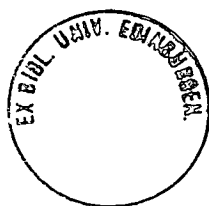


Knowledge Based Techniques in Plant Design for Safety

Anthony Waters

Thesis presented for the Degree of
Doctor of Philosophy
University of Edinburgh
February 1991



Abstract

This thesis is concerned with computer support for the Loss Prevention activities that take place during process design. The scope is deliberately wide because the central problem is to “get the design right”. This in turn requires consideration of “What is a good design?” and “How can one represent designs computationally?”. Added to this are strategic issues of how a plant design should proceed and how safety techniques can best be integrated into the overall design structure. Thus one needs to address both the fundamental questions concerning the nature of reasoning and the database/communication problems of managing a large design project. An example problem from each area is explored in this thesis, which has a common introduction but then divides into two parts.

The first part is devoted to “bottom-up qualitative reasoning”, which tries to predict system behaviour without performing detailed numerical simulations. The aim was to capture the sorts of reasoning that a HAZOP team might do whilst working through a set of failure scenarios. This problem is addressed both by the use of rules to directly represent causality and by the use of *qualitative simulation*, which is a technique from Artificial Intelligence (AI). The inferencing power of the latter approach is shown to be far superior but severe efficiency problems do result.

The second part addresses the problem of checking the validity and satisfaction of *designer's intention*, at all stages of design. To explore this the author has extended the features of an AI toolkit called Knowledge Craft, in order to allow its use as a Database Management System for design knowledge. The resulting software is called The Constraint Tools System, which has three main components:

1. The Constraint System, which is used to create constraints and to control their application throughout the design process.
2. The Propagating-relation System, which is used to specify mappings in data sharing situations and to automatically update equivalent values.
3. The Refinement Manager, which uses Knowledge Craft contexts to represent hierarchical design.

The use and interaction of these components is described in detail. There is also an ongoing discussion concerning the use of *decomposition* and *refinement*, in this and other AI-based modelling environments.

I hereby declare that this thesis has been entirely composed by myself, that except where otherwise attributed it describes my own original work and that except where indicated it has not been submitted, in whole or in part, for any other degree at any university.

Dedication

This thesis is dedicated to my family (who can now find out what I've been doing for the past 4 & 1/2 years), to my fellow researchers (who have to at least read the introduction), to my friends (who won't get past the list of contents) and to Nesta (whom I love very much).

Acknowledgements

I would like to acknowledge the following people for their help, or their distraction, during this thesis study:

- Jack Ponton (my supervisor) for providing resources, support and enthusiasm, without constraining too much the direction of my efforts. Alastair Struthers, Duggles Hutton and Murray Laing for technical discussions and general waffle sessions. Alastair, Gareth Jones and Neil Skilling for systems support, in chronological order. The members of the Computer Discussion Group (CDG) for the mutual support sessions. Elspeth Lakie for patience and perserverence when her afternoons work disappeared down Knowledge Craft's plug hole. The departmental secretaries, past and present, for keeping the department running. The other members of the department for the staff student football matches and the Burns Suppers.
- The Science and Engineering Research Council (SERC) for their financial support.
- The Artificial Intelligence Applications Institute for their support and resources in the early stages. In particular, Paul Chung and John Kingston for guidance and help with Knowledge Craft.
- ICI Engineering for financial support, and for taking the time to discuss my project aims and direction. In particular, Malcolm Preston and Jonathon Smithers for arranging many visits to the Process Systems Section; and the members of the process SHE sections for allowing me to participate in HAZOP studies for a real plant.
- My flatmates and friends for the leaving party and for not giving up on me in my more antisocial moments. Debbie Johnson for all the late night "Why am I doing a PhD?" discussions. Gordon MacAllan who'se been there from the very start. Alec Keith, for taking me out of myself on our once yearly ice (?) expedition. Gareth Yardley, for not being the climber. Ian Blackwood, for keeping my room and food cupboard in the manner to which they were accustomed. Annie, Ant (my namesake), Lis Cooper, John McKeever, Gareth Jones and Dave Sadler, for also being my flatmates - although they didn't stay the pace. Dave and Karen, for putting me up in their sumptuous abode. Wills, for calling it Ant's pad.
- My family, to apologise for all the birthdays I've missed over the last few years.
- Nesta, for all the love and attention while she was in Edinburgh; also for going off to Italy and leaving me to finish the thing in peace!

Close to the western summit there is the dried and frozen carcass of a leopard. No one has explained what the leopard was seeking at that altitude.

Ernest Hemingway, The Snows of Kilimanjaro

Contents

1	Introduction and Overview	1-1
1.1	Scope of the thesis	1-1
1.2	Loss Prevention at the Design Stage	1-3
1.2.1	Timing and manpower in Hazard Studies	1-6
1.2.2	Knowledge representation for Loss Prevention	1-8
1.3	Integrated Process and Plant Design	1-10
1.3.1	The payback from improved data handling	1-12
1.3.2	Multiple viewpoints and data mappings	1-13
1.3.3	The limitations of database technology	1-15
1.4	Relevance of Knowledge Based Techniques	1-18
1.4.1	Modelling, abstraction and decomposition	1-19
1.4.2	AI-based design environments	1-23
1.4.3	AI in Safety and Loss Prevention	1-26
1.5	Summary	1-30

Part I: Bottom-up Techniques

2	A Simple HAZOP Program Using Rules to Represent Causality	2-1
2.1	Introduction to HAZOP Studies	2-2
2.2	Other Work	2-5
2.3	Program Structure	2-7
2.4	The Deviations Explored in the Program	2-9
2.5	Deviation Generation and Propagation	2-11
2.6	Improving Causal Reasoning by the Use of Less Obvious Rules	2-16
2.6.1	Rules Which Model Wider Sections of Plant	2-17
2.6.2	Rules Which are Derived by Algebraic Manipulation	2-18
2.6.3	Rules Which deal with semi-numerical Information	2-19
2.7	Conclusions	2-20
3	Qualitative Simulation and Fault Propagation in Process Plants	3-1
3.1	Overview	3-1
3.2	Introduction	3-2
3.3	Example Problems	3-4
3.4	Approaches to Qualitative Simulation	3-9
3.4.1	The Steady state Approach	3-10
3.4.2	The Dynamic Approach	3-15
3.5	Extending the DeKleer method	3-18

3.5.1 Representation 3-18

3.5.2 Capabilities and Limitations 3-23

3.5.3 Further developments 3-29

 3.5.3.1 Confluence Solution 3-30

 3.5.3.2 Finding Interstate Behaviour 3-33

3.6 Related Work by Chemical Engineering Researchers 3-34

 3.6.1 Fault Diagnosis 3-34

 3.6.2 Operating Procedure Synthesis 3-37

 3.6.3 Fault-tree synthesis 3-41

 3.6.3.1 The tree versus the structure approach 3-42

 3.6.3.2 Applying principles from qualitative simulation 3-48

3.7 Summary and Conclusions 3-52

Part II: Constraint Tools

4 Managing Constraints within Design 4-1

 4.1 Introduction to Part II 4-1

 4.2 Different Uses of the term ‘Constraint’ 4-3

 4.3 Classifying Constraints by their Use and Purpose 4-5

 4.4 Further Classifying Requirement Constraints 4-7

 4.5 Representation & Implementation Issues 4-13

 4.6 What Constraint Tools Do We Need? 4-17

 4.7 Summary 4-18

5	An Overview of Knowledge Craft and The Constraint Tools System	5-1
5.1	Introduction to Knowledge Craft	5-1
5.2	Other Work Using AI-toolkits	5-4
5.3	The Extended Knowledge Craft Environment	5-7
5.3.1	Knowledge Representation Utilities	5-8
5.3.2	Pseudo Multitasking	5-8
5.3.3	Existing Graphical Workbenches	5-11
5.4	Tools for Constraint Representation	5-14
5.4.1	Object Hierarchies and Inheritance	5-14
5.4.2	Relations	5-17
5.4.3	Query Languages and Mapping Functions	5-20
5.4.4	Access Oriented Programming	5-20
5.4.5	Walnut and The Constraint System	5-22
5.4.6	The Path-Spec Special Syntax	5-24
5.5	Data Mapping and Propagation	5-26
5.5.1	Uses of Abstraction	5-26
5.5.2	The Propagating-relation System	5-28
5.6	Hierarchical Design	5-32
5.6.1	Contexts and Refinement Trees	5-32
5.6.2	The Refinement Manager	5-37
5.7	Summary	5-40

6	Design Examples Using The Constraint Tools System	6-1
6.1	Editing Background Information	6-1
6.2	An Example for the design of Continuous Plants	6-18
6.2.1	Editing Structure	6-18
6.2.2	Dealing with Versions	6-32
6.2.3	Saving all changes	6-40
6.3	An Example for the design of Batch Plants	6-43
6.4	Summary	6-49
7	Decomposition and Refinement in other Modelling Environments	7-1
7.1	MODEL.LA	7-2
7.1.1	Semantic Relationships	7-3
7.1.2	Multifaceted Modelling	7-6
7.2	ASCEND	7-11
7.3	CLAP	7-18
7.4	ModAss	7-19
7.5	Summary	7-20

8 Overall Summary and Conclusions 8-1

8.1 Overall Summary 8-1

8.2 Future Directions 8-9

Bibliography Bib-1

Part III: Appendices

A Some qualitative rules for bottom-up reasoning A-1

A.1 Causes of a Deviation A-1

A.2 Rules for the Propagation of a Deviation A-3

A.3 Rules for the Consequences of a Deviation A-7

B More Detail on the Qualitative Simulation Program B-1

B.1 Notation for Chapter 3 B-1

B.2 Code and Algorithms B-1

B.3 Unit Models B-7

B.3.1 Modelling a Relief Valve B-7

B.3.2 Modelling a Simple Resistance Element B-13

B.3.3 Modelling a Split, a Junction and a Line Blockage B-14

B.4 An Example Run B-17

C A list of Example Loss Prevention Constraints C-1

D Extensions to existing Knowledge Craft Tools D-1

D.1	Using Palm for Drawing Networks	D-1
D.2	Using Palm to Simplify Knowledge Craft Programming	D-3
D.3	Small Palm	D-4
D.4	The Fixed-menu Builder	D-4
D.5	The SUN COMMON LISP Editor	D-5
D.6	Graphics and the Command System	D-5
D.7	The Save Manager	D-6
E	File Used to Load the Constraint Tools Examples	E-1
F	Using CRL-OPS Rules for Partial Simulation	F-1
F.1	CRL-OPS and Propagating-relations	F-2
F.2	The heating-circuit example	F-3
G	Example File of Rules To Maintain Simple Data Dependencies	G-1

To climb steep hills requires a slow pace at first.
Shakespeare, Henry VIII, Act 1, Sc. 1

Chapter 1

Introduction and Overview

1.1 Scope of the thesis

This thesis will be primarily concerned with computer support for the identification of loss scenarios that arise through faults in chemical plant design. That is, with support for Loss Prevention through adequate design. However, it has not been appropriate to keep the study within too narrow bounds because we must address the fundamental problem of “getting the design right”. This in turn requires consideration of “What is a good design?” and “How can we represent designs computationally?”. Indeed, the skills needed for assessing the safety of a plant design are mainly those needed to do the design in the first place i.e an understanding of chemical engineering principles at both qualitative and quantitative levels. Added to this are strategic issues of how a plant design should proceed and how safety techniques can best be integrated into the overall design structure. For example, as with design, the success of a safety assessment method depends strongly on the reliability of the design information and the means of its communication. Thus one needs to address both the fundamental questions concerning the nature of (qualitative) reasoning and the database/communication problems of managing a large design project.

By expanding this discussion Chapter 1 tries to show how different parts of this thesis may be of interest to the following types of researcher:

1. Those interested in the provision of computer support for Loss Prevention activities which occur during process design.
2. Those interested in process engineering databases, integrated design and design environments.
3. Artificial Intelligence (AI) workers interested in design environments, qualitative simulation and the frame-based representation of knowledge.

Each of these are large research areas in themselves so some sub-divisions are necessary. In addition, not all of the thesis will be of interest to each area so pointers to different chapters or sections will be a help. Chapter 1 serves both of these functions by discussing each area in turn and describing the scope of the work carried out for this thesis. This chapter is therefore set out in the order in which the areas are listed above. However, the thesis itself is set out in approximate chronological order. That is, the different areas of work are described in the order in which they were carried out. There are two broad divisions within this work:

1. That concerned with 'bottom-up' qualitative reasoning, which was conducted using the language Prolog.
2. That concerned with the use of constraints within process design, which was conducted using an AI Toolkit called Knowledge Craft.

To some extent this thesis is also subject to these divisions; but it has not been appropriate to present entirely self contained parts. In particular, a common introduction is provided within this chapter and a common summary in Chapter 8. In addition, the Bibliography and Appendices for both areas are grouped together. Finally, some common themes, particularly representation issues, run through both areas.

1.2 Loss Prevention at the Design Stage

Loss Prevention is defined by the Institution of Chemical Engineers [44] as

..a general term used to describe a range of activities carried out in order to minimise any form of accidental loss, such as damage to people, property or the environment or purely financial loss due to plant outage.

This is a useful definition because it emphasises that hazards and operating problems are inextricably linked, particularly at the design stage. Good design will obviously have its payback in terms of efficiency and operability but will usually enhance safety and ease of maintenance also. However, there are trade-offs which must be addressed in future work. For example, a low inventory is good for inherent safety but makes plant control more difficult. Research into the design of control systems should be addressing this issue.

Lees [63] has reviewed the build up of industrial interest in loss prevention. He states that between 1962 and 1969 the maximum size of ethylene plants increased from 100 Kte/year to 450 Kte/year and other plants showed similar behaviour. This was accompanied by a tendency for plants to operate at more extreme conditions, made possible by the development of relatively sophisticated protective equipment such as that used on the ethylene oxide process in 1971. These trends have continued, enhanced by the sudden increase in interest in fuel conservation after the 1973 oil crisis. Large plants tend to be single stream and have large material hold-ups. Consequently, severe problems can arise during commissioning, start-up and operation. The disaster potential of modern plants is very large.

Public interest in the hazards of the chemical process industries has not been far behind industrial interest over the last 15 years. The explosion at Flixborough in June 1974, which caused \$36 million worth of damage and killed 28 people, attracted an enormous amount of media attention [43]. Since then further incidents, particularly Seveso (1976), Bhopal (1984) and Chernobyl (1986), have continued to stimulate the public debate. A working party of the I.Chem.E. [44] has attributed continued public

interest since Flixborough to a variety of causes. Among these are U.K. legislation recommended by the Advisory Committee on Major Hazards in 1976 & 1979, European legislation stimulated by the Seveso incident and the popularity of the European Green Movement. In addition, they mention the planning enquiry concerning Canvey Island, which introduced the public to quantitative risk assessment.

Looking to the future we can expect such legislative and public interest to continue. However, the emphasis will probably shift away from being primarily concerned with the immediate damage potential of plants to becoming increasingly concerned with the environmental impact of both continuous and accidental releases. This is causing some companies to widen the scope of their formal loss prevention activities to encompass Health and the Environment as well as Safety (e.g. see Turney [106]). This combined area is coming to be called Process SHE.

Duxbury and Preston [26] have described ICI's existing 6 stage approach to loss prevention, which is paraphrased in Figure 1.2. Turney [106] has shown how these stages can be extended to cover Process SHE by focusing on the nature of all the possible chemical releases from a plant. Thus we see that Environmental Protection has become a motivating force for change, or evolution, of the existing Loss Prevention methods¹. Another factor is the increasing emphasis on batch or semi-batch production [83, 90]. This is likely to lead to changes in both the timing and scope of the current hazard reviews; though to be fair, a flexible approach has always been needed for retrofits. Perhaps in the future we will see a formalisation of such concepts, together with the proper integration of Health, Safety and the Environment.

These issues are not explicitly pursued in this thesis, as the work described herein has been primarily concerned with the existing 'model' of Loss Prevention for continuous plants. In addition, most of the discussion centres on *Hazard Identification*; rather than its numerical counterpart *Hazard Analysis*, which is used to assess the probability and consequences of an identified scenario. Nevertheless, the thesis covers many different areas, some of which may be applicable to an enhanced SHE model of Loss Prevention or to the design of batch plants.

¹Indeed, Process SHE as a whole seems to be becoming a motivating force for change within large chemical companies, particularly with respect to computing.

Hazard Study 1 (project exploration stage): This ensures that the hazardous properties of all the materials involved in the process and their potential interactions are understood. Also that full consideration is given to constraints imposed by the local environment and by legislation.

Hazard Study 2 (Process Flow Diagram (PFD) available): The main plant sections are studied in turn and potential hazards identified in a top-down manner. Fault-trees and quantified hazard analysis are used where appropriate. Some hazards will be designed out, others will be prevented by protective equipment.

Hazard Study 3 (Engineering Line Diagram (ELD) Completed): This is conducted as a Hazard and Operability (HAZOP) study by the design team and a HAZOP study leader. The ELD is examined on a line by line basis. A list of 'guidewords' are used to stimulate the imagination of the study members into identifying causes and consequences of process deviations. Thus problems are identified in a bottom-up manner, which makes the study intensive and laborious, though thorough.

Hazard Study 4 (Pre start-up): The commissioning manager conducts this study to ensure that all the recommendations made during hazard study 3 have been carried out. In addition, he or she must ensure that operating and emergency procedures are available.

Hazard Study 5 (After construction): The operating management are responsible for conducting this on-site inspection to ensure that the workforce are protected against everyday physical hazards and have adequate means of escape. Physical hazards, such as tripping or slipping, continue to cause more deaths or injuries than process hazards.

Hazard Study 6 (After some operation): This study reviews changes made during commissioning and early operation. It provides feedback to the design team and information for ongoing control and auditing tasks.

Figure 1.1: ICI's Hazard review Procedures for New Plant

For further information about the hazards associated with chemical plants and the existing techniques used to prevent them see Lees [63] or Wells [114]. In addition, Kletz [50] presents some interesting accident case-studies.

1.2.1 Timing and manpower in Hazard Studies

This thesis discusses the applicability of Artificial Intelligence (AI) techniques to the support of Loss Prevention throughout the design process. With respect to this one can identify two aims for the provision of computer support:

1. To alleviate some of the burden of the hazard reviews. An obvious way of doing this is to partially or totally automate the study concerned. An alternative is just to help collate and present the data needed within, or produced by, the study.
2. To provide tools that can be easily used throughout the design process; thus facilitating an incremental approach to design-stage Loss Prevention.

These problems are related because the hazard reviews use reasoning techniques that could be usefully applied at many stages of design. Thus one would expect that tools to automate parts of a study would also be useful long before the study took place. Conversely, the early identification of design errors should make the hazard reviews easier by simply reducing the number of mistakes that the study team must consider.

One can see from ICI's hazard study procedure that the exploration of the design, and the identification of design faults, is only the main activity of the first three of the six review stages. However, it is these stages which consume the largest resources and have the greatest impact on the economic viability of the plant. In particular, Hazard study 3 is an extensive study that can tie-up the top design personnel two or three times a week, for several weeks (or even months) [49]. Thus there is an obvious financial incentive to support the hazard reviews in their present form. More importantly, however, one might hope to allow the review techniques to be used at more stages of the design. This is not possible at present because of the scale of the commitment involved, particularly for Hazard study 3. Given this, the only logical time to conduct this study is when the ELD has been finished i.e. when the chemical engineering is essentially completed. This is not the ideal situation since mistakes will be identified rather late, with significant financial penalties.

At a late stage :**Benefits :**

- Design well developed.
- Hazard Study an effective check

Disadvantages :

- The design is frozen; the main plant items are often on order.
- There is little opportunity for inherently safe design.
- Safety can only be achieved by 'bolt-on' systems.

At an early stage :**Benefits :**

- There is still enough flexibility to consider inherently safe design.

Disadvantages :

- The design details have not been fully thought through.
- The Hazard and Operability studies become design meetings.
- If there are too many changes then the final design will need to be re-studied.

Figure 1.2: Issues Affecting the Timing of Hazard and Operability Studies

Turney [106] has discussed these issues which affect the timing of Hazard study 3 (see figure 1.2). Similar considerations will apply to the other hazard studies; indeed, the central problem is one of using discrete review stages instead of integrating Loss Prevention considerations properly into the design procedure. An alternative might be to provide tools that can be used near-continuously, i.e. *incrementally*, throughout the design process. Figure 1.3 lists some of the advantages and disadvantages that may arise from this approach. From this one can see that the choice is not clear cut because the hazard reviews are useful for project control. A possible scenario therefore is the gradual introduction of incremental tools with the eventual scaling down of the hazard reviews to just provide an audit function. This has the advantage that one can test

out new tools without taking anything away, which is likely to be an important issue in an area as sensitive as Loss Prevention.

Advantages of Discrete Hazard Reviews:

- This approach allows for formal studies at “set” stages of design.
- It forces a Loss Prevention structure on the overall design process. This is good for project control because one knows that certain data should be available by a certain stage.
- It allows for the involvement of personnel outside the design team. They can then provide a more objective assessment of the quality of the design.
- It allows for the assessment of Loss Prevention procedures, both internally and by external licensing bodies.

Advantages of An Incremental Approach:

- It has the potential to identify design mistakes as soon as they arise.
- The SHE impact of design decisions can be considered at any stage.
- The chances of achieving inherent safety are very much greater.

Figure 1.3: Pros and cons of an incremental approach to Loss Prevention

1.2.2 Knowledge representation for Loss Prevention

Regardless of whether one is trying to support an incremental or a discrete approach to Loss Prevention, one must consider the problem of how to represent qualitative and judgemental knowledge. Traditional computer languages, such as FORTRAN or BASIC, are not well suited to this task. However, the field of Artificial Intelligence has provided us with tools and techniques that are specifically intended for such ‘symbolic’ and ‘non-deterministic’ (i.e. inexact) reasoning. Using these, one may hope to be able to construct tools for problems such as automatic fault-tree synthesis or tracing HAZOP deviations through a flowsheet. Indeed, the emphasis on qualitative knowledge means that the potential exists to create a whole set of tools that span the complete

design process and give the best answers possible, based on the information currently available. At the early stages of design this may necessitate reasoning in the partial or total absence of numerical information; while in the later stages, detailed structural and numerical information can be used.

This proliferation of tools would itself raise some further questions concerned with data handling:

1. How do we retrieve the information that a particular tool requires?
2. How do we pass it to the tool?
3. Where do we put the results so that they can be used by other tools?
4. How do we know when the information is available so that we can apply the tool at the appropriate points in the design?

Of course, these are the sorts of questions that have already been addressed by those concerned with *Integrated Process Design*. That is, with the provision of design environments that allow for the smooth integration and management of tools from many disciplines. There has been a large body of work in this area but little of it has directly addressed the requirements of Loss Prevention. This is unfortunate because the problems here seem to be particularly severe, for the following reasons:

1. We need to handle many types of non-numerical data e.g:
 - Structural data about the design topology.
 - Functional data and the *Designer's Intention* for particular systems or pieces of equipment.
 - Non-numerical design attributes.
 - Legislative requirements, constraints on the use of materials and so on.
2. To allow for an incremental approach we need to be able to reason about this data at different levels of detail, at different point in a design.

3. We can conceive of hundreds, if not thousands, of 'little checks' that it is worth making as the design proceeds. However, few of these will be used if the plant data must be typed into the terminal for each one. In addition, we need a structure to tell us when the different checks should be applied.

Again, tools from AI provide potential solutions to some of these problems, though they do ignore many of the efficiency problems addressed by traditional database research. Such issues will be discussed in the next section, which will review some of the work on integrated design. This is in preparation for Part II of this thesis, which considers the use of an AI Toolkit called Knowledge Craft² to support an incremental approach to Loss Prevention.

1.3 Integrated Process and Plant Design

Motard [72] discussed the future of computers in chemical engineering in 1983 and noted that most computing at that time took a program centred, rather than a data centred, approach. This results in a host of stand-alone programs that produce data in a non-standardised manner, which makes interaction between them very difficult. The solution is said to be a data centred approach to program development, perhaps using Database Management (DBM) techniques from computer science or the business community. Unfortunately, the type of data required for a process design is not well suited to traditional DBM systems. These issues are considered in this section, together with the possible advantages of using AI tools.

Existing solutions to the data handling problem have concentrated on extensions to commercial *Data-base Management Systems* (DBMS). The most recent system is Prodbas from Prosys Technology Ltd [117], which is based on the relational data model. This is probably the "state of the art", from a technology point of view. However, ICI [68, 81] have probably taken integrated design further than anyone with their suite of Provue systems (Provue-db, Provue-2d & Provue-3d). Indeed, they claim to have achieved a limited form of integrated process *plant* design within the recent LCA am-

²Knowledge Craft is a registered trademark of Carnegie Group Inc., Pittsburgh.

monia project, thus crossing the artificial demarcation between process design and plant design. There is still much to be done, however, just to integrate the use of tools from the same domain, using similar types of data, let alone to integrate tools from disciplines as different as chemical engineering and control engineering. In particular there is a need for powerful and robust ways of representing data from many different viewpoints and maintaining consistency between them. Struthers [99] has considered the requirements in this area and defined two levels at which information integrity must be considered:

The integrity of data which he defines as follows:

The consistent use of data in design activities in terms of the availability, uniqueness of value, and explicit functional constraints imposed on the data. The notions of uniqueness and availability are not fixed but wholly dependent on the implementation of a data access control mechanism which then defines what these terms mean. For example, in a multi-user system which employs a data release strategy, in which each user operates on a private version of part of the project data base until a new global data update is released, then uniqueness and consistency of data can only be defined relative to a user's local viewpoint. In such a system, therefore, global consistency can only be periodically enforced when data updates are released which may or may not be a problem depending on how interdependent user's activities are.

The integrity of knowledge which he also defines:

There must be an explicit representation of the assumptions and principles upon which the data is based as well as an explicit knowledge of how that data can be used and manipulated in terms of data dependency, both of which must be consistently maintained throughout the design process. This can be thought of as being consistent about the actual reasons behind data constraints, which in turn requires that one is aware of the background knowledge, theory, and decisions responsible for the creation of the data in the first place.

It is this second, higher level that must be addressed if one is really trying to solve the "integrated design problem".

1.3.1 The payback from improved data handling

James [46] has discussed the present methods of data management in process design projects. Reviews of time utilisation by engineers indicated that information management is a major work component. This is borne out by the experience of Esso Research and Engineering which yielded the following typical breakdown of engineering time utilisation:

Data retrieval	20-25 %
Analysis or calculation	20-30 %
Data issue or manipulation	35-40 %
Planning or administration	15-20 %

Information management therefore requires 55-65 % of the engineers available time, which suggests a large payback for improvements in the automatic handling of design information.

To back this up one needs to consider the value of a process engineer's time. In fact, on a simplistic analysis one may be misled because the process engineering cost is typically less than 5 % of the total project "spend" (Craft [17]). However, Craft also notes that up to 80 % of the total project cost can be fixed by the time the Process Flow Diagram is finalised. That is, decisions made in the initial stages of design severely restrict the options downstream. This means that improvements in front end design may have a large return due to improvements in the final design product. Furthermore, one should also consider the effect of speeding up the design process, which could be a secondary consequence of improved data handling. Again, this is a factor which may not directly result in reduced project costs but could nevertheless provide a large economic benefit; this time through commercial advantage and easier company planning. Consequently, there is significant payback to be had by improving the time utilisation of the process design team. Preston [81] portrays this situation by the diagram shown in Figure 1.4.

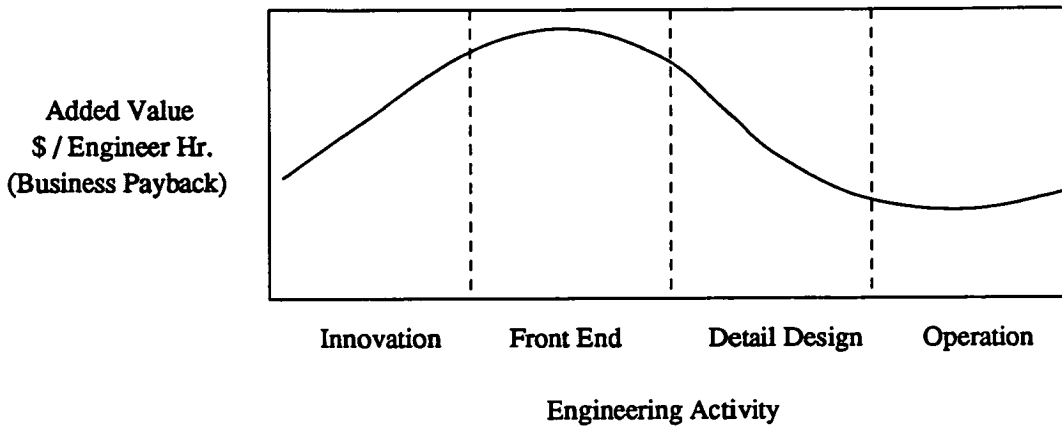


Figure 1.4: Payback of investment in different stages of a plant project

1.3.2 Multiple viewpoints and data mappings

Integrated design solutions based upon commercial DBMS technology seem to allow the integration of existing design programs in a relatively unstructured manner. Most of them provide a central repository for design data, which can be updated or read by any of the programs. Thus there is a central data model of the design. This means that each program only has to 'know' how to read data from the central model and how to write it back out again. Furthermore, one is given some guidelines as to the best I/O format for subsequently developed programs. The alternative to this approach is to proceed without a central model and so require translation programs to pass data between each pair of programs that might ever need to 'talk' to each other. This is unlikely to be a feasible approach unless the number of programs to be integrated is rather small³. Nevertheless, some researchers still choose to take this route (e.g. [80]).

Having decided on the concept of a central repository one now needs to ask whether one central model can be enough to cover all stages of design. This seems rather unlikely as one must allow for the representation of the plant at widely varying levels of detail. The alternative is to maintain several models of the design and map the

³ICI has over 80 programs in its Process Engineering Library!

information appropriately between them. For example, one might describe the design using the Block Flow Diagram (BFD), Process Flow Diagram (PFD) and Engineering Line Diagram (ELD) viewpoints that are currently used⁴. Existing systems seem to be able to maintain such views but not to map information between them in a general way. The result is that it may be difficult to carry information from the earlier stages of design and use it automatically later on.

One-off solutions to this problem may exist for particular tasks; for example, one may be able to take the temperatures and pressures calculated in a simulation run (PFD level) and place them on spec. sheets for the relevant pieces of equipment (ELD level). However, it is unlikely that one can do this for mappings that are complex and not fixed a priori. For example, it would probably be difficult to associate an effluent stream on the initial BFD with the effluent points on the final ELD. This makes it difficult to automatically check the effluent limits that are identified in Hazard Study 1.

Later chapters of this thesis describe the use of an AI Toolkit (Knowledge Craft) to explore such representational problems. The implication of this is not that such toolkits are themselves viable solutions to the integrated design problem. However, they do allow one to explore some of the issues involved. For example, the use of Knowledge Craft has facilitated the creation of a tool that allows the hierarchical representation of a design and the automatic propagation of information between the levels. This, in turn, allows for the easy representation of design alternatives, which can be done on two levels:

1. At the level of considering alternative structures for a given subsystem of the design.
2. At the level of considering alternative cases for the same structure. For example, one may simulate the operation of the plant at 30 % turn-down as well as at the base-case design. Most of the existing process design DBMS's already provide this facility.

⁴Note that we are not considering the implementation question of whether the data should be stored in more than one physical location. Rather we are considering the 'semantic' question of what the data means and how it inter-relates.

In addition, this tool allows the easy specification and automatic checking of constraints upon the design. To follow up a previous example one might use this facility to check that the final design obeys the effluent limits identified in Hazard Study 1. Furthermore, one might set up this check to run automatically, so that it fires whenever there is a change to one of the effluent stream flowrates.

1.3.3 The limitations of database technology

The above discussion has highlighted the inability of the current process design DBMS's to adequately support the use of multiple design viewpoints. This makes them very difficult to use as vehicles for providing a structure to the whole design process because they cannot be used continuously in a seamless manner. Thus it is difficult to attack the integrity problem along the time axis. Furthermore, the integration of tools from different disciplines becomes almost impossible because there is no way of transforming data between the different representations that each of them uses. Thus it is also very difficult to maintain integrity across different engineering functions or departments. Based on these limitations one can hardly claim that the existing solutions are capable of achieving truly integrated design.

So what are the reasons for these limitations? To some extent they stem from a basic mismatch between the technology and the problem. Various authors [8, 7, 14, 15] have discussed this when considering the data handling problems for a complete process plant design. Typically, a chemical process plant project uses one or two megabytes of data per million dollars of investment. Unfortunately the data is highly non-homogeneous and cannot be stored in the manner of files and records used in the common hierarchical data base systems. In addition access requirements to data are infrequent but complex, often with several users requiring access at once and so maintaining data integrity is likely to be very difficult. The differences between classical data management and that required for process plant projects is summarised in Table 1.1 [8].

The question now arises as to how AI technology can help one to remove some of these limitations. The simple answer to this is that AI tools are intended for the representation of knowledge in as natural and declarative a manner as possible. For

	Classical Data Management	Engineering Data Management
Object	Record	Set of non-homogeneous records (complex objects)
Data types	Formatted data: scalars, reals, short strings	Same, plus unformatted data: vectors, matrices, long strings
Data inter-relationships	Simple	Complex and many-faceted
Transactions	Frequent, short, few records, simple access	Infrequent, long, many records (operate on sets of records), multiple accesses, graphics

Table 1.1: A Comparison of Data Management Requirements

example, an AI environment would typically allow one to build any number or size of ‘objects’, ‘rules’ or strings of text to represent qualitative and structural knowledge. In contrast, a commercial database would probably require one to deal directly with records or tuples, thus hiding the structure within the database. Furthermore, some systems may have a limitation on the size of any single object, which would force one to make further artificial divisions. In such a system what one sees bears little resemblance to the knowledge that is implicitly represented. This can hardly fail to make knowledge manipulation more difficult and knowledge integrity unrealistic.

A further point made by Struthers [99] is that the structure of engineering data is much less static than that of the usual data held in business records. Partly this is because one is unlikely to be able to anticipate, in advance, all the data types and relationships needed for a large engineering project. Furthermore, some of these relationships will change anyway due to the existence of new products, new sources of information or changes in the engineering organisation. Indeed, the latter kind of change will probably come about precisely because of the introduction of an integrated environment which requires a large change in working practices. The conclusion is that one must accept

the need for a certain amount of structural maintenance, particularly when a system is first released. This is said to be a problem for commercial DBMS's, which are not designed for frequent changes to schema definitions⁵. In contrast, AI systems are usually interpreted, which means that any kind of object or link can be created dynamically.

Another problem is the effect of long, as opposed to short, transactions. Most commercial DBMS's maintain data integrity by locking users out of a record if it is already being used (or maybe just if it is currently being changed). In addition, if several records are related then users may be locked out of all of them whilst any of them are already being altered. This approach causes problems for process engineering because the data is highly interconnected and the transactions may take days, or even weeks. Consequently, large portions of the data-base could be closed for long periods at a time; which makes a mockery of integration.

These comments should not be taken as implying that current AI environments offer even a partial solution to the integrated design problem. This is certainly not the case because they fail to address some of the more basic database issues, such as efficiency or concurrency control⁶. However, what these comments do imply is that the principles of representing information in a natural manner, and providing mechanisms for maintaining alternative viewpoints, should be given high priority within database research. Thus there is a need for the integration of AI and database (DB) techniques. Fortunately there are many people working in this area so rapid progress can be expected. Indeed, Brodie [9] has conducted a survey which demonstrated that in 1988 over two thousand person years were spent on projects using a combination of both AI and DB technology!

⁵Though they are, of course, designed for frequent changes to the data that is stored using those definitions i.e. the data in the data-base. The distinction between structure and data is important in commercial DBMS's but becomes less rigid when one moves into the area of *knowledge bases*.

⁶Concurrency is wholly unsupported by most AI software, including Knowledge Craft.

1.4 Relevance of Knowledge Based Techniques

These sections will review some of the attempts at applying AI techniques to the problem areas associated with Loss Prevention and integrated design. Some parts of this discussion will be based on a review conducted in August 1989 [42]. However, the reader is referred to the original source for a comprehensive overview of AI applications in Chemical Engineering. In addition, earlier reviews have been given by Sririam et al [93, 94]. There are also several papers which discuss the nature of chemical engineering design and the applicability of AI techniques (e.g. [65, 97]).

This thesis shall not make too great a distinction between the terms *Expert System (ES)*, *Knowledge-based System (KBS)*, *Knowledge-based Expert System (KBES)* and *Intelligent Knowledge-based System (IKBS)*. The same thing applies to *Artificial Intelligence techniques* and *Knowledge-based techniques*. All these terms are used relatively interchangeably in the literature, so the terminology is not really worth worrying about. However, there are a couple of points worth making:

Firstly, using the term "Knowledge-based" does become significant if it means one is addressing Struthers' problem of maintaining the integrity of knowledge. To some extent this applies to the work in this thesis, though obviously it can do no more than scratch at the surface of this many faceted issue.

Secondly, the thesis work is not restricted to the usual narrow definition of a problem that is suitable for an "expert system" approach. Such a definition may go as follows [86]:

1. The task should be primarily symbolic rather than number crunching.
2. A friendly articulate expert should be involved from the start.
3. The task should be significant i.e. worthwhile.
4. The domain should be small. There should be at most a few hundred terms in the basic vocabulary.

5. The problem should be one that a human can do given something between a few hours and 2 weeks. The kind of problems that people handle over the phone are good candidates.
6. All the experts who will vet the finished product must be in agreement as to the nature and methods of the problem solving (i.e. they must be able to come to similar conclusions most of the time).
7. The system should start small and be expanded as confidence increases. It should be tried out on test cases.

If one is considering all the tools that may be needed to fully examine Loss Prevention at the design stage then obviously one is outside this narrow problem scope. In this case any analogies to 'classic' expert systems are not applicable. This is indeed the case for the problems addressed in this thesis, such as that of maintaining the integrity of knowledge (or even just that of data). This has implications for the cost of the systems one may plan to develop, the degree of automation that can be expected and so on. This explains why the last section carefully defined the payback to be expected from investment in integrated design.

1.4.1 Modelling, abstraction and decomposition

Lien, Suzuki and Westerberg [65] have discussed the role of Expert Systems technology in chemical plant design. In order to do this they pose the following questions:

1. What is the nature of design?
2. Why are experienced designers good at it?
3. How and to what extent can computers do it?

In answer to the second of these questions they discuss the ability of an expert designer to:

1. Model a chemical process on multiple levels and relate these models to each other.
2. Formulate strategies for attacking the problem.
3. Switch between goals and reformulate strategies as a response to preliminary results.
4. Accumulate knowledge, both factual and strategic, along the way.

The nature of design has always been a fairly open area for research. However, one of the observations that one can make is that design always involves modelling. Furthermore, it often involves the use of *abstraction* and *decomposition* to create and manipulate these 'models' at different levels of detail. The following are some examples of how one can use these complimentary techniques:

1. We use decomposition for strategic reasons when we split a problem into manageable sub-parts. Often those sub-parts will be further split until we get to parts small enough to consider as a whole (or to be bought off the shelf). The procedure may branch to look at alternatives for a given sub-system.
2. We *refine* a design through successive levels, again for strategic reasons. At each level we can perform some tests in order to determine if the design still looks promising at the current level of detail. If so, we go on to the next level. This is the approach of Douglas [48] who suggests six fixed levels of decision making when performing a design:
 - (a) Continuous vs. Batch.
 - (b) Input-Output Structure.
 - (c) Recycle Structure.
 - (d) Separation System (deal with vapour recovery and liquid recovery separately).
 - (e) Energy Integration.
 - (f) Detailed Process Alternatives.

The implication is that one refines the whole design to the next level at the same time. However, this restriction does not have to be made; indeed, refinement must also be used in the branched decomposition strategy just described in (1).

3. We can use multiple levels of representation simply to maintain an understanding of a design at different levels of detail. If one sees too much detail at once then the overall picture is obscured and it becomes difficult to see the purpose of what one is looking at. This problem is already addressed within a chemical plant design by describing it on three different diagrammatical levels i.e. the BFD, the PFD and the ELD.
4. Sometimes abstraction is needed to understand a more detailed solution. For example, in order to understand the results of a complex numerical simulation we might need to compare them with the results of a much simpler simulation of the same physical phenomena. An example of this would be modelling a multicomponent distillation as if only the two key splits were present in the column. That is, abstracting it into a binary distillation problem which can be represented using a McCabe Thiele diagram.
5. Similarly, abstraction is sometimes used to help solve a more detailed model e.g. by getting starting guesses for a numerical iteration.

The Knowledge Craft work described in this thesis has addressed some of these uses of abstraction and decomposition. In particular, it has used Knowledge Craft's *context* mechanism to allow a branched decomposition representation of the design, similar to that described in usage (1) above. However, this is not really for strategic reasons but rather to allow a multi-level understanding of the design so far (i.e. usage (3) above). An interesting point here is that this multi-level representation contains branches, so it can theoretically embed all the alternatives one has explored along the way. In addition, one can represent the notion of the current design by selecting between alternatives at all the choice points in the tree. As already described in a previous section, this allows one to distinguish between alternatives which are structurally different and ones which just differ in attribute values (i.e. are different cases of the same basic design).

This use of a hierarchical design representation is not really for strategic reasons. Neither is it to present a *history* of the design process; in no way can it do this since there is no time stamping of the data as it is entered. Instead, it just provides a retrospective view of the information that has been generated so far and the avenues that have been explored. Clearly there are many different uses for abstraction and decomposition.

Since they wrote this early paper Lien and Westerberg have explored several different ways of providing general, AI-based tools for design. Lien has constructed a blackboard system called AKORN D [66], which will be discussed in Chapter 5. Westerberg has had some involvement with this but also supervised other projects which take a rather different approach. One of these is the ASCEND modelling system [78] which helps with the creation and solution of mathematical models for a complex system. Most importantly, a complete model can be created by “merging” together models of the system sub-parts. Thus it addresses some of the hierarchical modelling issues discussed above. The system has been written in PASCAL but is clearly based on Object-Oriented techniques.

There are several other institutions working on modelling ‘languages’ comparable to ASCEND. In George Stephanopoulos’ group at MIT they have been constructing a system called MODEL.LA [97], which is based on an AI-Toolkit called KEE. One interesting aspect of this work is the use of a Backus-Naur Form not only to define the domain independent aspects of the language but also to define the construction of chemical engineering models. Thus they claim that MODEL.LA has a “well defined semantics”. Another system is called CLAP (Combined Logic and Procedures). This is a Prolog-based environment constructed by Struthers [99] at Edinburgh University. It differs from ASCEND by being aimed more towards the end-user than a modelling expert. For example, it provides the facility to reason about models so the best model to use can be automatically selected based on the currently available data. This operation could be made transparent to the eventual end-user but it does not have to be.

The work described in this thesis has not considered modelling in any clean or well-defined mathematical sense. However, it has explored several representational techniques which may be relevant to these modelling environments. Such comparisons will be made in Chapter 7. In addition, the next section reviews some work on process

synthesis and the more general problem of providing a design environment. Again, this thesis does not address these issues directly but may provide some insight in ways to proceed. Indeed, one way of considering the work described in part II is as an evaluation of the use of Knowledge Craft as a DBMS. Of course, this has been purely from the point of view of the representational tools provided by Knowledge Craft, so the issues of efficiency and concurrency have been completely ignored.

1.4.2 AI-based design environments

Having identified the use of strategic models in design Lien et. al. [65] go on to review some existing attempts at process synthesis. These broadly divide along the lines of taking an optimisation approach (e.g. [25]) and taking a heuristic approach (e.g. [48]). One might be tempted to call the latter an AI approach, which it is to some extent. However, although the heuristics and short-cut design methods have been worked out in some detail [22] the implementation of them in the PIP system [48] is rather naive. In particular, there is no notion of defining the constraints upon the particular design problem that one is trying to tackle. Consequently, all PIP will be able to do is to run through a pre-defined set of design rules, asking about the applicability of pre-defined constraints (i.e. ones not defined by the user). This is not to say that PIP cannot be useful but it does make it important to clearly define what it is intended for. Really it is not doing design but rather generating ideas at a pre-cursor stage. These would then have to be interpreted and developed by a design team in order to construct a viable flowsheet. The PIP concept cannot be scaled up to the task of doing a full design.

At the other end of the scale in terms of size and complexity Subrahmanian et al [100, 115] have been looking into the problem of communication during a large design project. The researchers spent a year observing the progress of a Westinghouse project for the design of a new control system for a coal fired power station. One of the strongest impressions they got from this was the sheer volume of textual information generated and the subsequent problems of document control. At this level of the design problem they identify with the view of Bucciarelli [11] that *design is a social process*. In particular, they discuss Bucciarelli's notion that each participant in such a process operates within his own "object world(s)". The primary problem then is that none of them see

the design from the same point of view, which requires the design team to undertake several types of *discourse*:

Constraining discourse in which constraints are negotiated for the design.

Naming discourse in which a common vocabulary is negotiated.

Decision discourse in which the decisions impacting the design are negotiated.

Thus, Subrahmanian et al are in agreement with this thesis in identifying the need for multiple design viewpoints. However, the approaches diverge due to the heavy emphasis that they place on memos and textual information. The work described in this thesis has been exploring avenues that require the design to be represented using well-defined tools and languages, such as LISP. This allows a system to be automatically active since it has an explicit representation that it can reason about. However, if a design really is memo-based then there is a rather difficult translation problem. In contrast, the early ideas of Subrahmanian et al were based around the notion of a hypertext system, which would be much less active. This has now grown into a project called NDIM but the textual emphasis seems to have been reduced. Indeed, they seem to be moving away from hypertext, so more correspondence may develop between the two approaches. It is really too early, however, to see the form that the NDIM project will take.

Subrahmanian et al have defined some desirable features of a preliminary design environment. One of the central features they mention is the notion that once a model starts to be used by other members of the design team then it must be "cast in stone". That is, it can no longer be directly changed because of the consistency problem with another user's view of it. One can, however, create an entirely new model which is a revision of the original one. This raises the problem of how to alert other users to the existence of this new model so that they can start to use it if they wish. In addition, there is the related problem of how a user can switch to the new model and still maintain consistency. The author's make no comments on how this is to be done.

By using the notion of models being "cast in stone" but also revisable Subrahmanian et al require that an extensive design history is maintained. If it can be elegantly done, this would be a great advantage of the system that they envisage. Smithers [92] also

considers a design history to be important, though the nature and overall architecture of his proposed environment is rather different. This architecture is shown in Figure 1.5. There are, however, some similarities between the two proposals. For example, they both talk about the need to resolve inconsistencies in the *initial design requirement description* and support its evolution into the *final design requirement description*. This is rather different to the usual use of a computer, which is to take *complete and consistent* input and transform it into a desired output form.

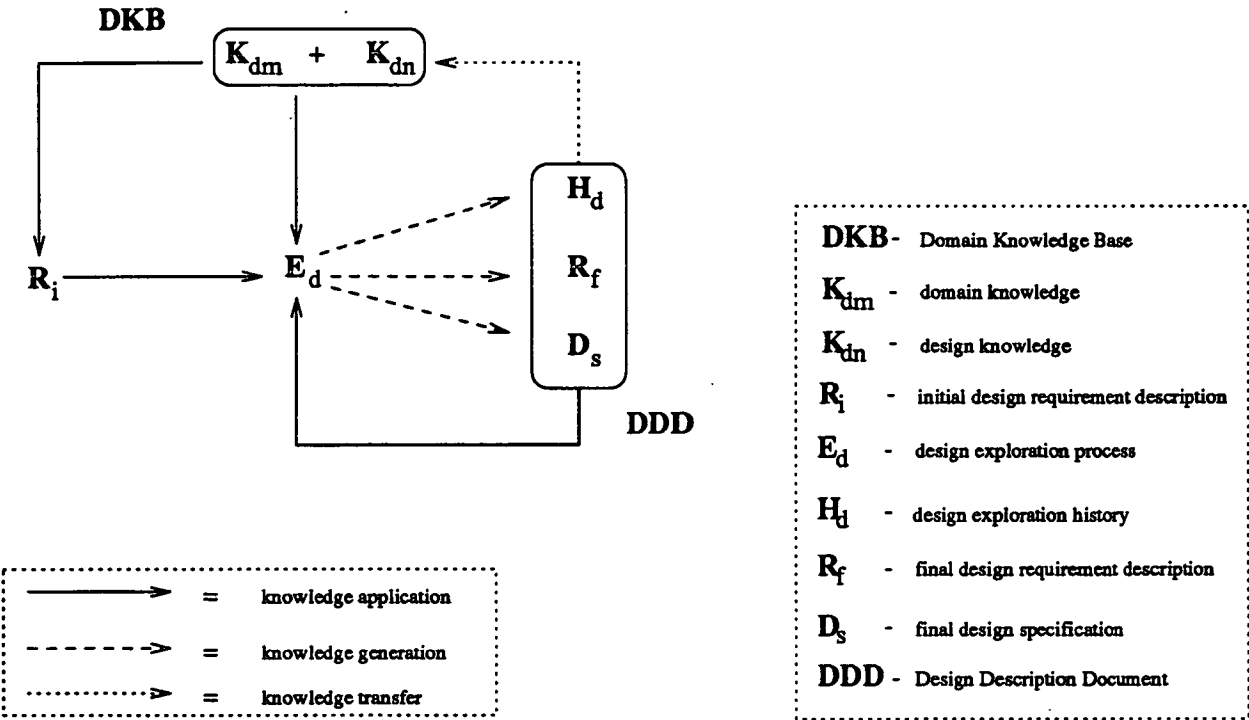


Figure 1.5: An Exploration-based Model of Design

Smithers places more emphasis on truth maintenance than do Subrahmanian et al. In addition, he considers the need to use a “blackboard” architecture to provide an adequate design environment. However, when it comes down to it both proposals are only in a preliminary form. In particular, they demonstrate a huge gap between what they are talking about doing and what actually they know how to do. One can but wait to see whether AI and databases can deliver an integrated product in this crucial

area.

1.4.3 AI in Safety and Loss Prevention

The sections above have reviewed attempts to apply Artificial Intelligence to the areas of modelling and design environments. This section will now bring the review closer to the specific area of Loss Prevention. Really this section is just an overview as in most cases it will just give a pointer to another chapter in this thesis. However, in a couple of cases it reviews work not covered elsewhere. To give some structure to this section it will follow the stages of ICI's hazard review procedure.

Andow [4, 2, 35] has given several previous reviews of AI in Loss Prevention but these are now somewhat out of date. In particular, they tend to concentrate rather heavily on rule-based approaches rather than considering AI planning, qualitative simulation and frame-based plant representation.

Fault-tree Synthesis

Hazard Study 1 is essentially an information gathering exercise to obtain as much data as possible about the hazards likely to be associated with the plant. This is important as one can sometimes immediately rule-out alternatives on the basis of safety. In addition, it may be necessary to begin experimental programs to obtain vital data which is missing. There is not much AI research specifically relevant to this stage although recent work on physical property prediction [79] could be useful. More importantly, however, Hazard Study 1 will probably benefit from improved techniques of information management and retrieval. Such problems have already been discussed above.

Hazard Study 2 is performed when the process flow diagrams are available and reasonably fixed. Usually this means that a small number of process routes have been chosen for further consideration and have been compared by numerical simulation and optimisation. At this stage one starts looking at the hazards that need to be avoided, such as vessel overpressurisation, and at how they can arise in a particular process

configuration. There are two ways of doing this, which can be called the 'bottom-up' approach and the 'top-down' approach. The first requires simulation in some quantitative or heuristic manner to assess the outcome of plausible scenarios such as pump failures. The second approach takes a top event of interest, such as 'overpressurisation of tank-1' and reasons backwards to the trigger-events that may cause it. The 'bottom-up' approach is inherently more combinatorial than the 'top-down' one because it may require the simulation of events that do not lead to a hazard. However, simulation is more capable of correct reasoning because time and sequence can be properly captured and alternative worlds modelled in a consistent way. Such comparisons are more fully explored in chapter 3. This chapter also reviews some of the older work on automatic fault-tree synthesis (a top-down approach). This did not use AI techniques, although the concepts are definitely applicable. Indeed, work is currently underway to apply LISP & OPS5 to this problem [6, 37] but the results are not yet available.

Part I of this thesis is primarily concerned with methods for bottom-up reasoning. Two different approaches have been attempted during the course of this work, as described in chapters 2 & 3.

Hazard and Operability Studies

In between Hazard Study 2 and Hazard Study 3 the detailed design proceeds and culminates in the production of the Engineering Line Diagram (ELD)⁷. During this phase detailed safety questions will be considered, such as "Do we need a pressure relief valve on this tank?" or "Should we fit fire insulation around this structure?". Bunn [12] has examined a large range of such small, detailed decisions in order to consider their suitability for expert system techniques. For each type of task he discusses the sort of expertise involved and briefly points to the technology that may be useful. In addition, he gives the example of deciding whether or not to fit emergency isolation valves as a case where the expertise exists in the form of case studies. He then describes an attempt to apply rule-induction techniques to this problem. Really he is only considering rule-based technology, though he makes passing reference to frames.

⁷This can also be known as the Piping and Instrumentation Diagram (P & ID).

ICI's Hazard Study 3 is conducted as a Hazard and Operability (HAZOP) Study. The applications of AI techniques to this task will be reviewed in Chapter 2. This chapter then describes an early attempt to replicate the 'simulation' side of the HAZOP study technique. The approach was not really successful but led on to more fundamental work on the nature of qualitative reasoning.

Representing Designer's Intention

The use of databases in process engineering has already been discussed at length. Whilst doing this there was also some discussion of the use of Knowledge Craft to represent and check constraints upon design. This can be thought of as representing the *designer's intention* for sub-parts of the design, which is one of the most important concepts in the HAZOP method. If one doesn't know why something was done then it is obviously very difficult to reason about whether or not it is correct. In addition, if one doesn't keep the designer's original specification and assumptions in mind then it is very easy to make later changes to the design that invalidate his work. These are often the kinds of problems that the HAZOP study is intended to identify by exploring in detail the interaction between different parts of the design. However, this is really rather a late stage to perform such checks. Part II of this thesis addresses this problem by describing a new Knowledge Craft extension called the *Constraint Tools System*.

Qualitative Simulation

Qualitative simulation is an AI technique which has recently attracted a certain amount of interest in the Chemical Engineering literature. Chapter 3 describes some new work in this area, which was a natural extension of an interest in bottom-up qualitative reasoning, such as is done during HAZOP studies. In addition, this chapter reviews the qualitative simulation work of other chemical engineering researchers. From this, one can see some hope of capturing adequate knowledge but also some severe efficiency problems. The conclusion is that more work is required, particularly that which also considers 'semi-quantitative' techniques. However, it is important that further development of qualitative reasoning is conducted in parallel with that of the problem solving

systems that depend upon it. The construction of stand alone qualitative reasoning programs will probably not lead in the best direction.

Automatic Operating Procedure Synthesis

ICI's Hazard Study 4 checks that all the recommendations from Hazard Study 3 have been carried out and that the plant operating procedures have been written. However, Hazard Study 3 requires a complete ELD which even shows lines and equipment only needed for start-up and shut-down. Consequently, operation must be partly considered in Hazard Study 3 and there is some confusion as to when operating procedures should be written. The provision of computer-aids for the synthesis of operating procedures would perhaps facilitate their earlier production, thus allowing them to be more formally considered during the HAZOP study. In addition, one should remember that HAZOP studies are said to identify many more operating problems than hazards. Some of these may lead to redesign so there could be a significant payback from a tool which allows their earlier identification. Finally, there is the possibility of synthesising operating procedures on-line to plan recovery from damaging situations. This is rather an ambitious aim however, so most researchers have just concentrated on synthesis at the design stage. This allows them to avoid real-time issues.

Some of the more interesting attempts to use AI in chemical engineering have focussed on the problem of automatically synthesizing operating procedures. Typically this has involved the use of techniques from "AI planning", which is not directly relevant to the rest of this thesis discussion. Nevertheless, such work is reviewed in Chapter 3 because its success is closely related to the power of the qualitative modelling used. For the same reason Chapter 3 also describes some of the work on Fault Diagnosis, though this time the coverage is minimal. The reader is referred to [42] for a more extensive review.

1.5 Summary

This chapter began by defining the main area of interest as being design-stage Loss Prevention. However, it was quickly shown that this area cannot really be tackled without thinking about the wider context of the design problem. In particular, it was shown that one must provide a design structure to control the application of the “thousands of little checks” that are needed to ensure the validity of a design. There was then a discussion of the differences between *discrete* and *incremental* approaches to Loss Prevention. The conclusion was that the ideal approach would be the near-continuous assessment of design quality, in order to provide for inherent safety, together with discrete formal design reviews, which may eventually be reduced to just performing an audit function. To facilitate this one must be able to reason about the design at many different levels of detail, perhaps even in the complete absence of numerical information. This explains why Part I of this thesis is devoted to the problem of using causality and qualitative reasoning to ‘understand’ process flowsheets.

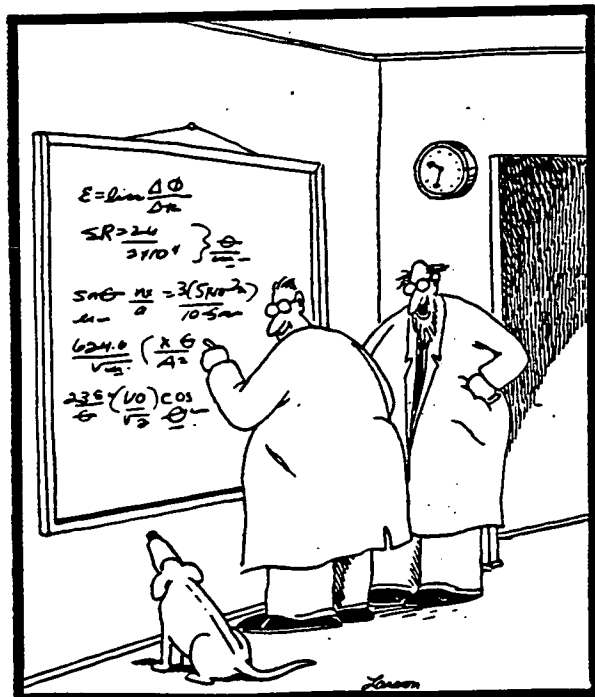
The discussion then progressed to the notion of Integrated Process Design, which could provide the structure that was deemed to be necessary. Past attempts at solving this problem have concentrated on database technology. This has many shortcomings because of a basic mismatch between the data types and transactions needed for engineering and those used by the business community. In contrast, more recent work, particularly that of Struthers [99], has taken a knowledge based, rather than a databased approach. This time the data types may be adequate but it is the traditional database issues which are neglected, such as efficiency and concurrency control. There is therefore much benefit to be had from a synthesis of database and knowledge based technology. The work in this thesis does not address this issue but there is much database research which does. In addition, this thesis partly follows on from the work of Struthers by considering the problem of representing designer’s intention. This is part of what he calls the problem of “maintaining the integrity of knowledge”.

This chapter concluded with a review of previous attempts to apply AI techniques to the problems of Loss Prevention. This also included a discussion of some attempts to provide AI-based design and modelling environments. Part II of this thesis is relevant

to these areas of integrated design and AI-based modelling. Before this, however, there will first be a discussion of the problem of bottom-up qualitative reasoning.

Part I

Bottom-up Techniques



"Ohhhhhh ... Look at that, Schuster
... Dogs are so cute when
they try to comprehend quantum mechanics."

But we are not likely to find science returning to the crude form of causality believed in by Fijians and philosophers, of which type is "lightning causes thunder".

Betrand Russell

Chapter 2

A Simple HAZOP Program Using Rules to Represent Causality

This chapter describes the author's initial work on the application of AI techniques to assessing the safety of plant designs. This work actually took place during his honours year as an under-graduate, under the auspices of his fourth year research project. As such it pre-dates work on the Ph.D. proper. Nevertheless, it is included in this thesis because it gives a good introduction to the problems encountered when trying to computerise qualitative reasoning, which is such a vital part of tasks such as HAZOP.

The project concentrated on the bottom-up, or simulation, aspect of HAZOP reasoning, as opposed to the top-down reasoning that is used to construct fault-trees. As will be shown below, this is only a part of the HAZOP problem and other aspects will be discussed at other points in this thesis. Nevertheless, it as an important area and led quite naturally to other interesting work, such as the work on qualitative simulation described in Chapter 3.

2.1 Introduction to HAZOP Studies

Comprehensive descriptions of HAZOP studies are given by the Chemical Industries Safety and Health Council (CISHC) [16] and by Kletz [49]. The description of HAZOP studies given below is taken from these references. Example studies relating to continuous processes are given by Lawley [61, 62], CISHC [16], Ozog [76], Wells [114], Piccinini [77], Kletz [51] and by Austin and Jeffreys [5]. An example relating to a batch process is given by CISHC [16]. The work described in the early chapters of this thesis focuses on continuous processes but the later chapters describe tools that are essentially “domain independent” and so can be applied to processes of either type.

The CISHC recommend the following procedure for a study:

1. Define objectives and scope.
2. Select the team.
3. Prepare for the study.
4. Carry out the examination.
5. Follow-up.
6. Record the results.

The form of the study will depend upon the objectives. A HAZOP is a lengthy task so it is imperative that the scope is clearly defined and that the team does not get carried away. If the study is for a modification then a quick answer may be required.

The HAZOP study itself is a systematic line by line examination of a flowsheet or Engineering Line Diagram (ELD). To begin the study a line on the ELD is identified and marked. Its purpose is then clearly defined and guide words applied to this purpose to establish possible deviations from designer's intent. For example, the guide word “high” is applied to the purpose of conveying fluid at a certain pressure, in order to propose the deviation “high-pressure” in the line. For each such deviation that is established the team considers the following questions:

- 1. Is there a possible cause for the deviation.
- 2. Does it result in a hazard or an operability problem
- 3. How easy is it to prevent this problem and how likely is it to occur?

When all the conceivable deviations in a line have been discussed then the actual vessels on the line are considered with regard to the guide word “other than”. When this is completed then the line is said to have been “hazopped” and a new line is picked to repeat the process. During the course of the study a table is drawn up listing the problems identified, the causes concerned and the action recommended. The action can often be requests for design modifications, or for a quantified hazard analysis, but the team should refrain from doing detailed design during the study.

New Plant	Existing Plant
Design engineer	Plant manager
Process engineer	Process foreman
Commissioning engineer	Plant engineer
Instrument design engineer	Instrument manager
Research chemist (optional)	Process investigation manager
Independent chairman	Independent chairman

Table 2.1: Recommended team members for a HAZOP study

Table 2.1 shows the HAZOP team members recommended by Kletz [49]. The team members must be top level personnel, rather than deputies, as they must have the authority to agree changes on the spot. Kletz reports that a HAZOP takes 1 & 1/2 hours per main plant item and recommends 3 hour sessions two or three days per week. HAZOP of a large plant may therefore take several months. This, together with the status of the people involved, usually limits the use of HAZOP to a single application when the ELD is finished and the chemical engineering essentially completed. Chapter 1 has already discussed the implications of this one-off approach.

Kletz describes the success of HAZOP as lying in its ability to bring experts together

and force them to communicate. A balance between safety, cost and speed of commissioning is usually achieved because of the differing priorities of the team members. The CISHC also point to the communication aspects of HAZOP in that they see the success or failure of a study as depending upon the following:

1. The accuracy of drawings and other data used as a basis for the study.
2. The technical insights of the team.
3. The ability of the team to use the approach as an aid to their imagination in visualising deviations, causes and consequences.
4. The ability of the team to maintain a sense of proportion when assessing the seriousness of the hazards which are identified.

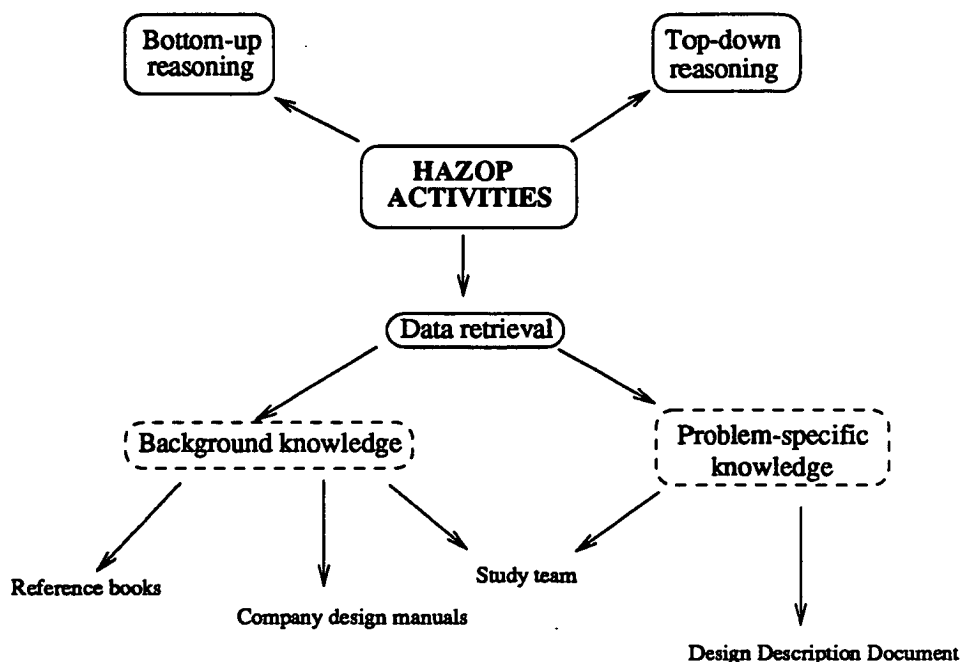


Figure 2.1: Knowledge-based activities within HAZOP studies

Thus there are several dimensions to the HAZOP task, as is shown by Figure 2.1. Some features of actual studies are discussed by Roach & Lees [85] who examined a number

of study records to see what really went on. In particular, they considered the amount of fault propagation that was done during a study and found it to be fragmentary and unsystematic. They consider that it is not the place of a HAZOP to conduct a detailed analysis of causes and effects but nevertheless some such reasoning is necessary. The method can consequently be regarded as an incomplete method of exploring fault propagation in process plants.

2.2 Other Work

The result of the work described in this chapter was a HAZOP “simulation” program, written in Prolog. The most important feature of this program was that it tried to reason about a flowsheet in a causal manner, in order to predict how a deviation could arise and be propagated to a damaging effect. In this respect the program was essentially rule-based¹. However, the rules were divided into cause rules, propagation rules and consequence rules and were applied by an explicit algorithm, rather than by control strategies embedded in the rules themselves. Thus the program was structured differently than if it had been written in a forward-chaining rule-based language such as OPS5. Nevertheless, it could have been written in OPS5 and the central point is the commitment to causality. This is an intuitive approach but, as will be shown below, it relies upon some assumptions which are not really tenable. Indeed, the approach is only fully successful for problems without feedback of any kind.

Rule-based approaches to HAZOP study aids have also been attempted by Weatherill and Cameron [113] and by Suokas et al [101]. The first authors report the use of a pc prolog to construct an interactive aid for HAZOP studies. They claim that their system is modular and easily extended for serious industrial use. However, their tool seems little more than a check-list that asks the user the questions he should be asking himself. There does not appear to be any significant reasoning done by the system and the representational methods are simplistic, which makes their claim barely credible.

¹The causal reasoning approach actually pre-dates rule-based shells and languages but only in its application to tasks other than HAZOP, such as automatic fault-tree synthesis. This will be further discussed in Chapter 3.

Similar criticisms can be applied to the work of Suokas et al, although this is on a larger scale and uses the AI toolkit KEE. The failure here is that the system does not seem to do anything approaching simulation, which is a large part of the HAZOP process. This can be seen from the example in the paper, which presents a dialogue between the computer and a user in order to HAZOP a single vessel. The problem with this dialogue, and the results produced by it, are that the user is left to do significant chunks of the causal reasoning himself. For example, in response to the question of what can cause "high concentration" in the vessel the computer simply suggests "high input concentration". Thus it is left to the user to find out if there is a possible cause for this "high input concentration" in the rest of the process. In addition, causes do not seem to be properly associated with consequences and corrective action, so one does not know which cause will result in which consequences and how to protect against it. Thus to use this program the user will have to significantly interpret its output. This seems to suggest that its value would be as a large repository of hazard information rather than as an interactive aid for HAZOP studies.

Another author (Andow [2]) has quoted Ferguson [28] as having briefly investigated the knowledge needed for HAZOP studies. It was found that several problems exist in defining HAZOP study knowledge as a knowledge base for expert system implementation:

1. The ELD contains a lot of pictorial information that it is not easy to represent textually.
2. One of the guide-words used is essentially a "catch-all" for anything not yet mentioned.
3. A lot of the knowledge is common sense, or at least common sense to engineers. It is not explicitly stated in design manuals.
4. Humans know what they don't know. Being able to ask for further information is an essential part of the study.

Andow concludes that the knowledge needed for a HAZOP study is "broad and deep", which is the antithesis of the classic application area for expert systems. As will be

shown later this is roughly in agreement with the conclusions of this chapter. However, chapter 1 has already shown that the support of hazard studies could be done in ways other than totally automating the study concerned; for example, by just providing easy access to databases and design information.

2.3 Program Structure

The structure of the HAZOP program was as shown by the flowcharts in figures 2.2 and 2.3. The first flowchart determines the order in which lines and deviations are explored while the second describes the fault-propagation process. In a real study the order of line selection is important to maintain the continuity of the study and to focus the attention of the team upon a particular area of the flowsheet. The team will find it easier to keep in mind the designer's intention for a line if they have just dealt with the line upstream of it and are next going to deal with that downstream. In addition, continuity will help the team to build up section-wide mental models of plant behaviour. This can be very important as the fault-propagation in a real study is never exhaustive or complete.

In contrast, in the program being described here the order of line selection is not of crucial importance because all faults are propagated as fully as possible anyway. Consequently, the results of hazopping one line are not strictly needed when hazopping any other. However, the fault propagation process relies upon the assumption that the result of a deviation in a line is independent of how that deviation arose. If this is valid then duplicated deviations are equivalent and need only be explored once. For example, the propagation of "high-temperature" from the start of a flowsheet may cause this deviation to be explored throughout the whole flowsheet. One can therefore avoid replicating this reasoning when the downstream sections are hazopped directly. The order of line selection can thus affect efficiency by determining how much of the propagation take place at once.

In the HAZOP program duplication is avoided by stage 5 on figure 2.2. In addition, the lines are hazopped in a depth first manner, starting from the process input streams.

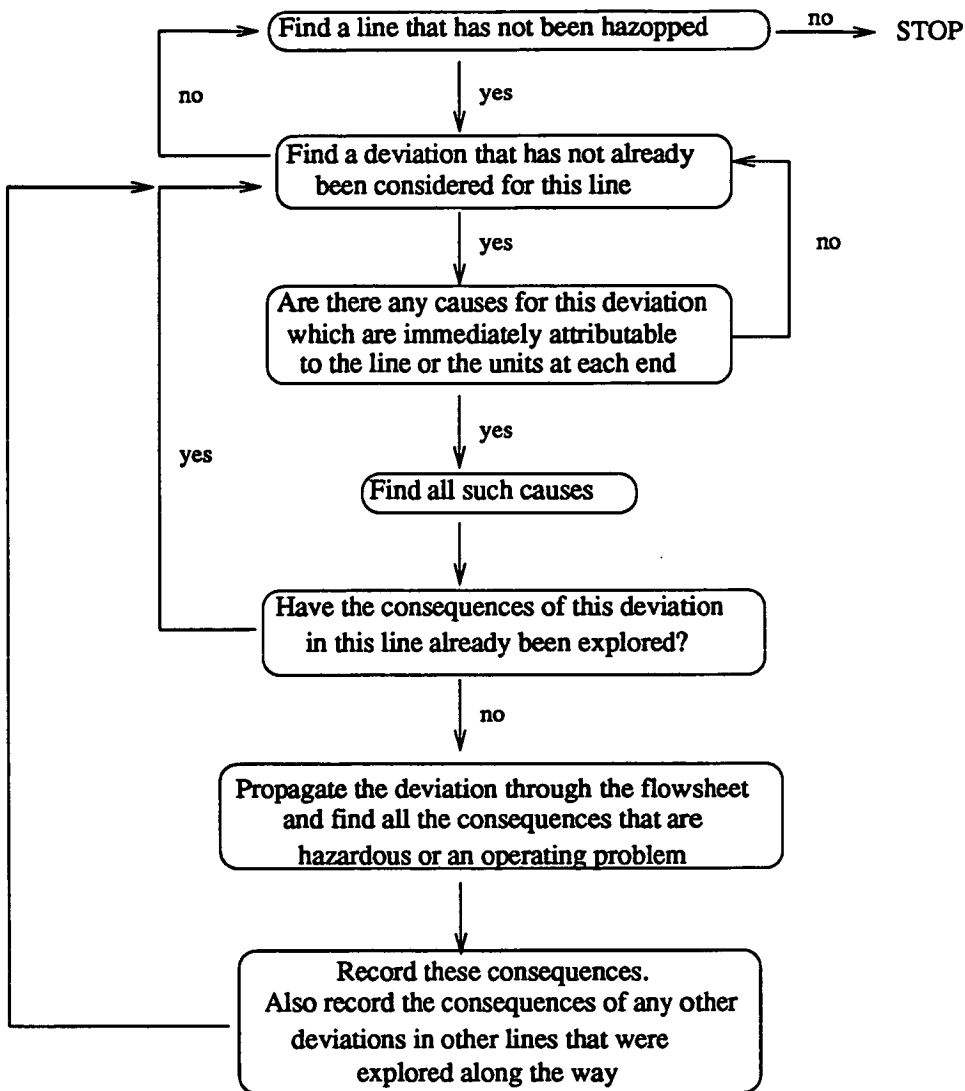


Figure 2.2: A Flowchart for a Simple HAZOP Program

This presents the maximum chance of propagating deviations throughout the flowsheet in one pass. Unfortunately, the advantage of this is much overshadowed by the inadequacies of the fault propagation assumption.

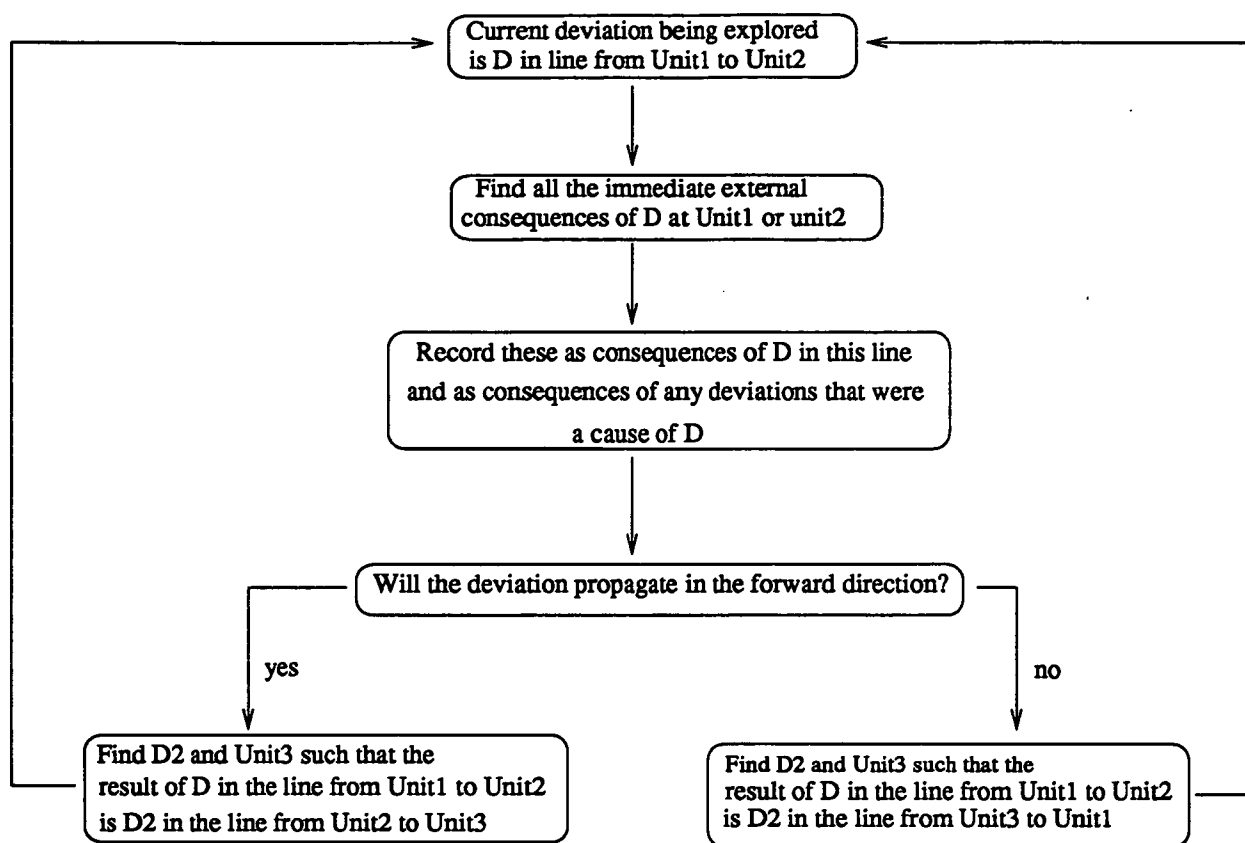


Figure 2.3: A Flowchart for fault propagation in a Simple HAZOP Program

2.4 The Deviations Explored in the Program

The deviations used in the program are shown in figure 2.4. This is obviously a very restricted list, which involves only the deviations that are easiest to deal with. In particular, any deviations dealing with concentration are missing from the list, despite being vital for a real chemical process. The problem here is that concentration only seems to fit neatly into a rule-based representation if one uses a separate deviation for perturbations in each component (i.e. high-conc-component-1, low-conc-component-2 etc.). This is unsatisfactory because it multiplies the number of rules and requires an input routine to fix the number of components and set things up. Furthermore, one has

the additional confusion of dealing with the interdependence of the mole-fractions, i.e. that they sum to 1. This is a tricky problem which continues to plague any knowledge-based chemical engineering research which uses the notion of causality. The problem was bypassed during this early project but will be further discussed in later chapters.

high-forward-pressure
low-forward-pressure
high-back-pressure
low-back-pressure
no-forward-pressure
full-back-pressure
reverse-flow
high-temp
low-temp

Figure 2.4: Deviations Explored in the Simple HAZOP Program

In the list of deviations there appear to be six deviations dealing with pressure and none dealing with flow. This is the result of the following considerations:

- Pressure acts in both the upstream and downstream directions, usually at the same time. Consequently, the pressure deviations are divided into ones which propagate in the normal direction of flow (e.g. low-forward-pressure) and ones which propagate against the flow direction (e.g. low-back-pressure).
- Flow is the result of pressure and deviations such as “high-flow” and “high-forward-pressure” are virtually synonymous. In addition, the consequences of “no-flow” are mostly ones of high or low pressure and so are best treated in these terms.

The need to model pressure as acting in both directions was first described by Martin-Solis et. al [71] in early work on automatic fault-tree synthesis. Their solution was similar to that described in this chapter except that they propagated pressure in the opposite direction to flow and other variables, instead of using two types of pressure deviation. The fault-tree synthesis work is also relevant when one considers how many

discrete levels to split variables into, in order to form deviations. In the list in figure 2.4 there were six levels for pressure but only two for temperature. Other researchers have used different sets; for example Taylor [102] used the following levels:

- Very high/low: Cannot be cured by a safety system.
- High/Low: Can be cured by a safety system such as a trip loop
- Slightly high/low: Can be cured by a controller if working
- Normal:

How many levels one uses determines the resolution that can be expected from the reasoning. For example, in order to determine the consequences of “no-flow” at a pump one has to know whether the flow is blocked upstream, which may result in cavitation, or downstream, in which case the pump is said to be ‘deadheaded’. This distinction would be difficult to make if the “no-flow” deviation was not split into “no-forward-pressure” and “full-back-pressure”. Similar examples could be found for the other process variables, so one might consider the introduction of more levels than are currently used by the HAZOP program. This approach will always be limited however, as the real problem is the concept of using pre-defined discrete levels for tasks such as fault-tree synthesis. In particular, this is because such levels embed both information about the direction in which a value changes and the approximate magnitude of that value at a certain stage in the reasoning process. This makes it difficult to reason about magnitudes properly, which makes the consideration of worst-case scenarios particularly difficult. Chapter 3 will show how qualitative simulation avoids some of these problems by properly distinguishing between magnitudes and values.

2.5 Deviation Generation and Propagation

There are three stages to the reasoning process, as follows:

1. Generation of local causes for a chosen deviation in a chosen line.

2. Propagation of that deviation to other deviations in other lines.
3. Generation of undesirable consequences from the deviations considered.

The knowledge for each of these stages can be described using rules, although the HAZOP program actually uses Prolog facts. Figures 2.5, 2.6 and 2.7 give example rules for each stage (in a pseudo OPS5 format), while Appendix A gives a fuller list of the rules that resulted from this project.

English form :

```

IF
    There is a line which has a valve on it
    and the valve is not locked open
THEN
    record that accidental closure of the valve is a
        possible cause of full-back-pressure in the line
    and record that accidental closure of the valve is a
        possible cause of no-forward-pressure in the line

```

Pseudo OPS5 syntax :

```

IF
    (line <L> ^has-valve <V>)
    (valve <V> ^locked-open NO)
THEN
    (record-cause ^line <L>
      ^deviation full-back-pressure
      ^cause (accidental-closure <V>))
    (record-cause ^line <L>
      ^deviation no-forward-pressure
      ^cause (accidental-closure <V>))

```

Figure 2.5: An example rule to find the *causes* of a deviation

Figure 2.8 shows a simplistic flowsheet that could be partly hazopped by the program. For example, the program could correctly show that stoppage of pump P-1 was a possible cause of full-back-pressure in line-1 and of no-forward-pressure in line-2. It

Specific Form :

```

IF
    (pump <P> ^deviation high-temp ^port inlet)
THEN
    (new-deviation ^item <P> ^deviation high-temp ^port outlet)

```

More general form :

```

IF
    (pump <P> ^deviation <D> ^port inlet)
    (member <D> <<high-forward-pressure low-forward-pressure
                no-forward-pressure
                high-temperature low-temperature>>))
THEN
    (new-deviation ^item <P> ^deviation <D> ^port outlet)

```

Figure 2.6: An example rule to describe how a deviation *propagates*

```

IF
    (pump <P> ^status running
             ^deviation full-back-pressure ^port outlet)
THEN
    (record-consequence ^item <P>
                        ^deviation full-back-pressure
                        ^port outlet
                        ^consequence (pump deadheaded - probable fire))

```

Figure 2.7: An example rule to find the *consequences* of a deviation

could then trace this latter deviation downstream to show that this led ultimately to no-forward-pressure at end-1 and high-temperature at end-2 (among other consequences). Thus it could trace forwards and backwards along flow lines and also propagate heating

effects across a heat-exchanger. However, the program also concluded that “racing” of pump P-2 would lead to high-forward-pressure (i.e. high-flow) in lines 4 and 5. It then concluded that this would cause a spillage in tank T-2. This is patently ridiculous since all the extra flow going into the tank via line-5 has to come from the tank via line-3. In other words, the program was unable to cope adequately with a feedback loop in the flowsheet².

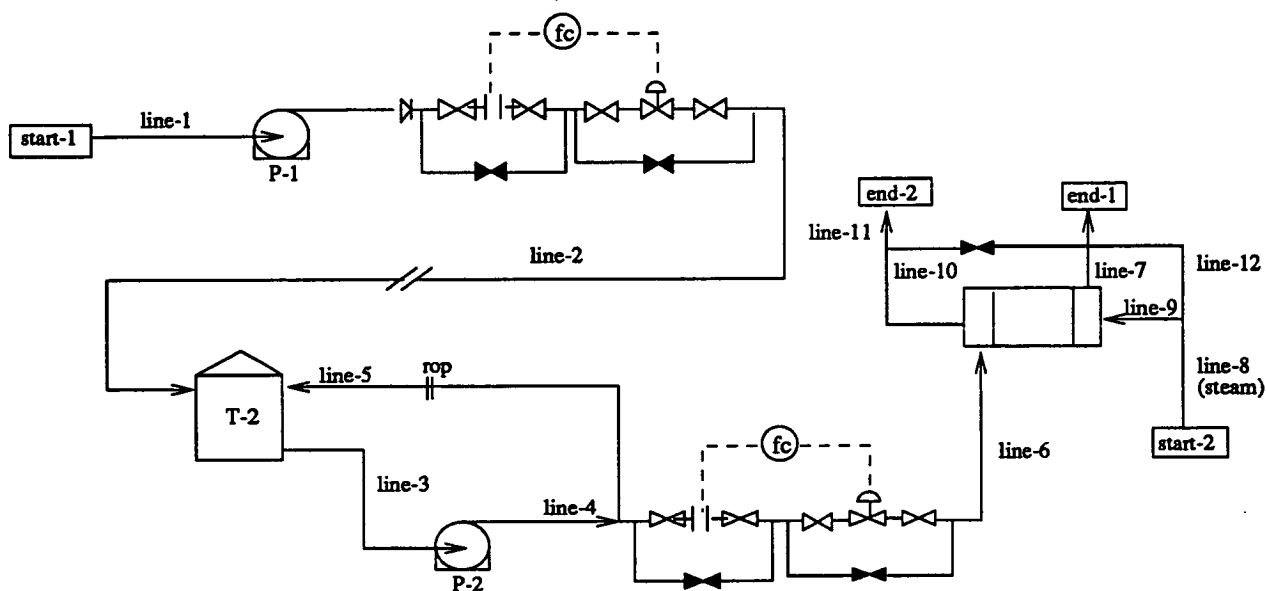


Figure 2.8: An example flowsheet problem for the simple HAZOP program

To understand better these successes and failures one needs to consider the assumptions behind the chosen representation. This can be done by defining three premises upon which the method is based. The first may be phrased as follows:

It is possible to determine the causes of a deviation by a purely local analysis i.e. without reference to the rest of the flowsheet.

²although it did have the checks necessary to prevent infinite propagation of a deviation around the loop.

This works in many cases, such as for the generation of temperature deviations when the tubes of a heat exchanger become fouled. However, Figure 2.9 presents a simple counter-example if one considers what happens when valve V-1 is partially closed. By a local analysis one would conclude that the result is low-forward-pressure (i.e. low-flow) in line-3. Unfortunately this is incorrect because the flow controller will compensate by opening valve V-2. Note also that the same effect would arise if the flow controller was downstream, not upstream, of valve V-1. That is, a compensation can occur in either direction. On the other hand, if V-1 closes completely then the flow controller can no longer compensate. Thus one must identify if a compensatory effect can occur but also try to consider its limitations.

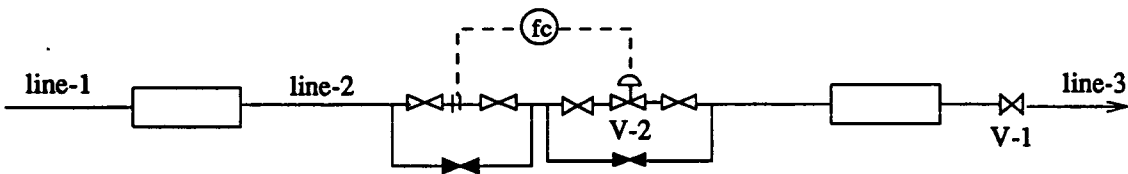


Figure 2.9: A counter-example to the first premise for a rule-based approach

Similar arguments will apply to the second premise, which goes as follows:

It is possible to determine a qualitative description of the outputs from a device, given just its current state and a qualitative description of the inputs.

As before, this premise is partly invalidated by the need to consider the magnitude of a process deviation. To some extent this information can be ‘carried’ within the deviation itself (e.g. by using “very-high-temperature” instead of “high-temperature”). However, this is undesirable because one is then relying upon implicit assumptions of what these levels mean, and how they will be used. This could cause many consistency problems because the levels may have to be ‘retuned’ whenever a new type interaction has to be represented. This chapter has already described these problems, which arise from trying to improve the resolution of the reasoning by just increasing the number of

discrete levels. As already mentioned, qualitative simulation provides an alternative approach.

A further difficulty with the second premise is the question of what to do if an input is not yet known when a unit needs to be simulated. This problem can arise in feedback loops, such as the kickback loop shown in Figure 2.8. As with numerical modelling one needs to “tear” the recycle stream in order to obtain a solution. However, in qualitative terms this means guessing one of the inputs and later testing if the guess was correct. If not, one must be able to “backtrack” and try again. Once more, it will be shown that qualitative simulation provides a much cleaner method than a strictly rule-based approach.

A final difficulty is what to do if the inputs counteract each other. That is, if they affect the outputs in opposite ways. In this case any qualitative scheme is going to have difficulties, as will be discussed in chapter 3.

The final premise for the rule-based approach is as follows:

The consequences of a qualitatively described deviation are independent of how that deviation arose.

Again the question of levels is the sticking point. For example, the above discussion has already shown that to determine the consequences of “no-flow” at a pump one needs to know whether the flow is blocked upstream or downstream. Similar examples could be constructed for the other process variables.

2.6 Improving Causal Reasoning by the Use of Less Obvious Rules

The last section has shown that the limitations of the HAZOP program were mostly attributable to the use of a purely qualitative, local analysis. This section will now consider whether it would be useful to remove these restrictions whilst still retaining

the commitment to using rules to represent causality. In other words, can the situation be improved by the use of less obvious rules?

2.6.1 Rules Which Model Wider Sections of Plant

The rules described in Appendix A are written from a device-centred viewpoint. That is, they model causes, propagation and consequences by only considering the nature of the device local to the current deviation. From a users point of view this is the ideal situation as it means that one can analyse a flowsheet by just feeding in descriptions of the plant items and their connectivity, as opposed to having to make wider statements about the nature of the flowsheet. However, from a reasoning point of view the representation is impoverished. In particular, it can only cope with feedback by the use of local models that make assumptions about the rest of the flowsheet.

This was demonstrated by the binary split and binary junction rules in Appendix A.2. These assumed that the relative effect of a blockage in one of two lines was related to the fraction of flow that each of them normally took. Unfortunately, this makes assumptions about the use and purpose of the lines and is not successful where one of them is part of a kickback loop. A possible solution to this problem to this is to explicitly recognise the kickback loop and so model its required behaviour. There are several ways of doing this:

1. Require that the user declares the group of objects to be a kickback loop and write rules accordingly.
2. Have rules that are capable of automatically deciding that a group of objects should be regarded as a kickback loop, as well as rules to model kickback loop behaviour.
3. Combine both types of rule from (2) so that the first few elements of a rule test for the presence of a kickback loop.

In all of these cases one needs to be able to apply rules in some rather complex or ad-hoc manner, in order to check that the kickback loop can perform its desired function.

In addition, the structure of kickback loops is not so fixed that all behaviour will be the same. For example, some will kickback to just before a pump and some to a tank; some will use restriction orifices and some will use relief valves. This makes it important that the kickback loop rules only model and check the kickback behaviour i.e. that a downstream flow blockage is safely relieved. For other behaviour, not related to the main purpose of the loop, it must be possible to model the system using different rules, such as by looking at its constituent parts.

To reason about a flowsheet it may be necessary to regard parts of it as belonging to several different systems at once. For example, a pump may need to be represented as a unit in its own right, as well as being represented as the prime mover in a recycle loop. Recognition of such requirements is the basis of the constraint representation work described later in this thesis. Indeed, that work provides a way of writing checking 'rules' that apply to wider sections of plant and as such is an extension of this early work, though not so qualitative in nature.

2.6.2 Rules Which are Derived by Algebraic Manipulation

The rules given in Appendix A are heuristic, in that they were justified by intuition rather than a formal proof. As an alternative, one can derive rules by algebraic linearisation and the taking of total derivatives. Such an approach to the construction of behaviour models has been described by Taylor [102] who was constructing models for a fault-tree synthesis algorithm. Unfortunately, the procedure is generally difficult and will require the use of simplifications to obtain analytical expressions.

Algebraic manipulation is generally tedious but the burden may be alleviated somewhat by the use of computer packages such as MACSYMA [82]. Typically, such packages can perform the manipulation steps with ease but are incapable of making strategic decisions about what to do next. For example, it is a simple task to completely differentiate a set of algebraic equations in MACSYMA. This results in a set of equations which are linear in the "difference variables" but probably not in the variables which make up their coefficients. Consequently, one can definitely eliminate some of these difference variables, in order to express finite output differences in terms of finite input

differences. This procedure was briefly explored in this project as a possible way of constructing qualitative models that span whole plant sections.

The questions that must be answered are as follows:

1. Can the algebra be done with reasonable speed and memory usage?
2. Can the resulting answers be interpreted qualitatively?
3. Can one model units that are only properly modelled by a mixture of algebraic and differential equations? (e.g. plug flow reactors).

Unfortunately, the answers to all three of these questions appears to be no. For example, setting up the procedure using MACSYMA was relatively straightforward but the expressions generated were often several thousand lines long! In fact, machine memory was quickly exhausted so it was not possible to solve the problem that was initially chosen. A restricted problem was then considered and a solution obtained. However, this solution still had to be interpreted in a qualitative manner.

The basic idea was to construct parametric cases such that all the terms in the final expressions affected the outputs in the same direction. For example, if the terms are all added together then to remove ambiguity they must all have the same sign. This process is manually applied to individual units when constructing models for qualitative simulation. Unfortunately, it seems much too difficult to apply automatically on a system-wide scale because of the length and complexity of the expressions involved. Consequently, the answer to question 2 must be no. The same applies to question 1, while question 3 remains undecided. This was an interesting idea but one must conclude that complete algebraic linearisation of whole plant sections is not a useful or practical proposition.

2.6.3 Rules Which deal with semi-numerical Information

This chapter has discussed the use of rules to propagate qualitative deviations throughout a flowsheet. In Chapter 6 there is a similar discussion on the use of rules to prop-

agate other values, such as numerical information. For example, one might have a rule which fires whenever the temperature into a pump changes and changes the temperature out of the pump to the same value. This is a slightly easier application because there is the possibility of performing numerical tests on the values involved. Nevertheless, it is still tied to the concept of causality and so will be subject to most of the problems and considerations already raised in this chapter.

2.7 Conclusions

This chapter has described a rule-based Prolog program which tried to copy the simulation side of HAZOP reasoning exactly. The central concepts were the use of a local, qualitative analysis and the use of rules to represent causality. Unfortunately, this combination was capable of only limited success. In particular, the approach ran into difficulties on problems with feedback of any kind, which occurs in all but the simplest of cases.

To understand the failings of the approach the underlying assumptions were formalised as three basic premises. Each of these were then invalidated by the presentation of counter examples. For example, problems were shown to arise from the use of a device-centred viewpoint, rather than examining the flowsheet as a whole. A possible solution to this is the use of 'less obvious' rules, which model wider sections of plant. Reasoning of this type will be further discussed in Part II of this thesis. In addition, it was shown that further problems arise from the use of fixed, discrete levels to represent qualitative magnitude. In particular, it proves difficult to reason about maximum and minimum variable values in a general and consistent manner. In contrast, such concepts are easily represented using qualitative simulation because it allows one to distinguish between the direction in which a variable changes and its qualitative magnitude at a certain point in the reasoning process. These issues will be the subject of the next chapter.

In the physical world one cannot increase the size or quantity of anything without changing its quality. Similar figures exist only in geometry.

Valery

Chapter 3

Qualitative Simulation and Fault Propagation in Process Plants

3.1 Overview

This chapter is an extended version of a paper which appeared in Chemical Engineering Research and Design [111]. The original paper was first written in early 1988 and covered most of the author's qualitative simulation work before that date. This chapter and its associated appendices expand on this previous discussion. Appendix B.2 gives more detail about the Prolog program that was written, Appendix B.3 presents some of the unit models that were used and Appendix B.4 describes the input and output for some example runs¹. In addition, this chapter describes some more recent work which ties up a few loose ends. It should be noted that the author's work in this area essentially ended with the writing of the paper so there are no major advances to report. However, Section 3.5.3 does describe a new algorithm for the first stage of the qualitative method. This has been implemented in LISP and tried on some representative problems. The results show that it is up to 140 times faster than the original Prolog code. Also, in Section 3.6 there is a review of related work by other researchers in the field of Chemical Engineering. Of this, the work of Mark Kramer's group at MIT is particularly relevant and could solve some of the problems identified

¹The notation for the original paper has also been put in Appendix B

in the original paper.

3.2 Introduction

Assessing the hazards or operability problems that may be associated with a detailed chemical plant design is a complex and time consuming task. For this reason a number of formal analysis techniques have been created, for example Hazard and Operability (HAZOP) Studies [16] and Fault Tree Analysis [63]. In addition, there has been substantial interest in providing computer support for these techniques. At first this was limited to the automatic analysis of fault trees that had been synthesised by hand. Subsequently, attempts were made to synthesise fault trees automatically, given the system topology and the chosen “top” event. It now seems appropriate to consider the provision of computer support for HAZOP studies, although this may be a more difficult task.

Fault tree synthesis is a top-down technique because one works from a chosen top event back to the scenarios that could cause it. This is a recursive process which involves finding the immediate causes of the top event and treating these in the same manner. In this way branches are explored until they terminate when one can no longer define the immediate causes of an event; either because it has none or because it is considered to be a “primal” event. The overall result is an AND/OR tree which shows the causal links between the top event and the sets of primal events that ultimately cause it. Simple examples of fault-trees are discussed in section 3.6.3.

In contrast, a HAZOP study can be said to pursue a bottom-up methodology because it involves asking a series of “What happens if?” questions about the design. In particular, the team members work through a list of deviations, which define initial scenarios, and try to predict the possible outcomes of interest. Thus the process is one of ‘simulation’ from causes to consequences, rather than the derivation of causes for a specified top-event. This suggests that recent work on “qualitative simulation”, in the field of Artificial Intelligence, might prove to be highly relevant. In particular, because it provides a technique for reasoning about system behaviour in the partial or total

absence of numerical information.

Thus one can make a useful distinction between top-down and bottom-up approaches, although on first appearances they have much in common. For example, in both cases there is the need to be able to reason from first principles, the need to sometimes use numbers or estimates of scale, the need to use heuristics and the need to deal with sequential or time-dependent behaviour in a reliable manner. There are, however, some striking differences due to the relative speeds and efficiencies of each technique. A top-down method will usually only follow the pathways that lead to the top event of interest whereas a qualitative simulation will explore all behaviours that are consistent with the initial state of the system and the input deviations. The result is that qualitative simulation is potentially more combinatorial than fault tree synthesis because many of these explored behaviours are "uninteresting" from a safety point of view. On the other hand, a bottom-up simulation uses a clearer notion of system state than the top-down approach which means that there is much more information available to resolve ambiguities. Consequently, the inferencing power of a bottom-up approach is much greater than that of a top-down one since one can introduce extra information to "prove" the connection between the initiating events and the hazard.

Such issues will become clearer in later sections which make further comparisons between the two approaches. Firstly, however, there is a description of the authors' experience with a qualitative simulation method similar to that used by De Kleer [20]. This resulted in a program, written in Prolog, which has many interesting features although it is far too slow to be used directly. The reasons for this are discussed at length, together with possible improvements. This leads on to a review of related work performed by other chemical engineering researchers. Most of this is not primarily concerned with qualitative *simulation* but rather with other uses of qualitative representations. Nevertheless, such work certainly has things to add to the qualitative simulation debate. Even more importantly, however, such work demonstrates the central importance of qualitative reasoning to a whole range of loss prevention problems.

The chapter concludes with a description of some of the approaches to automatic fault-tree synthesis that have been taken by other researchers. This shows that most of the current algorithms for this task either suffer from weak inferencing power or from a

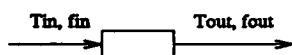
complexity that approaches that found in qualitative simulation. Section 3.6.3 discusses possible reasons for this and tries to show how some of the qualitative simulation work might be applicable. Finally, Section 3.7 summarises this chapter and makes some concluding remarks.

3.3 Example Problems

This section will discuss the application of qualitative simulation to a range of test problems. All the problems assume step changes in process inputs, and require a “plot” of the long-term system behaviour against that deviation. Such step changes are called “difference variables” within this chapter and are represented by terms such as “del(Tin)=+”, which means that there is a step increase in process variable ‘Tin’.

The problems in figures 3.1(a) to 3.1(f) are assumed to transfer (or transit) smoothly to their new states so one is not particularly interested in transient behaviour. Let us say that these are problems that can be solved in a “quasi-steady-state” manner. In contrast, problems 3.2(a) to 3.2(c) require consideration of how a transition actually takes place because the long-term behaviour depends upon the actual *history* of process change. These problems must therefore be treated in a more correctly transient or dynamic manner. Finally, the problem in figure 3.3 will remain qualitatively ambiguous, even when one has considered all the devices on the flowsheet. In this case it is not possible to obtain a unique solution without recourse to numerical arguments or arguments involving estimates of scale.

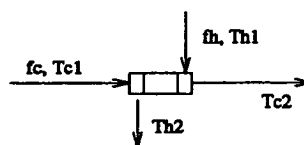
Problem 3.1(a) concerns a stream passing through a “resistance element” which could be a vessel, a valve or anything else that has a resistance to flow. Such elements will be used throughout this chapter and should be taken as implying that one may need to model the pressure/flow relationship. This is not the case, however, in problem 3.1(a), which is very straightforward and could be solved by expressing the relationship between ‘Tout’ and ‘Tin’ as an unconditional rule. This is because there is no interaction between temperature and flow in a simple resistance element i.e. there is only one input effecting ‘Tout’. Such simplicity also extends to a system which has several of



Boundary conditions: $\text{del}(T_{in}) = +$

Find: $\text{del}(T_{out}), \text{del}(f_{out})$

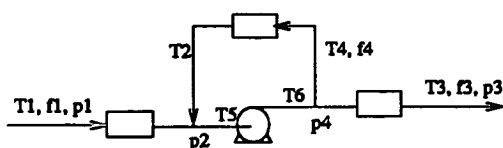
(a) Temperature propagation in a resistance element



Boundary conditions: $\text{del}(f_c) = +, \text{del}(T_{c1}) = ?, \text{del}(f_h) = ?, \text{del}(T_{h1}) = ?$

Find: $\text{del}(T_{c2}), \text{del}(T_{h2})$

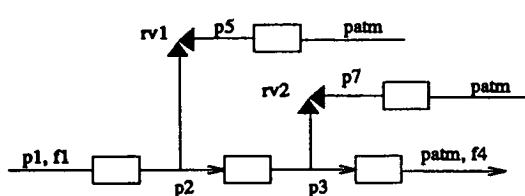
(b) A problem with multiple inputs



Boundary-conditions: $\text{del}(T_1) = +, \text{del}(p_1) = 0, \text{del}(p_3) = 0$

Find: $\text{del}(T_2), \text{del}(T_3)$

(c) A problem involving self-limiting feedback



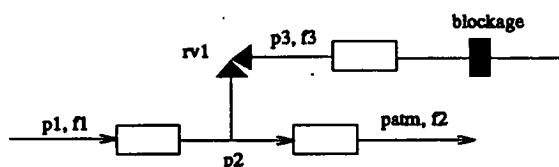
Boundary-conditions: $\text{del}(p_1) = +, \text{del}(p_{atm}) = 0$

Initial information: $rv1 = \text{closed}, rv2 = \text{closed}, f1 > 0$

Parameter information: $\text{set-point}(rv1) < \text{set-point}(rv2)$

Find: Behaviour of $rv1$ and $rv2$

(d) A problem that can be solved by reasoning with inequalities

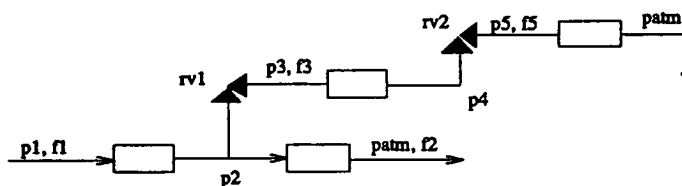


Boundary conditions: $\text{del}(p_1) = +, \text{del}(p_{atm}) = 0$

Initial information: $rv1 = \text{closed}, \text{blockage} = \text{full}$

Find: behaviour of $rv1$

(e) A problem involving a blockage downstream of a relief valve



Boundary conditions: $\text{del}(p_1) = +, \text{del}(p_{atm}) = 0$

Initial information: $rv1 = \text{closed}, rv2 = \text{closed}, f1 > 0$

Find: behaviour of $rv1$ and $rv2$

(f) A problem that requires simultaneous device state changes

Figure 3.1: Problems that can be solved by quasi-steady-state reasoning

these units linked together in series, since the larger system can be easily solved by chaining through the rules in the order of linking. Let us call this simple technique

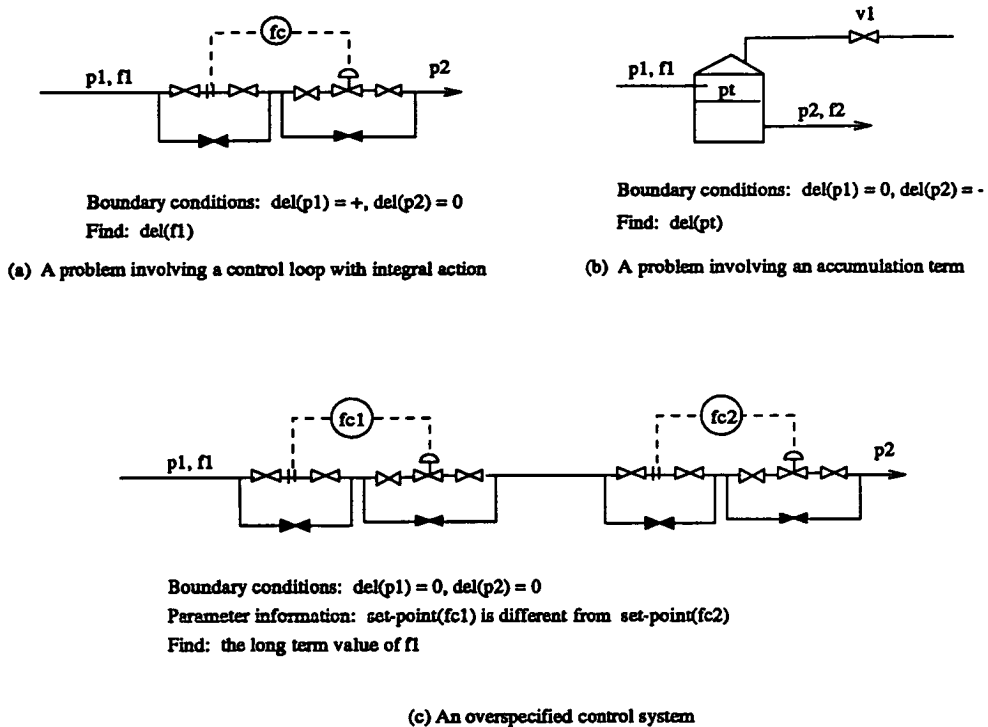
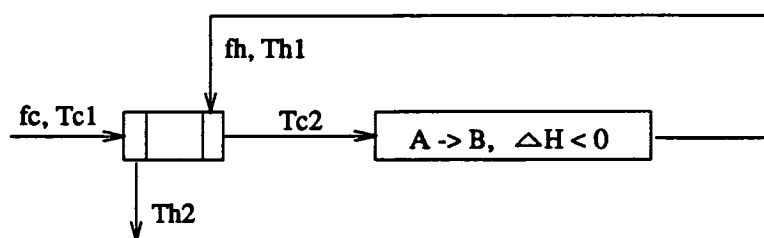


Figure 3.2: Problems that depend upon transition behaviour

“rule-tracing”.

Problem 3.1(b) is slightly more complex because an interaction between temperature and flow now exists. Consequently one must know what is happening to all the inputs, not just ‘ $Tc1$ ’, in order to determine what happens to the outputs. Perhaps one might propose a general heuristic that variables are undeviated unless there is explicit information to the contrary. Unfortunately this approach fails as soon as feedforward and feedback loops appear between variables in the system.

Problem 3.1(c) involves a self-limiting, positive feedback loop. To trace the effect of $\text{del}(T1) = +$ one needs a rule describing the propagation at the junction. This rule must involve the sign of $\text{del}(T2)$, which is not yet known, so let’s suppose that it is assumed



Boundary conditions: $\text{del}(fc) = +$, $\text{del}(Tc1) = 0$

Find: $\text{del}(Th2)$

Figure 3.3: A feedback problem with competing effects

to be zero. Further tracing will soon show this to be inconsistent with the rest of the system. One must then backtrack until the correct answer is obtained, i.e. $\text{del}(T2) = +$ and $\text{del}(T3) = +$. Thus it is important to be able to make and check assumptions, as well as to ensure that the simulation does not trace around the loop indefinitely.

Problem 3.1(d) is of quite a different nature because at least one of the devices on the flowsheet is going to move to a new operating state. What is needed now is a “plot” of system behaviour versus the deviating variable ‘p1’. This should only terminate when nothing more can happen to the system, because there is no information on a limit to ‘p1’ or $\text{del}(p1)$. For the example shown, this will involve the opening of one of the relief valves followed by the opening of the other². To determine which of the valves will open first one can note that the upstream valve has the lower set-point of the two but also the higher inlet pressure because of pressure drop along the pipe. Consequently the upstream valve has to open first. This kind of reasoning has been formalised within the work described in this chapter by the use of relational facts, equations and algebra to qualitatively describe system states. For example, the program reasons about facts such as “ $p2 > p3$ ” and “ $(p2-p5) < \text{set-point}(rv1)$ ”. This will be more fully explained

²It is taken to be extremely unlikely that the two valves will open simultaneously. This helps to make the modelling more efficient but can cause some problems. In particular, there are some situations where a simultaneous state change of several devices is the only feasible answer (e.g. problem 3.1(f) above).

in section 3.5.

The problem in figure 3.1(e) also involves the transition of a relief valve to a new state but this time the valve cannot remain in this new, open state because of the downstream blockage. Consequently, the only description one can have of the long-term behaviour is that the valve will “flutter” about its opening point. To represent this it is necessary to divide relief valve behaviour into at least the 3 states “open”, “fluttering” and “closed”.

Problem 3.1(f) is also difficult because it is impossible for the system to be in a steady-state with one valve open and the other closed. To model the behaviour of this system in a quasi-steady-state manner one must therefore allow both device transitions to occur simultaneously. This will be a serious problem in a large flowsheet if it means that all combinations, of all possible transitions, must be explored in full.

Problem 3.2(a) presents an additional difficulty because the long-term output signal from the control unit to the control valve is not just a function of the long-term value of the input signal to the controller but also depends upon its history. Indeed, if the PI controller is working properly then the input signal will be driven back to its desired value but the control valve will remain perturbed for as long as the system disturbance occurs. This means there is absolutely no way that one can describe the behaviour of the control-unit in terms of a steady state relationship between its input and output signals. One can, however, describe the long-term behaviour of the loop as a whole since it should always fully compensate for an increase in upstream pressure by a slight closure of the valve. The problem with this is that one loses the ability to look within the loop to check that it is indeed working as expected. This will be a serious problem for more extensive loops that encompass several processing units. The concept of modelling loops as a whole has been explored with respect to fault tree synthesis and is further discussed in section 3.6.3.

Problem 3.2(b) encounters similar difficulties due to the accumulative behaviour within the tank. More specifically, the pressure within the tank is not determined solely by the values of ‘p1’ and ‘p2’ but also by whether or not the vent valve ‘v1’ is open when the level changes. Consequently one cannot define the long-term behaviour of the tank (e.g

whether or not it collapses) without considering how transitions actually take place.

Problem 3.2(c) is one where it does not even make sense to talk of a long-term solution because the system as designed is inherently unstable. This is a very simple example of an over-specified control system and would be easy to spot. There may, however, be less obvious cases with similar behaviour. Probably it would be best to pick these up by a degrees-of-freedom analysis before the problem is passed to a qualitative simulator.

Finally, the problem in figure 3.3 is indeterminate unless one has additional “numerical” information. This is because the deviation $\text{del}(\text{fc}) = +$ increases both the cold-side and the hot-side flow into the exchanger, so the overall effect on ‘Th2’ is uncertain. This is further complicated by the effect of $\text{del}(\text{fc})$ on the residence time, and hence the heating effect, in the reactor. The digraph for this flowsheet would contain at least one negative feedforward loop between ‘fc’ and ‘Th2’.

These problems show part of the full range that it would be interesting to tackle. Other problems might involve the role of human operators, maintenance personnel etc.. Dealing with these adds several extra layers of complexity so they will not be considered here.

3.4 Approaches to Qualitative Simulation

A distinction was made in the previous section between problems that could be treated in a quasi-steady-state manner and those that required a more proper treatment of system dynamics. This distinction is not clear cut, since any problem is essentially dynamic, but perhaps a quasi-steady-state approach could sometimes give reasonable answers for less work. Certainly the approaches are quite different since, in the former, the qualitative equations describing the behaviour will be derived from algebraic equilibrium equations, while in the latter they must come from truly dynamic (i.e. differential) equations. This is a distinction that has been unclear in the literature and resulted in some confusion as to whether a method really is dynamic or not.

This section will describe the work of two groups of researchers on qualitative simu-

lation. The first group is De Kleer and Brown [20] who are best regarded as taking a quasi-steady-state approach. The second group is led by Kuipers [55], who takes a truly dynamic approach which requires every variable to be represented as a continuous function of time. With respect to this thesis work it was decided that representing transitions between steady-states should be a pre-cursor to representing more complex dynamic behaviour; consequently, it was decided to explore and extend the method of De Kleer and Brown. These extensions will be described in later sections.

3.4.1 The Steady state Approach

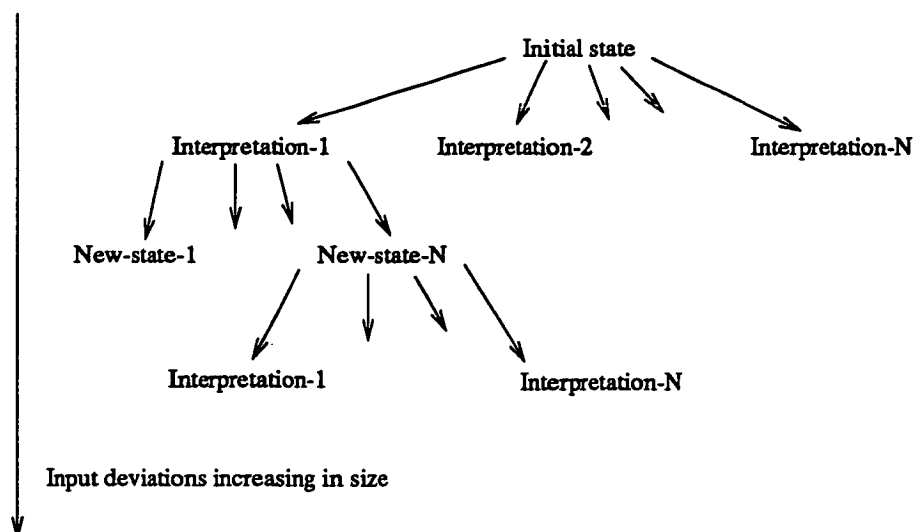


Figure 3.4: The structure of the output from the De Kleer method

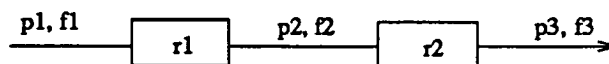
The De Kleer method is split into two stages which are repeatedly applied as shown in figure 3.4. The system is in an initial state which is defined by the states of all the devices involved. This provides one with information that can be used to determine the possible directions in which the deviations may cause the process variables to change. Sometimes there will be a unique answer to this but sometimes one may find several possible complete sets of directions of change; let us call such sets *global interpretations*.

Having obtained these interpretations, each of them must be examined in turn to see what device state changes could possibly result from it. One must then consider the device state changes one at a time and repeat the whole process with the system now in its new state. The method is therefore in two stages which can be called *finding intra-state behaviour* and *finding inter-state behaviour*. These stages are repeated for each of the possible new system states until no more new states are possible. The simulation can branch at either stage.

The De Kleer approach is device centred because one tries to determine system behaviour from that of its constituent devices rather than from more 'structural' knowledge of the system. The concept of device states is crucial because they determine the qualitative descriptors (i.e. equations and facts) that represent the behaviour of the device at a particular moment. For instance, if a valve is closed the behaviour may simply be that the flow is always zero but if a valve is open one must represent the relationship between pressure drop and flow. Thus it is possible to split complex behaviour into a set of simpler cases provided one can switch between them at the appropriate point in the simulation. The need for such a switch is usually due to the values of the process variables but a transition could also be "caused" by external means e.g. a decision to simulate the manual opening of a valve.

The directions of change are represented by difference variables, such as $\text{del}(T1)$, and are only allowed to take on the values +, 0 or -. Device behaviours are represented by *confluence equations* which are used to generate feasible combinations of these difference variables around the device concerned. Let us call such combinations *local interpretations*.

As an example, consider simulating the resistance element problem shown in figure 3.5, which requires one to model the pressure-flow relationship. To do this one can use confluences to describe the behaviour of each individual resistance element. One can then generate the local interpretations around the two devices. These can then be combined to produce a single *global interpretation*. The next few paragraphs will demonstrate this procedure using the representation developed within this thesis work. This is different to that used by DeKleer, although for the aspects discussed here the functionality is very similar. The more important representational differences will be discussed in the



Boundary conditions: $\text{del}(p1) = +$, $\text{del}(p3) = 0$

Find: $\text{del}(f1)$, $\text{del}(p2)$

Figure 3.5: Two resistances in series

next section.

To perform a qualitative simulation of the flowsheet segment in figure 3.5 one can begin at element $r1$, for which the behaviour is represented *qualitatively* by the following confluence equations:

1. $\text{del}(f1) = \text{del}(p1) - \text{del}(p2)$
2. $\text{del}(f2) = \text{del}(f1)$

These are “qualitative algebra” equations which define the possible combinations of signs for the difference variables involved. The first confluence forbids combinations such as $(\text{del}(p1)=+, \text{del}(p2)=0, \text{del}(f1)=0)$; the second confluence says that $\text{del}(f1)$ and $\text{del}(f2)$ must have the same sign. If one considers confluence (1) in isolation then one finds 13 valid combinations for the difference variables $\text{del}(f1)$, $\text{del}(p1)$ and $\text{del}(p2)$. Including confluence (2) doesn’t reduce this number at all but simply introduces the extra difference variable $\text{del}(f2)$. However, when one also considers the input deviation $\text{del}(p1)=+$ this list can be reduced to the following 5 “partial” interpretations:

- $(\text{del}(p1)=+, \text{del}(p2)=+, \text{del}(f1)=+, \text{del}(f2)=+)$.
- $(\text{del}(p1)=+, \text{del}(p2)=+, \text{del}(f1)=0, \text{del}(f2)=0)$.
- $(\text{del}(p1)=+, \text{del}(p2)=+, \text{del}(f1)=-, \text{del}(f2)=-)$.

- ($\text{del}(p1) = +$, $\text{del}(p2) = 0$, $\text{del}(f1) = +$, $\text{del}(f2) = +$).
- ($\text{del}(p1) = +$, $\text{del}(p2) = -$, $\text{del}(f1) = +$, $\text{del}(f2) = +$).

So far this has only considered the “local” behaviour of the device r1, which has resulted in a localised ambiguity because:

1. The sign of $\text{del}(p2)$ is not yet known.
2. When $\text{del}(p1) = +$ and $\text{del}(p2) = +$ then the sign of $(\text{del}(p1) - \text{del}(p2))$ is ambiguous.

This ambiguity is resolved when one includes element r2 in the analysis, due to the introduction of the extra *boundary condition* $\text{del}(p3) = 0$. Using this one can combine the partial interpretations so far created to produce a single global interpretation that displays the correct behaviour:

$$(\text{del}(p1) = +, \text{del}(p2) = +, \text{del}(f1) = +, \text{del}(f2) = +, \text{del}(p3) = 0, \text{del}(f3) = +)$$

This procedure has therefore arrived at the correct solution to the problem but only via an initial branching. If the number of devices had been greater this branching would have increased, along with the number of combinations to be explored.

The confluences that represent the intra-state behaviour of a device are derived from the algebraic equilibrium relationships for the device concerned. As an example, suppose that one is representing flow through a valve, which is *approximately* described by the following algebraic equation:

$$f_{in} = c1 * A * (P_{in} - P_{out})$$

By differentiating this one gets:

$$df_{in} = (c1 * A) * (dP_{in} - dP_{out}) + (P_{in} - P_{out}) * c1 * dA$$

This is a *mixed* equation because it contains both difference variables and “actual” variables. The De Kleer method requires a clear separation of these two types so one must split the device behaviour up into the *pure* confluences shown in Figure 3.6.

Confluence	Conditions
$\text{del}(\text{fin}) = \text{del}(A)$	$A=0, (\text{Pin}-\text{Pout})>0$
$\text{del}(\text{fin}) = 0$	$A=0, (\text{Pin}-\text{Pout})=0$
$\text{del}(\text{fin}) = -\text{del}(A)$	$A=0, (\text{Pin}-\text{Pout})<0$
$\text{del}(\text{fin}) = \text{del}(\text{Pin}) - \text{del}(\text{Pout}) + \text{del}(A)$	$A>0, (\text{Pin}-\text{Pout})>0$
$\text{del}(\text{fin}) = \text{del}(\text{Pin}) - \text{del}(\text{Pout})$	$A>0, (\text{Pin}-\text{Pout})=0$
$\text{del}(\text{fin}) = \text{del}(\text{Pin}) - \text{del}(\text{Pout}) - \text{del}(A)$	$A>0, (\text{Pin}-\text{Pout})<0$

Figure 3.6: Linearised regions to model a valve

To decide which of these to apply one must keep track of the “values” of ‘A’ and (Pin-Pout). In this way one can capture the behaviour of the valve but only by increasing the possibilities for simulation branching. Indeed, one may need some more states to represent valve blockages or leakages, which can lead to serious complexity problems. In addition, it is only possible to decide which confluences should apply at different stages in the simulation if one has a flexible representation for the “values” of the “actual” process variables such as ‘A’, ‘Pin’ and ‘Pout’.

In the work on automatic fault-tree synthesis there has often been poor distinction between difference variables and “actual” variables. Typically all process variables can only be described by the same set of discrete levels, such as “high”, “normal” and “low”. This approach is used in qualitative simulation work to describe the difference variables as having one of the values +, 0 or - . However, such an approach is much too limited to be used for the process variables themselves because of the need to represent when interesting device state changes can occur. For this reason, a process variable is described in qualitative simulation work by reference to a chosen set of values, which Kuipers calls its *landmark* values. For example, one might say one of the following facts:

$$\text{min-design-pressure} < \text{Pin} < \text{max-design-pressure}$$

$$\text{Pin} < \text{Pout}$$

$$\text{Pin} < 10$$

The problem then becomes to update this information when changes are known to have occurred. For instance, if one starts with the information “Pin = 10” and then finds that $\text{del}(\text{Pin})=+$, one can then get the information “Pin > 10”.

The flexibility that is allowed in such a description depends upon the method that is being used. Both De Kleer and Kuipers allow a totally ordered set of landmarks to be predefined for each variable and allow a variable to be described as being at a landmark or between two of them. Kuipers also allows additional landmarks to be created as the simulation proceeds, in order to represent limits reached in oscillations. However, the method pursued in this thesis work provides an additional layer of description by the use of binary relational facts. This allows one to describe variables by reference to other process variables. This will facilitate the kind of reasoning with inequalities that was necessary to decide which of the relief valves in figure 3.1(d) would open first.

3.4.2 The Dynamic Approach

The representation used by Kuipers is really quite different, and perhaps more flexible, than that used by De Kleer and Brown since the approach is functional rather than device centred. All the variable interactions are specified by relationships of the following types:

Add(f,g,h) i.e. $f(t)+g(t)=h(t)$

Mult(f,g,h) i.e. $f(t)*g(t)=h(t)$

Minus(f,g) i.e. $f(t)=-g(t)$

Deriv(f,g) i.e. $g(t)=(df/dt)$

Mplus(f,g) ‘f’ is a monotonically increasing function of ‘g’.

$M_{\text{minus}}(f,g)$ 'f' is a monotonically decreasing function of 'g'.

These can be used to represent complex behaviour by the creation of pseudo-variables.

As with De Kleer's confluences the applicability of these equations is defined by the use of variable landmarks. Again, the simulation may branch due to ambiguous behaviour and terminates when no new system states can be found, either because the system is in a steady state or because one has discovered an oscillation cycle.

Dalle Molle et. al. [19] describe the application of the "Qsim" programs to some representative chemical engineering systems. Initially they discuss the modelling of first, second and third order systems with a single input and a single output topology (SISO systems). Qsim correctly generates all possible behaviours, and no others, for step inputs and ramp inputs to such systems. Next the authors discuss an example involving a hot/cold mixing tank. They demonstrate that the construction of a Qsim model by a straight-forward transcription of the differential equations for this system results in unnecessary ambiguity. This is attributed to the loss of global information about parameter dependencies (e.g. that the hot stream temperature is greater than the cold stream one!). For the tank example this problem is solved by the algebraic-manipulation of the system equations to produce a redundant constraint.

The authors then show that Qsim correctly predicts off-set and overshoot when modelling a proportional feedback control loop around a tank. Finally, they mention that Qsim can also correctly predict the lack of off-set if a proportional-integral controller is used instead, but allude to further problems with trying to model such integral action.

The Qsim models used by Dalle Molle were constructed by hand and could not be straight-forwardly combined to model a network. Indeed, the algebraic manipulation needed for the tank example suggests that model building is something of an art. In addition, Dalle Molle [18] describes several ambiguity problems that beset Qsim simulations. The most important of these seems to be "chattering" which can cause a simulation to run indefinitely. The problem arises because Qsim uses a truly dynamic method of qualitative simulation and so must keep track of derivatives of all orders. This means that it is possible for the higher-order derivatives to "chatter" continuously

without any changes ever reaching the lower orders that one is interested in. A possible solution to this problem is simply to ignore the higher-order derivatives but this prevents the modelling of oscillations. A better solution is to place additional constraints on them to prevent the chatter. For some problems the required constraints can be derived automatically but for others they must be derived by hand.

In his thesis Dalle Molle also describes later versions of the Qsim suite of programs which allow quantitative information to be used during a simulation. In particular, numerical ranges can be assigned to landmark values and this information used to restrict interpretations. This can successfully eliminate ambiguity in the initial stage and terminating branches of a simulation but is less successful in the intermediate steps. Thus, if there are several possible paths between the same initial and final points these will probably not be reduced by this method. The use of such information has also been a central part of the work described in the next section, though the approach taken is very different. Qsim uses a numerical iteration method to restrict interpretations. This should be quite efficient but suffers from the disadvantage that one cannot directly use facts which relate two variables (or even two landmarks). In contrast, the approach taken in this thesis is to include the quantitative information using a general mechanism for reasoning with such binary relational facts. This is more powerful but severe efficiency problems result.

In summary, Kuipers' method has been extensively developed and has been successfully applied to some simple chemical engineering problems. Its main advantages over the DeKleer method might be seen as follows:

1. It takes a dynamic approach and so can potentially capture behaviour that depends upon the history of a transition.
2. It has been formulated mathematically, which might be useful for proving the validity of the associated algorithms.

However, these advantages are also its weaknesses since it is rather more complex than DeKleer's method and less well-suited to non dynamic problems. In particular,

it is more difficult to use when one's problem is indeed best represented in a quasi-steady-state manner. A good example of this is for modelling the flow of a liquid through a pipe, where a dynamic approach will have to unnecessarily consider liquid compressibility. In addition, the functional approach taken by Kuipers makes the method unsuitable for network problems where there are several options for which device will change state. This poses a difficulty for its use in chemical plant design since flowsheets are network problems by definition. Kuipers' method was therefore rejected in favour of the DeKleer method for the first stage of this thesis study. The results of this work will now be described.

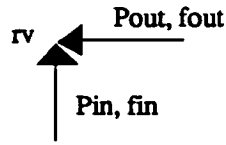
3.5 Extending the DeKleer method

The work carried out for the first part of this thesis resulted in a qualitative simulation program written in Prolog. Most of the ideas behind this program will be described in this chapter but see Appendix B for more detail on the code itself and the program input/output.

3.5.1 Representation

The program follows the two stage methodology described in the previous section and illustrated in figure 3.4. Only one device transition is allowed to occur at a time which means that when a new system state is created all devices but one remain in their previous state. A system state is marked as a "repeating state" if all its device states correspond to one that has been explored already. This arises due to an ambiguity in the system behaviour, as represented by a set of confluences, rather than because one has discovered a true oscillation in time. Consequently, a repeating state is considered no further.

If a state is genuinely new then all the information that applies within that state is collected together and used to continue the simulation. One such piece of information is that the boundary conditions (input deviations) are the same as they were in the last state. Other pieces come from the representation of the individual device states,



Variables: P_{in} , f_{in} , P_{out} , f_{out} , A (open area of valve)

Parameters: $\text{set-point}(rv)$

State = Fluttering

Confluences :

1. $\text{del}(f_{in}) + \text{del}(P_{out}) = \text{del}(A) + \text{del}(P_{in})$; the pressure balance.
2. $\text{del}(f_{out}) = \text{del}(f_{in})$; the mass balance.
3. $\text{del}(A) + \text{del}(P_{out}) = \text{del}(P_{in})$; force balance on spring.
4. $\text{del}(f_{in}) = \text{del}(A)$; an additional confluence

Infacts : $f_{in} = 0$, $f_{out} = 0$, $A = 0$, $(P_{in} - P_{out}) = \text{set-point}(rv)$

Relational equations : None

Transitions :

Fluttering -> Closed :

Trans-infacts : $f_{in} = 0$, $f_{out} = 0$, $A = 0$, $(P_{in} - P_{out}) < \text{set-point}(rv)$

Fluttering -> Open :

Trans-infacts : $f_{in} > 0$, $f_{out} > 0$, $A > 0$, $(P_{in} - P_{out}) > \text{set-point}(rv)$

Figure 3.7: A Partial Representation of a Relief Valve

examples of which are given in Figures 3.7 & 3.8. For each state of the system one can get the following types of information:

1. The **boundary conditions** that are being applied i.e. the input deviations.
For example, " $\text{del}(p_1)=+$ " and " $\text{del}(p_3)=0$ " are the boundary conditions in

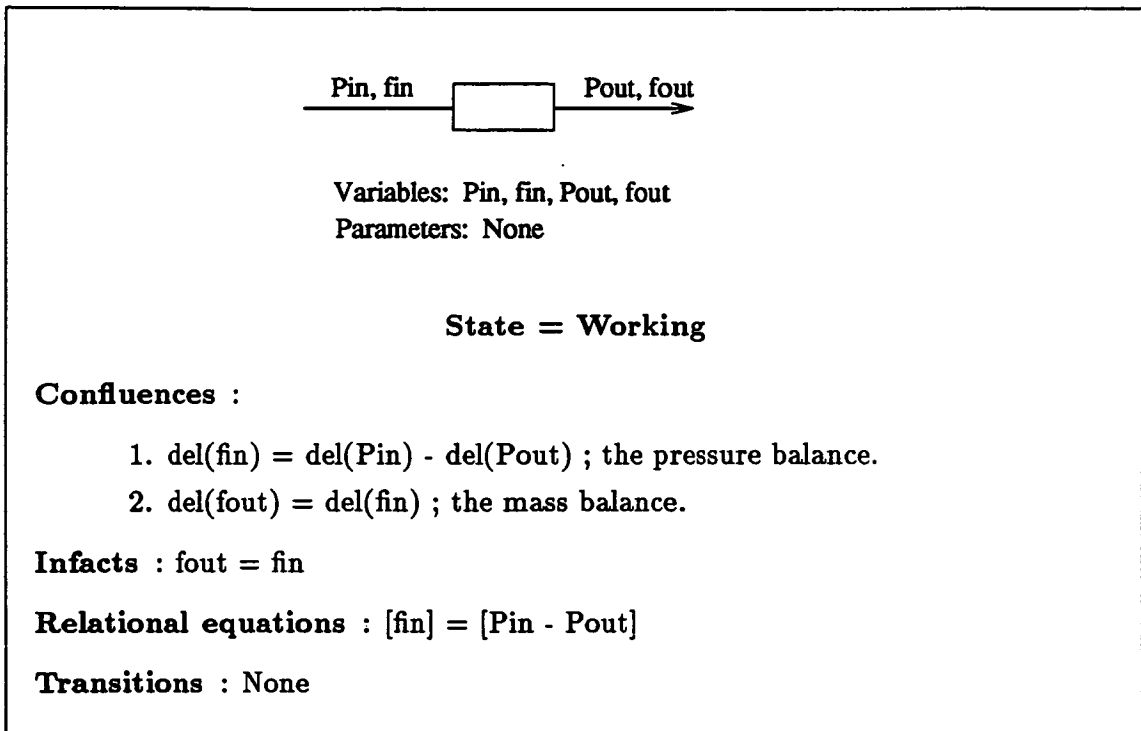


Figure 3.8: A Partial Representation of a Resistance Element

Figure 3.5.

2. A set of **infacts** which are relational facts that must hold if the system is *within* the state (but not necessarily when the system is at the boundaries of the state). For example, the fact:

$$(\text{Pin} - \text{Pout}) = \text{set-point}(\text{rv})$$

must be true while a relief valve is in the “fluttering” state (see Figure 3.7).

3. A set of **confluence equations** that must be satisfied within the state. For example, the confluence:

$$\text{del}(\text{fin}) = \text{del}(\text{Pin}) - \text{del}(\text{Pout})$$

must be satisfied within the “working” state of a resistance element (see Figure 3.8).

4. A set of **relational equations** that must be satisfied within the state. For example, the relational equation:

$$[\text{fin}] = [(\text{Pin-Pout})]$$

must be satisfied within the "working" state of a resistance element (see Figure 3.8).

In addition, one also needs information about the device state transitions that could possibly occur. Associated with each such transition is a set of **trans-infacts** which are relational facts that must hold when the transition is occurring. For example, the following fact:

$$(\text{Pin-Pout}) > \text{set-point}(\text{rv})$$

must be true for a transition to occur from the "fluttering" to the "open" state of a relief valve (Figure 3.7). These facts can be used, together with the relational equations, to try to disprove the feasibility of some of the transitions.

The use of confluences is similar to that described in the last section (when solving the problem in figure 3.5). The algorithm simply generates the local interpretations for each confluence and tries to pick one from each set to produce the complete global interpretation. A choice is known to be inconsistent with those made before it when a difference variable has been assigned two different values. In this case it is necessary to backtrack to make other choices. In addition, backtracking is forced when a complete global interpretation has been found, in order to find all other possibilities as well.

The relational equations are analogous to De Kleer's confluence equations except that they concern 'actual' variables instead of difference variables. For example, the relational equation:

$$[\text{fin}] = [\text{Pin-Pout}]$$

has the following local interpretations:

- $([fin]= +, [Pin-Pout]= +)$.
- $([fin]= 0, [Pin-Pout]=0)$.
- $([fin]= -, [Pin-Pout]= -)$.

which are equivalent to:

- $(fin>0, Pin>Pout)$.
- $(fin=0, Pin=Pout)$.
- $(fin<0, Pin<Pout)$.

These can be used to rule-out a transition by showing that none of these local interpretations is consistent with the facts that are already known. This is analogous to the method used to find the global interpretations for a set of confluences and a set of boundary conditions. It is, however, rather more complex because one must prove inconsistency by chaining through the facts involved; as opposed to just checking that the same difference variable has not been assigned two different values.

As an example consider the first local interpretation above, which introduces the facts " $fin > 0$ " & " $Pin > Pout$ ". To check the consistency of these with the partial interpretation already built-up is quite complex as it involves extensive reasoning with inequalities. This is done through a set of general goals that exhaustively try to prove or disprove a given relational fact from a set of such facts. To improve efficiency the allowable facts are restricted to the following forms:

$ \begin{array}{l} X > N, \quad X = N, \quad X < N, \\ (X-Y) > N, \quad (X-Y) = N, \quad (X-Y) < N, \\ (X-Y) > (P-Q), \quad (X-Y) = (P-Q), \quad (X-Y) < (P-Q) \end{array} $

where 'X','Y','P' and 'Q' must be variables or numbers, rather than algebraic expressions, and 'N' is just a number. These forms are sufficient for the purposes described here. In addition, there is a further restriction that any substitute facts which are generated only compare like with like (i.e. it is not necessary to try to prove facts which are dimensionally inconsistent, such as $P_{in} > f_{in}$). Even with these restrictions, however, the use of relational equations still turns out to be extremely costly. This will be discussed in the next section.

3.5.2 Capabilities and Limitations

This section discusses the problems that the program was able to solve and those that it was not. Perhaps it could be extended to include some of the latter but there are numerous additional difficulties which will arise if this is attempted. In addition, there are serious efficiency problems with the general approach, which will be seen to severely limit its usefulness. This will be shown to be strongly dependent upon the order in which one attempts to solve confluences and relational equations. Nevertheless, the order was sensible for the examples attempted here so the poor statistics should not be dismissed as unrepresentative.

At present the program is really set up to be able to solve problems involving relief valves, such as that in figure 3.1(d), so the only device types allowed are:

relief valves
resistance elements
binary junctions
binary splits
total blockage points

and the only process variables considered are pressure and flow. Theoretically this list could be expanded ad-infinity but in practice some problems appear because of differences in the kinds of inferences that one may wish to draw with the new and the old units. This is a problem that will arise with any program that can supposedly

be extended by simply including extra units. Indeed, such scepticism should also be applied to programming techniques, such as rule-based programming, about which similar claims are made. The point is that generality cannot be deduced from the mere physical capability of including extra rules or units but only by actually doing so. Needless to say, such extension must also result in correct inferences *without* any changes having to be made to the current program structure.

Section 3.3 has already described a range of problems that one may wish to solve. As it stands the program is capable of solving those in figures 3.1(a), 3.1(b), 3.1(d), and 3.1(e). The feedback loop in figure 3.1(c) is not in itself a problem, as the program uses an appropriate backtracking algorithm, but this example does require the modelling of temperature propagation in a split. This will be shown below to be problematical. The solution of the problem in figure 3.1(f) is not possible until there is an answer to the question of when to consider more than one transition at a time. Finally, as already described, the problems in figures 3.2(a), 3.2(b) and 3.2(c) cannot be appropriately solved by a quasi-steady-state methodology, while that in figure 3.3 requires recourse to numerical arguments.

The program was initially written in Edinburgh Prolog and implemented on a Sun 3/50 Workstation. Prolog runs quite slowly but is very convenient for implementing and testing new ideas in a modular fashion. The program input is in the form of very simple prolog facts (as described in Appendix B.4). This isn't particularly user-friendly but is appropriate to the general usefulness of the original program. This is limited because the time taken for the simulation exceeded, substantially, that required for a numerical simulation involving considerable detail! Later sections will discuss possible improvements arising from subsequent thesis work and that of other researchers in related fields. These have the potential to improve the efficiency of both stages of the Dekleer method. However, it remains to be seen whether the combinatorial problem can be reduced enough to obtain a usable tool.

The combinatorial problem is partly dependent upon the order in which the confluences and relational equations are solved. To see this, consider finding the valid global interpretations for the problem in figure 3.9, which involves ten simple resistance elements in series. For each of these one can write confluences of the following form:

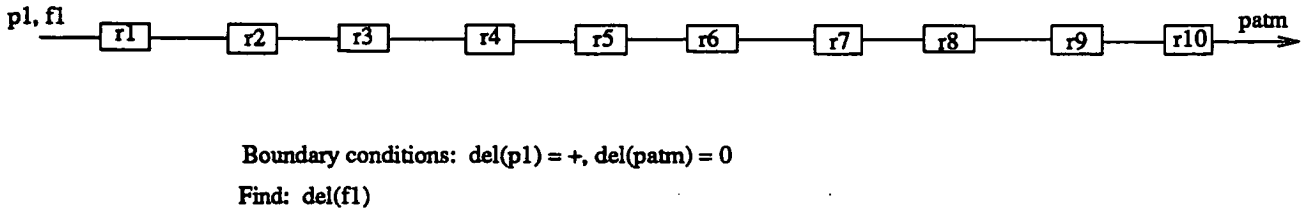
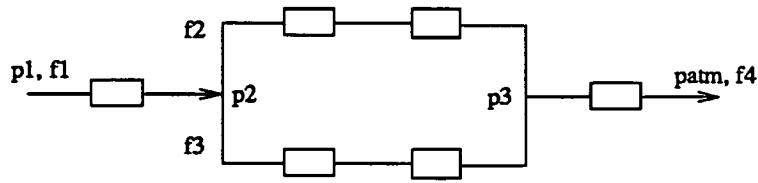


Figure 3.9: Ten resistance elements in series

1. $\text{del}(fin) = \text{del}(Pin) - \text{del}(Pout)$
2. $\text{del}(fout) = \text{del}(fin)$

As shown in the last section these lead to 13 local interpretations for the difference variables $\text{del}(Pin)$, $\text{del}(Pout)$, $\text{del}(fin)$ and $\text{del}(fout)$. Now, if one starts the analysis with the boundary conditions at one end and works *causally* along the ten elements then many of these choices will be rejected early on as inconsistent. This means that such choices do not make combinations with those further down the line. On the other hand, if one doesn't involve the boundary conditions (Bc's) until all ten elements have been combined then the number of partial combinations is going to be very large. Indeed, the absolute maximum for this number is of the order of 13^{10} i.e. 1.38×10^{11} combinations! Fortunately by including the boundary conditions at the start and considering the units in a sensible order one can reduce this to a reasonable number.

One therefore gets the best efficiency by using causality to drive constraint satisfaction. Unfortunately, this is only really effective for the case of straight sections of flowsheet because in a branched section it is difficult to know the best strategy for searching the branches. In the thesis program the order of solution is simply defined by a fact in the input file. This is somewhat unsatisfactory because this order should be optimised as the simulation proceeds and new pathways are opened up (e.g. by the opening of relief valves). Nevertheless, for the problems described already the chosen order was fairly sensible.



Boundary conditions: $\text{del}(p1) = +$, $\text{del}(patm) = 0$

Find: $\text{del}(f1)$, $\text{del}(f2)$, $\text{del}(f3)$, $\text{del}(f4)$

Figure 3.10: A branched network of resistance elements

One can see from this discussion that branching makes it difficult to choose a confluence solution order that will be processed efficiently. Unfortunately, this is not the only problem that branching causes for qualitative simulation since it can also make it difficult to make the right inferences. Consider the network in figure 3.10. ‘By eye’ one knows this to be deterministic but the application of confluences alone will produce a great number of global interpretations. This is because of the ambiguities inherent in qualitative algebra. For instance, one cannot show that $\text{del}(f2) = +$ and $\text{del}(f3) = -$ is an infeasible combination because when they are combined at the junction the sign of $(\text{del}(f2) + \text{del}(f3))$ is ambiguous, hence so is that of $\text{del}(f4)$.

One can reduce the number of possible interpretations from the branched network by including *extra* confluences within the modelling. At the extreme this could mean the confluence:

$$\text{del}(f2) = \text{del}(f3)$$

but this will be difficult to justify without complicated arguments. A more “natural” one to include might therefore be one representing the overall mass balance for the network:

$$\text{del}(f1) = \text{del}(f4)$$

This is easy to justify and reduces the number of interpretations somewhat. However, it does still leave the following five alternatives:

- $(\text{del}(p1)=+, \text{del}(Patm)=0, \text{del}(f1)=+, \text{del}(f2)=+, \text{del}(f3)=+, \text{del}(f4)=+)$
- $(\text{del}(p1)=+, \text{del}(Patm)=0, \text{del}(f1)=+, \text{del}(f2)=+, \text{del}(f3)=0, \text{del}(f4)=+)$
- $(\text{del}(p1)=+, \text{del}(Patm)=0, \text{del}(f1)=+, \text{del}(f2)=+, \text{del}(f3)=-, \text{del}(f4)=+)$
- $(\text{del}(p1)=+, \text{del}(Patm)=0, \text{del}(f1)=+, \text{del}(f2)=0, \text{del}(f3)=+, \text{del}(f4)=+)$
- $(\text{del}(p1)=+, \text{del}(Patm)=0, \text{del}(f1)=+, \text{del}(f2)=-, \text{del}(f3)=+, \text{del}(f4)=+)$

The problem can therefore only be fully resolved by the inclusion of the first extra confluence mentioned above.

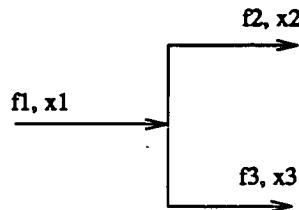
At present the inclusion of extra confluences is done in the input file, rather than by generating extra confluences automatically. Doing so should be feasible but it might be difficult to decide which extra ones are useful and which are not. This will particularly be a problem in a highly branched network. Some of the extra confluences, such as the second one above, are derived from equations which would be redundant in a numerical scheme but remain useful in a qualitative scheme. This concept has also been discussed by Stephanopolous [96].

This discussion has so far considered problems of efficiency and inference that arise from loops and branches in the flowsheet. Now let's consider problems that arise in trying to expand the program by the inclusion of additional processing units. Refer back to figure 3.1(b) and consider creating a model for the 'normal' state of the heat exchanger involved in this example. Two of the confluences for this might be:

1. $\text{del}(\text{Th2}) = \text{del}(\text{Th1}) + \text{del}(\text{Tc1}) + \text{del}(\text{fh}) - \text{del}(\text{fc})$
2. $\text{del}(\text{Tc2}) = \text{del}(\text{Th1}) + \text{del}(\text{Tc1}) + \text{del}(\text{fh}) - \text{del}(\text{fc})$

Semantically this is correct but one should note that each of these has 181 valid local interpretations whereas the confluence which related pressure and flow in the simple re-

sistance elements in figure 3.5 had only 13. Consequently, any combinatorial problems that were encountered before will now be magnified many times. Indeed, an indeterminate problem involving only a few heat exchangers would probably exhaust memory space, even on a large computer.



(flows are positive as shown)

Figure 3.11: Representing composition in a binary split

Now consider including composition or concentration as a variable in the models. Take the binary split shown in figure 3.11. This is represented with the flows in certain directions but these may change as a result of pressure disturbances. This is unfortunate because one can only reason about composition if the flow directions are known. To see this let's take the mixed differential equations that represent the component balance:

1. $f1 \cdot dx1 + x1 \cdot df1 = f2 \cdot dx2 + x2 \cdot df2 + f3 \cdot dx3 + x3 \cdot df3$
2. $df1 = df2 + df3$

and try to split them into a set of pure confluences. This will involve behaviours such as:

- $\text{del}(x2) = \text{del}(x1)$, $\text{del}(x3) = \text{del}(x1)$, $\text{del}(f1) = \text{del}(f2) + \text{del}(f3)$...if $f3 > 0$, $f2 > 0$, $f1 > 0$
- $\text{del}(x2) = \text{del}(x1) + \text{del}(x3)$, $\text{del}(f2) = \text{del}(f1) - \text{del}(f3)$, $\text{del}(x2) = \text{del}(f1) + \text{del}(f3)$...if $f3 < 0$, $f2 > 0$, $f1 > 0$, $x1 > x3$

and many others. Indeed, one needs at least one device state for each of the 13 possible combinations of the flow difference variables $\text{del}(f1)$, $\text{del}(f2)$ and $\text{del}(f3)$, and maybe more for cases such as " $x3=0$ ". Similar problems will also arise if using temperature as a variable and will be compounded if using both temperature and concentration. This demonstrates the difference between the low dimensionality problems considered in the AI literature and the high dimensionality problems that must be solved for the qualitative simulation of real industrial processes.

Additionally, one should note that it may not even be possible to always split up a mixed confluence in the required way. This is shown by considering a general algebraic equation and its differential form:

$$z + f(x,y) = 0 \rightarrow dz + (df/dx)*dx + (df/dy)*dy = 0$$

To split this up into pure confluences requires defining suitable landmarks for 'x' and 'y' that can determine the signs of (df/dx) and (df/dy) . A minimum criteria to do this is that algebraic solutions exist for the following turning points:

$$(df/dx) = 0 \text{ and } (df/dy) = 0$$

which is by no means possible in the general case.

3.5.3 Further developments

The discussion so far has primarily just repeated the findings of a paper which appeared in *Chemical Engineering Research and Design* [111]. Now, however, the discussion will turn to more recent work in this area. Firstly this will focus upon possible improvements to the DeKleer method per. se. but Section 3.6 will then widen the discussion to bring in the work of other researchers.

The first problem addressed is that of the first stage of the DeKleer method. That is, solving a set of confluences to find the intrastate behaviour.

3.5.3.1 Confluence Solution

The original Prolog code responsible for confluence solution was based on a generate and test backtracking strategy. This was similar to the algorithm demonstrated in Section 3.4 by working through the example in Figure 3.5. This algorithm was very easy to implement in Prolog due to the declarative nature of Prolog programs. Indeed, it was a striking example of the advantages of a clean representation, compared with the ad-hoc rule-based approach discussed in Chapter 2. Unfortunately, however, the code was inefficient for the following reasons.

1. Prolog is a slow language compared to other languages such as C, or even LISP.
2. The step which generated the local interpretations around a confluence assigned values to all the variables involved. This resulted in unnecessary intermediate branching of the search.

To understand the second point better consider the following confluence:

$$\text{del}(f1) = \text{del}(p1) - \text{del}(p2)$$

Now, suppose that one already knows the assignment $\text{del}(p1) = +$. The original algorithm would then obtain the five partial interpretations already listed in Section 3.4. However, in reality the confluence would be completely satisfied by any *one* of the following assignments:

1. $\text{del}(f1) = +$
2. $\text{del}(p2) = +$

It is therefore sufficient to just generate and test these *partial assignments* because if any of them are made then the remaining variables in the confluence are no longer constrained by it. Of course, one may later find that these free variables are constrained by other confluences but that can be dealt with in turn.

1. Propagate the assigned deviations as far as possible (i.e. perform *constraint propagation*).
2. Are there any confluences not yet satisfied?

 Yes -> Go to (3).
 No -> Record the global interpretation just found.
 Backtrack to (4) or (3).
3. Choose one of the confluences not yet satisfied. Generate the *partial* assignments that satisfy it. Remove it from the confluence list. Go to (4).
4. Choose one of the partial assignments from (3). Go to (1) and propagate it.

Figure 3.12: A new algorithm for Confluence Solution

In order to explore these observations further a new algorithm was implemented for confluence solution (Figure 3.12). The program is written in LISP and makes use of the partial assignment concept described above. The constraint propagation stage of the algorithm (step 1) takes the assignments one at a time and applies them to the confluences they are involved in. This can result in several things happening to the confluence:

1. It is satisfied and removed from the list.
2. It is violated; in which case propagation stops and backtracking begins early.
3. It results in a new assignment (i.e. a confluence with only one variable). In this case the old confluence is removed from the confluence list and the new assignment added to the list of ones to be propagated.
4. It results in a new confluence. For example, a previously undisturbed confluence is now known to have a positive right hand side. In such a case the old confluence

is removed from the confluence list and the new one included.

Figure	Device states	Boundary conditions	Speed-up	LISP time (s)
3.9	All working	$\text{del}(p1) = +, \text{del}(p11) = 0$	34.5	0.8
3.9	All working	$\text{del}(p1) = 0, \text{del}(p11) = 0$	14.5	0.7
3.9	All working	$\text{del}(p1) = -, \text{del}(p11) = 0$	34.5	0.8
3.9	All working	$\text{del}(p1) = -, \text{del}(p11) = +$	6.7	4.3
3.9	All working	$\text{del}(p1) = +, \text{del}(p11) = +$	14.2	1.9
3.9	r6 blocked	$\text{del}(p1) = +, \text{del}(p11) = 0$	24.8	0.46
3.1(d)	rv1 = open, rv2 = fluttering	$\text{del}(p1) = +, \text{del}(\text{patm}) = 0$	142.2	1.14
3.10	Five resistors on each branch (instead of 2). All working	$\text{del}(p1) = +, \text{del}(\text{patm}) = 0$	39.6	4.12

Figure 3.13: Speed-up Statistics for the new confluence solution algorithm

The LISP program is rather more complicated than the original Prolog version, for two reasons. Firstly, the algorithm itself is more complicated and secondly, LISP programs have to use recursion and iteration to implement backtracking. Nevertheless, it is considerably more efficient. Indeed, Figure 3.13 shows that for some representative problems the compiled LISP program is up to 140 times as fast as the original version. Further speed-up could be achieved by rewriting it in C. This would be relatively straightforward since it is algorithmic, rather than declarative, in nature.

This subsection has described a new algorithm which greatly improves the efficiency of the first stage of the De Kleer method (confluence solution). Section 3.6 will describe other work relevant to this stage, particularly that of Mark Kramer's group at MIT. First, however, there is a discussion of work relevant to the second stage of the method.

3.5.3.2 Finding Interstate Behaviour

This thesis has discussed in detail an extension to the second stage of the DeKleer method. In particular, it discussed the need to be able to prove or disprove the consistency of a set of binary relational facts. In the original Prolog program this was done by the use of rewrite rules to generate substitute facts to consider as well. Such substitution continues until an inconsistency has been identified or all possible substitutions have been performed. Equalities are treated specially to prevent infinite recursion so the search is guaranteed to terminate. However, the only efficiency technique used is to require that all substitute facts compare like with like. Not surprisingly, therefore, this approach is exceedingly combinatorial and hardly a practical proposition.

This may not be the end of the story, however, because AI researchers concerned with robotics appear to be using such binary relational information to reason about space and time [108, 87]. One of the approaches considered is to use the first stage of a linear programming system to see if the set of constraints (facts) has a feasible starting point. However, Valdes-Perez [108] considers such an approach an inefficient method for solving the *simple linear programs* (SLP's) that are involved. As an alternative, he describes his own algorithm and program, which can be used to check the consistency of facts of this form. He claims that the complexity of his method is of the order of:

$$E * \min(L, V),$$

where E = number of constraints (relational facts)

V = number of quantities (variables)

L = number of facts whose right hand side number is negative.

This looks extremely promising since the complexity of the approach described in the previous sections will probably be "exponential" in the number of facts. However, Valdes-Perez does say that "to solve" the set of inequalities means to assign *non-negative integers* to the quantities involved so perhaps there is something which restricts application in the general case. Unfortunately, this area of work was identified after this thesis study had moved away from qualitative simulation, so time constraints have prevented a proper evaluation.

3.6 Related Work by Chemical Engineering Researchers

The last two sections have discussed possible ways of improving the efficiency of the DeKleer method. The first subsection below continues this discussion by considering work which is not directly aimed at qualitative simulation. Real improvements might result from these new directions but one must bear in mind the large number of problems identified just within this limited thesis study. One is tempted to conclude, therefore, that the hope for the DeKleer method as a practical *simulation* tool is limited. Neither is it clear that a method such as is used by Kuipers would escape all of these problems.

This is not to say that the development of qualitative simulation has been fruitless because one may still be able to apply some of the representational techniques to other ways of reasoning about process plant behaviour. Sections 3.6.2 and 3.6.3 try to demonstrate this by considering two very different problem areas. The first is the problem of automatically synthesizing operating procedures for a chemical plant. Mostly this has been tackled using techniques from AI planning but many of the authors involved have recognised the central importance of qualitative modelling. Indeed, the success or failure of a planning algorithm appears to be closely related to the power of the underlying qualitative representation. The second area is that of automatic fault tree synthesis, which is a top-down technique. Previous approaches at solving this are discussed and shown to be limited by the poor semantics of the representation used. In particular, the failure to distinguish different types of event makes it difficult to apply all the information that is potentially available. The section concludes by showing the concepts from qualitative simulation that might be incorporated to solve this problem.

3.6.1 Fault Diagnosis

The work which bears the closest resemblance to that already described above is that of Mark Kramer and his students at MIT [53, 75]. They are really concerned with the problem of fault diagnosis but have been exploring the use of qualitative simulation to construct large-scale diagnostic rules. From a fault diagnosis viewpoint their approach

is somewhat limited because it can only allow for one fault at a time and relies upon a steady state, rather than a dynamic, analysis. In addition, the qualitative simulation is limited to confluence solution, which is only the first stage of the DeKleer method. Thus they avoid the use of landmark values, which means there is no general mechanism for accounting for device transitions. Indeed, the only transition considered is the saturation of a control loop. Nevertheless, they take a rather different approach to confluences which enables them to tackle some of the complexity and ambiguity problems identified in this chapter. Thus they do significantly extend the debate.

The major advance achieved by Kramer et. al. is to start the analysis from a *signed directed graph* (SDG) of the process instead of from a set of devices (and confluences). This enables them to reduce the graph before solution, in order to remove some of its inherent complexity and ambiguity. They describe a manual procedure for this in the earlier paper [53]. It is not clear whether this has yet been fully automated.

Kramer et. al. are concerned with the derivation of diagnostic rules from the SDG. One approach to this is to find all possible interpretations of the SDG, under some hypothesised fault. One can then combine these into rules to remove redundant or uninteresting information. However, they consider that this is not feasible, in general, because the number of interpretations increases combinatorially with the number of control loops and the number of nodes where paths converge. Instead, they first employ the following strategies to reduce the SDG:

1. Unmeasured nodes are removed from the SDG using an algorithm that is careful to preserve the propagation paths that they represented.
2. Conditional branches are added connecting disturbances and manipulated variables in a control loop. These branches are conditional on the controller working i.e. not being saturated. So long as this is the case then the disturbance perturbs the manipulated variable, in order to counteract itself. However, if the controller is saturated then the manipulated variable can change no more and there will be no compensatory response.
3. Special consideration is given to positive feedback loops to remove some of their ambiguous behaviour. The problem can be seen by referring back to Figure 3.1(c)

in this chapter. The SDG for this example would contain a positive feedback loop between T2 and T5. This means that if $\text{del}(T2)$ and $\text{del}(T5)$ have the same sign then the loop confluences are satisfied, irrespective of the boundary conditions to the problem! In reality, however, perturbations in T5 and T2 need some external cause. Kramer et. al. get around this by recognising such situations in the SDG and placing extra conditions on their rules to ensure the existence of such a cause. In contrast, the work described in previous sections has only considered how to use causal information to improve the efficiency of confluence solution, rather to improve the answer obtained.

Kramer's later paper [75] considers the derivation of confluences from the SDG and extends some of the earlier notions. Ambiguity is reduced both by changes to the SDG before the construction of confluences and by the addition of extra confluences to the derived set. The latter approach might utilise confluences based on the overall mass balance or other conservation equations. This choice is described as heuristic, which supports observations made in previous sections about the difficulty of automatically selecting extra confluences. However, the changes to the SDG revolve around an important heuristic that has not yet been discussed in this chapter:

“An effect cannot completely compensate for its own cause”.

As an example, consider an adiabatic CSTR containing an endothermic reaction. If the temperature of the inlet stream is increased then this will tend to increase the temperature within the reactor. This, in turn, will increase the reaction rate and so the amount of heat consumed by the reaction. The latter effect will tend to decrease the temperature within the reactor again but it is obviously impossible for it to drop back to its original, unperturbed value. Unfortunately, the SDG that you would intuitively construct for this situation would probably allow for such *compensatory response* (CR) or even for *inverse response* (IR).

To solve the problem Kramer et. al. introduce the notion of a simple causal path (SCP) which can be traced from the primary deviation to any subsequent event without repeating a node i.e. an acyclic path in the SDG. They then prevent a node from changing sign unless there is an SCP between the new event and the primary deviation.

This has the effect of cutting out all negative feedback loops i.e. all compensatory response. However, this is not completely valid since a change in tank level can indeed compensate fully for a change in inlet flow and controllers can exhibit inverse response. Consequently, Kramer et. al. introduce the Extended Signed Directed Graph (ESDG) which is able to capture the integral behaviour of such units but still allow the use of the SCP assumption. The SDG uses a self cycle to represent integral behaviour. This enables the authors to construct an algorithm to automatically derive the ESDG from the SDG.

Clearly, Kramer et. al. have been doing some work which is relevant to the problem of finding intrastate behaviour. Dohnal [21] also describes some work using DeKleer type confluences but with much less theory and detail. He mentions two qualitative simulation programs that his group has written. The first is in Prolog and can be used for the solution of qualitative non-linearities for a limited number of confluences. The second is written in FORTRAN and can deal with an unlimited number of linear confluences. Presumably the difference is that in the former case the non-linear equations are treated in a piecewise linear fashion by keeping track of transitions between regions. The limit on the number of confluences certainly suggests some efficiency problems but it is not clear whether these are to do with speed or memory limitations.

The use of confluences in fault diagnosis has also been mentioned by Krishnamurthy [54] and by Ventkatasubramanian [109] but their attempts seems to be more like the kind of reasoning discussed in Chapter 2, which we have called "rule-tracing". In addition, Niida et. al. [73] describe a diagnosis kit which is supposed to contain a qualitative simulator but no details are given. Finally, Tsuge et. al. [105] describe some qualitative simulation work which is broadly aimed towards use in fault diagnosis. Their approach seems to be essentially rule-based, though they claim to have no difficulty in dealing with recycles or bypasses.

3.6.2 Operating Procedure Synthesis

The automatic synthesis of operating procedures is another safety related task which makes extensive use of qualitative reasoning. The original work in this area dates back

to 1974 [84] and so pre-dates the wide distribution of AI techniques. However, more recent work has utilised techniques from AI planning and qualitative simulation.

The aim of the synthesis task is to construct a sequence of steps that will safely take a system from its initial state to some desired, final state. Typically, this might involve two types of reasoning:

- Top-down reasoning to obtain a set of candidate operations to consider at a particular point in the sequence so far.
- Bottom-up reasoning to model the effect of these operations on the system in its “current” state.

Both of these require some kind of qualitative reasoning but there are some issues that complicate the direct application of qualitative simulation. For example, planning is usually conducted hierarchically rather than jumping straight in at the greatest level of detail possible. Thus if one is given the task of starting up a plant then one will probably begin by considering the best order in which to perform high level sub-goals such as “start-up feed section” or “start-up reactor section”. Only much later will one then consider very detailed operations such as “open valve v1”. This means that the qualitative techniques employed must be able to reason at different levels of detail during the planning process. There has to be a close relationship between a planning algorithm and the modelling representation that it uses.

This relationship between qualitative simulation and planning has been explored by Drabble [24, 23]. His main aim was to construct a planning system that did not require the “closed world assumption” i.e. that could react to an unexpected event during the execution of a plan. To do this he has constructed a blackboard system, called *Excalibur*, which can monitor the execution of plans and patch them up when they go wrong. *Excalibur* consists of a temporal reasoner, a planner and a qualitative simulation system based on Forbus’ qualitative process theory [30]. Unfortunately it is written in POP11, which has prevented its evaluation within this thesis work.

Fusillo and Powers have written several papers describing their attempts at automatic

operating procedure synthesis [34, 32, 33]. In the second paper they describe the following strategy:

1. Decompose the system (plant section) into smaller sub-systems that can be physically isolated from each other.
2. Enumerate the constraints that the system is subject to.
3. Identify the goals to be satisfied.
4. Solve the goals by a double-backtracking planning routine.

The first two of these sub-problems are performed manually. The system decomposition is for efficiency purposes and is likened to “island driven planning”. To facilitate this the decompositions should result in a system that can be brought close to its desired final state and held there. In addition, the decomposition facilitates modelling since the authors take a lumped-parameter approach and use a STRIPS like functional model. For example, they use a single value for the temperature of the fluid in a recycle loop and model the direct effect on this value as more heat is put into the loop. Thus they avoid the complication of modelling different temperatures at different parts of the loop. However, this is at the expense of rather poor qualitative modelling.

The constraints divide into the following forms:

- Global constraints that apply throughout the plant.
- Local constraints that apply to individual pieces of equipment.

The former are typically constraints on streams, such as forbidding certain combinations of chemicals in order to prevent explosions, whilst the latter are more specific to a piece of equipment, often to prevent equipment damage. In the later paper [33] they further sub-divide local constraints into:

- Pre-conditions on the equipment concerned.

- Exogenous constraints that prevent damage to another piece of equipment and are derived by constraint propagation.

Again the constraint enumerations are all manual, although constraint propagation is a classic qualitative simulation technique. The operating goals are constructed by a means ends analysis to determine the difference between the initial state and the final, desired state. They can then be satisfied by the selection and ordering of operators to perform the necessary increase, decrease or other change. Constraints are checked by a “simulation” after each operator selection, which determines the suitability of the choice. The backtracking algorithm should allow for all combinations and orders of operators to be attempted but no comments are made upon the efficiency of this.

One of the main weaknesses of this method must be the use of the STRIPS model for the qualitative representation. This is amply demonstrated by Lakshmanan and Stephanopoulos, who describe a non-linear planning methodology for operating procedure synthesis [57, 58]. In part one of the paper they review some of the previous planning work, particularly with regard to the modelling approaches used. They point out the deficiencies of the STRIPS approach when modelling distributed, network structures such as chemical plants. In particular, they point to the dependencies between different parts of a network, which make it very difficult to model a local effect without accounting for the state of the rest of the system. For example, one must make the outcome from opening a valve dependent upon whether or not there is fluid in the attached pipe and a pressure drop across it. Unfortunately the STRIPS model doesn't do this but instead tries to reason about influences unconditionally. It is therefore equivalent to the use of rules to represent causality, the limitations of which have already been extensively discussed in Chapter 2.

Lakshmanan and Stephanopoulos describe their own planning strategy in part two of the paper [58]. They then describe a case study which requires the determination of a sequence of valve actions to regenerate the catalyst in a reactor. The non-linear planner performs automatic constraint propagation but can only manipulate constraints that are temporal in nature i.e. that enforce one operation to be performed before another. The paper demonstrates how some of the operational constraints can be directly put

into this form by the exploitation of domain specific features. In addition, they describe an additional planning phase which takes the partial plan from the non-linear planner and resolves remaining constraints by a systematic, linear, generate and test strategy. The papers describe various algorithms for manipulating constraints and demonstrate that they are provably correct.

O'Shima [74] and Foulkes et. al [31] also describe computer-aids for operating procedure synthesis. However, rather than using a rich knowledge representation and applying techniques from AI planning these authors simply rely upon Prolog type clauses to represent connectivity and other boolean information. Really they are addressing themselves to simpler problems that reduce to path-finding, such as finding all the valves that need to be opened in order to create a flow path between two points. Such approaches lack the modelling power needed to properly represent and check plant constraints but could nevertheless provide useful tools for plant operation.

Similar criticisms might be applied to the work of Tomita et. al. [103, 104] whose qualitative reasoning is also rather impoverished. In particular, constraints can only be placed on individual units and can only be phrased as pre-conditions that must be satisfied before the unit is operated. Indeed, a constraint must be phrased in terms of the flow status of the lines into a unit i.e. whether or not they are open for flow. This would seem to make it virtually impossible to model temporal constraints, such as "purge with nitrogen before admitting hydrogen"! Actually the system is not quite as limited as this because there is a general purpose inference engine which can be used to implement ad-hoc reasoning. One suspects that for a real problem the majority of the reasoning will be done in this manner.

3.6.3 Fault-tree synthesis

So far this chapter has described a computerised, bottom-up technique, called qualitative simulation, in some detail. This section will now give a brief description of a computerised, top-down approach to representing process plant behaviour. This is shown to have the potential to avoid the combinatorial problems associated with qualitative simulation, though current approaches do not necessarily manage this. Un-

fortunately, however, the inferencing power of fault-tree synthesis methods is shown to be limited by its poor semantics and its inadequate representation of the loss scenarios envisaged. The second section discusses this and suggests which concepts from qualitative simulation might be used to advantage.

3.6.3.1 The tree versus the structure approach

There appear to be two main approaches to the representation of process plant behaviour for the purposes of automatic fault-tree synthesis. One of these can be called the *tree approach*, which has been used by Martin-Solis et al [71] and by Taylor [102]. The other is a *structure approach*, which has been used by Shafaghi et al [89] and by Lapp and Powers [60]. This section will discuss the relative merits of each of these approaches, with a view to assessing whether or not the best top-down method can overcome the problems encountered in qualitative simulation.

The tree approach is somewhat akin to De Kleer's method in the sense that it is device-centred. In contrast, the second approach involves 'stepping back', examining the overall process structure and applying algorithms that are specific to types of substructures within the whole. This will be explained in more detail below.

In the simplest device-centred approach one would just go from device to device 'tracing' the causes of the top event. It is therefore analogous to the procedure used in section 3.3 to trace through the simplest kind of simulation problem. Such tracing is unconditional and so this method only works if the same relationship between input and output *always* applies. Unfortunately, as was demonstrated earlier, simple tracing strategies easily fail when there are feedback or feedforward loops between process variables. To solve this problem one must make the relationships between inputs and outputs conditional upon other events, such as the deviations of other input variables or the state of the device involved. Within fault-tree synthesis this is done by the use of "mini-fault-trees" which represent the behaviours of the individual device. An example of this is given in figure 3.14(b) which represents part of the behaviour of the control valve shown in figure 3.14(a) (derived from Martin-Solis [70]). Most of the relationships on this tree are unconditional, so they can be expressed using simple OR gates. However, control

loop failure can only be represented using conditional events such as $Zin(\text{no-change})$, for which an AND gate is needed.

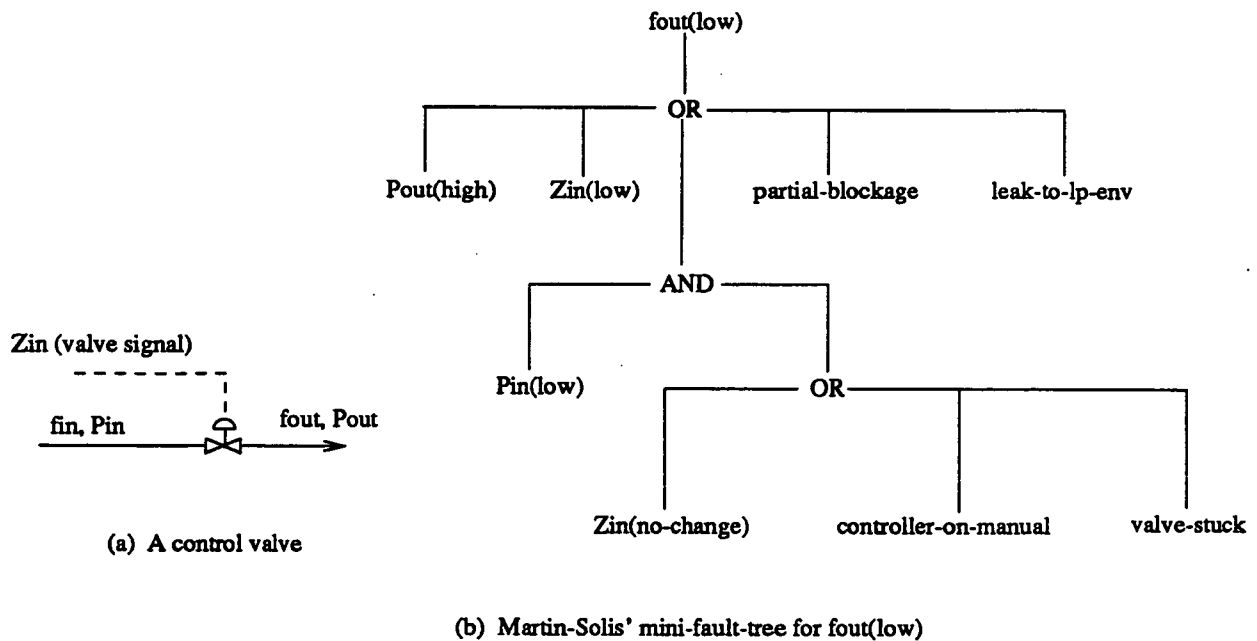


Figure 3.14: A Mini-fault-tree for a control valve

Programs taking this approach usually have a library of mini-fault-tree templates. These can then be filled in and connected to construct the fault-tree for the entire system. In principle the algorithm is very simple as it just involves joining up templates in a pre-specified manner. However, one does have to look out for events that are repetitions of, or inconsistent with, their ancestors.

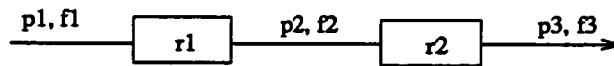
In contrast, the structure approach is quite complicated and is particularly aimed at dealing with control or trip loops. These always defeat the unconditional tracing strategy and so merit special attention. Typically a fairly complex algorithm will be used for each loop of a certain type. This presents problems of trying to get the algorithm correct, about which there has been much discussion (e.g. see [59, 69]). Moreover if loops are joined or nested, as occurs in cascaded control systems, then one has to decompose the system into separate loops and decide which loops and which

algorithms dominate. Nevertheless, an approach based upon flowsheet structure seems liable to give a clearer understanding of where ambiguities arise in the analysis and to give a better chance of their resolution. In some ways this use of structure is similar to the idea of automatically including extra confluences in a qualitative simulation. In particular, in both cases one is making wider statements about the flowsheet as a whole. To explore such techniques one can make useful reference to the more general A.I. problems of abstraction and decomposition.

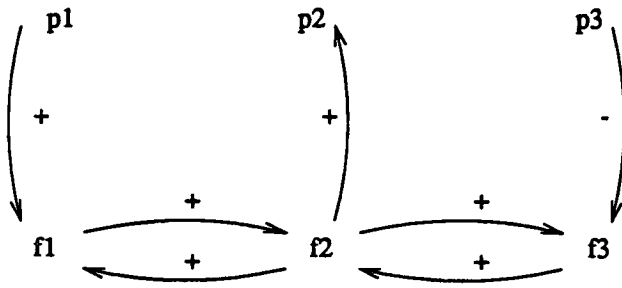
Typically an approach based upon flowsheet structure will represent the process interactions by means of a signed, directed graph with conditional edges. The nodes of such a graph can represent continuous process variables (e.g. $P1$), 'events' concerning continuous process variables (e.g. $f1 > 0$) or discontinuous events such as 'pump failure'. In the first case a node value of zero will usually mean "normal", while positive or negative values mean the variable is "high" or "low" respectively. In the latter two cases the nodes can usually only take either the value 0, which means event is not occurring or the value 1 which means that it is. There is therefore poor distinction between different kinds of events, which can make such a representation confusing and difficult to handle. Additionally, one must have a suitable procedure for assigning nodes to loops. All of the published work seems to have required this to be done by hand.

When the graph and loops are represented one can then apply loop based algorithms to process the digraph and construct the tree. This is more likely to be successful if the number of loops on the digraph is not very large. Unfortunately this is not the case when representing pressure/flow relationships by a digraph.

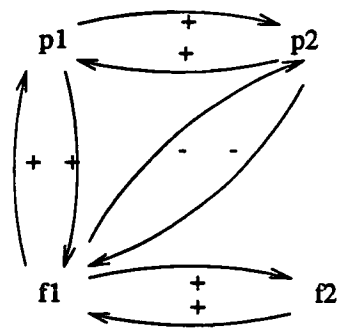
Consider representing the behaviour of two simple resistance elements in series, as shown in figure 3.15(a). To do this one needs to create a basic building block to represent a resistance element on its own and then join two of these together. This is unfortunate because the best digraph to describe the behaviour of this system is shown in figure 3.15(b) but this cannot be constructed by the union of two identical digraphs. Indeed, one probably needs to consult the confluence that represents a resistance element to obtain the digraph in figure 3.15(c) as the basic building block for the system. Consequently, a digraph for a flow network will contain a great number of large and small loops and be difficult to process. In contrast, the tree representation for pres-



(a) Two resistances in series



(b) The digraph that best represents them



(c) The minimum digraph segment to represent a resistance

Figure 3.15: Digraph representation of two resistances in series

sure/flow behaviour would use mini-trees similar to that in figure 3.14(b) and hence be fairly easy to process.

Thus the tree approach seems to offer the best potential for modelling pressure/flow behaviour because the structure approach may be overwhelmed by the large number of loops in the digraph. However, the tree approach runs into serious problems with more spread out loops because of the 'localised' nature of its analysis. To see this, refer back to the conditional event on the example mini-fault-tree in figure 3.14(b). The validity of this AND gate relies upon certain assumptions about the loop behaviour that are built into the mini-tree events. In particular, one is assuming that if Z_{in} doesn't change then the controller has failed but if Z_{in} does change (at all) then the controller is working perfectly and will compensate fully for the event $P_{in}(low)$. The

representation is therefore really rather rigid but appears to be flexible because such assumptions are not made explicit.

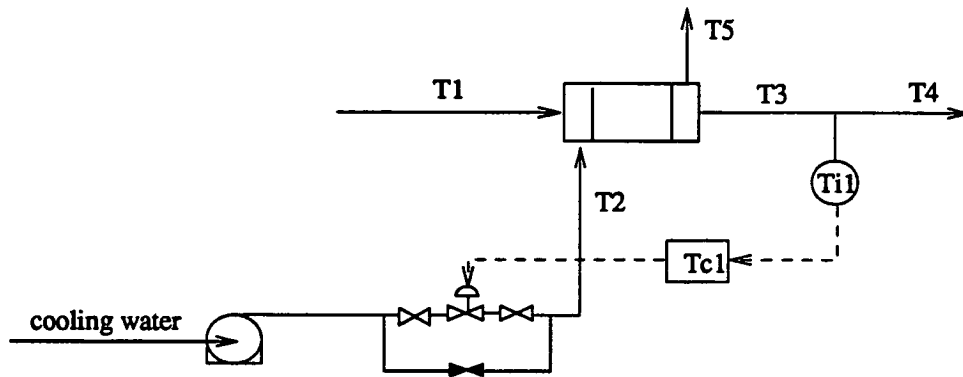


Figure 3.16: The Lapp & Powers example without the trip-loop

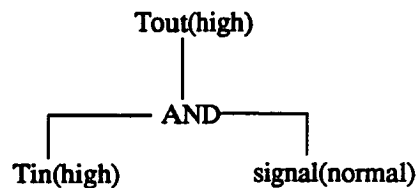


Figure 3.17: A Possible Mini-tree for a temperature sensor

As a further example of this refer to Figure 3.16 which is based on the now classic Lapp & Powers [60] fault-tree synthesis problem. Consider how to use a tree approach to represent the causes of $T4(\text{high})$ by applying mini-trees for each unit on the flowsheet. Whilst doing this, note that if one traces high temperature from $T4$ back to $T1$ then the only control item that is met on the way is the temperature indicator, $Ti1$. This means that the only possibility one has for capturing the behaviour of the temperature control loop, $Tc1$, is via a mini-tree such as the one shown in Figure 3.17. Unfortunately, this figure assumes that if a signal is sent to the controller then it will fully correct for the

disturbance i.e. one cannot model controller failure in this case. Furthermore, if one constructed a tree for T3(high) instead of T4(high) then one wouldn't even meet the temperature indicator before tracing the cause back to T1. This is despite the fact that T3(high) is also implicitly controlled by the loop. Martin-Solis' algorithm runs into this problem but he fails to recognise it. Instead, he just presents T1(high) as an unconditional cause of T4(high) [70]. One might then ask what the control loop was for!

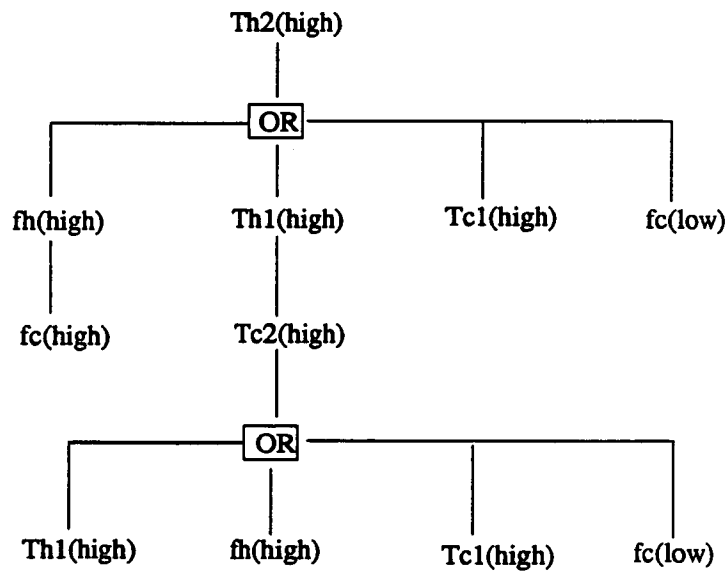


Figure 3.18: Naive top-down analysis of the problem in Figure 3.3

For yet another example of the problems with the tree approach refer back to the example in figure 3.3, for which a simplistic tree might be as in figure 3.18. The problem here is that fc(high) and fc(low) both appear on the tree as causes of the top event. Is this consistent? Does this mean that either of these will cause the event or does it mean that exactly one of them will but one doesn't know which? Such a tree is ambiguous and therefore difficult to interpret. How can this ambiguity be removed using the tree approach? The answer would have to be by writing the events on the mini-fault-tree for the heat-exchanger as conditionals involving all the process variables. For instance, one knows that the relationship between fh(high) and the top

event Th2(high) can be cancelled by deviations in the other process variables. One might therefore try to replace the event "fh(high)" with the conditional event:

fh(high) AND Tc1(normal) AND fc(normal) AND Th1(normal)

Unfortunately this is not quite the same as with the control valve example because one has no reason to assume that the deviations in the various process variables will be of the right magnitudes to cancel each other out. In addition, if every event is made conditional then one will end up by concluding that there are no possible causes of the top event because none of the deviations can exist in isolation. Clearly this is erroneous reasoning and limits the application of the tree approach in this case. Additionally, one should note that if every event is made conditional on the values of the other process variables for the unit concerned, then one shall be approaching a complexity that rivals that experienced in qualitative simulation!

In such an example it is probably best to explicitly recognise the existence of a 'natural', negative feedforward loop between 'fc' and 'Th2'. This is essentially what the structure approach would do so clearly there are advantages to both approaches. Indeed, the best method might be a hybrid that reserves the use of trees for representing the pressure/flow relationship. On first appearance it seems that such an approach could escape some of the worst combinatorial problems of qualitative simulation. However, it also seems that the inferencing powers of the usual top-down techniques are weakened by their failure to distinguish between different types of events and to have a clear notion of device states and the information that these provide. The next section will consider this and try to suggest where aspects of qualitative simulation could be of use.

3.6.3.2 Applying principles from qualitative simulation

Some features of the work on automatic fault-tree synthesis become evident if one considers it in the light of the work on qualitative simulation. Among these are:

1. There is no sharp distinction between 'events' that represent changes in continuous process variables (e.g. $\text{del}(p1)=+$), those that represent value ranges of

those variables (e.g. $\text{fin} > 0$) and those that represent device states or device state transitions (e.g. "pump stopped" or "pump stops").

2. All variables must be expressed in the same number of discrete levels, such as $(-1, 0, +1)$, rather than being described with respect to sets of landmark values or the values of other process variables.
3. There is poor definition of whether a scenario is being treated in a quasi-steady-state or a qualitatively dynamic manner. The treatment of sequential behaviour is, in general, fairly poor.
4. Constraints are only used for generating possible descendants in the tree, rather than for additional layers of inferencing to prove or disprove some of these. This is partly because the constraints are usually only represented as the conditional edges of a signed directed graph.

In the De Kleer approach difference variables were allowed to take on the values $(+, 0, -)$ and the process variables to be described by reference to their landmark values. This is possible because there is a clear separation of difference variables and 'actual' variables. In contrast, as noted in (1) above, there is no such separation in the current algorithms for automatic fault-tree synthesis. This means that process variables can only have the same set of possible values as the difference variables i.e. (2) is a direct consequence of (1). Similarly (4) is also a consequence of (1) because the lack of distinction between 'events' concerned with device states and other types of event leads to a poor definition of the system state that is being envisaged. A result of this is that one cannot conduct proofs by the use of device state information, such as the "infacts" used in the qualitative simulation program described in previous sections. Consequently, the fault-tree synthesis algorithms may generate some events that could, in fact, be proven not to be causes of the top event. This may be acceptable in fault diagnosis and unquantified fault-tree analysis, although it does multiply the number of hypotheses to be considered. In the case of quantified fault-tree analysis, however, it will lead to an erroneous calculation of probabilities.

The need for events that involve the actual 'values' of process variables, rather than just their directions of change, is shown by referring back to the discussion at the end of

section 3.5.2. This concerned the simulation of composition propagation in the binary split in figure 3.11. Suppose now that one wishes to find the *causes* of $\text{del}(x3)=+$ in this split. As was the case before, one must consider the signs of the flows when tackling this problem. For example, for the flows as shown in figure 3.11 the only possible cause of $\text{del}(x3)=+$ is $\text{del}(x1)=+$. However, if the sign of $f2$ is reversed then $x3$ becomes a mean concentration and therefore dependent upon 'f1', 'x1', 'f2' and 'x2'. The result of such dependencies is that the propagation in the split must be represented by a rather complicated tree such as the one in figure 3.19. This involves events that define the values of 'f1' etc. so it appears that fault-tree synthesis also needs to be able to represent process variables by reference to landmark values. In addition, one needs device state information to decide under what conditions such values arise. With the present approaches to automatic fault-tree synthesis none of this is formalised which leads to complexity and confusion. For example, one doesn't know which value of 'P1' an 'event' such as "P1(high)" refers to. A more formal approach is probably necessary to deal properly with examples such as the one discussed here. Clearly, even top-down approaches to the qualitative analysis of process plant behaviour cannot completely escape the complexity of high dimensionality problems.

Andow [3] has already drawn attention to the generally poor treatment of sequential behaviour in fault-tree analysis and pointed out some consistency checking difficulties with allowing time lags on fault trees. These are problematical but must be tackled because one often needs to represent the sequential behaviour of 'sharp' dynamic processes, such as trip activation. Andow suggests the use of cause-consequence diagrams, rather than fault-trees, for the expression of time delays. Another possible method may be via the cause-effect graph used by Umeda et al [107] which should adequately capture the distinction between fast and slow processes. A further alternative might be to refer to some of the temporal reasoning work performed in A.I. (e.g. [36, 1]). Such work provides a notation for describing sequences of events using primitive relations between intervals of time (e.g. "during", "before" etc.). As with the qualitative simulation work already discussed such work is expected to be useful because it provides a representation that is both concise and formalised.

Both the top-down and the bottom-up approaches need real explanation facilities that allow the analyst to comprehend predicted process behaviour properly. This problem is

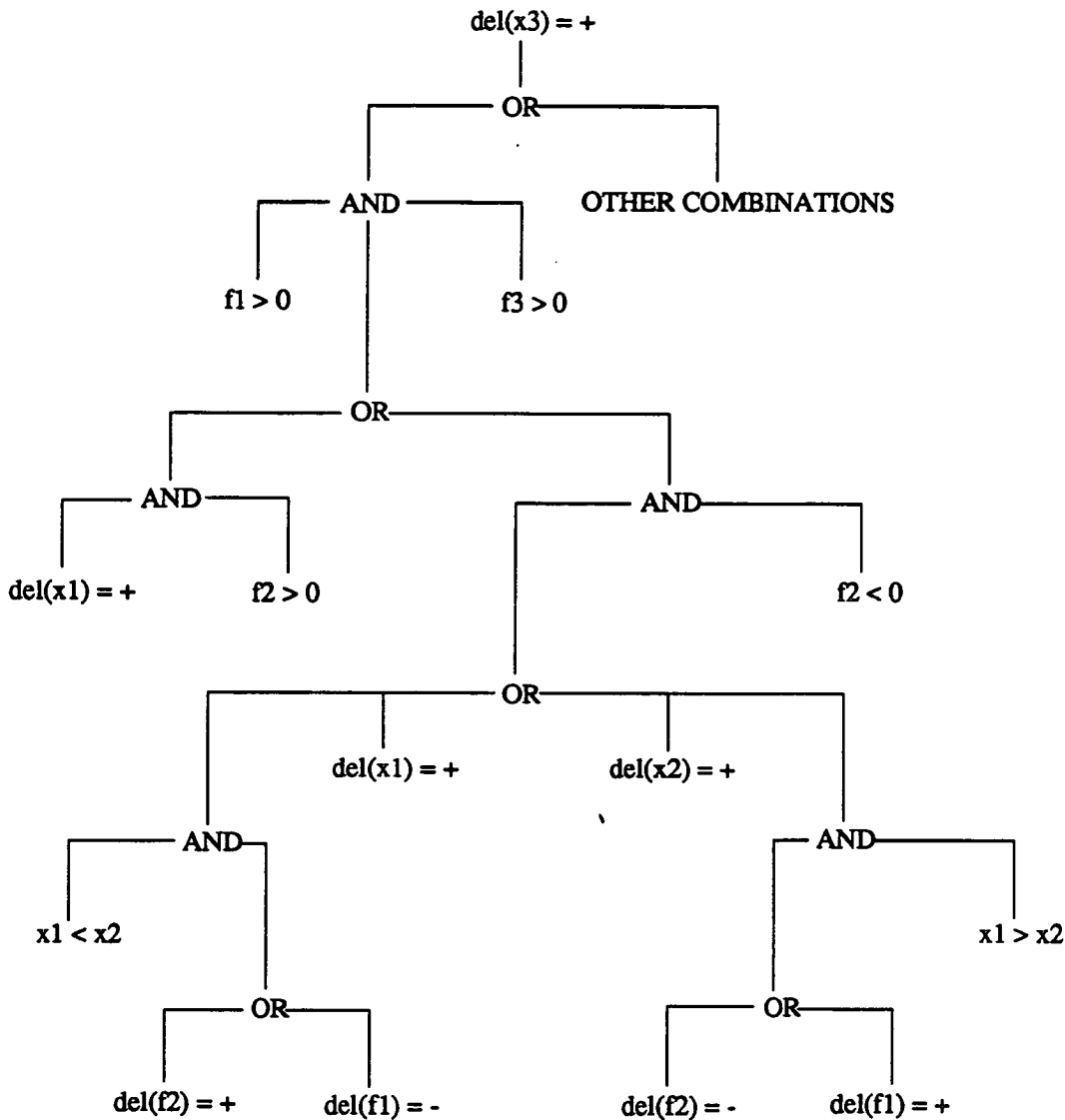


Figure 3.19: Partial top-down analysis of the split in Figure 3.11

not solved by the rule displaying and rule tracing that is possible with solely rule-based Expert Systems. As this section has tried to demonstrate, a rule-based (i.e. tree-based) approach will only work if the mini-fault-trees use implicit assumptions about the behaviour of the rest of the process. Consequently, a display of such a rule is likely to be rather cryptic. Also one should note that people understand complex processes



on several different levels, most of which involve examining the overall process structure rather than taking a device-centred viewpoint. Rule tracing is unlikely to aid this type of comprehension.

Finally, both approaches encounter problems when one tries to pick the event or scenario of interest i.e. the top event in a fault-tree or the initial system state and boundary conditions for a qualitative simulation. Picking such events or scenarios requires the integration of the analysis into the wider task of design. As Chapter 1 has already shown, this is one of the major themes of this thesis.

3.7 Summary and Conclusions

This chapter has discussed the similarities and differences between top-down and bottom-up methods of hazard identification. To do this it has described in some detail a bottom-up technique called qualitative simulation and a top-down one called 'automatic fault-tree synthesis'. The main emphasis was on the former, as this is a novel AI technique not much explored in chemical engineering. Compared with Chapter 2 the central concept is now *constraint satisfaction*; indeed, causality is only used to help with the efficiency of this process. This allows one to resolve many of the feedback problems identified earlier. In addition, qualitative simulation provides a well formulated representation, which makes it easier to reason about limiting values and relative magnitudes. Unfortunately, severe efficiency problems were found with the technique.

The chapter began by discussing a range of flowsheet situations that one may be interested in modelling. It was shown that some of these examples can be adequately modelled in a 'quasi-steady-state' manner, but for others it is necessary to take proper account of the actual history of a transition. In addition, there was a discussion of one example where it is not possible obtain a unique solution without recourse to numerical arguments. These distinctions were seen to be useful in the next two sections, which discussed the previous work of two sets of AI researchers. The first was the work of De Kleer and Brown [20], which is quasi-steady-state, and the second was that of Kuipers [55], which is more truly dynamic. For the work described in this thesis it was

chosen to extend the method of De Kleer and Brown, by providing a means of reasoning about binary relational facts. This significantly increased the inferencing power of the approach but turned out to be an extremely costly improvement. In addition, some serious problems were identified, such as how to represent composition in a binary split. These were felt to severely limit the applicability of qualitative simulation in any of its current forms.

This is unfortunate, since the later sections of the chapter demonstrate the central importance of qualitative representation to a number of loss prevention problems. For example, the work on automatic operating procedure synthesis has identified the close relationship between modelling and planning. This leads to the conclusion that one cannot separate a planning algorithm from the modelling representation that it uses. Similarly, fault diagnosis is now using qualitative simulation techniques to enhance the robustness of the diagnostic rules; in particular, by making them less brittle to changes in flowsheet structure. Finally, fault-tree synthesis relies upon qualitative knowledge to define the connections between the discretised events on the fault-tree.

The remaining sections of the chapter described some representational differences between qualitative simulation and the techniques used for automatic fault-tree synthesis. Most of these seemed to stem from the failure of the latter to distinguish between 'events' that represent changes in continuous process variables (e.g. $\text{del}(p1) = +$), those that represent value ranges of those variables (e.g. $\text{Fin} > 0$) and those that represent device states or device state transitions (e.g. "pump stopped" or "pump stops"). It was felt that qualitative simulation could provide some valuable input here but the form it might take was not discussed in detail.

In conclusion then, qualitative reasoning is a vital area for many of the problems associated with loss prevention but current techniques of qualitative simulation suffer from both severe efficiency problems and inference limitations. More work is therefore required, particularly that which also considers 'semi-quantitative' techniques. However, it is important that further development of qualitative reasoning is conducted in parallel with that of the problem solving systems that depend upon it. The construction of monolithic qualitative reasoning programs is not necessarily the best way to proceed.

This concludes Part I of this thesis, which has been concerned with the qualitative representation of process plant behaviour. Part II now moves away from this area to consider the wider problem of Loss Prevention within integrated design. In addition, it addresses some different representational problems, such as the use of abstraction and refinement in hierarchical design. Chapter 1 has already mapped out in some detail the scope and purpose of this work but the first section of Part II will provide a further brief introduction.

Part II

Constraint Tools



**"Mr. Osborne, may I be excused?
My brain is full."**

The essence of engineering, as a crusty veteran once told his engineering rookies, is to be only as complicated as you have to be. What he left unsaid, though not undemonstrated, is that you must also be as able to get as complicated as the problem demands.

D.E. Gushee, Ind. Eng. Chem.

Chapter 4

Managing Constraints within Design

4.1 Introduction to Part II

At present, chemical plant design is typically split into partially independent sub-tasks and tackled using a host of stand-alone computer packages. In recent years, however, the need has been recognised for more integrated process design, which uses a data-based approach. This has resulted in a body of research into the uses of Data-bases and Data-base Management Systems (DBMS's) for process plant design. Ultimately, the aim is for a *complete design environment* that provides a data-base for all aspects of design, provides the tools necessary to advance the design and provides significant help on what to do next.

The most important feature of such an ideal environment is that it must allow the timely, *and parallel*, consideration of most aspects of design, in order to allow the early identification and correction of design errors. A particularly important example of this is the need to check the validity and satisfaction of *designer's intention* at all stages of design. In particular one must check the following:

- The designer's reasoning is sound i.e. what he intends a particular system to do makes sense in the context of the rest of the design problem.
- The designer's decisions are valid i.e. his intention can be appropriately realised by the system suggested.
- Neither of the above are invalidated by later changes to design i.e. his intention continues to remain sensible and his decisions valid.

As an example, consider the simple problem of deciding to fit pressure relief to a vessel and of choosing the size of the relief valve subsequently used. These decisions are strongly influenced by the duty and design of the pressure vessel concerned. Consequently, it is imperative that they be reconsidered whenever these factors change. Such a change could easily happen in the course of an ongoing design project. Furthermore, new sources of overpressurisation may arise, which were not accounted for in the original decisions. One must therefore keep track of the dependencies between the different parts of the design.

Currently, designer's intention is only formally checked towards the end of design, when the *Hazard and Operability* (HAZOP) Study takes place. However, one really needs to do such checking continually so that problems are identified much earlier. Part II explores this problem by describing the use of tools from Artificial Intelligence to represent and check designer's intention, in the form of constraints, throughout the design process. In particular, it describes an attempt to use an AI toolkit called Knowledge Craft as a Data-base/DBMS for design knowledge. To this end, the author has explored several ways of representing design information within Knowledge Craft and has created some tools that allow the easy description and checking of constraints. The result is a domain independent set of tools which are collectively called The Constraint Tools System. These extend Knowledge Craft by providing some completely new modules, as well as by providing some additional user-interface commands.

The current chapter will outline further what is meant by the term 'constraint' and introduce some of the representation issues involved. Chapters 5 & 6 then describe the Constraint Tools System in detail. Both of these chapters are concerned with how to represent process engineering knowledge within the system. However, Chapter 5

presents an abstract overview of the tools provided whereas Chapter 6 runs through an extensive set of examples. Neither are intended to be a software manual, however, as a separate document has been written for that purpose [112]. Further background to this part of thesis work can be seen in a book chapter [110], which was published in early 1990. This should not be taken as a description of the Constraint Tools System, however, as the software has developed considerably since the article was first written in the summer of 1988.

4.2 Different Uses of the term ‘Constraint’

A successful design must satisfy certain constraints which define its intended function. These are constraints upon normal operation. In complex situations their enumeration may require negotiation between the parties interested in the final product and the cost of making it. This part of the design problem has been called “Constraining discourse” by Bucciarelli [11], as already described in Chapter 1. However, in simpler cases the design constraints can be easily enumerated and perhaps used to automatically derive acceptable design solutions. This is the assumption behind attempts to use AI-planning for design synthesis tasks, particularly for sub-problems such as space planning [27]. To some extent it is also the assumption behind an optimisation approach to synthesis, if this requires pre-enumeration of all the constraints.

Unfortunately, the need for discourse and argument makes pre-enumeration of all constraints rather difficult. In addition, a design must also satisfy constraints upon what could possibly happen during probable deviations from normal operation. This second type of constraint is even less well defined than the first because it involves a predictive element. Typically a series of ‘What happens if’ questions must be applied to the design in a systematic manner in order for potential hazards to be identified and noted. There is therefore a need both to be able to represent the potential hazards, perhaps in the form of constraints, and to be able to perform some qualitative or quantitative simulation to show how an initiating event could lead to one of them.

Thus one can see that there are several possible meanings of the word *constraint*. Obvi-

ously the term always refers to some sort of limitation on a model or a design. However, the form and use of that limitation varies greatly between different approaches. Here are some examples:

Numerical methods use mathematical constraints (equations) in *generative mode*.

That is, to generate output information from input information. Typically the solution method will involve iteration, which requires that the problem be well formulated.

Optimisation methods use equations, intermingled with the use of integer variables.

Constraints are used in two ways [88]:

Hard constraints define allowable solutions, so they must be satisfied. Often these are expressed as inequalities, which define allowable regions for the optimisation variables.

Soft constraints have to be approximately satisfied but some trade-offs are allowed. This is achieved by weighting the constraints and adding them into the objective function. This is an appealing viewpoint but it does require that the constraints be expressible in numerical terms so that one can measure how well they are satisfied.

Constraint propagation methods use symbolic or mathematical constraints in generative mode. Typically the solution involves symbolic substitution of 'known' information to produce a restricted constraint. When such a constraint only involves one variable then it defines new 'known' information. If an unambiguous solution cannot be directly found then various relaxation procedures may be applied. For example, in a problem using a discretised representation one might just guess the 'value' of one of the variables. This new value could then be 'propagated' until one of the following occur:

1. All the constraints are satisfied (it was a valid guess).
2. One of the constraints is violated (it was not a valid guess).
3. Another impasse is reached and further relaxation (guessing) is needed.

Constraint propagation methods are used in AI-planning and qualitative simulation.

Database integrity methods use *semantic integrity constraints* to prevent inconsistencies within a database [10]. These constraints are hardly ever used in generative mode but rather just as checks which return “yes” or “no” answers, according to whether or not the constraint is satisfied. However, their usage can be quite powerful if applied at the right time. In particular, most databases can undo any effects from a transaction that does not succeed completely. Consequently, constraints applied at this stage might prevent inconsistencies at source.

Previous chapters have described work on qualitative simulation, which uses constraint propagation methods to manipulate a qualitative representation. This work identified numerous problems with these techniques, particularly from an efficiency point of view. The work to be described in the second part of this thesis has therefore avoided using constraints in this way. In addition, it has avoided taking the optimisation or numerical view of constraints, due to a basic mismatch between their formalisms and the non-numerical information that one needs for Loss Prevention. Instead, the current work has concentrated on the use of constraints to check the integrity of the design database or to implement a design check-list. At first sight this seems to be very similar to the database viewpoint. However, important differences will be demonstrated when one considers how constraints are checked and what action results from a constraint violation. Furthermore, the constraints that this work has been concerned with require a much deeper knowledge representation than the ‘type checking’ constraints usually represented in a database. Such issues will be discussed below but also see Brodie et al [10] for a discussion of the differences between *databases* and *knowledge bases*. In addition, Appendix C presents some examples of the sorts of Loss Prevention constraints that one needs to represent.

4.3 Classifying Constraints by their Use and Purpose

The section above has shown that a constraint can be classified by its purpose, its syntax and its usage. This results in distinctions between using constraints for numerical methods, for constraint propagation or just as procedural checks upon knowledge or data. These differences have already been described but another way of representing

them is with a hierarchy, such as that shown in Figure 4.1. This presents three new classifications, which can be further described as follows:

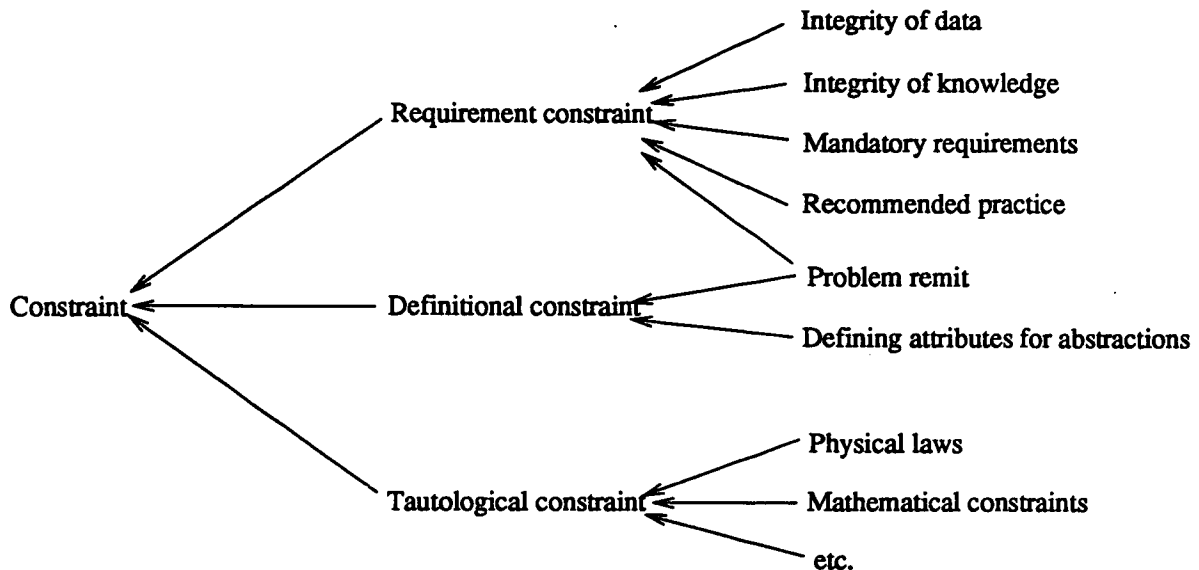


Figure 4.1: A Possible Classification of different constraint types

Tautological constraints must always be satisfied for any physically realisable system. Usually they are used in generative mode but occasionally one may check such a constraint because a violation signifies an error of reasoning or abstraction. This is particularly useful in fault diagnosis.

Requirement constraints are usually too complex to be used in generative mode. Indeed, it is difficult to provide full expressive power at the same time as constraint propagation. The latter requires that the constraint syntax be limited enough to allow its manipulation and ‘comprehension’. This makes it difficult to provide ‘browsing’ facilities that can roam over the whole data set. Levesque and Brachman discuss this fundamental trade-off between expressive power and inferencing capability [64]. In particular, they show that languages as expressive as First Order Predicate Logic (FOPL) are not *decidable*. That is, one cannot guarantee being able to prove a sentence of the language. Furthermore, even if one trivialises FOPL by removing all quantifiers

the decision problem is still believed to be computationally intractable. This presents a serious problem for AI languages as it implies the need for limitations on expressive power. For example, one must usually make a *closed world assumption*, thus allowing sentences to be 'proved' by complete enumeration. This paper will be further discussed in a later section.

Definitional constraints lie partway between the two other types. Sometimes one may use them in generative mode, such as is done in simple planning problems, but sometimes one may just use them as checks upon reasoning or the design so far. For example, the defining attributes of an ammonia plant are that it has a net production of ammonia. This is fairly simple as it could be checked by a comparison of the inputs and outputs to the proposed plant. It could also be used to synthesize the design by a means-ends analysis. In contrast, the defining attributes of a 'safe ammonia plant' are much more complex and unlikely to be of direct use in design synthesis. One could, however, still consider the use of fault-tree synthesis to generate simpler constraints. In this case the definitional or requirement constraint becomes associated with the top event in the fault-tree. This is an intriguing possibility but it will run into problems which are similar to those that arise with constraint propagation. For example, the constraint syntax would have to be limited in a similar manner. This avenue has not been seriously explored in this thesis work. It will, however, be further discussed in the sections below.

4.4 Further Classifying Requirement Constraints

The sections above have mentioned that part II of this thesis is concerned with the use of constraints as *procedural checks* upon data or knowledge. This subsection will now consider different ways of classifying constraints of this type. Firstly, however, it is worth discussing the purpose of this exercise. There are at least two views on this:

1. One can classify constraints for control purposes. That is, in order to control their behaviour and use, in a manner appropriate to their purpose and structure. Some of the issues involved are:

- (a) How does one know when a constraint might need checking?
- (b) To what level of detail should one check the constraint?
- (c) How 'expensive' is it to check the constraint?
- (d) How serious is a violation of the constraint?
- (e) Should the constraint be checked automatically?
- (f) What action should be taken if the constraint is violated? For example, should a warning be issued, should action be taken automatically or should the user be forced to take rectifying action?

The tools produced in this work allow control over the answers to most of these questions. However, only one pre-defined constraint class is provided. The user or developer is therefore left to experiment with his own domain.

2. One might classify constraints for representation purposes. For example, one might find that constraints of the same type need the same sort of knowledge or representation. At the highest level this may determine the syntax and primitives of the constraint language. Such possibilities have already been addressed during the discussion of different uses of the term 'constraint'. At a lower level, however, the classification may just determine common utilities between constraints of a similar type. For example, one may find the need for general graph searching tools.

This thesis work has certainly provided common utilities for constraint representation and checking, such as the Propagating-relation system described later. In addition, some of these could be partitioned so that they behave differently for constraints of different types. However, the partitioning has really been used to enhance the general flexibility of the utilities themselves, rather than specifically for the classification of constraints. Knowledge Craft encourages one to do this because the system itself is programmed in a modular and flexible way. This work has clearly demonstrated the benefits of this approach by the integration of separately developed modules in unexpected ways. In software engineering this is called a 'bottom-up' approach to software development.

Some of the constraints that one will need to represent will be fairly simple, others will be more complex. Below are some possible distinctions:

1. Constraints which involve a single attribute of a single object. For example:
 - The temperature in pipe-3 must be greater than -10 Degrees C..
2. Constraints which describe a forbidden combination of values of different attributes of a single object. For example:
 - Don't allow fuel flow to a burner while the air flow is zero.
3. Constraints which involve the comparison, perhaps arithmetic, of values of different attributes of a single object. For example:
 - The operating pressure of any piece of equipment must be less than its upper design pressure.
4. Constraints which compare values of attributes from different objects. For example:
 - The upper design pressure of a vessel must be greater than the set-point of any relief valve which relieves it.
5. Constraints which are the inverse of ones that have already been attached. For example, the inverse of that in (4) is:
 - The set-point of a relief valve must be less than the upper design pressure of any vessel which it relieves.

Of these, constraints of types 4 & 5 are the most complex because they require one to first identify the multiple objects involved before one can compare attribute values to check the constraint. How this is to be done strongly affects the software that is needed for a constraint system. The following are several alternatives:

1. Attach a constraint directly to a prototypical object (schema); from now on this will be called the *home-schema* for the constraint. The constraint then applies to other objects which are suitably related to this one. For example, one might use a generic "tank" object as the home-schema for constraints that apply to any type of tank. For a constraint that refers to more than one schema, such as the fourth example above, it will probably be necessary to create more than one version of the constraint and attach them to different prototypes. This is so that the data

can be found from either end, which is necessary if constraints are to be checked automatically.

2. Don't attach a constraint directly to a prototypical object but instead apply it through pattern matching. One must then do this pattern matching whenever a constraint may be in danger of violation.

When choosing between these one comes back to the question of how the choice affects the constraint representation. This is particularly important with respect to pattern matching as one will probably be restricted to using rule systems such as CRL-OPS or Prolog. One must then ask whether the rule system has to be in control, how brittle it is to changes and how much control it gives one over how and when constraints are checked. Some of these questions will be expanded upon below but the Constraint Tools Manual [112] gives a more complete discussion from a Knowledge Craft point of view. In particular, it justifies the rejection of the pattern matching approach, in favour of attaching a constraint directly to a home-schema.

This approach works well for constraints that are obviously 'local' to a particular piece of equipment but may not work so well for constraints that are more 'global' to the whole design. For a chemical plant design one can partly solve this problem by attaching constraints to streams. For example, one might be able to prevent explosive mixtures by simply checking the composition of all streams, rather than having to consider vessels and so on. However, some constraints will have to apply across whole subsystems and thus be difficult to attach to any single ELD item. One must therefore consider representing larger sections of plant as objects in their own right. More general constraints can then be attached to these more general systems. Unfortunately, for this to work one must also be able to map or propagate data that is shared between the different levels of detail. These problems have already been discussed in Chapter 1 but some solutions will be discussed below.

If constraints are attached at different levels of detail then one may consider linking them together by some sort of 'refinement' relation. In other words, one can recognise that some of the lower constraints, taken together, ensure the satisfaction of some of the higher ones. For example, a constraint such as "Don't overpressurise a vessel" is

completely satisfied by constraints that require the vessel to be relieved and constraints that ensure the relief system can always function. One might then consider completely replacing the original constraint with the set of more detailed ones. This is not really necessary, however, since the less detailed constraint is merely redundant, rather than inconsistent.

This is an important point because it distinguishes the multi-level reasoning used for refinement constraints from that needed for tasks such as mathematical modelling. In the latter case, less detailed models will usually be approximations that are actually inconsistent with the set of more detailed models. Thus one cannot simply get the whole lot and solve them together but instead must filter out the levels that are redundant. In contrast, less detailed requirement constraints should merely have a larger range of satisfaction than more detailed ones i.e. they should still be satisfied when the more detailed ones are. Thus one merely sacrifices efficiency by the application of redundant constraints. This even applies to constraints that have already been used at a previous design stage.

A further point with respect to refinement is whether one can use it to construct a formal method for constraint enumeration, particularly from the point of view of Loss Prevention. One might consider doing this by using a fault-tree representation, such as the example in Figure 4.2. This starts at an undesirable 'event' called overpressurisation and expands upon how that event could occur. One can easily turn the top event into a constraint by inverting it (i.e. "Don't allow overpressurisation"). Similarly, one could turn lower events into constraints but it may be difficult to decide whether or not to combine them. For example, does one construct constraints which *separately* prevent the build-up of explosive mixtures and the existence of a source of ignition or does one construct a single constraint which only prevents the combination of such factors. There is no clear cut answer to this as it depends upon how well constraints can be used and checked, how much redundancy one wants to build in and so on.

The author's viewpoint is that engineering judgement must be used to interpret legal, institutional and company policy in the context of the design concerned. This makes it difficult to pre-enumerate a set of constraints (i.e. a check-list) that will apply in every situation, unless it is rather general. For example, most people would accept a

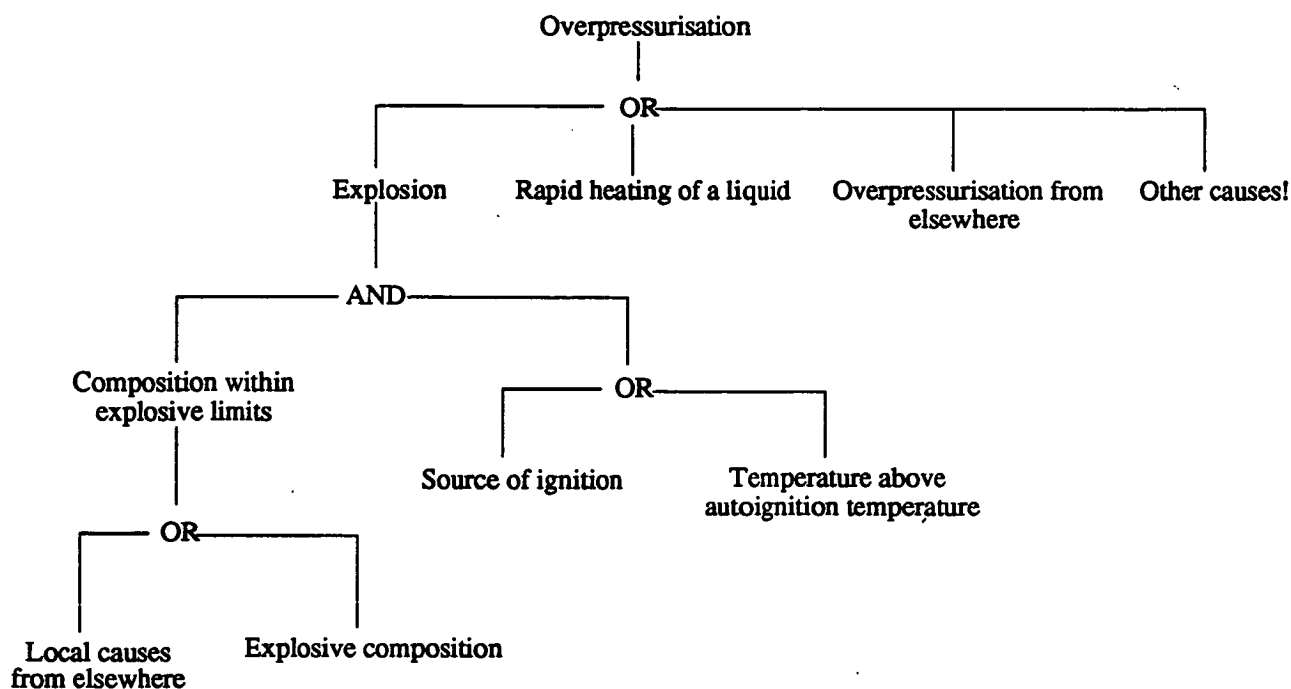


Figure 4.2: An example fault-tree describing vessel overpressurisation

check-list that refers to constraints such as “prevent overpressurisation” but they would not all agree on whether to satisfy this by relief methods or by designing a vessel for all foreseeable pressures. This thesis work accounts for such disagreements by allowing constraints to be easily turned on and off, as well as providing other flexibility. Thus it has tried to provide an automated check-list which is easily specialised for a particular company, type of design or design team.

A final way of considering constraints is in terms of the types of data that they refer to. In particular, one may want to distinguish between the following types:

1. Data that can be changed as a result of operator action and may therefore be changed as a result of simulating operator action e.g. temperatures, pressures, the running status of a pump (“running” or “stopped”).

2. Data that is fixed by design and so cannot change during a simulation e.g. design limits, pipe sizes or equipment connections.

This distinction can be used to decide how often data is likely to change, which may affect how one checks the constraint involved.

4.5 Representation & Implementation Issues

The sections above have already discussed some of the issues that affect how constraints are represented and implemented. For example, they considered whether to attach constraints directly to objects or to use a pattern matching approach for their application. Such discussions were considering a mixture of *representation* issues, which are concerned with what one is trying to say about the world, and *implementation* issues, which are concerned with how one says it. Workers in the knowledge representation area try to separate these two types into *knowledge level* issues and *symbol level* issues respectively. In contrast, this thesis takes a more pragmatic approach by addressing representation issues from the point of view of the specific software that has been used. Hopefully the comments will generalise to software of similar types. However, to extend the experience outside this narrow scope will require the services of someone with a more theoretical AI/Computer Science background. This research has been very much 'example based'. Nevertheless, this section will try to give an informal theoretical justification for some of the implementation decisions taken in this work. In addition, it sets out some ground rules for what constitutes a good representation.

Levesque and Brachman [64] have quoted Brian Smith [91] as defining the following *Knowledge Representation Hypothesis*:

Any mechanically embodied intelligent process will be comprised of structural ingredients that (a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and (b) independent of such external semantical attribution, play

a formal but causal and essential role in engendering the behaviour that manifests that knowledge.

This is a useful starting point for considering the distinctions between database programming, 'conventional' programming and knowledge based programming. In particular, it places knowledge based systems as lying partway between database systems, in which the data structures only mean something to the external user and 'conventional' programs, in which the data structures often only mean something to the program. In the first case the data is all important but the system cannot itself reason about it. In the second, only the program outputs are of importance, rather than the data transformations performed along the way. Obviously this distinction is becoming blurred, as procedural languages become more structured (e.g. by using C++) and databases more flexible. However, AI languages still seem to be the best tools for addressing the middle ground.

The power of AI languages comes from their ability to represent a wide variety of knowledge in a natural, and declarative, manner. Why is this important?

Firstly, a close match between a representation mechanism and the natural structure of a domain makes it easier to set-up and maintain a knowledge based system¹. This is not to say, however, that AI languages have solved the representation problem. Many forms of knowledge are still difficult to represent, such as the assumptions and goals in a complex design problem.

Secondly, by representing knowledge in a declarative manner one can potentially make it available for tasks other than that for which it was originally intended. This is obviously important for problems as large as the construction of design environments, where it will be almost impossible to pre-enumerate all the different forms in which information will be needed.

Levesque and Brachman give an excellent example of this second point by considering

¹Anyone with doubts about this should compare writing a simple fault-tree synthesis algorithm in FORTRAN, using integers to represent events and stack handling to simulate recursion, with writing the same algorithm in a naturally recursive and symbolic language such as Prolog!

a program which prints out the colour of a given item. In the first implementation the program uses strings which embed the answers within them, e.g. "the colour of snow is white". This does the job intended but the knowledge that snow is white is not available in a form that could be used for any other task. In contrast, the second version maintains explicit facts which match an item to its colour. The color is then inserted into a string template to return the answer as a sentence. Thus the factual information is separated from the user-interface and could be used in other applications. Perhaps one is tempted to say that any decent programmer would take the latter approach. However, by their nature AI languages are modular and encourage one to make knowledge as declarative as possible. This is particularly true of frame-based systems.

The astute reader may be confused at this point because he has remembered that this thesis work uses constraints as *procedural checks* upon data or knowledge. This doesn't sound very declarative. Indeed it's not, for good computational reasons that will be outlined below. However, although the constraints themselves are procedural most of the knowledge they refer to is stored declaratively. Furthermore, one is given full control over the balance between these methods. That is, over how much knowledge one tries to represent declaratively. Unfortunately, this is at the expense of having to think in procedural terms about how knowledge will be retrieved and when constraints will be checked.

Levesque and Brachman take First Order Predicate Logic (FOPL) as the benchmark for a fully expressive language. In doing this they note that its power comes not so much from what can be said *but from what can be left unsaid*. For example, one can say which facts aren't true without having to enumerate all the facts which are. However, they also discuss the computational intractability of complete FOPL theorem provers, as has already been mentioned. They then go on to show how databases and other systems get around these problems by limiting what one is able to say.

To see this, suppose that one is given the following fact in a design situation:

The feed to the plant must be either chlorine or a chloromethane.

One common way of dealing with this *disjunction* would be to explore two alternative “worlds” in which the options are separately true. Alternatively, one might just do the design and later check that this constraint is satisfied. However, neither of these alternatives is using the fact as a whole i.e. as a theorem to produce other facts. Indeed, most computationally tractable systems, including Prolog, do not allow the use of disjunctions except in a procedural manner. The same applies for many other expressive forms, such as negations or universal quantifications. Thus expressive power is limited by the need for tractability.

One can now see that the constraints used in this thesis work are procedural by necessity. This makes it difficult to be clear about their ‘semantics’ i.e. about what they say about the real world. In addition, it is left to a system developer to ensure that data is mapped as required and that representational gaps are plugged. Many tools are provided to facilitate this, particularly for the problem of automatically propagating data between shared locations. Unfortunately it still remains a serious problem, as will be seen from later examples.

In order to further address this in later chapters it is worth establishing some ground rules for what constitutes a good representation. Obviously the concerns already raised must be satisfied. That is, a representation must be tractable and expressive enough for the use that is being made of it. However, even if one can satisfy these ‘theoretical’ limitations there are still some pragmatic issues to be addressed:

1. One must consider how hard it is to describe the relevant knowledge using the representation. For example, how many ‘sentences’ does one have to write for a given piece of knowledge, how complex are they, can the syntax be checked, does one get graphical help and so on. This may be thought of as the end-users viewpoint; although what is meant by this depends upon the task concerned and a company’s organisational structure (i.e. who is allowed to change what knowledge).
2. One must consider how hard it is to change a representation once it has been partially completed. For example, if one builds the same mapping information into many different constraints and rules then it will be tedious and complex to

change the mapping. In contrast, if one can keep the mapping explicit then only a few changes should have to be made. In addition, the mapping may also be available for other tasks. This work has addressed this problem at length, which has resulted in several different ways of traversing intermediate objects to obtain required data. This problem can be thought of as the developers viewpoint.

3. One must give the developer or user some help with seeing how well a partial representation ties together. This is particularly a problem if many of the representational tools are procedural, such as are rules and constraints. In addition, one must give him some help with assessing the effect of a representational change. For example, if an item is renamed can one find which constraints and rules referred to it? This is the weakest part of the Constraint Tools System as it stands, mainly because Knowledge Craft does not force or encourage you in this direction.

To some extent there may be a trade-off between these aims. One must then decide who is the most important customer. This is a real problem for the AI-toolkits as they seem to provide enhanced development tools at the expense of end-user friendliness. In particular, the fact that they are transparent and intended for rapid prototyping encourages the developer to enhance the original interface. The user is therefore exposed to the complexity of the development software. Some further comments on this will be made in the next chapter.

4.6 What Constraint Tools Do We Need?

The above sections have already addressed many of the issues associated with how constraints are represented, used and controlled. Some more will be discussed in later chapters, when the Constraint Tools System is described in detail. In the meanwhile, however, it is worth clarifying some of the features that the system is intended to provide:

1. Constraints must be easy to see and cross-reference. For example, one should be able to ask to see "all the constraints concerned with pumps". This may have a bearing upon how one chooses to store and represent the constraints.

2. Constraints should be easy to write and use so that the “automatic safety check-list” is easily specialised to the needs of a particular design team.
3. The constraint languages must be flexible enough to say complex things.
4. Some constraints should be checked automatically, others only on request. It should be possible to enable and disable automatic checking at any level of detail. The types of things that one might want to demand are “Automatically check all constraints on tanks” or “Switch off automatic checking of any constraints on tank-1”.
5. One should have control over many of the system features, such as what action is taken when a constraint is violated.

The author believes that he has demonstrated the applicability of AI-Toolkits to the provision of these features. In particular, the use of objects and inheritance makes control and partitioning rather easy. However, other serious problems have been identified, such as the difficulty of maintaining a partly procedural knowledge base. In addition, later chapters demonstrate other complications that arise from some of the features listed above. For example, setting up automatic checking is conceptually simple but causes some difficulty when more than one object is involved in a constraint.

4.7 Summary

This chapter began by discussing the different uses that are made of the term ‘constraint’. The distinctions were primarily concerned with the way in which a constraint is used. In particular, distinctions were drawn between the use of constraints for numerical simulation, optimisation, constraint satisfaction or just as procedural checks upon a data or knowledge base. This thesis work was shown to be concerned with constraints of the last type, despite the fact that they cannot be used in *generative mode*. This was justified by a discussion of the fundamental trade-off between expressiveness and efficiency, as identified by Levesque and Brachman [64]. In addition, there was a discussion of some other pragmatic issues that should be addressed when comparing two forms of knowledge representation. The chapter concluded by listing the features

that the Constraint Tools System is intended to provide. The way it achieves these features will now be described in some detail, together with how the software relates to the existing tools provided by Knowledge Craft.

The perfect computer has been developed. You just feed in your problems, and they never come out again.

Al Goodman

Chapter 5

An Overview of Knowledge Craft and The Constraint Tools System

5.1 Introduction to Knowledge Craft

This chapter gives an introduction to the Knowledge Craft features that have been found to be useful and an overview of the extensions that have been made. Knowledge Craft is an AI-toolkit marketed by Carnegie Group Inc. in Pittsburgh. It consists of several core modules which allow the integration of the following programming paradigms:

- The frame-based definition and manipulation of knowledge, using a set of LISP functions which are collectively called CRL.
- Forward chaining, rule-based programming using a version of OPS5.
- Backward chaining, rule-based programming using a version of Prolog.

In addition, it provides graphical workbenches to access these features and a *Command System* to create new graphical interfaces of one's own.

Knowledge Craft can be run on several platforms, such as SUN workstations or Symbolics LISP machines, which means that it is possible to construct applications such that they are portable between several machines. It has been written in Common Lisp and the LISP windowing system that a particular platform provides. Thus it is essentially an extended LISP environment. However, software written in conventional languages, such as C or FORTRAN, can be integrated in two ways:

1. Load the program directly into LISP if the particular LISP provides this facility. For example, SUN Common LISP provides an interface to C and FORTRAN programs.
2. Use the operating system to run the package externally and talk to it through files or pseudo files, such as UNIX pipes.

Unfortunately, both of these will reduce the portability of an application.

Knowledge Craft is a fairly expensive tool to buy, though a large academic discount is available. In addition, a large workstation with substantial disk space is required, which adds to the cost. This thesis work has been conducted using Knowledge Craft 3.2, running under SUN Common LISP 2.1.1, on a SUN 3/60 with 16 Megabytes of real memory and 60 Megabytes of swap space. Knowledge Craft was chosen for this project because of the academic discount and because the author was able to conduct an extended trial of the product, prior to purchase¹. Comparable products are Inference ART² and KEE³ but Knowledge Craft appears to have been the right choice. ART has several advantages in the way it manipulates contexts but seems to be more geared towards rule-based reasoning than is desired. KEE is more like Knowledge Craft but seems to be more closely integrated. This should make it easier to pick up at first, but less flexible once experience is gained. In addition, KEE does not support user-defined relations in the manner that Knowledge Craft does. However, these are only

¹Researchers sponsored by The Science and Engineering Research Council (SERC) can conduct extended trials of AI software at The Artificial Intelligence Applications Institute, Edinburgh

²"Inference ART" is a registered trademark of Inference Corporation, Los Angeles. "ART" stands for Automated Reasoning Tool.

³"KEE" is a registered trademark of IntelliCorp Inc., MountainView, California. "KEE" stands for Knowledge Engineering Environment.

first acquaintance observations since it has not been possible to conduct a detailed comparison of ART, KEE and Knowledge Craft within this work. Some Chemical Engineering applications using KEE will be described in the next section. There do not appear to have been any using ART.

Knowledge Craft is split into modules, which can be purchased and loaded separately if desired. Most modules are completely accessible either by the use of a graphical workbench or by the use of LISP commands. This provides multi-level access since a novice user can be guided through the features available while a more experienced user can write LISP commands to do many things at once. In addition, the Command System is very transparent, which makes it possible to alter existing workbenches or copy parts of one for another application. Indeed, the Command System is one of Knowledge Craft's strongest features as it encourages programming in a very modular and re-usable way. This tends to result in a set of tools which are only loosely integrated and require the user to type-in schema names to tie things together. This approach is clumsy for the end-user but greatly benefits the system developer by encouraging the rapid production of code that is transparent and easily maintained.

The Constraint Tools System makes heavy use of Knowledge Craft's *CRL* module and its Command System. In addition, this work has considered the use of *CRL-OPS* for maintaining simple data dependencies. Some additional features have therefore been provided to allow the use of *CRL-OPS* rules in conjunction with the new Knowledge Craft extensions. However, the system does not actively use such rules itself. Nor does it use *CRL-PROLOG*; since initial trials suggested that it was extremely inefficient, both for development and during execution.

CRL provides a set of LISP functions to describe and manipulate schemata (also known as objects or frames). A schema is a data area which has access to a set of slots, each of which have access to a set of values. A typical use of a schema is to represent a physical object, such as a pump. One would then use a slot to represent an attribute of that object (e.g. its impeller diameter); and a value to represent the attribute value (e.g. 30cm). One can, however, use schemata to represent more abstract objects such as plant subsystems, chemical data, graphical objects or operator actions. Indeed, it is also useful to represent abstract programming entities as schemata since one can

then use inheritance to obtain the necessary information. This approach has been taken within Knowledge Craft itself and continued within this thesis application. For example, later sections describe a new module, called The Constraint System, which is almost entirely controlled by schemata. This makes it easy to group together similar constraints and so control them by editing a single parent object. In addition, built-in schema editors make it easy to change this control information.

One can program using schemata in two different ways:

1. Schemata can be directly edited using LISP commands or graphical workbenches. This is the usual way of using an AI-toolkit.
2. One can program by sending *messages* directly between objects, instead of using an external LISP function to exercise control.

In computer science and AI the second of these is called *Object Oriented Programming* (OOP) but there is no specific term for the first technique. This thesis therefore uses the terms more loosely. In particular, the terms Object Oriented Programming and *Frame-centred Programming* should be taken to refer to any sort of programming that uses objects and *inheritance*, rather than just to the second of these two approaches.

5.2 Other Work Using AI-toolkits

This section will review some other Chemical Engineering work that makes use of AI-toolkits or OOP. Whilst doing this it will make some initial comments on the use of frames and inheritance but the reader is referred to Jackson [45] for a more general introduction. In addition, more detail on Knowledge Craft can be found in [47] and [13]; and on LISP in [95] & [116].

Stephanopoulos et. al. [98] have constructed Design-kit as a process engineering design environment. Curiously, they have used KEE for the frame-based and rule-based parts of the system but used the Symbolics Common LISP Flavours system to construct

graphical workbenches. Unfortunately, it is not clear whether this is for efficiency reasons, or because KEE's graphics facilities are inadequate, or just because of the past history of the project.

The cited paper gives a good introduction to the use of OOP within Process Engineering. For example, they discuss the use of multiple inheritance, which allows an object to inherit information from more than one 'parent'. In particular, they show how it can be used to modularise knowledge while still allowing for the creation of unusual or composite objects. For example, instead of having an object called "adiabatic reactor" one can inherit the properties of such objects from a "reactor" object and an "adiabatic" object. This prevents one having to duplicate "adiabatic" information within different composite objects, such as an "adiabatic CSTR", an "adiabatic PFR" or an "adiabatic compressor". Unfortunately, it does mean that one might inherit conflicting information.

Design-kit uses "internal logic" to protect against inconsistent descriptions. This seems to be at the level of objects, rather than slots or values, and is done when objects are defined. For example, they prevent an object from being linked to both the "adiabatic" and the "external heating" objects since these are based on inconsistent assumptions. However, they do not appear to do further reasoning when equations are inherited so the composite objects must inherit a consistent set.

A similar multiple inheritance approach has been taken within the process engineering examples within this thesis work. In particular, it has adopted the idea of separating topology from function, since it is common for systems with the same function to have slightly different topologies. This is a useful distinction because topology often defines mathematical relationships, which must be satisfied, whereas function only defines designer's intention, which it is physically possible to violate. For example, for any plant item that has *Single Input Single Output* (SISO) topology one knows that at steady state, mass flow in equals mass flow out. This is regardless of what the item is actually intended to do.

Design-kit is said to be an "experimental apparatus" which allows the development and testing of new methodologies for a whole range of process engineering areas. However,

within the paper the authors actually demonstrate its use for the following tasks:

- Graphical construction of process flow diagrams.
- Generation of an equation set to compute a given output.
- Degrees of freedom analysis and graphical selection of design variables.
- Newton-Raphson solution of the equation set.
- Order of Magnitude Reasoning (see section 3.6) above.

Presumably they are planning to bring in other work from Stephanopoulos' group at a later date.

KEE has also been used by Hofmeister et. al. [40] for the construction of a batch simulation and scheduling system called Batchkit. They are running KEE, and its co-product Simkit, on a symbolics machine. This is expected to provide them with an environment that is flexible enough to formulate a large range of batch problems and to easily implement new algorithms for their solution. The paper discusses some of these but does not go into knowledge representation in any great detail.

Yet another KEE application is that of Suokas et. al. [101], who have tried to construct an expert system to support HAZOP analysis. This work has already been discussed in Section 2.2.

Knowledge Craft has been used by Lien [66] for the construction of a *Blackboard* system called AKORN D. Essentially this is a rule-based system, based on CRL-OPS. However, conflict resolution (deciding what to do next) is done dynamically, in a problem specific manner. This is achieved by the use of knowledge sources to reason about solution strategy, as well as about the problem actually being solved. Thus there is a clear separation of domain and control knowledge.

The paper discusses Blackboard systems in detail but does not say much about how Knowledge Craft is actually used in this application. One can, however, anticipate

large differences from the use of Knowledge Craft within this thesis work since the approaches to design are very different. With AKORN D, Lien appears to be trying to automate some parts of the design process, presumably with little user interaction. In contrast, the loose association of tasks already described means that the Constraint Tools System relies upon the user exercising a large amount of control. Obviously tasks may have error checking to ensure that they are not used with insufficient information, such as could arise from using them at the wrong stage of design. In addition, some tasks could be advice modules that consider the current state of design and recommend the next course of action. Nevertheless, it will still be up to the user to decide what to do next, even if it is just invoking such a module.

Various other workers are using inheritance and OOP for symbolic or semi-numeric tasks. For example, Venkatasubramanian [109] describes the use of object oriented methods in fault diagnosis. Thus one can see that there is a growing appreciation of the power of this approach and can expect many future applications to use this programming style.

5.3 The Extended Knowledge Craft Environment

The result of this project is a set of tools which extend the facilities of Knowledge Craft. These are programmed in a domain independent manner so they could be used by any task for which they appeared to be appropriate. However, they are, of course, aimed at the process engineering design domain, with which this thesis is primarily concerned. Some of the tools are merely extensions to existing Knowledge Craft features, such as additional commands for The Palm Network Editor. Some of these are described in Appendix D. Other tools provide entirely new knowledge representation modules. These will be briefly described below but the reader is referred to the Constraint Tools Manual [112] for a more complete description. Virtually all of the programming has been done by Anthony Waters, the author of this thesis⁴.

⁴However, applications for the source code should be made to Jack Ponton, University of Edinburgh, Department of Chemical Engineering, Kings buildings, West Mains road, Edinburgh, UK.

5.3.1 Knowledge Representation Utilities

Knowledge Craft's CRL module provides core facilities that are needed for the frame-based representation and manipulation of information. These go beyond the methods provided by C++ type OOP environments by providing for symbolic relations, contexts (or Worlds), meta information and so on. Such features will be described in Sections 5.4, 5.5 & 5.6 of this chapter, which deal with the three main components of the Constraint Tools System. This description is important since the reader may not fully appreciate the nature of the extensions unless he has some grasp of the existing methods of knowledge representation. Firstly, however, there will be a description of the way in which a user (or developer) interacts with the Knowledge Craft environment.

5.3.2 Pseudo Multitasking

When one logs on to the extended version of Knowledge Craft the icons along the top of the screen will probably look something like that in Figure 5.1(a)⁵. Note that the row of icons has a scroll bar along the bottom; this is an extension which stops there being any hard limit on the number of icons that can be used.

From left to right, the icons shown in Figure 5.1(a) give access to to the following tasks:

- The Constraint Tools Shell, which simply allows one to exit or pause Knowledge Craft.
- The P-edit schema editor.
- The Fixed-menu Builder.
- The Palm Network Editor.
- The Small Palm workbench.
- The CRL-OPS workbench.

⁵The qualification is because it is a trivial task to move icons around and add new ones, so the basic appearance of the system can be easily customised.

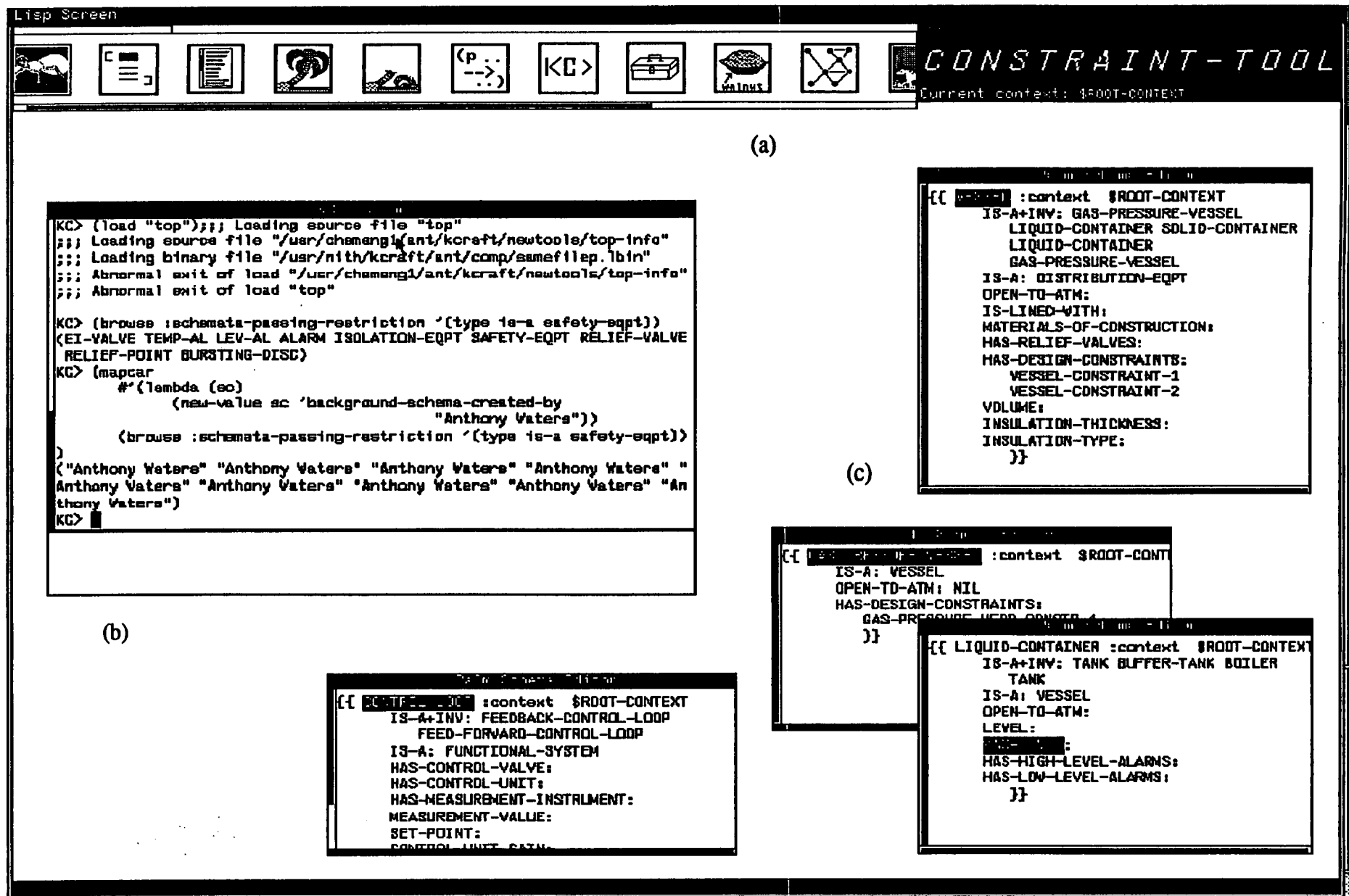


Figure 5.1: (a) Task icons, (b) a KC-listener and (c) some schema edits

- The KC Listener
- The Toolbox, which gives one access to some miscellaneous features, such as the The Context Editor.
- The Walnut Constraint Editor.
- The Refinement Manager workbench.
- The CRL-PROLOG Workcenter

Some of these will be described below, others in Appendix D.

A tool is started by clicking the left mouse button once on its icon. Let us call this action **clicking mouse-l-1**⁶. After that, a further click of mouse-l-1 on this icon will resume the existing task instance (i.e. will cause the existing workbench to come to the top of the screen). One can create a new instance of a task by clicking **mouse-r-1** on its icon and selecting the relevant option from the menu that is popped up. However, not all tasks allow multiple instances to be created. If multiple instances of a task do exist then clicking mouse-l-1 on its icon will resume the one that was most recently used. In addition, if a task instance has any of its viewports showing then clicking mouse-l-1 on that viewport will immediately place the instance at the head of the task queue. It will then be resumed at the first opportunity. Thus one can quickly switch focus between tasks that exist in parallel. This is a sort of 'pseudo' multi-tasking, in that 'jobs' can exist in parallel but one can only switch between them at pre-defined points in their execution.

The multi-tasking environment is provided by Knowledge Craft's Command System. The transparency and modularity advantages of this have already been described above. In addition, one might also mention that it encourages a consistent use of mouse buttons and window commands. For example, mouse-l-1 nearly always selects an icon, mouse-r-1 always gives help or a menu and simultaneously holding down the "meta" key and the letter "r" nearly always reshapes the workbench. These conventions have been

⁶From now on we will say that such an action is to "click mouse-l-1" on the icon concerned. Similarly, to "click mouse-m-1" and to "click mouse-r-1" mean to click the middle and right buttons once (respectively). Also, to "click mouse-r-3" means to click the right mouse button 3 times in quick succession. Similar conventions apply to "mouse-l-2" and so on.

followed in the design of the Constraint Tools System, thus providing a common 'look and feel' to the application.

The Command System allows one to set up command hierarchies and switch focus between tasks that exist in parallel. Each task uses Knowledge Craft graphics to provide mouse and menu access to its commands. The simplest workbench possible is just a prompt viewport, but typically one will have several viewports displaying icons and some displaying pictorial information. Knowledge Craft graphics distinguish between a canvas, a window looking at a portion of the canvas and a viewport which displays the window on a specified part of the screen. This distinction makes it easy to scroll windows, zoom in and out, display windows in several viewports at once and so on. Unfortunately, the graphics are very inefficient, both in terms of the speed of execution and in the amount of LISP 'garbage' that they create. This is a serious disadvantage compared to the graphics provided by state of the art window systems, such as the NeWS toolkit developed by SUN Microsystems.

5.3.3 Existing Graphical Workbenches

This section will describe the graphical workbenches that are automatically supplied with Knowledge Craft. These are all still available under the Constraint Tools System.

The **KC Listener** is a window in which one can type LISP commands. Its most significant feature is that it allows command line editing, using a subset of EMACS commands. Furthermore, it retains a history of the most recent commands, which can then be yanked into a buffer for re-use. A typical instance of a KC Listener is shown in Figure 5.1(b).

P-edit is a schema editor which can be used to create slots, add values, delete values and so on. Typical instances of P-edit are shown in Figure 5.1(c). The P-edit functions can be accessed from a menu or just by single keystroke commands ("s" for create slot and so on). A particularly useful feature of P-edit is that if a slot value refers to a schema then one can mouse select it to edit that schema instead. Furthermore, one can then return to the original schema or go on to another. Thus P-edit provides a simple

way of browsing related schemata and editing them as required.

The **Palm Network Editor** is a powerful tool for displaying and editing schemata hierarchies. Figure 5.2 shows an example instance of Palm, which is editing a hierarchy of process plant objects. The small viewport along the bottom of the Palm workbench is the prompt window where user-input is typed. The small viewport above it is a key to the links shown in the main viewport. In the Palm shown only the is-a link is used, but others could appear on the same diagram. Palm has been used extensively within this thesis work as a basic graphical interface for setting up design problems. Some customisations were necessary, however, as the original tool could only be used for drawing hierarchies, not networks. These customisations are described in Appendix D.

Coconut is a slot filler which prompts the user for slot values one at a time. Thus it is a schema editor, but it cannot be used to create slots or to traverse schemata in the same way as P-edit. However, it is easily called from a developer's own application and provides a simple way of getting data from a user. In particular, it is easy to set checks on the data that the user supplies or to perform some action just before or just after the slot has been filled. Figure 5.8(b) below shows how The Constraint System uses Coconut to edit constraints.

The **Context Editor** provides a graphical display of the context tree and an easy way of creating or destroying contexts. An example tree is shown in Figure 5.11 below. A definition and description of Knowledge Craft contexts is left until this figure is described in Section 5.6.1.

The **CRL-OPS Workcenter** allows the graphical display and editing of CRL-OPS rules. In addition, one can see the current working memory, the current conflict set and a graphical trace of rules as they fire. This seems to be a good interface to OPS but has not been extensively used within this work. However, an example instance of the CRL-OPS Workcenter is shown in Appendix F

The **CRL-PROLOG Workcenter** provides similar facilities to the CRL-OPS Workcenter. Again, it is reckoned to be a good interface to the rule system but has not been much used within this work.

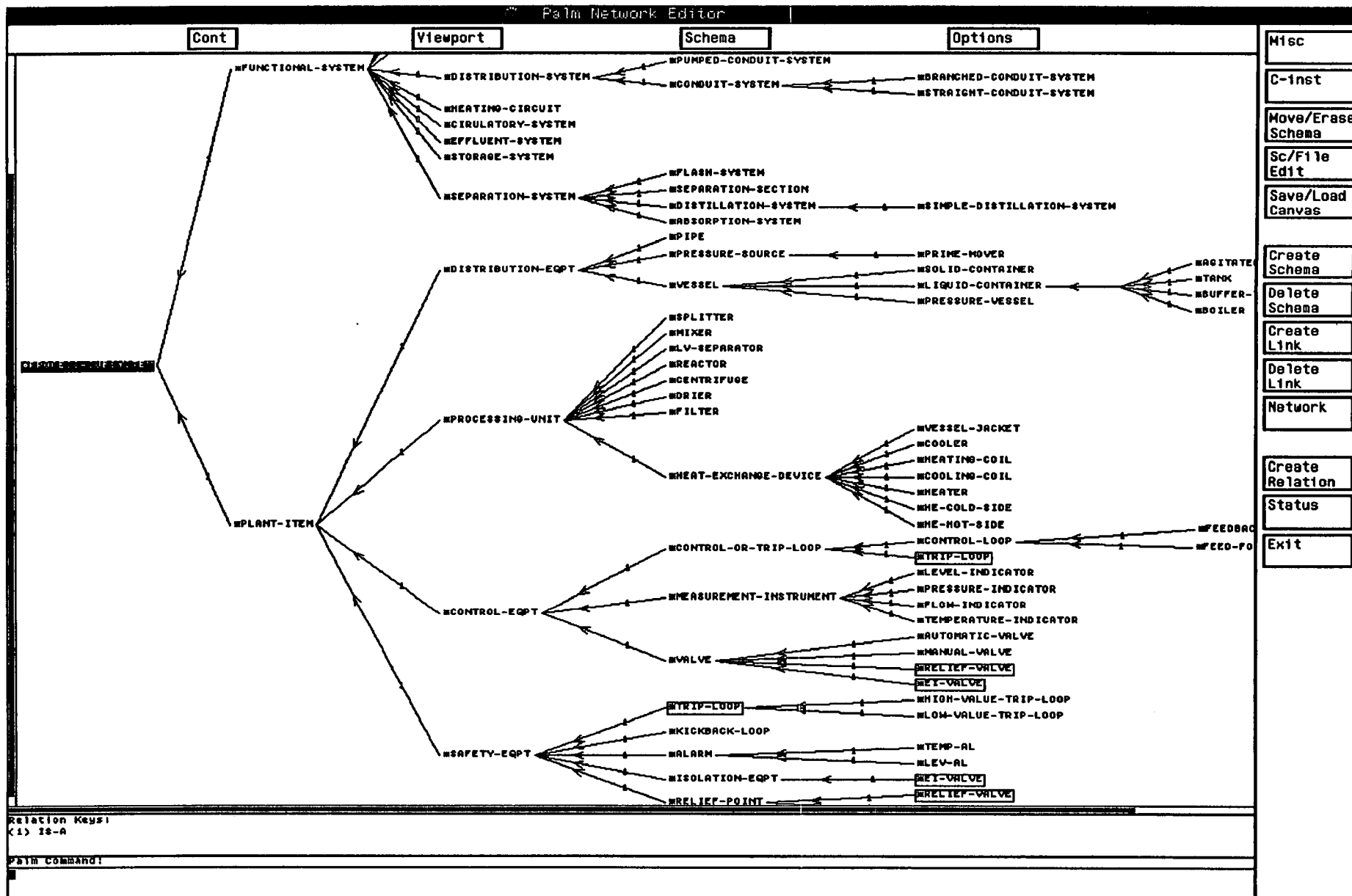


Figure 5.2: The Palm Network Editor

5.4 Tools for Constraint Representation

This section will first describe those of Knowledge Craft's representation utilities which are particularly useful for constraint definition. The next two subsections will then briefly describe the Constraint System, which is one of the major products from this thesis work.

5.4.1 Object Hierarchies and Inheritance

Inheritance is a mechanism which allows an object to access the slots and values of another object. Typically, this is done by forming the objects into hierarchies and allowing objects from lower down to inherit from those higher up. A hierarchy is specified by declaring relationships between the objects involved. In most cases this will just involve the "is-a" and "instance" relations that are built into Knowledge Craft. However, Knowledge Craft allows new relations to be defined by the user or developer, which is one of its major advantages.

The is-a relation is used for relating two object classes; for example, one might say that a class called 'pressure-vessel' is a sub class of a class called 'storage-eqpt' (i.e. "a pressure-vessel is-a storage-eqpt"). In contrast, the instance relation is used to say that an object is an *instance* of a particular class (e.g. "vess-1 is an instance of pressure-vessel"). That is, the 'pressure-vessel' schema represents the generic class of all pressure vessels whereas the 'vess-1' schema represents an actual vessel that must be specified and designed. This should become clear if one considers the small example hierarchy shown in Figure 5.3; and the schemata from it shown in Figure 5.5. These schemata are printed in the format described in Figure 5.4. All schemata will be printed in this format from now on. The hierarchy is a simplified version of that already shown on the Palm display in Figure 5.2.

Inheritance is extremely useful for maintaining consistency and avoiding duplication. For example, one knows that every piece of equipment on a plant must be subject to constraints that prevent operating limits of temperature and pressure being violated.

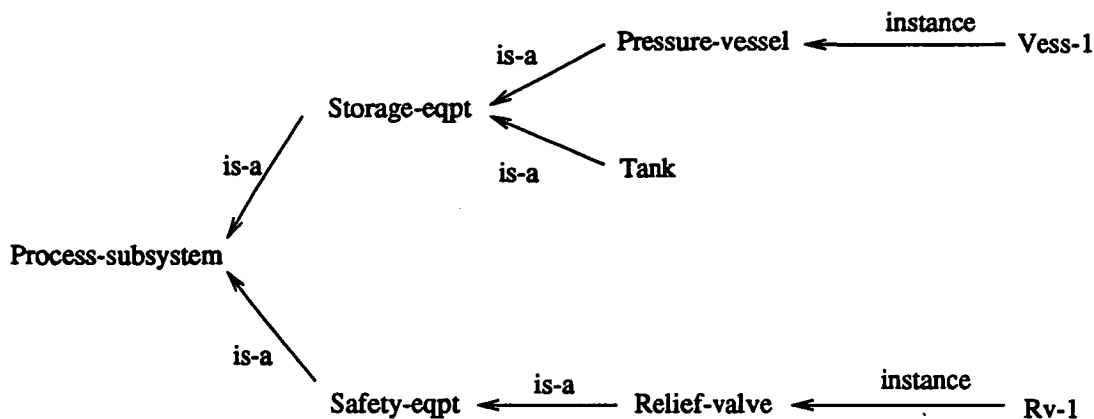


Figure 5.3: A small example hierarchy

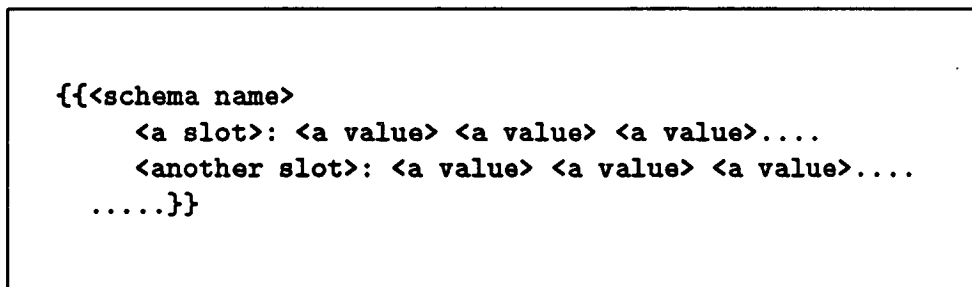


Figure 5.4: The format for printing schemata

It therefore makes sense to store general representations of these constraints near the top of the hierarchy and apply them to all the schemata below. Not only does this lessen data input and storage but it also means that future changes to the constraint representation can be made easily. In contrast, if one wrote new versions of the constraints for each schema then data would be heavily duplicated and changes would require extensive editing.

Knowledge Craft allows the unrestricted use of multiple inheritance, but gives one a little control over which values are returned and used. However, one is not given explicit access to the inheritance algorithm. Consequently, it is not, in general, possible to decide which values will be inherited and in which order. This makes multiple

```
{{process-subsystem
  is-a+inv: storage-eqpt safety-eqpt
  upper-design-temperature:
  lower-design-temperature:
  upper-design-pressure:
  lower-design-pressure:
  has-constraints: temp-ok-constraint
                  press-ok-constraint
  }}

{{storage-eqpt
  is-a: process-subsystem
  is-a+inv: pressure-vessel tank
  has-constraints: check-relief-constraint-1
                  check-relief-constraint-2
  is-relieved-by:
  }}

{{pressure-vessel
  is-a: storage-eqpt
  instance+inv: vess-1
  has-constraints: design-for-total-vacuum
  }}

{{vess-1
  instance: pressure-vessel
  upper-design-temperature: 440
  upper-design-pressure: 10
  }}
```

Figure 5.5: Some example schemata

inheritance fraught with difficulties unless one of the following criteria are obeyed:

- Although schemata exhibit multiple inheritance a particular slot can only be inherited from one place i.e. the information from different sources is orthogonal.
- One will always ask for all the values in the slot concerned and these values will never be in conflict.

The Design-kit system described in a previous section uses a mixture of these criteria. In addition, a depth-first procedure is used to reduce the set of inheritable equations to those relevant to the computation concerned. In contrast, this thesis work uses constraints in a way that obeys the second criteria, since all inheritable constraints can be checked for satisfaction. The only disadvantage of this is that one may check constraints which are redundant because one is also checking more stringent versions. Similar criteria to those above need to be applied to the use of CRL-OPS rules, since the ability to match certain rules may also be affected by multiple inheritance.

5.4.2 Relations

A relation in Knowledge Craft provides one with a way of specifying a connection between two objects and using that connection for inheritance, or other purposes. The most important feature of relations is that they always have an inverse which is user or system defined. For example, the in-built “is-a” relation has the in-built “is-a+inv” relation as its inverse. This means that when one introduces information such as:

“schema-a is-a schema-b”

then Knowledge Craft automatically introduces the inverse information:

“schema-b is-a+inv schema-a”.

This is extremely useful because one can now explore the connectivity from either end i.e. one can easily find all the descendants of a schema as well as all of its ancestors. This can be regarded as a cross-referencing system.

The second important feature of relations is that one can specify their *transitivity* and is provided with a *path grammar* to do so. This is not needed if schemata are directly linked by the relation concerned, as the link will simply be specified by a slot which takes the other schema as its value. However, in many cases the link will be indirect, so one must use a more complicated method to test if two schemata are related in a certain way. This is done by specifying alternative paths that one must be able to follow between the two schemata for them to obey the relation. For example, in the is-a hierarchy one knows that "schema-a is-a schema-b" if one can get from schema-a to schema-b by stepping across any number of is-a links. Similarly, one can define the transitivity of domain specific relations. An example of this is given in Figure 5.6, which defines the relations "is-upstream-of" and "is-downstream-of".

The transitivity of the is-a relation is actually more complicated than as just described, since it allows an optional instance link at the start of the path. Indeed, the path grammar is very flexible and allows one to define extremely esoteric relationships. For example, to look for the possibility of a pump being forced backwards by another pump one may want to see if there is a junction with a pump on each upstream branch. One way of doing this is to start at the junction and see if it has any *relatives* under relations called:

is-a-pump-upstream-of-left-branch-of
is-a-pump-upstream-of-right-branch-of

This is possible within Knowledge Craft because one is provided with a function to obtain all the relatives of a given schema, under a given relation. Most importantly, this function call automatically does loop checking, so there is no danger of the search continuing indefinitely. There is therefore no difficulty if the junction in the example above is part of a recycle loop.

Thus the path grammar gives one an easy way of searching flowsheets and networks for certain structures or information. This can be a powerful tool for describing constraints to protect against undesirable situations, such as that just used as an example. In much the same manner, one could use relations for automatic abstraction and analogical reasoning. In fact, these two applications are extremely similar, the only real

```
{{is-downstream-of
  is-a: relation
  inverse: is-upstream-of
  transitivity: (repeat is-downstream-of 0 inf)
}}
```

```
{{is-upstream-of
  is-a: relation
  inverse: is-downstream-of
  transitivity: (repeat is-upstream-of 0 inf)
}}
```

Example usage:

```
{{u1
  is-downstream-of: u2
}}
```

```
{{u2
  is-upstream-of: u1
}}
```

Figure 5.6: An example of a user-defined relation

difference being whether the abstraction is done explicitly or is implicit in the constraint definition. Chapter 6 will say more about constraints which embed abstractions within them.

The transitivity syntax is quite powerful but rather verbose. In addition, it is only intended for finding the schemata at the end of a path, rather than slots or values within those schemata. As an alternative, therefore, a new grammar has been constructed, which is called the *Path-Spec Grammar*. This is modelled on a UNIX pathname,

complete with the use of wildcards, which makes it very concise. It will be described in a later section.

The final feature of relations is that one can define the inheritance that takes place along them. With the is-a relation virtually every slot can be inherited to the child but none to the parent. One can change this, for one's own relations, to include or exclude extra slots. However, this thesis work has tended to use domain specific relations just to specify connectivity and to use the is-a relation whenever inheritance is required. This is not to say, however, that inheritance is used for every data sharing situation. Indeed, it is not used to map between levels in the refinement tree, for reasons that will be explained in Section 5.5 below.

5.4.3 Query Languages and Mapping Functions

Knowledge Craft provides a second grammar which allows the specification of restrictions upon schemata, slots and values. In isolation it is useful for constraint definition since one is provided with a function for testing a schema against a restriction. However, it can also be used in conjunction with a "browse" function to search the entire knowledge base for schemata passing or failing a given restriction. This thesis work has made extensive use of this facility in the Knowledge Craft extensions to be described shortly. Some examples of using the *restriction* grammar are given in Figure 5.7. From this one can see that it is quite powerful and easy to use, though it does suffer from a lack of pattern matching. In addition, this figure shows the use of a LISP mapping function, in conjunction with a call to the browse function. This allows one to perform the same operation on a whole set of schemata at once.

5.4.4 Access Oriented Programming

Access Oriented Programming is a facility which allows one to attach procedures (demons) to a slot and have the procedures instituted whenever the slot is accessed in a certain way. The simplest form of procedure will just test the values in a slot whenever a new value is inserted. Knowledge Craft provides a special facility for this

- `(type is-a process-subsystem)`
..returns all the objects on the process-subsystem hierarchy.
- `(and (type instance pressure-vessel)
 (not (slot lower-design-pressure (some 0))))`
..returns all pressure vessels not designed for total vacuum
- `(mapcar
 #'(lambda (p) (new-value p 'status 'stopped))
 (browse :schemata-passing-restriction
 '(type instance pump)))`
..Simulates power failure (i.e. all pumps turned off at once).

Figure 5.7: Some examples of using the CRL Restriction Grammar

special case, as well as a more general facility for the attachment of demons. Demons are created as schemata but their active code is written using LISP and CRL. They are very flexible because one can define the type of slot access that should cause them to fire; for example, when a new value is put into the slot, when a value is deleted or when one just asks for the value in the slot. In addition, one can decide whether the demon code is fired before the slot access is carried out or afterwards, so one has a lot of control. Within the Constraint Tools System these features are used to automatically check a constraint when a new value is put into a slot involved in that constraint. They are also used for the automatic propagation of information between design levels.

The following are some of the advantages and disadvantages of using demons:

1. Once they have been set up one can forget about them and leave them to work automatically. This could be confusing, however, if attempted in too arbitrary a fashion. One must beware of circularity since it is entirely legal to write demon code that alters slot values and so fires other demons.
2. One can attach several demons to the same slot so that they all fire when the slot value changes. This allows one to maintain several data dependencies at once and to check several constraints. However, one can have problems in deciding in which order demons should fire.
3. One can create demons which are instances of a generic schema holding the necessary LISP code. This makes it easy to attach demons automatically, and so avoid the tedium of doing it "by hand".

5.4.5 Walnut and The Constraint System

The previous sections have described the Knowledge Craft features that may be useful in the definition and control of constraints. These are all brought together within the Constraint System, which represents a major programming effort. In some sense it is like another rule system, and could perhaps have been implemented using one. However, it gives one a lot more control over the inheritance and behaviour of the "rules" (constraints) than would be possible using CRL-OPS.

The Constraint System provides one with the following functionality:

1. Facilities for the creation, representation, display and editing of constraints.
2. Facilities for the explicit checking of constraints.
3. Facilities for the automatic checking of constraints when the information (slots) involved change. Full control over which constraints are automatically checked for which objects.

TYPE: DESIGN-CONSTRAINT, CONTEXT: PLANT1-BFD1

C/D Options Exeunt Cont Save/Clean C/W Switches

History Window

Checking/ Violation Q's

Undo/Compile Demons

Set Up Demons

Check Constraints

Checking constraints for the schema TOL-FEED
in the context V1-PLANT1-BFD1-1 and the constraints (STREAM-CONST1)
Done...
Setting up automatic checking for the schema TOL-FEED
Done...
in the context PLANT1-BFD1 and the constraints (TOL-FEED-CONST-2 TOL-FEED-CONST1 STREAM-CONST1)
Done...

Fn help/ Misc

Edit/Create Constraints

Translate Constraints

Fts Event

Examine or fire checking queue[Fire]:

Form filler for tol-feed-const-2

```

{{T2851181567+1414 :context PLANT1-BFD1
IS-DESIGN-CONSTRAINT-FOR*: TOL-FEED
DESCRIPTION*:
  "This constraint checks against sulphur impurities
  in the toluene feed stream. These can arise
  from the upstream plant and would poison our
  catalyst."
  
```

Form filler for vessel-constraint-1

```

{{T2851181567+1305 :context $ROOT-CONTEXT
IS-DESIGN-CONSTRAINT-FOR*: VESSEL
DESCRIPTION*:
  "Check that all the relief valves attached to
  a vessel will open before the vessel is
  overpressurised."
EXTERNAL-FORM*:
  "(all-less-than
    #!~|has-relief-valves|*|set-point|*
    #!~|upper-design-pressure|?)"
LOCAL-SLOTS-INVOLVED*: UPPER-DESIGN-PRESSURE
IS-INVERSE-CONSTRAINT-FOR*: RELIEF-VALVE-CONST-1
  }}
  
```

Form filler for stream-reverse-flow-constr

```

{{T2860405553+696 :context $ROOT-CONTEXT
IS-DESIGN-CONSTRAINT-FOR*: STREAM
DESCRIPTION*:
  "
  This constraint checks against reverse flow in
  a stream, as this usually means that an error has occurred.
  EXTERNAL-FORM*: "<>= #!~|total-kmolar-flow|? 0)"
LOCAL-SLOTS-INVOLVED*: TOTAL-KMOLAR-FLOW
IS-INVERSE-CONSTRAINT-FOR*:
  }}
  
```

External form (do not enclose in quotes)[<>= #!~|total-kmolar-flow|? 0]:

Figure 5.8: (a) Walnut and (b) some constraint edits

4. The ability to change behaviour whilst checking constraints (such as turning-off automatic checking, placing checks on a queue for later attention or changing the behaviour when a violation occurs).
5. Graphical facilities for accessing these functions and a save-manager to save the current state of the system to files. Thus one can alter the knowledge base with the graphical editors and reproduce the changes in a subsequent Knowledge Craft session.
6. The ability to partition constraints so that different types of constraint can be checked and managed in different ways.

Following standard Knowledge Craft practice this functionality is provided both as a set of user-callable LISP functions and as a graphical workbench which sits on top of them. Thus there is both a graphical interface to the Constraint System and a programmers interface. The former is a workbench called Walnut while the latter is a set of LISP functions which can be used to interface to other tasks or to create a different workbench. Figure 5.8(a) shows a typical instance of Walnut and Figure 5.8(b) some typical constraint edits. The schemata and icons shown on these figures will be explained in the next chapter.

Constraints are written using LISP and the new Path-Spec grammar (which makes it easier to access slot values). This gives the user expressive power but makes it difficult to use the constraints in generative mode. Thus they cannot be used for the purposes of constraint satisfaction. Such trade-offs have already been discussed in Chapter 4.

5.4.6 The Path-Spec Special Syntax

The Constraint Tools System defines a new syntax, called the Path-Spec syntax, which is introduced by the macro-characters “#!”. That is, whenever the LISP interpreter encounters these two characters in sequence it parses the subsequent characters according to the new syntax, until a terminating character is seen (e.g. a space, a newline or a “”). The syntax is analogous to a UNIX pathname, complete with “?” and “*” wildcard characters, except that one is accessing schemata, slots and values instead of

files and directories. The starting point for the path is taken to be whatever is currently bound to the variable “THIS-SCHEMA”. The syntax is particularly intended for constraint definition but could be used elsewhere provided “THIS-SCHEMA” is bound. Some example Path-Specs are shown in Figure 5.9. The syntax is fully described in the Constraint Tools Manuals [112].

- `(let ((this-schema 'pump-1)) <path spec>)`
..temporarily binds 'this-schema' during a <path spec> call.
- `#!~`
..just returns the value of 'this-schema' i.e. the starting point.
- `#!~|temperature|?`
..returns the value in the temperature slot of 'this-schema'.
- `#!~|*`
..returns all the slots local to 'this-schema'.
- `#!~|has-relief-valves|*|set-point|?`
..returns the set-point's of all the relief valves attached to 'this-schema'.

Figure 5.9: Example usage of the Path-Spec Grammar

5.5 Data Mapping and Propagation

This section will introduce the data mapping and propagation issues that have been considered within this work. Mostly these stem from the desire to use abstraction and decomposition, as already discussed in Chapter 1. In particular, the desire to reason about the same system at different levels of detail requires that data can be shared between the levels. This is further discussed in the second subsection below, which notes that one of the central questions is whether or not data is duplicated (and if so how consistency is maintained). It then goes on to describe the solution taken within this work, which does indeed duplicate data but also uses demons to automatically update equivalent values.

5.5.1 Uses of Abstraction

A simplistic approach to OOP tends to encourage a device-centred viewpoint in which most of the objects represent single pieces of physical plant equipment. This can make it difficult to look at a design in a more “global” manner because the overall structure is only implicit in the connectivity of the parts, rather than being explicitly represented. This can be remedied by creating instances of more abstract objects that represent larger design systems, such as control loops, heating circuits, recycle loops and distillation columns. Thus one can use schemata to explicitly represent systems at different levels of detail. This supports reasoning in the following ways:

1. A user can be asked to perform this abstraction if it is too difficult to do it automatically. One can then record his judgement by creating the necessary instances. By doing this one is allowing him to say “Regard this set of units as constituting a such and such system”.
2. One can write constraints or rules that cover larger sections of plant, without having to rederive the abstraction each time the rule or constraint is used. For example, one can say “If the flow around a heating circuit fails then it will fail to provide the required heat”.

3. One can use abstract objects to hold intermediate results during a long reasoning process. In a simulation problem this might involve a higher level representation of the current state of the system. For example, a heating-circuit might have a "status" slot into which one can put the value "failed". In a design problem the intermediate results might also be concerned with status but this time the values might be "undergoing conceptual design", "undergoing detailed design", "signed off" etc.
4. One can make analogies to standard design cases, and look for missing or extraneous features.

These are powerful possibilities but they cause their own problems:

1. Information needs to be mapped between schemata at different levels. For example, the specification of a system *determines* the structure and size of its constituent parts. Consequently, this information goes down to the more detailed levels during the design process. In contrast, the actual behaviour of a system is *determined by* that of its constituent parts, so this behaviour goes up the refinement tree. One can summarise this by saying that inputs go down and outputs go up.
2. A piece of equipment will often belong to several non-exclusive subsystems. One can therefore have multiple viewpoints for such equipment and perhaps some confusion as to its role in the plant.
3. One will eventually need to make analogies automatically, to identify when they are no longer valid and to recover from the analogy in such cases.

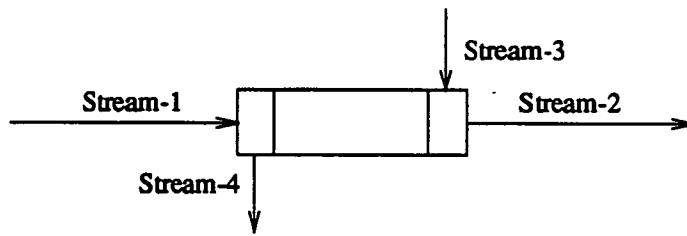
Thus one may need to be able to view a design as a set of connected subsystems with boundaries that are neither exhaustive, exclusive or fixed. Only then can one support a design view which reasons about abstract notions such as current level of completion, degree of safety and so on. Thus one of the most important tasks must be to develop a flexible representation that can support the design team in maintaining such a multi-faceted view of the project.

The representation utilities described in the last section go partway towards those needed for drawing analogies and testing them, though there are many questions of control which remain unanswered. For example, although the constraints could be used to watch and test analogies automatically it is by no means certain that one could construct a violation behaviour that was intelligent enough to recover from an invalid viewpoint. Indeed, it is virtually impossible to do so in isolation since the Constraint System was intentionally designed in a way that places few restrictions on the rest of the system. One therefore needs to incorporate a design structure that maintains additional dependency information (e.g. where a value came from) and is capable of recovering when inconsistencies are identified. Such a structure is partly provided by the Refinement Manager, which is described in Section 5.6 below. However, a user is still given virtually complete freedom over the order in which he tackles the design. Thus the Refinement Manager is also more of a representation tool than a module for doing the design. The same applies to the data mapping system, which will now be described.

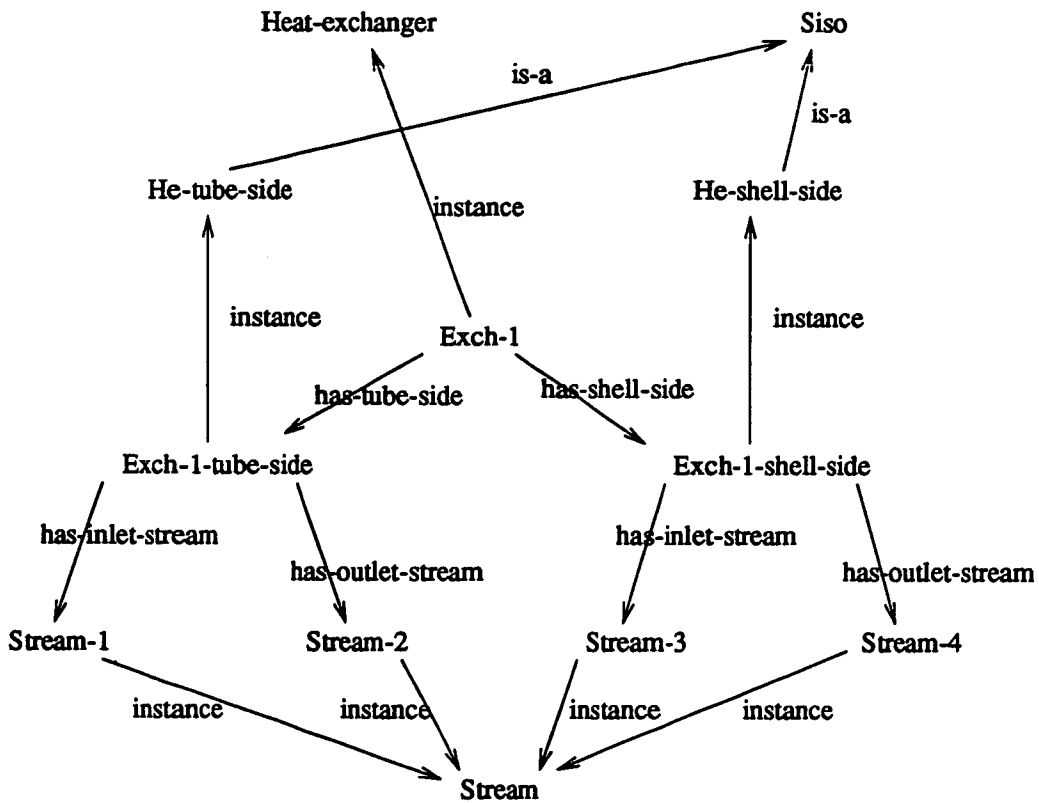
5.5.2 The Propagating-relation System

This work has explored various ways of splitting up information and accessing it when required. This is desirable for a variety of efficiency and representational reasons. For example, a previous discussion showed that using inheritance could lessen the duplication of information and increase its maintainability. In addition, there was a discussion of the notion of multiple inheritance and the separation of topology from function. This also lessens the duplication of information; for example, because one can write rules or equations that apply to any SISO system, regardless of its function. However, it does cause some problems as to how information is accessed and whether or not one needs to store it in more than one place. In addition, the next section discusses hierarchical design, which runs into similar data sharing and propagation problems.

Consider the heat exchanger in Figure 5.10(a). This is made up of a shell-side and a tube-side, each of which is a SISO system. Thus one can try to inherit information that is common to such systems. For example:



(a) An example heat exchanger



(b) A multiple inheritance network to represent it

Figure 5.10: Representing a heat exchanger by multiple inheritance

“mass flow in equals mass flow out”.

In addition, one needs to inherit information which represents the interaction between the two sides of the heat exchanger. For example:

“increasing the cold stream inlet flow
decreases the hot stream outlet temperature”.

To do this one can try to use multiple inheritance, or its rule-based equivalent (i.e. firing rules which match the different parents). This can be achieved by actually creating separate schemata for the two sides of the heat-exchanger, as well as a schema to tie the two together. In addition, one can use separate objects for the streams involved and link the whole lot together using user defined relations. Thus one can construct a network like that in Figure 5.10(b), which displays the schemata involved and labels the links between them. A similar approach would be valid if one was using equations instead of rules.

Some values have to be shared between the schemata in the network. For example, both Exch-1-tube-side and Exch-1 may need to refer to the temperature of stream-2. This network contains three levels of shared values but one could easily need more because the heat-exchanger will itself be part of a larger system.

For standard items such as a heat-exchanger one can use the same design level⁷ for the overall system (e.g. Exch-1) and its component parts (e.g. Exch-1-tube-side, stream-2 etc.). This is because there are no alternative configurations to consider. However, for larger systems, such as a distillation system, one may store the overall system schema at a higher level than its components. This would allow one to compare alternative types of component subsystem by branching the *refinement tree*. For example, one might compare the use of a thermosyphon reboiler with that of a glycol heating circuit. Thus values may need to be shared between schemata at different design levels as well between those at the same level. The implications of this will become clearer during the discussion of the Refinement Manager in Section 5.6 below.

⁷i.e. the same context, as described in the next section.

One can identify several possible solutions to this information sharing problem:

1. Only ever store the value in one place and use special access functions to retrieve it. For example, in the process engineering domain one might only store shared values within streams.
2. Store copies of the value in every schema that needs it. Update these copies automatically whenever the information changes.
3. Some combination of the above.

In a standard database system the first of these alternatives would probably be taken. However, within Knowledge Craft the second alternative makes more sense, for the following reasons:

1. If the value was stored in one place then all demons which refer to that value would have to be attached to the same slot. For example, most constraint checking demons would have to be attached to slots within streams. This is undesirable from both an efficiency and a control point of view, as many constraints are more properly attached to systems.
2. Similarly, CRL-OPS rules would have to fire on changes to streams, despite the rules really referring to systems.

Furthermore, it is important that rules, constraints and demons do not all have to be redefined whenever the representation is changed. For example, this would be the case if the representation of a CRL-OPS rule relied upon the three information levels that have just been described for a heat exchanger, and then more were introduced. This problem can be minimised by explicitly recording how information is mapped between two levels. One can then refer to this mapping, rather than directly to the information concerned. This means that if extra levels are then added one only needs to change the mappings, instead of having to change all the rules and constraints which use the information they map.

The Propagating-relations provide a means of describing such mappings, for any number of information levels. In addition, demons are attached so that information is automatically propagated when changes occur. Such propagation can take place between contexts as well as between schemata in the same context. The use of Propagating-relations will be further discussed in the next chapter but see Appendix F for more detail and some example definitions. In addition, the reader is once again referred to the Constraint Tools Manual for a complete description of this software.

5.6 Hierarchical Design

Previous sections have described the use of abstraction to reason about a design at different levels of detail, perhaps at the same time. One way of doing this is to construct a *refinement tree*, which represents design refinements in a hierarchical manner. This is already done in chemical plant design by the use of the Block Flow Diagram (BFD), the Process Flow Diagram (PFD) and the Engineering Line Diagram (ELD) levels. Part of this thesis work has been aimed towards extending this concept to allow an unlimited number of levels in the refinement tree. This has resulted in the use of Knowledge Craft contexts to implement a Refinement Manager, which is the subject for discussion in the rest of this chapter.

5.6.1 Contexts and Refinement Trees

A context is a workspace that allows one to explore alternative versions of a problem by changing copies of schemata, rather than the originals. In Knowledge Craft, contexts are arranged hierarchically, with the “\$root-context” at the top of the tree. Any context can be accessed at any time but by default the system focuses on whatever is the *current* context. The name of this is displayed under the ‘Constraint-Tools’ banner in the top right hand corner of the Knowledge Craft screen (e.g. see Figure 5.1 above). One can easily change the value of this focus.

Schemata are not automatically copied when a new context is created but only when they are accessed in the new context. If a schema exists in a higher context then

it can be accessed by any of the ones below it in the tree. If this occurs then the schema is copied into the lower context, even if the access just involves looking at it. Unfortunately, this means that one may have to explicitly update lower contexts when one changes schemata in contexts above them.

Figure 5.11 shows the use of Knowledge Craft contexts to institute hierarchical design. In the \$root-context one can place schemata representing background knowledge, such as class objects and constraints that may apply to any plant. In the next level down one can place instance schemata that describe a particular design problem at the most general level. These might be schemata representing plant inputs, design goals or constraints describing the initial design remit. One can then go on to refine the plant into sections, subsections and so on.

The tree below the “plant2” context on Figure 5.11 shows how this is done. The first level down defines alternative block flow diagrams. The subsequent levels then go on to define alternatives for the various plant sections. At each level, a child context represents *one* possible alternative for a system (schema) in the parent context. The context tree can, in theory, embed all alternatives for all possible systems and subsystems to satisfy the design. A complete design is then formed by starting at the first level and picking one alternative for each choice point, working one’s way down to the bottom of the tree.

The Refinement Manager uses contexts in this way but there are several complications. Firstly, it distinguishes between structural alternatives for a given system and alternatives which just involve different numbers (e.g. because one is considering a different throughput). The latter are said to be different *versions* of the same *structure*. To allow for this one has to spread the contexts in a second dimension and explicitly propagate some information between them. This is shown by the tree below “plant1” in Figure 5.11, which has many ‘leaf’ contexts whose names begin with “V3”, “V2” or “V1”. These are the names of the versions that are currently being explored. In contrast, the tree below “plant2” has no such contexts because this design problem has so far only been considered from a structural viewpoint. This use of contexts will be further explained in the next section.

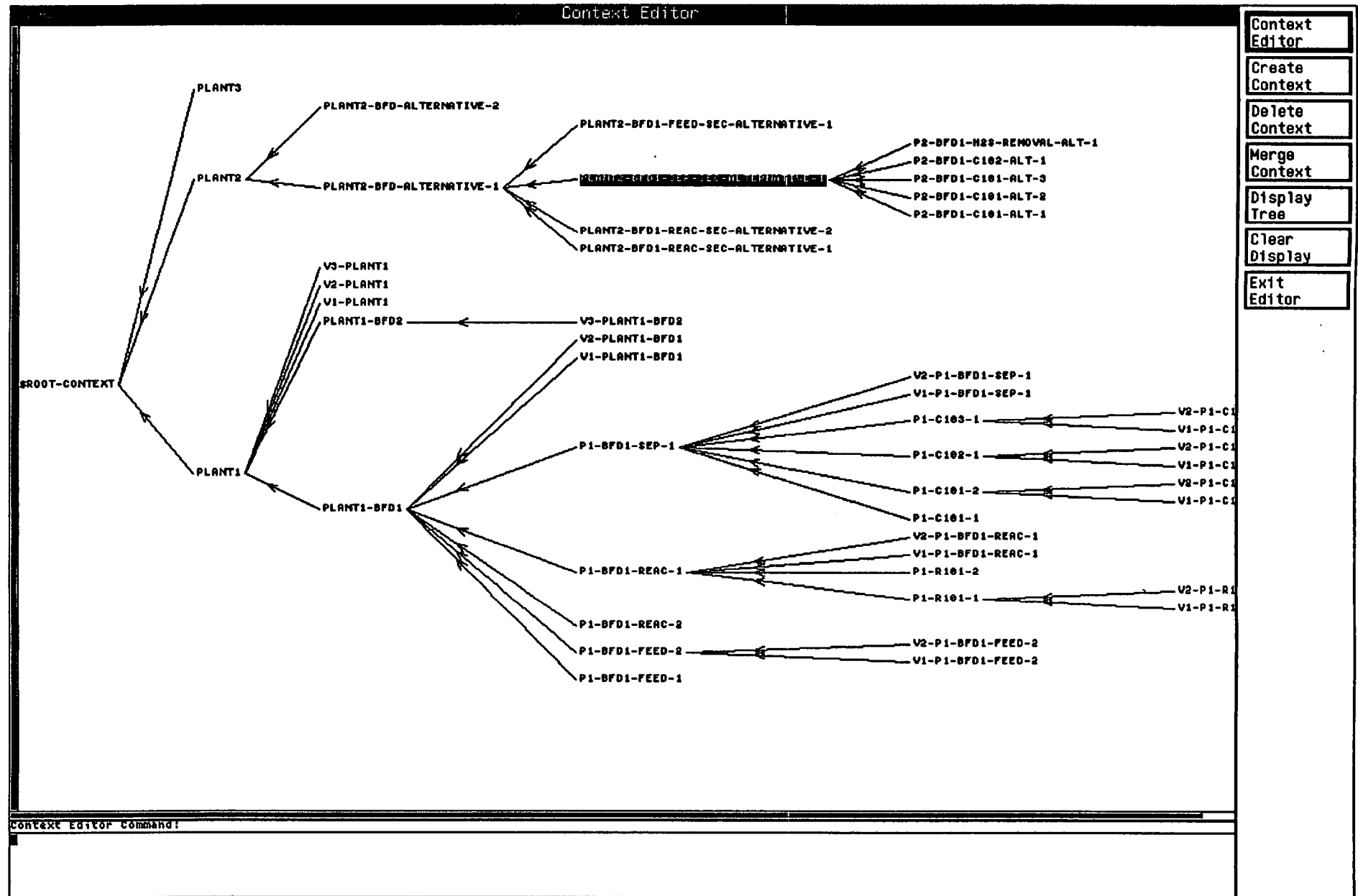


Figure 5.11: Using contexts for hierarchical design

The maintenance of contextual information is complicated by a Knowledge Craft implementation decision which means that changes to a parent context are not automatically reflected in its descendants. This is a curious decision because a change to a parent context implies that one didn't get it right first time. In this case one obviously wants the change to be reflected lower down the tree. Possibly this choice was made to satisfy the demands of CRL-OPS, since one is not supposed to change non-leaf contexts whilst using it. Unfortunately, this approach is too restrictive for use in the Refinement Manager.

The use of CRL-OPS causes another difficulty when using contexts, since all class objects must exist in the \$root-context. This prevents one from splitting up the background knowledge by adding new levels between the \$root-context and the specific design levels. Thus if one wants to use CRL-OPS then one is *prevented* from using a context tree like the one in Figure 5.12.

The Knowledge Craft context mechanism is conceptually very powerful but rather messy in implementation. Nevertheless, it provides sufficient functionality to demonstrate the possibilities in this area. Ideally it would now be reworked to better support this behaviour. In addition, it would be worth considering the relationship of databases to contexts. Knowledge Craft supports an interface that allows the use of an SQL database to store schemata. However, it appears that an explicit command has to be given to retrieve a schema from a database and to save it back again. A better solution might be if the databases acted as "super contexts", such that schemata were automatically pulled in if they could not be inherited from the context tree.

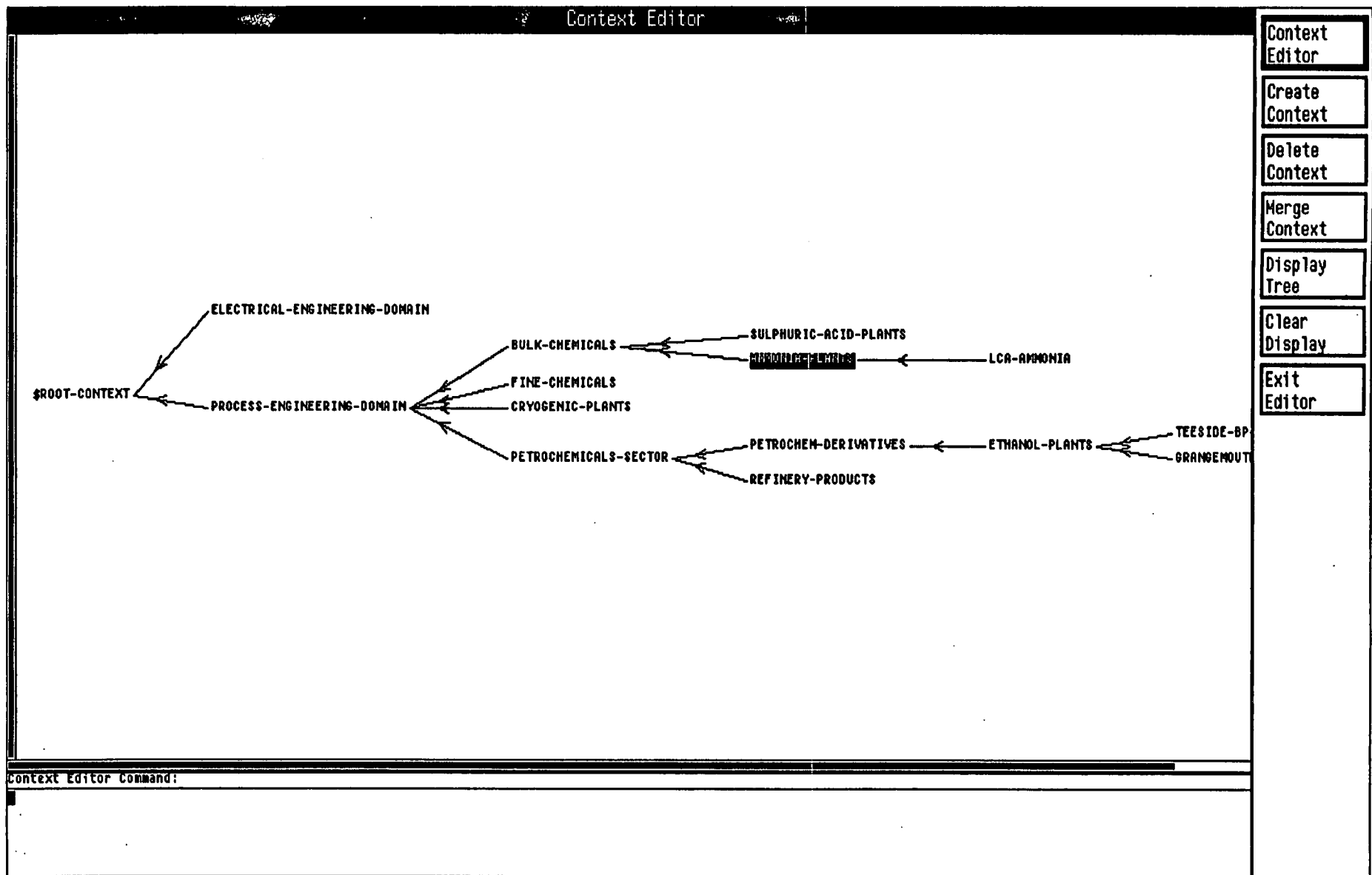


Figure 5.12: A Context tree with several levels of background information

5.6.2 The Refinement Manager

The last section has described the use of the the Knowledge Craft context mechanism to institute hierarchical design. In particular, it showed how one could construct a refinement tree to embed all possible design alternatives. A typical tree was shown in Figure 5.11 above. From this one can see that there are three types of context:

1. The \$root-context.
2. Version contexts, the names of which are made up of a version name, followed by a "-" and the name of the parent context. For the tree shown the version names were "V1", "V2" and "V3". A version context will always be a leaf of the tree but a leaf does not necessarily have to be a version context.
3. Structural contexts, which are all the other contexts on the tree.

The \$root-context holds background information and class objects; all other contexts refer primarily to instance schemata. All schemata should be created in structural contexts and inherited down to version contexts. It is intended that one can look at two types of alternative:

1. Alternatives which are structurally different, but implement the same system. Usually the requirements of this system are defined by the rest of the design.
2. Alternatives which are structurally identical but differ in equipment sizes, temperatures, pressures and so on. That is, they use exactly the same plant items (schemata) and connections, but their slot values may differ. This would be the case if one was looking at the effect of different throughputs on the same design. One would say that these are alternative versions of the same structure.

This distinction explains the use of structural and version contexts. All schemata must be defined in the former, since one must be able to look at different versions of the same structure. However, most of the slots will only be filled in within version contexts (for

the same reason). Note that for a version to make complete sense one must choose the structural alternatives for the entire plant. That is not to say that all versions have to have the same structure, one just has to be sure of what one is comparing if they don't. That is also not to say that the structure has to be fixed. One can dynamically swap structural alternatives within a given version. However, in this situation it is currently left to the user to decide if the slot values still make sense. For example, the results of a numerical simulation, for a given throughput, will no longer apply if one changes the throughput. That is, if one considers a different version. In addition, however, they will almost certainly no longer apply if one keeps the same throughput but makes structural changes to the plant!

If one examines the tree in Figure 5.11 one will see that a non-leaf, structural context can have two types of children:

- Version contexts which contain its slot values under different versions. For example, context "V2-PLANT1" contains the slot values of the schemata in context "PLANT1" under version "V2".
- Structural contexts which each give an alternative refinement of *one* of the schemata defined within the parent.

In the example tree the structural contexts have sensible names which tell one which system they are refining. For example, context "P1-BFD1-SEP-1" refines the separation system from context "PLANT1-BFD1". However, the names do not strictly matter because the refinement information is recorded as the contexts are created. This is one of the functions of the Refinement Manager. Its other facilities are as follows:

1. One can create new structural contexts, versions and version contexts. The necessary refinement information is automatically recorded on LISP hash tables.
2. One can ask to visit an existing structural or version context. There are three ways in which the context can be visited:
 - A Palm workbench can be started and the schemata network for the context automatically loaded from a file.

- A 'fixed menu' can be created, which permanently lists all the schemata that would otherwise be on the Palm network. The schema names can be mouse-selected from the menu for most other Knowledge Craft tasks.
 - The context can just made the current context.
3. One can use a schema restriction to specify the schemata in which one is "interested". These are the schemata that will be listed on the fixed menu.
 4. One can automatically save all the "interesting" schemata to files. In addition, any relevant Palm displays will also be saved. There are several options for the type of save one can perform (e.g. save everything, just save structural contexts, save a particular version and so on).

In addition, the information recorded by the Refinement Manager is used for data propagation and mapping around the refinement tree. For structural information this is easy as the data only ever needs to go down to more detailed levels. Thus one can use the normal Knowledge Craft context mechanism. In contrast, however, some of the information stored in a version context may need to go up to other version contexts at higher levels. Also, a version context has to be a leaf context and so there can be no downward inheritance. Instead, one must use some other way of propagating information between the refinement levels of a particular version.

This is not quite as difficult as it may sound because only the schemata at the boundaries of a system should exist higher up the refinement tree. In the process engineering domain this means that it is mostly just the stream objects which are shared between different contexts. In addition, one will probably still be working under the strategy that "inputs go down and outputs go up", though in some cases it may be difficult to decide whether a boundary schema is an input or an output. Anyway, automatic propagation is achieved by direct use of the Propagating-relations described in the last section. In addition, there is a command which can be called explicitly to tidy-up contexts, if automatic propagation is not appropriate. To some extent one is just trying to achieve the multiple inheritance of contexts. Some AI-Toolkits seem to provide this feature (e.g. KEE with their KEE-Worlds system [29]) but Knowledge Craft does not.

The refinement system and propagating relations are integrated in the manner just described. In addition, the constraint system is integrated by storing constraints within structural contexts. One can then use the normal context inheritance mechanism to apply them down the tree. The integration of these three key modules will be further described in the next chapter.

5.7 Summary

This chapter began by discussing Knowledge Craft at the most general level and how it can be integrated with software written in conventional languages. It then reviewed some previous chemical engineering research using AI toolkits. This discussion was mostly devoted to the Design Kit system, produced by George Stephanopoulos' group at MIT, as this was a good way of introducing the notions of inheritance, multiple inheritance and the separation of topology from function. These were shown to be powerful concepts but they can cause some difficulties with data sharing and consistency. Similar problems were also seen to arise from the representation of abstract systems as schemata in their own right, which is useful for constraint representation and other purposes. Lastly, the chapter also discussed hierarchical design, which encounters similar difficulties due to the need to share information between different design levels.

The bulk of the chapter was devoted to an abstract description of the three main components of the Constraint Tools System:

1. The Constraint System, which is used to create constraints and control their application.
2. The Propagating-relation System, which is used to specify mappings in data sharing situations and to automatically update equivalent values.
3. The Refinement Manager, which uses Knowledge Craft contexts to represent hierarchical design.

In each case the discussion began with a description of existing Knowledge Craft features that are useful for the area concerned. This set the scene for the subsequent description of the new software; which was vital in understanding how it all ties together. In addition, there was a description of how the user interacts with Knowledge Craft's graphical environment, in a manner that can be called 'pseudo multi-tasking'.

This concludes this rather abstract overview of Knowledge Craft and the Constraint Tools System. The discussion will now turn to more concrete ground as the reader is taken through some detailed examples of how the software might be used.

Few things are harder to put up with than the annoyance of a good example.

Mark Twain

Chapter 6

Design Examples Using The Constraint Tools System

This chapter will display a large series of 'screendumps' in order to lead the reader through some example sessions using the Constraint Tools System¹. In this way it describes the user interface that has been provided, together with some recommended ways of interacting with it. In addition, the examples give enough detail to show how the Constraint Tools System may be used in a design problem. The chapter is not intended for use as a software manual, however, as many details are skipped over or left out. Indeed, some of the command sequences are abbreviated in order to emphasise the main features. Once again the reader is reminded that a software manual has been written as a completely separate document [112].

6.1 Editing Background Information

When one logs on to the Constraint Tools System the terminal screen will be completely covered by the Knowledge Craft graphics window. Initially this window simply contains a scrollable row of icons along the top left edge and a context flag in the top right corner. The icons are used to start or resume instances of their associated task, as described in the last chapter. When a task is first started one is usually asked to define the overall size of its workbench. This area will then be divided between the viewports and icon

¹Unfortunately, photo-reduction has resulted in some parts of these screendumps being hard to read. If this causes problems then larger copies can be obtained from Professor J.W. Ponton, Department of Chemical Engineering, The University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh, EH9 3JL, Scotland.

viewports that need to be displayed. Only one task can be active at a particular time but one can switch from one to another by clicking `mouse-l-1`¹ on the latter's icon or on any of its workbench viewports. When a task is resumed then all of its viewports are automatically exposed.

After the Constraint Tools System has been fully initialised the next thing to do is to load up some existing files containing schemata definitions and LISP functions. These will represent the background and design specific knowledge that one has previously defined. The load can be achieved by starting a KC Listener and using the COMMON LISP "load" command. At present this is not a very efficient operation, so the load may take some time.

The next thing one may want to do is to start instances of Walnut and the Refinement Manager. For both of these one will be prompted for the name of a prototype schema which contains control information used by the workbench commands. This allows one to partition the representation into different types of constraint or refinement. For the examples in this chapter, however, only one of each type of prototype is used. These are called "design-constraint" and "design-refinement" respectively.

Among other things, the refinement prototype contains information describing some Fixed-menus. These menus will be automatically created when the prototype is chosen; thereafter, they are available for general use. This is purely for convenience so one can turn off this feature if it becomes a nuisance.

After one has loaded one's files and created the two workbenches the screen may look like that shown in Figure 6.1. This shows an instance of Walnut towards the top right corner and an instance of the Refinement Manager in the bottom right corner. In addition, some Fixed-menus have been automatically created along the left edge of the screen. These partly cover the KC Listener which was used to load a file called "ex1-top.lisp". This file then loaded up all the other files needed for the examples described in this chapter.

¹As described in the last chapter, this means to click the left mouse button once on the object concerned. See the previous footnote for similar definitions of `mouse-m-1`, `mouse-r-1`, `mouse-r-3` and so on.

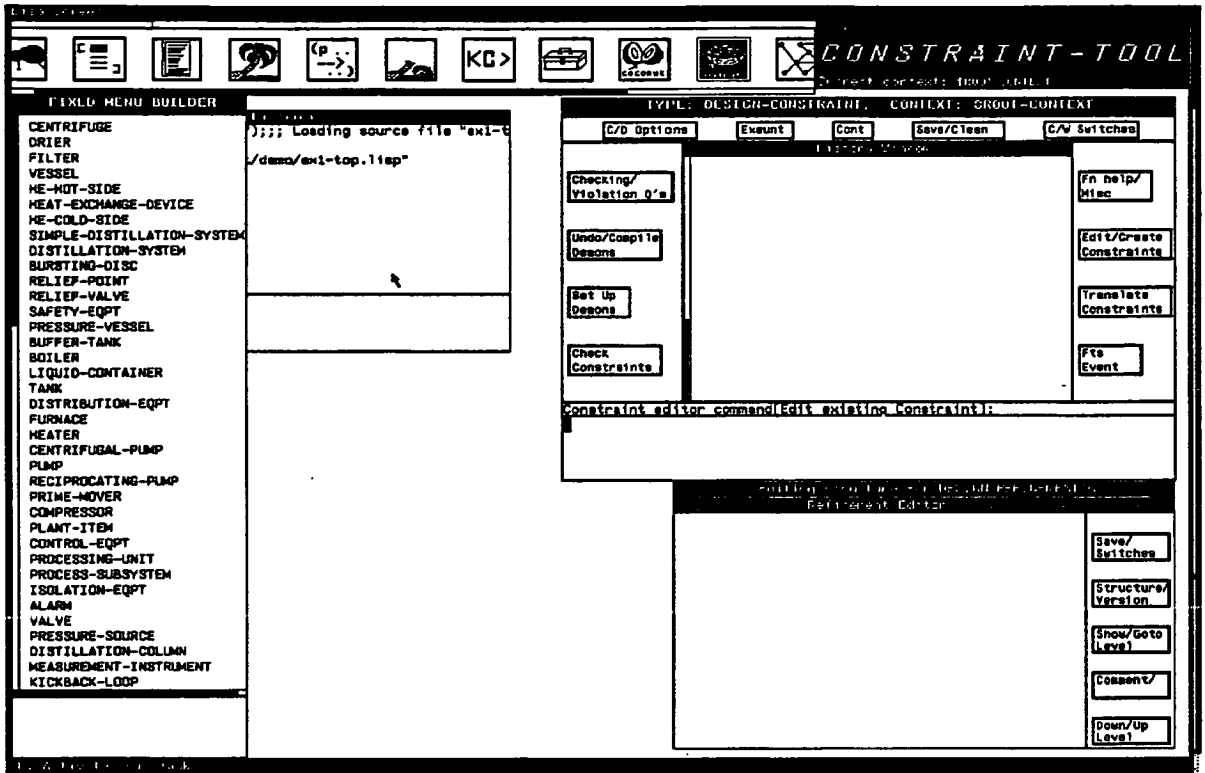


Figure 6.1: The Knowledge Craft screen after starting a Walnut and a Refinement Manager

The file "ex1-top.lisp" is reproduced in Appendix E. A file such as this has to be created by hand for one's own use. However, all it needs to do is to call certain LISP functions which load up other files, or even whole directories. Most of these files or directories will have been automatically created during a previous Knowledge Craft session.

Only one of the Fixed-menus is actually displayed in Figure 6.1, the others lie stacked underneath it. One can get access to them by clicking mouse-r-1 on the Fixed-menu task icon, which is the third icon in the row along the top edge of Figure 6.1. The menu shown on Figure 6.2 will then appear.

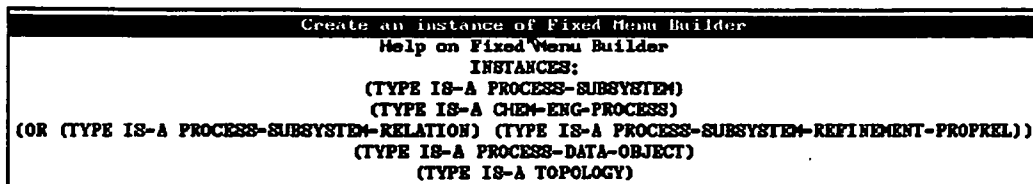


Figure 6.2: An example pop-up menu from clicking mouse-r-1 on a task icon

This menu presents one item on each line. As the mouse is moved down the menu the item currently underneath it is inverted. An inverted item is selected by clicking mouse-l-1.

The item currently inverted in the example menu allows one to create yet another instance of the Fixed-menu task. The last five items on the example menu describe the instances which currently exist. Or more exactly, they use a Knowledge Craft schema restriction to describe the schemata names which are displayed on the existing instances. For example, the first of these items says that the associated Fixed-menu displays the names of all the schemata which are is-a related to the "process-subsystem" schema. If one clicks mouse-l-1 on this item then this Fixed-menu instance will be immediately exposed and resumed.

The Fixed-menus that are associated with the design-refinement prototype display the following five hierarchies:

The process-subsystem hierarchy: which relates schemata describing the types of functional system or physical item found on chemical plants. Some of this hierarchy has already been shown in Figure 5.2 in the last chapter.

The process-data-object hierarchy: which relates schemata used to describe streams, hold-ups, materials or chemicals. Part of this hierarchy is reproduced in Figure 6.3.

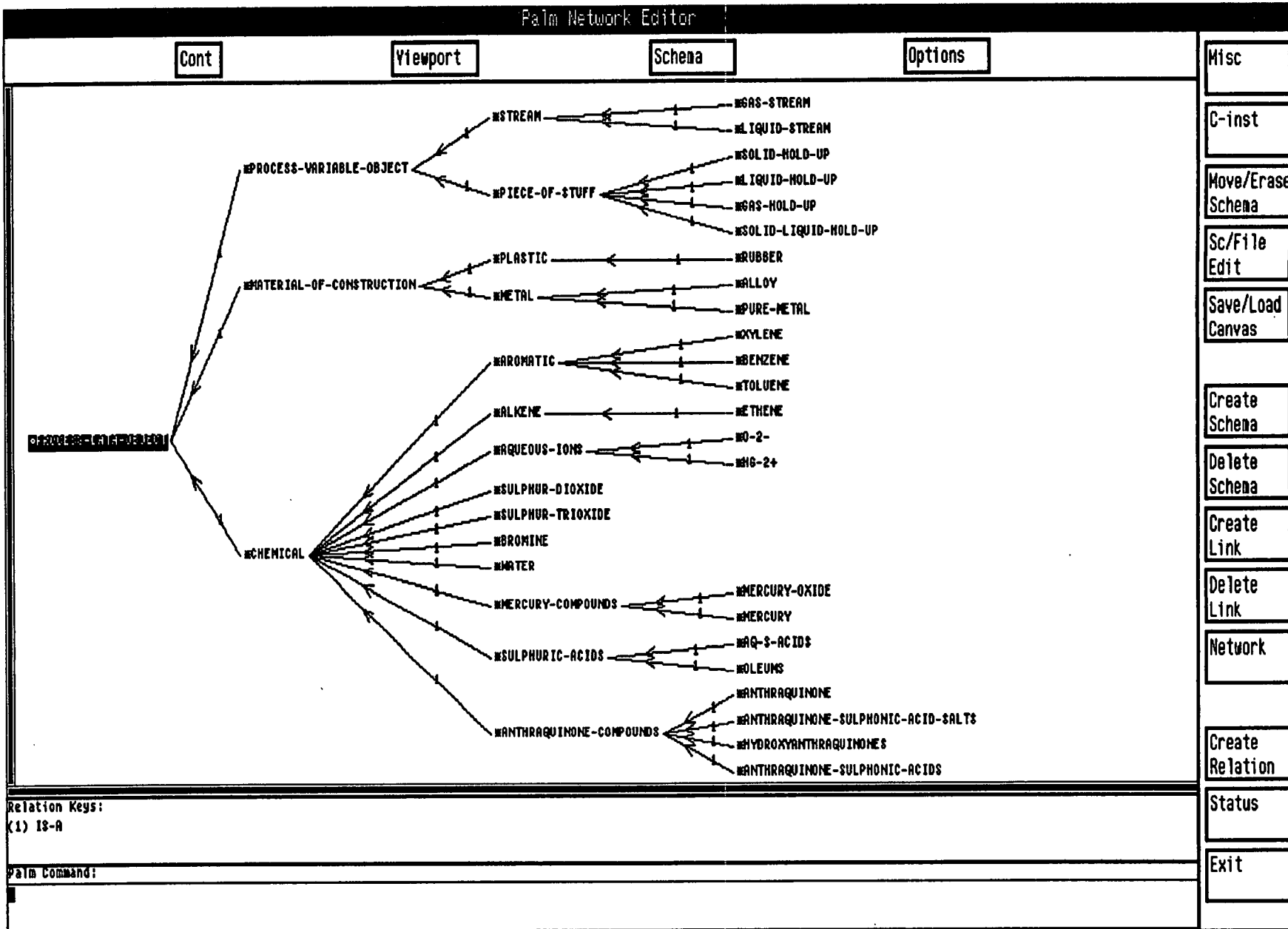


Figure 6.3: The process-data-object hierarchy

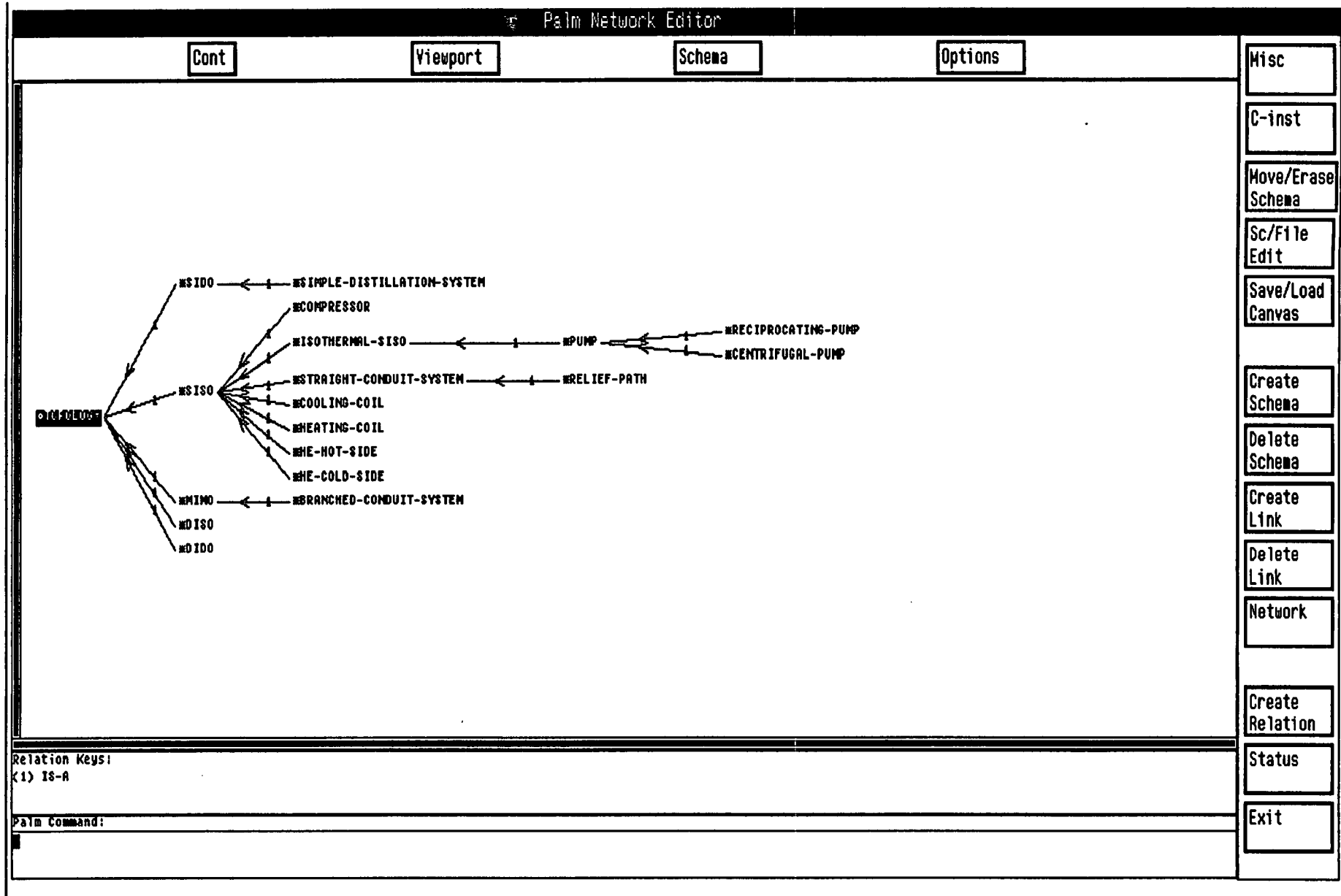


Figure 6.4: The topology hierarchy

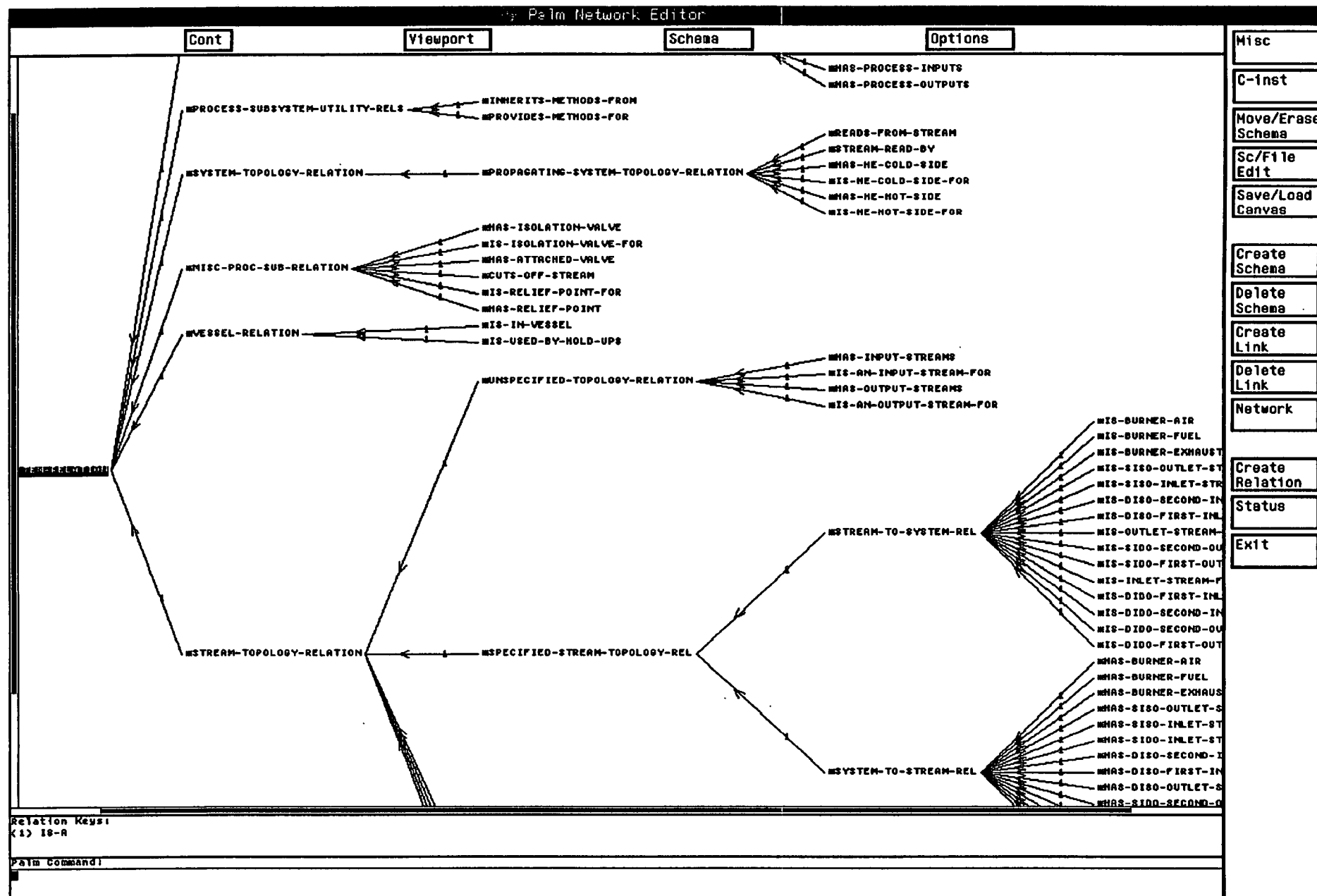


Figure 6.5: The process-subsystem-relation hierarchy

The topology hierarchy: which relates schemata used to describe the topology of a system or section. For example, a system with a single input and a single output is said to have “siso” topology. The schemata on this hierarchy will be accessed via schemata on the process-subsystem hierarchy, either by use of an is-a link or of a more specific link called “has-topology”. For example, one might say that a “prime-mover” has topology siso. This characterisation is then available for mathematical modelling, data propagation or the application of CRL-OPS rules. Part of the topology hierarchy is displayed in Figure 6.4. Most of the schemata on this figure were also on the process-subsystem hierarchy shown in Figure 5.2.

The process-subsystem-relation hierarchy: which classifies the relations and Propagating-relations that have been created for the process engineering domain. Some of these relations are fairly specific because they are only intended for use in a very particular situation. For example, “has-burner-air” should only be used to relate a burner object to its air stream. This use of specific links is clumsy but necessary, as one needs to directly describe the function of a connection and so allow the easy description of constraints. For example, for a burner one needs to know which stream is *intended* to be its air stream, which its fuel stream and which its exhaust stream. Only then can one define constraints to check this intention. Part of the process-subsystem-relation hierarchy is shown in Figure 6.5.

The chem-eng-process hierarchy: which is similar to the process-subsystem hierarchy but is entirely concerned with the function or behaviour of “processes”, rather than with manufactured items of equipment. This hierarchy was actually created for a batch plant design problem, which will be described in a later section.

The author believes that the ideas behind these hierarchies are sound but accepts that the divisions are by no means complete. In addition, there may be some confusion over how best to match a “concept” to a schema on one of the hierarchies. Really the hierarchies have been structured as they are for the purpose of best implementing the set of examples, as opposed to being based on a more fundamental study of chemical engineering concepts. For example, much of the grouping on the process-subsystem hierarchy is so that constraints can be conveniently applied in the right situations.

Unfortunately, the classification needed to do this is not necessarily the same as that needed for other tasks. Thus there is a different classification on the topology hierarchy in order to allow for the application of rules derived from mathematical models. Indeed, it is difficult to make any representation general enough for it to properly apply to tasks other than those for which it was originally constructed. Thus, in general, one will have to perform procedural translation steps to share information between co-existing tasks. Chapter 4 has already alluded to the problems of maintaining such a partly procedural knowledge base.

This problem is particularly serious for constraint definition because one can always phrase a constraint by arbitrarily creating slots and schemata with no real physical meaning. In the most ridiculous case one might have a slot called "constraint-is-satisfied". All the constraint definition then has to do is to check that this slot contains the value "yes". Unfortunately, to ensure that it always contains this value when the constraint is satisfied, and the value "no" when the constraint is violated, effectively requires a further definition of the constraint. One therefore has to limit the schemata and slots used in a constraint definition to prevent them from becoming too abstract and task specific.

This is a tricky problem because one must strike a balance between generality and detail. If the constraint definition uses terms that are too general or abstract then these terms are unlikely to be maintained by standard reasoning systems, such as numerical simulators. On the other hand, if the constraints are very specific then not only are they harder to phrase but also they may be brittle to changes in design structure. For example, suppose that one is using a constraint to protect against failure of a heating circuit. If the constraint refers directly to a 'status' slot in an abstract 'heating-circuit' object then the constraint definition is trivial but one must ensure that this status is updated when appropriate failure events occur (e.g. failure of the circulation pump). Thus one has the 'procedural' problem of knowing how (and when) to translate between the two representations. On the other hand, if the constraint refers directly to the primal events then the constraint definition will be more verbose but easier to apply. However, it will also be brittle to changes in design (such as changing the name of the pump). This example will be further discussed in Appendix F, which considers using CRL-OPS rules to perform the procedural steps.

Thus it is difficult to phrase constraints with confidence without having a large proportion of the total reasoning system in place. This is part of the reason why one should not place too much significance on the classifications and constraints produced within this work. Indeed, the primary aim has been the provision of general tools, rather than to solve problems for a given domain. Most importantly, these tools have deliberately been constructed in a manner that does not place too many restrictions on the other modules that are needed for a complete design environment.

The names of some of the constraints that have been created in this work are shown on the new Fixed-menu towards the left hand edge of Figure 6.6. This Fixed-menu was created by selecting the "Create" option from the pop-up menu in Figure 6.2. It was then placed with the mouse and supplied with a LISP expression to find the display items². In addition, Walnut has been used to pop-up a description of one of the constraints on this menu. In a real session, this description would disappear as soon as one hit any key or mouse button.

When Walnut is asked to describe a constraint it prompts the user for the constraint name. This can either be typed in or mouse selected from a Fixed-menu. Indeed, most commands in most tasks allow one to mouse select input from any existing Fixed-menu. Thus they provide a simple means of task communication.

A constraint is represented as a schema with a fixed set of slots. The most important is the "external-form" slot, which holds a LISP expression that actually defines the constraint. The example shown uses the Path-Spec syntax to obtain data from the schema being tested. Note the use of the "status" slot, which is expected to take the value "working". Also, note the use of the "in#total-kmolar-flow" slot, which refers to the 'total-kmolar-flow' of the inlet stream. This slot resides within an instance of 'SISO' but will contain data that is duplicated within a stream object. Thus it must be maintained by a Propagating-relation. Such slots nearly always have a '#' within their name.

Now suppose that one wants to create a new constraint which checks that a pressure

²A Fixed-menu can either be supplied with a schema restriction, in which case it browses the current context for all the schemata which satisfy that restriction, or it can be given a LISP expression, which is just evaluated to return the list of symbols to display.

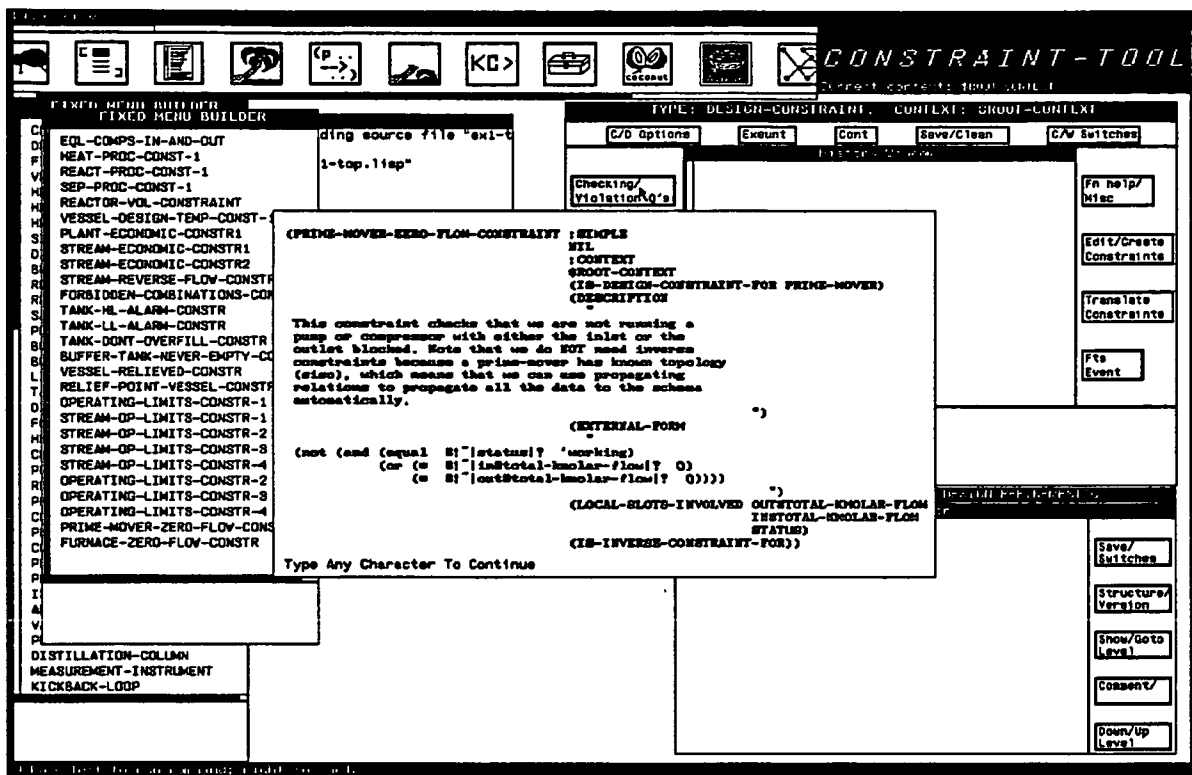


Figure 6.6: Using a Fixed-menu and Walnut to display constraints

vessel is designed for total vacuum. First one must find the best schema to which to attach the constraint. If there is already a schema called pressure-vessel then one can use this directly; if not, then one has to decide between using the next best thing or creating a schema of this name. To perform the search one should resume the process-subsystem Fixed-menu, which was mentioned in 6.2. One can then ask to search the menu, to match a string such as "press". All the items on the Fixed-menu which contain this partial string are then placed on a pop-up menu, such as the following:

Matching Menu Items
PRESSURE-INDICATOR
PRESSURE-CNTKL
PRESSURE-SOURCE
COMPRESSOR
PRESSURE VESSEL

In this example the required schema does already exist and can be selected from the pop-up menu. This places the item in a buffer so that it can be retrieved by other commands in other tasks.

To create the constraint one can now resume Walnut and click mouse-m-1 on the "Edit/Create Constraints" icon. First one is prompted for the name of the new constraint, which is typed in directly (e.g. "pressure-vessel-constr1"). One is then prompted for the schema to attach it to. At this point one can click mouse-r-1, followed by hitting the "Return" key. The first action causes the last selected Fixed-menu item to become the default input for the current command. The second action accepts this default value. Thus one can easily use the items selected from a Fixed-menu.

The example constraint has now been created so one can ask to edit it. In doing this one will need to know the slots which can currently be inherited by the pressure-vessel schema. To obtain these one might resume the Kc Listener and type in an expression using the CRL "get-slots" function. The results are then printed out:

```

Kc Listener
KC> (load "ex1-top.lisp");; Loading source file "ex1-
top.lisp"
#P"/home/nith/kcraft/ant/demo/ex1-top.lisp"
KC> (get-slots 'pressure-vessel)
(HAS-RELIEF-POINT OPEN-TO-ATM IS-LINED-WITH MATERIALS-
OF-CONSTRUCTION SAFETY-PRECAUTIONS UPPER-DESIGN-PRESSU
RE UPPER-DESIGN-TEMPERATURE LOWER-DESIGN-PRESSURE LOWE
R-DESIGN-TEMPERATURE STATUS IS-A+INV IS-A HAS-DESIGN-C
ONSTRAINTS INHERITS-METHODS-FROM HAS-INPUT-STREAMS HAS
-OUTPUT-STREAMS)
KC>

```

One is now ready to define the constraint, which is done by using a Coconut slot filler. Figure 6.7 shows the required edit after all the slots have been filled. Note that the external-form refers to the “lower-design-pressure” slot listed in the Kc Listener. No new slots were needed for this constraint definition.

```

Form filler for pressure-vessel-constr1
{{T2860847552+1568 :context $ROOT-CONTEXT
IS-DESIGN-CONSTRAINT-FOR*: PRESSURE-VESSEL
DESCRIPTION*:
"
    This constraint checks that a pressure vessel has been
    designed for total vacuum. Not all companies may want
    to enforce this.
"
EXTERNAL-FORM*:
"
    (= #!"|lower-design-pressure|? 0.0)
"
LOCAL-SLOTS-INVOLVED*: LOWER-DESIGN-PRESSURE
IS-INVERSE-CONSTRAINT-FOR*:
}}
Type in any any text you like (do not enclose in quotes)[...]:

```

Figure 6.7: Using Coconut to edit the pressure-vessel-constr1 constraint

Suppose that one now wishes to create an example problem to check the constraint definition. To do this one will need to create an instance of the pressure-vessel schema. However, one will also want to be able to destroy this instance afterwards. A clean way to do this is to create the temporary schema in a temporary context. One can then clean up everything by simply deleting the context.

A new context is created by starting the context editor, drawing out the context tree and invoking the "Create Context" command. Figure 6.8 shows the example context tree after the creation of a temporary context called "junk". This context has also been made the current context, which is reflected by the change in the context flag in the top right corner of that figure.

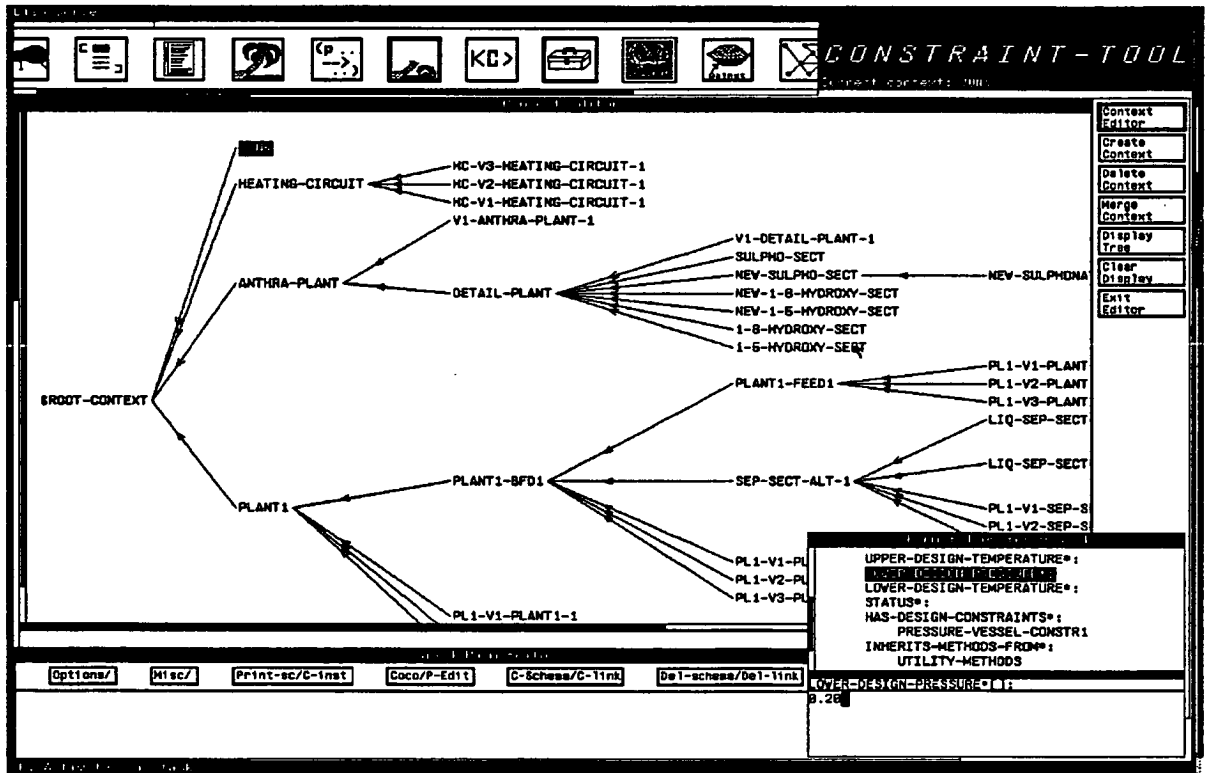


Figure 6.8: A context tree, a Small Palm and a P-edit

Figure 6.8 also shows an instance of another tool called Small Palm, which has been started and placed along the bottom edge of the screen. This has been used to create an instance of pressure-vessel called "vess-1" and to begin an edit of that schema. This edit is shown in the bottom right corner of Figure 6.8. The important thing to note about it is that the value 0.20 is being put into the "lower-design-pressure" slot. Thus the constraint should be violated as vess-1 is not being designed for total vacuum.

To check this one can resume Walnut and invoke the “Check Constraints” command. First one is prompted for the schema name, then the names of the constraints to be checked. At the second prompt one can click mouse-r-1 to be offered a menu of all the constraints that can be inherited by the chosen schema. The menu for the vess-1 schema is shown in Figure 6.9. This is a multiple choice menu, which means that it will keep reappearing until one selects a terminating option i.e. one of those beginning with an “*”. Thus one can check several constraints at once.

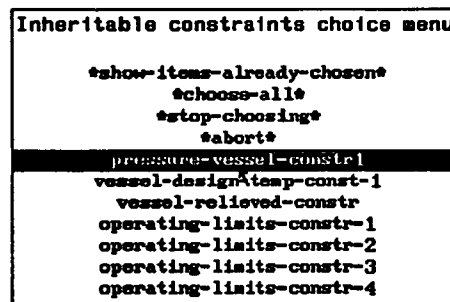


Figure 6.9: A pop-up menu of inheritable constraints

When a constraint is found to be violated then the action to be taken is defined by the constraint prototype. The default behaviour is to pop-up a warning message, such as that shown in Figure 6.10. However, one can easily change this behaviour at any time. For example, one might decide to write a warning message to a file instead of to the screen. A later section will describe how this is done.

The new constraint has now been defined and tested, so the junk context is no longer required. It should therefore be destroyed using the commands provided by the context editor. This will automatically destroy the temporary schemata and put one back into the \$root-context by default. In addition, one may want to redisplay the Fixed-menu of constraints as the new one will now be included in the list.

The next thing to consider is whether or not the new constraint should be checked automatically from now on. If so, then Walnut can be used to institute this behaviour.

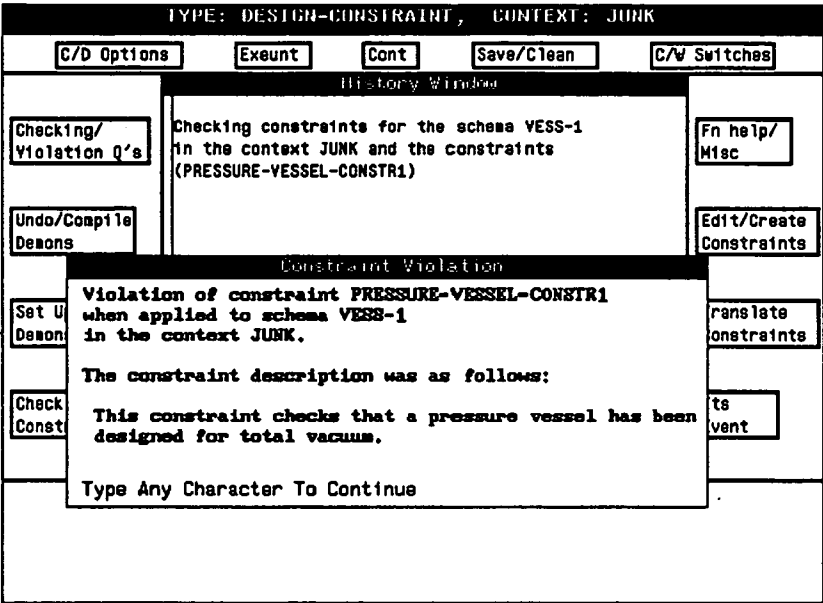


Figure 6.10: An example message warning of a constraint violation

The procedure is analogous to that used for explicitly checking a constraint. It is begun by resuming Walnut and clicking mouse-l-1 on the “Set Up Demons” icon. One is then prompted for the schema to which automatic checking is to be attached. This will be called the *attachment-schema* from now on. It can either be typed in or selected from a Fixed-menu. A pop-up menu then appears, much like that already shown in Figure 6.9. This offers all the constraints that can be inherited by the attachment-schema supplied, so one can set-up checking at any level. For example, the “operating-limits-constr-1” constraint is directly attached to the “plant-item” schema and applies to any schema which can inherit from this object. However, one could decide to only automatically check this constraint for instances of the “pressure-vessel” schema, rather than also for tanks, valves, pipes and so on. This is easily done by using the pressure-vessel schema as the attachment-schema and selecting this constraint from the menu that pops up.

One may also want to see which automatic checks have already been set-up, and perhaps undo some of them. To do this one should click mouse-l-1 on Walnut’s “Undo/Compile

Demons" icon. One is then prompted for a LISP expression which will find the previous attachments. The default expression is as follows:

```
(dominated-attachments '<constraint prototype>
:context nil :schema nil :constraints nil :inclusion-fn nil)
```

If this default value is accepted then one will find every automatic check that has been set-up; regardless of the attachment-schema, constraint or context involved. Usually one will only want to find a restricted subset of these, so the default is yanked into the command buffer and edited using EMACS commands. For example, one might restrict it to the following:

```
(dominated-attachments '<constraint prototype>
:context nil :schema 'vessel :constraints nil :inclusion-fn nil)
```

This will then only find checks which were attached to the "vessel" schema or a schema below it in an is-a hierarchy. When one has supplied an expression then one can get access to a pop-up menu which describes all the attachments found. For the example just given, the menu is shown in Figure 6.11. All but one of the checks on this menu were set up in a previous Knowledge Craft session; Walnut's save functions allow the current state of the Constraint System to be accurately reproduced the next time one logs on.

Each item on this menu describes a single demon instance that was created. The format of this description is as follows:

```
(<attachment-schema> <constraint> <slot> <demon-instance> <context>)
```

This says that a demon called <demon-instance> was created to watch the <slot> and to check the <constraint> whenever that slot changes in an appropriate schema and context. Note that <context> refers to the context in which the check was set up; this is the highest context in which the <demon-instance> schema exists. The automatic

```

Attachments found

*show-items-already-chosen*
*choose-all*
*stop-choosing*
*abort*

(vessel vessel-relieved-constr upper-design-pressure t2857461725+1231 $root-context)
(vessel vessel-relieved-constr has-relief-point t2857461725+1232 $root-context)
(buffer-tank buffer-tank-never-empty-constr level t2857461725+1233 $root-context)
(tank tank-dont-overfill-constr max-level t2857461725+1236 $root-context)
(tank tank-dont-overfill-constr level t2857461725+1237 $root-context)
(tank tank-ll-alarm-constr ll-alarms t2857461725+1238 $root-context)
(tank tank-hl-alarm-constr hl-alarms t2857461725+1239 $root-context)
(pressure-vessel pressure-vessel-constr1 lower-design-pressure t2860742740+3425 $root-context)

```

Figure 6.11: A menu showing some existing attachments for checking constraints

check will apply in this context and any below it in the context tree. Also note that several attachments on the example menu seem to be checking the same constraint. This is because a different demon is created for each slot involved in a constraint i.e. for each slot named in the constraint's "local-slots-involved" slot.

Once the menu has popped up one can select attachments to have them removed. Remember that one may need to remove several attachments to completely stop a constraint being checked. For more information on this and the dominated-attachments function see the Constraint Tools Manual[112].

6.2 An Example for the design of Continuous Plants

6.2.1 Editing Structure

The previous section has shown how the Constraint Tools System can be used to edit background information and to create constraints which apply to a large number of designs. In contrast, this section will discuss the management of information about a specific plant design project. These two types are somewhat different since the first is stored almost entirely in the \$root-context, while the second is spread throughout

the lower levels of the tree. Indeed, project specific information is almost exclusively represented using schemata that are instances of prototypes which already exist in the \$root-context.

The discussion will centre around the set of design levels that have already been 'loaded up' in the last section. One can begin by looking at the top level of the current set of examples. To do this one should resume the context editor that was shown in Figure 6.8. Note that the first level down the tree contains the three contexts "heating-circuit", "anthra-plant" and "plant1". These are the top levels for the three different problems that will be described. This section, however, is just interested in the "plant1" problem.

In order to look at the plant1 level one should make it the current context and then resume the Refinement Manager instance that was shown in Figure 6.1. One can then click mouse-m-1 on the "Show/Goto Level" icon, thus asking to "Goto" the current level. By default this results in the creation of a Palm instance to display the schemata associated with that level. One can, however, change this behaviour.

The overall size of a new Palm is specified when one first starts the Refinement Manager. A context level may have an associated Palm display if one had been created and automatically saved in a previous session. In this case the relevant file name will be made the default for the "Load Canvas" command in the new Palm. Consequently, a previous display can be reloaded by simply clicking mouse-m-1 on the "Save/Load Canvas" icon and hitting the "Return" key to accept the default value.

Figure 6.12 shows the display for the plant1 context. At this level all one has is an instance of "plant" called "plant1", one instance of "gas-stream" called "fuel-gas" and three instances of "liquid-stream". The stream objects perform the dual function of stating what the plant should produce (i.e. xylenes from toluene) and what the mass balance says the plant will also produce (i.e. benzene and fuel-gas). In addition, there are a few constraints at this level which further define the design problem. These are shown on the Fixed-menu at the left hand edge of the figure. This is actually the same Fixed-menu instance as was used in the \$root-context.

The five constraints displayed on the Fixed-menu are shown in Figures 6.13 & 6.14.

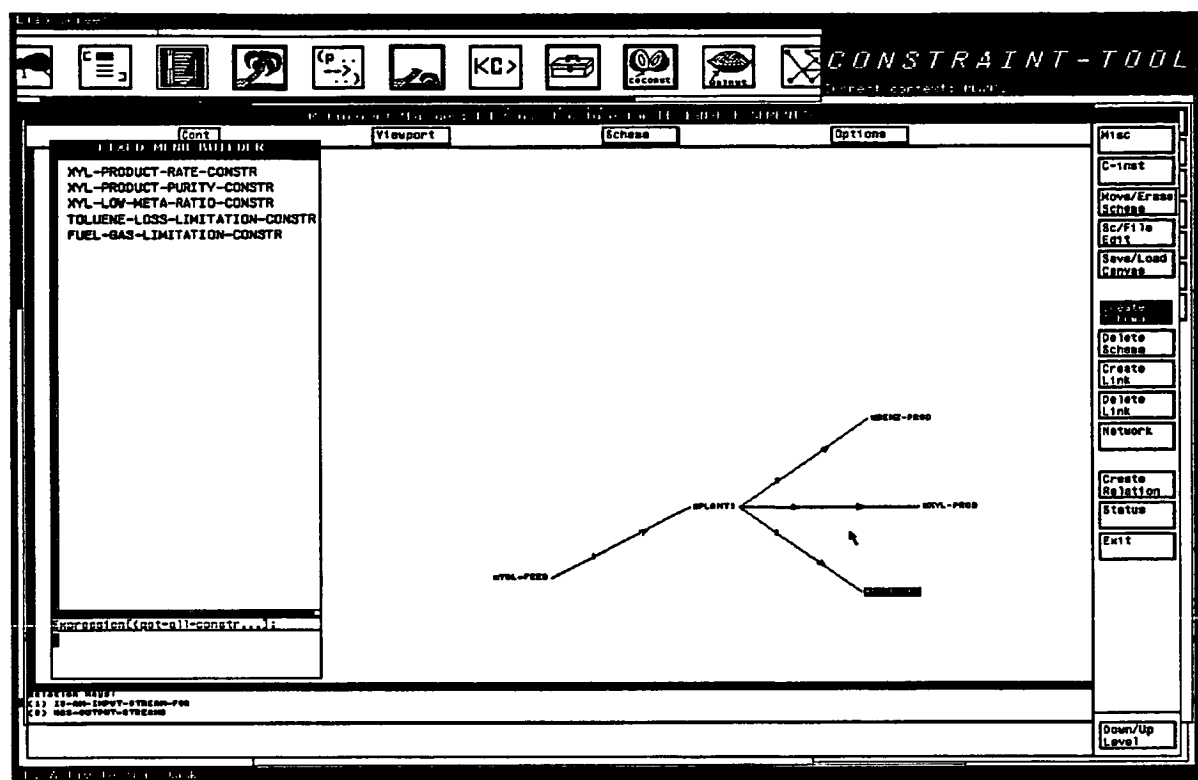


Figure 6.12: The top level for the plant1 design problem

Note that they are attached to instance schemata, such as ‘xyl-prod’, rather than to prototypes, such as the ‘liquid-stream’ object. This demonstrates that constraints can be attached and checked at any level.

```

{{xyl-product-rate-constr
  is-design-constraint-for: xyl-prod
  description:
    "This constraint checks that the plant is approximately
    producing its remit product rate of 10 Ktpa of mixed
    xylenes (= 3.6E-03 Kmol/s). We allow for a 30% turn-up
    or turn-down ratio."
  external-form:
    "(and (> #!~|total-kmolar-flow|? 2.52E-03)
          (< #!~|total-kmolar-flow|? 4.68E-03))"
  local-slots-involved: total-kmolar-flow
  }}

;; -----

{{xyl-product-purity-constr
  is-design-constraint-for: xyl-prod
  description:
    "This constraint checks that the mixed xylenes
    product is at least 99% pure."
  external-form:
    "(>= (+ (get-mole-fraction #!~ 'para-xylene)
             (get-mole-fraction #!~ 'ortho-xylene)
             (get-mole-fraction #!~ 'meta-xylene))
          0.99)"
  local-slots-involved: has-components has-component-mole-fractions
  }}

;; -----

{{xyl-low-meta-ratio-constr
  is-design-constraint-for: xyl-prod
  description:
    "This checks that the fraction of meta-xylene is less
    than the equilibrium value of 0.5. If not, we may as
    well use the standard 'conchem' catalyst process."
  external-form:
    "(< (/ (get-mole-fraction #!~ 'meta-xylene)
           (+ (get-mole-fraction #!~ 'para-xylene)
              (get-mole-fraction #!~ 'ortho-xylene)))
          0.5)"
  local-slots-involved: has-components has-component-mole-fractions
  }}

```

Figure 6.13: Some example constraints on the xyl-prod stream

```

{{fuel-gas-limitation-constr
  is-design-constraint-for: fuel-gas
  description:
    "This constraint limits the fraction of the toluene
     feed that is allowed to end up as fuel gas."
  external-form:
    "(<= (/ (get-total-mass-flow #!~)
             (get-total-mass-flow 'tol-feed))
          0.03)"
  local-slots-involved: total-kmolar-flow has-components
                        has-component-mole-fractions
  is-inverse-constraint-for: toluene-loss-limitation-constr
  }}

;; -----

{{toluene-loss-limitation-constr
  is-design-constraint-for: tol-feed
  description:
    "This constraint limits the fraction of the toluene
     feed that is allowed to end up as fuel gas."
  external-form:
    "(check-constraints 'fuel-gas
                        '(fuel-gas-limitation-constr)
                        'design-constraint)"
  local-slots-involved: total-kmolar-flow has-components
                        has-component-mole-fractions
  is-inverse-constraint-for: fuel-gas-limitation-constr
  }}

```

Figure 6.14: Some example inverse constraints

To use these constraints one first needs to define some additional LISP functions, such as "get-mole-fraction" and "get-total-mass-flow". Obviously, for a given application one would build up a library of functions which are useful for constraint definition. However, it remains an open question as to how much LISP one can reasonably expect an end user to write. There will always be a trade-off between supplying a confusing

number of pre-defined constraints and functions, and requiring a user to do too much programming himself.

The constraints shown in Figure 6.14 are inverses of each other i.e. they are both checking for the same scenario. Unfortunately, such duplication is necessary because one must be able to automatically check the constraint upon changes to either the “fuel-gas” or the “tol-feed” schemata. However, note the way in which one can avoid redefining the constraint within the tol-feed version. Instead, one can just call for an explicit check of the “fuel-gas-limitation-constr”, upon the fuel-gas schema. This means that changes to the constraint need only be made in place, which is an advantage. Unfortunately, one does lose some of the control over what happens when a constraint is violated. Such problems could be ironed out in a future implementation.

The plant1 context in this example has a number of constraints directly associated with it. To record this information we say that the plant1 context is the *home-context* for these constraints. They can then be applied in the plant1 context and any context below it. Thus one can use contexts to partition the application of constraints. In addition, one can set-up an automatic check for a constraint in its home-context or in any context below this. This check will then be carried out on relevant changes to this *attachment context* and any below it. For example, one could undo the automatic check that was set-up for “pressure-vessel-constr1” in the \$root-context and instead attach it to the plant1 context. The check would then only watch the contexts that belong to the plant1 problem, rather than those which belong to the heating-circuit problem or the anthra-plant problem. Similarly, one could set-up automatic checking of “xyl-product-rate-constr” for the whole of the plant1 problem, or just for a lower context which is only concerned with a subsection of the design.

The plant1 context has now been discussed in some detail so it is time to descend to the next level. This is requested by clicking mouse-1-1 on the “Down/Up Level” icon in the Palm workbench on Figure 6.12³. One is first prompted to mouse select the schema that one wishes to refine. In the current example the “plant1” schema is the only sensible choice since the other schemata are just streams. One is then prompted

³This is an extra icon that only appears when a Palm instance is automatically started by the Refinement Manager. It does not appear when one explicitly starts a Palm by clicking on its task icon.

for the type of visit to make. At this point one can click mouse-r-2 to be offered the following menu of choices:

Command Choices
New Refinement
Existing Refinement
MOUSE-R-3 : Menu of Global Commands

For the current example suppose that one selects the "Existing refinement" option. One will then get a further menu which shows the contexts that have already been created to refine this schema:

Refinement choices for system PLANT1
plant1-bfd1
abort

These contexts are simply immediate children of the current context. However, remember that there is a distinction between structural and version contexts. Only structural contexts are being edited at the moment so version contexts do not appear on the menu. At the present moment in this example there is only one possible choice but a later discussion will show how alternative refinements can be created and explored at this point.

When the next context has been chosen then a new Palm instance is started automatically. The new Palm also has a "Down/Up Level" icon and is the same size as its parent. In addition, it also has a default display, which can be loaded by using the "Load Canvas" command. Figure 6.15 shows the new Palm, which is displaying the BFD level for the plant1 problem. In addition, the Refinement Manager has been resumed, and the "Show Level" command used to pop-up a message describing the

new context.

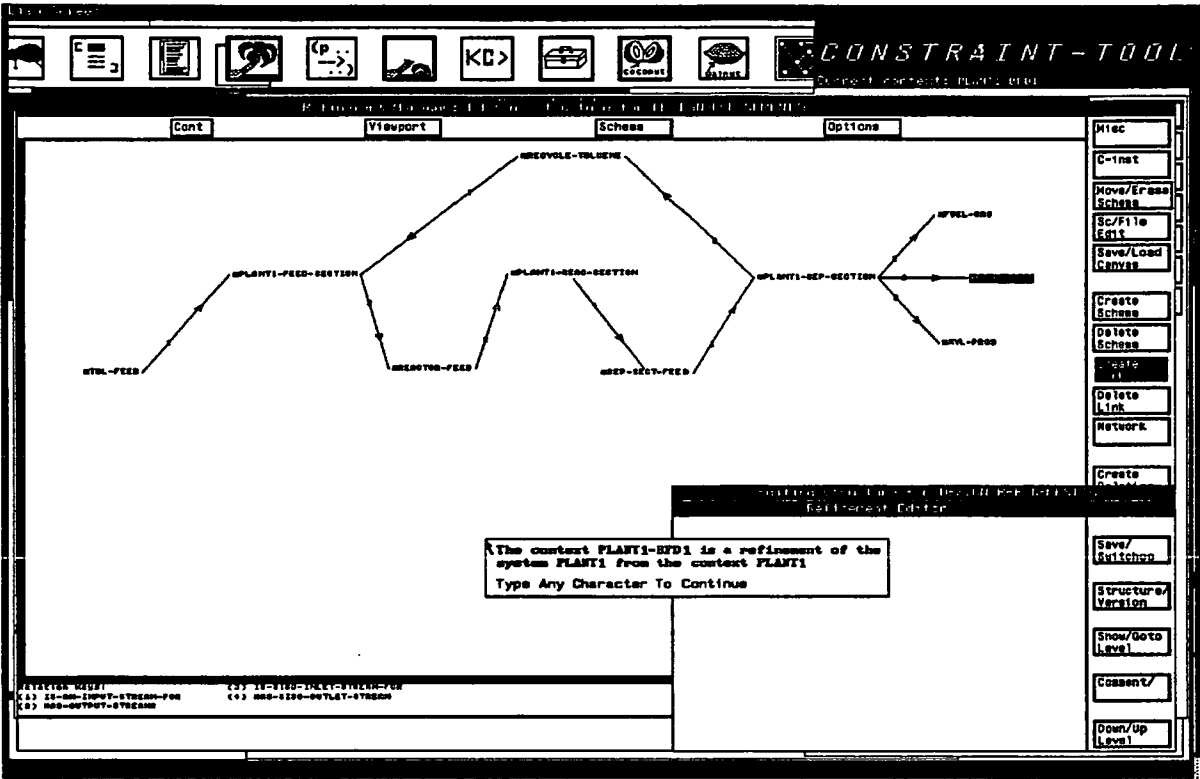


Figure 6.15: The BFD level for the plant1 design problem

In a similar manner one can look at a refinement of the “plant1-feed-section” from the BFD level. This has been done in Figure 6.16. The reader is advised to find this new context on the context tree in Figure 6.8 to properly understand this sequence of actions. Note that 9 different links are displayed on this new Palm. To see which is which one must match the numbered arrows on the main viewport with the numbered links in the relation key viewport. For example, the arrows in Figure 6.16 which refer to the ‘has-siso-outlet-stream’ relation are labelled with the number 4.

Figure 6.16 also displays an edit of the plant1-feed-section schema in this new context. What one should notice about this is that the “has-input-streams” slot refers to schemata which do NOT exist in the context above. In other words, this refinement

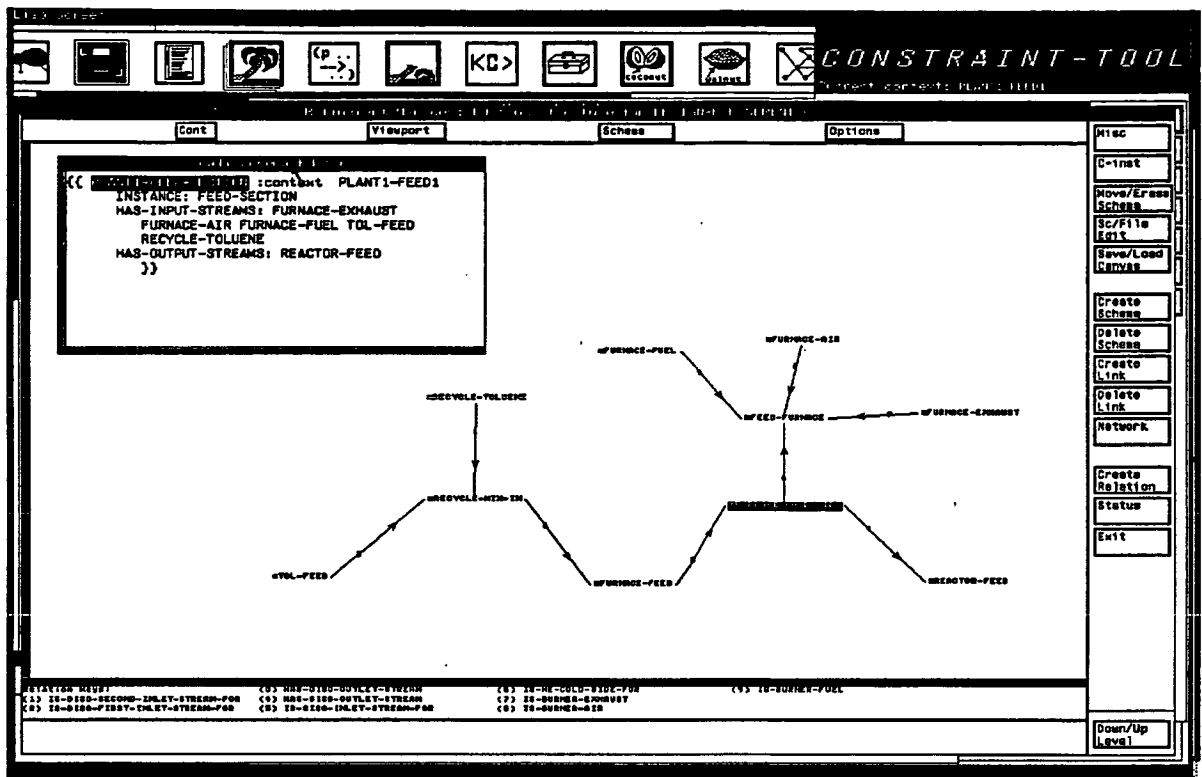


Figure 6.16: The plant1-feed1 context for the plant1 design problem

of the plant1-feed-section object has more input streams than the initial specification of it at the BFD level. One might consider passing this additional information up the context tree but this will cause problems if one creates a different refinement which doesn't use these additional inputs. Instead, a new version of the object is automatically created in the lower context and used to record the additional information. Thus the use of contexts allows one to maintain link consistency whilst exploring alternative refinements.

Whilst in the plant1-feed1 context one might browse for some more constraints. Whilst doing this one would find a constraint called "tol-feed-pressure-constr", which is shown in Figure 6.17. This is a constraint on the "tol-feed" schema, which also exists above the plant1-feed1 context. As such, one might consider attaching this constraint in a

higher context, such as at the plant1-bfd1 or the plant1 level. However, it has been attached in the lower of these three because it is specific to this choice of refinement. Thus it need not necessarily apply for alternative refinements of the plant1-feed-section schema.

```
{tol-feed-pressure-constr
  is-design-constraint-for: tol-feed
  description:
    "This constraint just checks that the pressure of the
     feed stream is high enough to not have to pump it into
     this section. This is an example only."
  external-form:
    "< #!~|pressure|? 8.0)"
  local-slots-involved: pressure
}}
```

Figure 6.17: A constraint specific to a refinement choice

Examination of the context tree reveals that all the children of plant1-feed1 are version contexts i.e. the feed section is refined no further along this branch. Consequently, one might now want to jump across to a different branch of the context tree to explore other parts of the design. This is done by resuming the context editor and changing the current context to that which one wishes to visit. One should then resume the Refinement Manager and ask to "Goto" this context (by clicking mouse-m-1 on the "Show/Goto Level" icon). Again, a new Palm instance will be started and the default display set appropriately. For example, one could jump directly across to the sep-sect-alt-1 context, which is shown in Figure 6.18. This context shows a refinement of the "plant1-sep-section" schema from context plant1-bfd1.

This figure also shows the menu of refinement choices after one has made a request to go down further, to look at existing refinements of the "liquids-sep-sect" object. This time there is more than one choice, so the context tree is branching to represent alternative refinements of the same system. Figure 6.19 shows one alternative for this separation

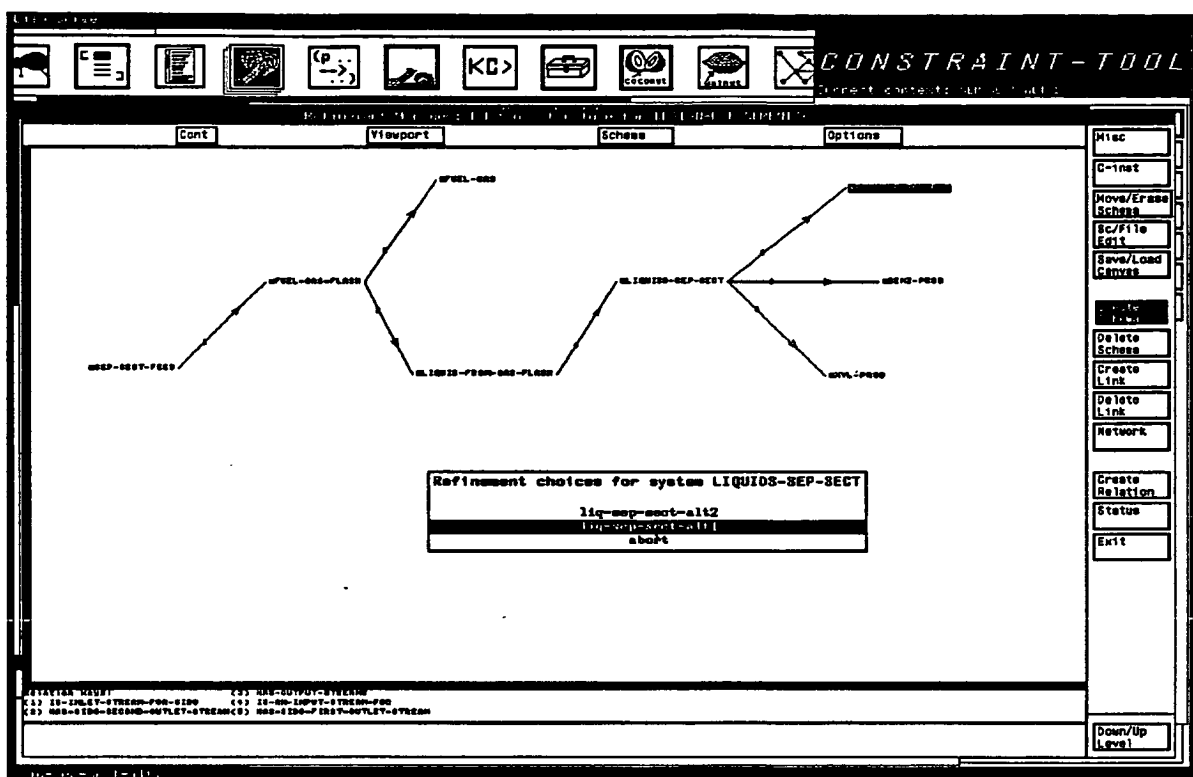


Figure 6.18: A refinement of the plant1-sep-section object

sub-system, which is required to split a single stream into two product streams and a stream of recycle toluene. The other alternative does the same job but takes product xylenes from the bottom of the first column and a benzene-toluene stream from the top. The second column then produces the benzene product as its top stream and recycle toluene as its bottoms. This is a classic example of the alternative separation sequences for a three product split.

Figure 6.19 is also showing how one creates a new refinement for a system in the design. In this example the user has already clicked mouse-1-1 on the "Down/Up Level" icon and selected "dist-col-1" as the schema to refine. Now he is being asked what type of visit he wishes to make. After selecting "New Refinement" he will be prompted for the name of the new context that will be created. At this point he can supply any name

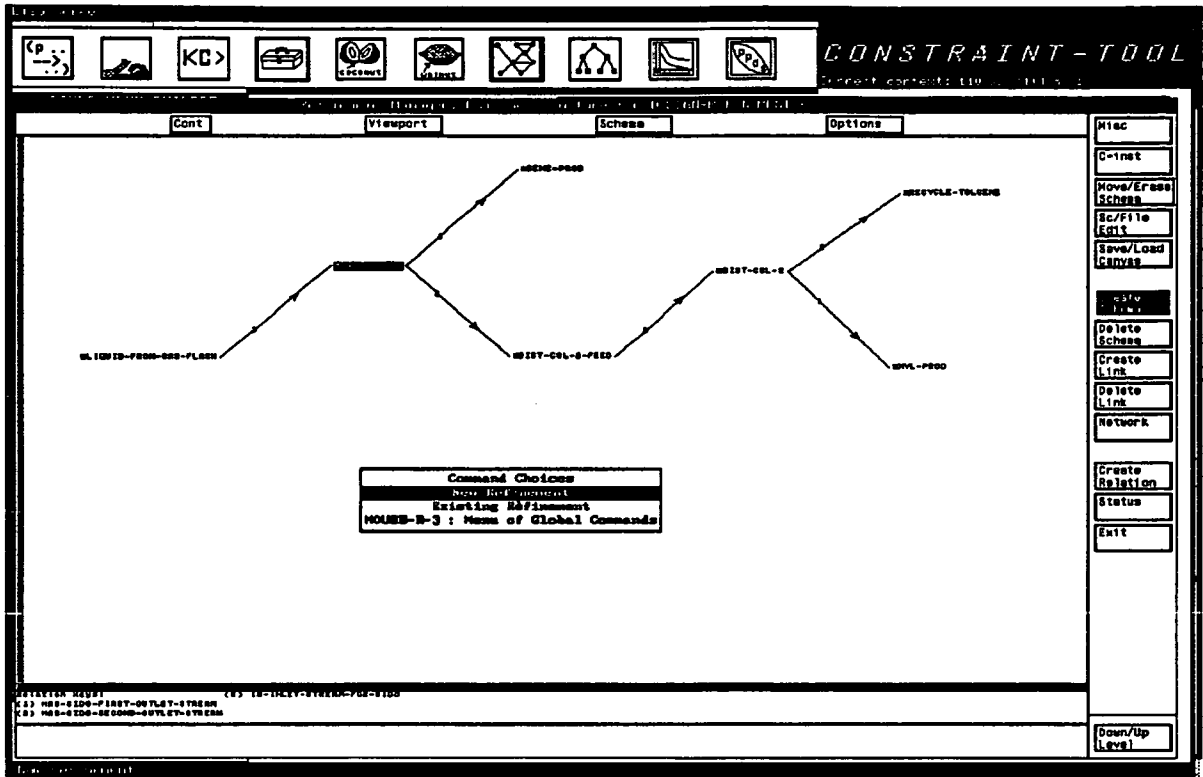


Figure 6.19: One of two alternative refinements for the liquids-sep-sect object

because the Refinement Manager keeps track of where it came from. However, for his own convenience he calls the new context “dist-col-1-alt-1”.

The new context will now be created as a child of the current context. A new Palm will then be started, much the same as before. Again, a default display can be loaded by using the “Load Canvas” command. However, this time the display is derived from the Palm for the parent context, rather than having been saved from a previous session.

Figure 6.20 shows the initial display for this example. Compare this with Figure 6.19 to see how it is just a part of the Palm display for the parent context. In fact, the new display always consists of the schema that is being refined, all the arrows that were attached to it and all the schemata on the ends of those arrows. The rationale for

this is that the refinement should contain all the boundary objects that were attached to the schema itself in the higher context. That is, everything that goes into the abstract representation of the system as a whole must also go into the more detailed representation somewhere. Indeed, it is mostly through these boundary objects that the more detailed objects connect to the rest of the design. The initial display makes it easy for one to specify this connection.

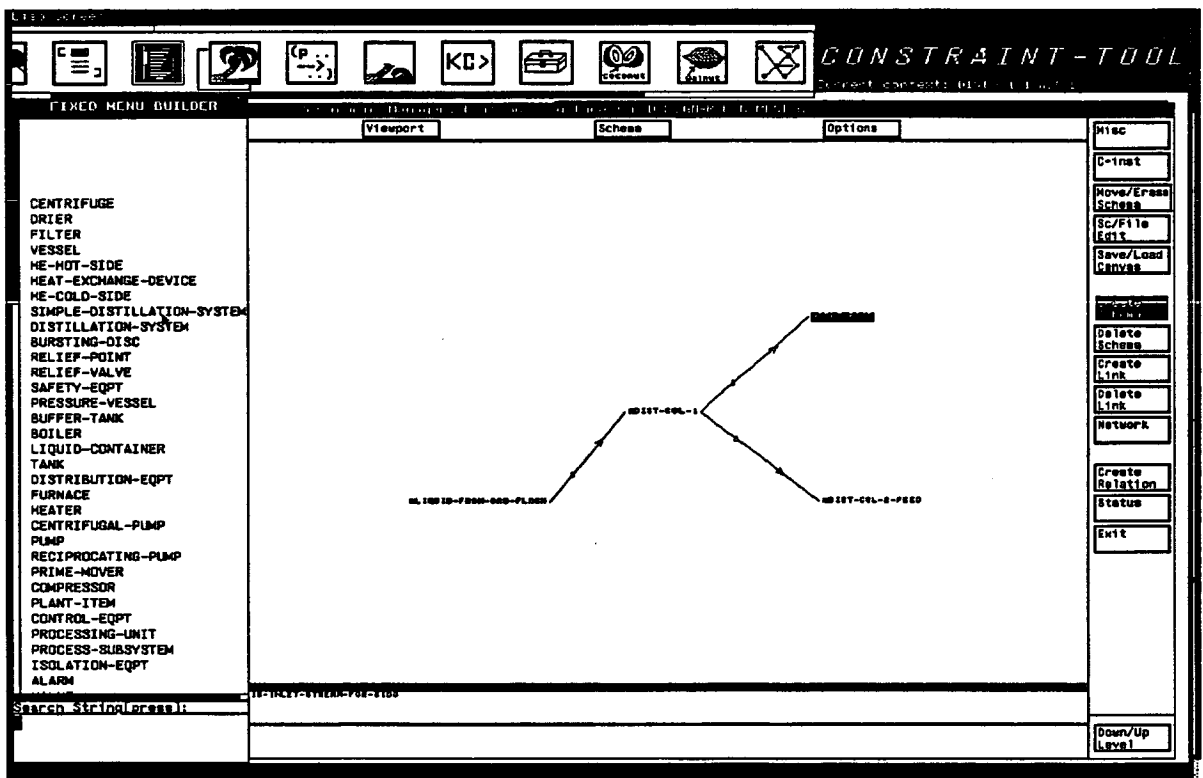


Figure 6.20: The initial display for a new refinement of the dist-col-1 schema

Usually the more detailed display does not need to show the schema that is being refined. In this case one can simply 'erase' it from the display, thus erasing all the links as well. One can then define the new system by creating new schemata, linking them to the boundary objects and tidying the display using the "Move Schema" command. Figure 6.20 shows how an existing Fixed-menu can be used to select prototypes for the creation of new instances. This makes it relatively easy to create complicated

networks, such as that shown in Figure 6.21. To understand this network the reader is recommended to locate the boundary objects that appeared on Figure 6.20.

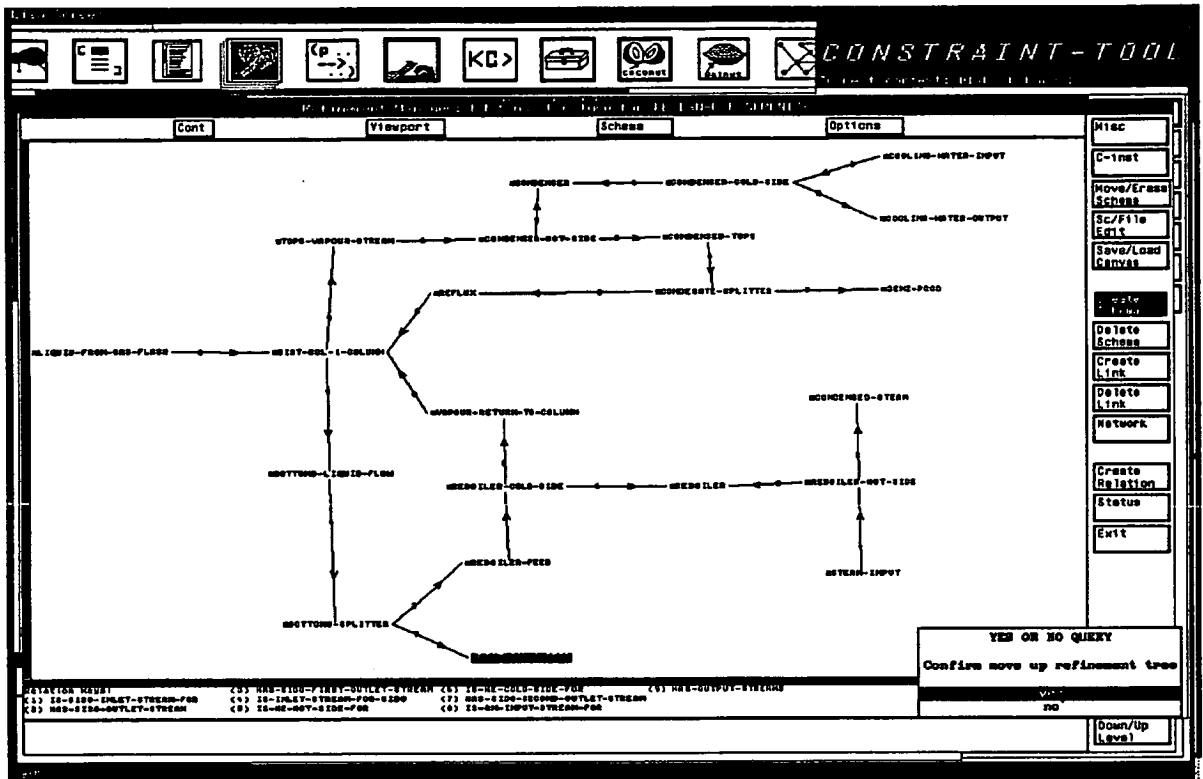


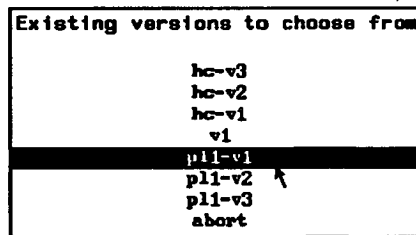
Figure 6.21: More detail on the refinement of the dist-col-1 schema

Figure 6.21 also shows the use of the “Up Level” command to move back up the refinement tree. This command is invoked by clicking mouse-m-1 on the “Down/Up Level” icon. A pop up menu then asks one to confirm the move. If one selects the yes option then one shall visit the context above. In the current example, this will resume the Palm instance already shown in Figure 6.19. Note that a new Palm is not started because one already exists displaying the correct context. For most tasks one can find out which workbenches already exist and what they are currently displaying.

6.2.2 Dealing with Versions

After having set up the refinement tree for the design problem, one can start tying the pieces together, in order to talk about the “current design”. Probably this will involve choosing between structural alternatives for whole design sections, or even for the whole plant. One will then have a complete structural design for large parts of the problem and can start to apply tools that rely upon this complete information. In particular, one can fill in the values of the necessary design parameters and perform numerical simulations upon this current design. This section describes the use of versions and version contexts which allow one to do this but still maintain alternative structures. Thus one can explore alternative parameterisations of the same design structure.

To start editing a version one must resume the Refinement Manager and click mouse-m-1 on the “Structure/Version” icon. One will then be prompted as to whether to start a new version or to resume an existing one. For the current example suppose that the latter is chosen, which causes the following menu to appear⁴:



Suppose that version “pl1-v1” is selected from this menu. The only immediate effect of this will be to change the Refinement Manager title banner to the following string:

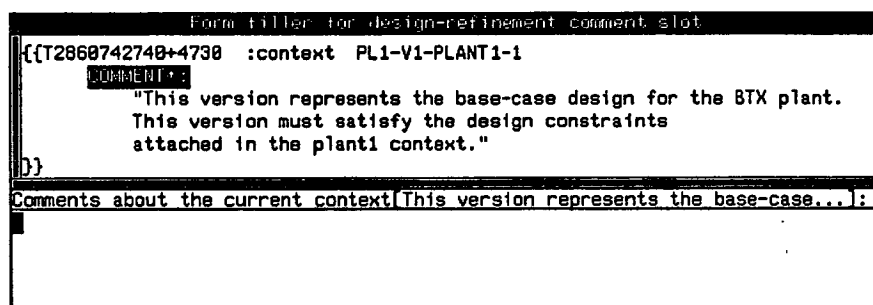
“Editing version pl1-v1 for DESIGN-REFINEMENT’s”

⁴This menu contains all existing versions, not just those connected with the plant1 design problem. This should be changed at a later date.

However, the real significance of the change is that one is now in *version* edit mode, rather than *structural* edit mode. This affects the behaviour of all the commands that have already been described. For example, the “Goto” command will now always visit a version context, even if it is invoked from a structural context.

The commands that one can use in version edit mode are highly analogous to their counterparts when in structural edit mode. Indeed, the user will access the same icons for them both. However, the prompt messages will have changed slightly because the version commands tend to be more complicated. One can switch back to structural edit mode at any time by resuming the Refinement Manager and clicking mouse-l-1 on the “Structure/Version” icon.

Now suppose that one wants to look at the top level of the plant1 problem for version pl1-v1. This context will automatically have been called “pl1-v1-plant1-1” and will be an immediate child of the plant1 context, so it should be easy enough to locate it on the context tree. Consequently, it is fairly simple to use the context editor to make it the current context. To examine it one could then resume the Refinement Manager and click mouse-l-1 on the “Comment/” icon. This allows one to see a comment describing the current context:



Form filler for design-refinement comment slot

```
{T2860742740+4730 :context PL1-V1-PLANT1-1
  COMMENT :
    "This version represents the base-case design for the BTX plant.
    This version must satisfy the design constraints
    attached in the plant1 context."
}
```

Comments about the current context (This version represents the base-case...):

This comment is stored in the refinement prototype schema in the context concerned. Thus one can store a unique comment for each context. This applies to structural as well as version contexts.

Now suppose that one wants to use Palm to edit versions instead of structure. As before, one should resume the Refinement Manager in the context of interest and then ask to "Goto" it. An appropriate instance of Palm will then be created or resumed. However, there are some differences to the situation previously described:

1. One does not have to be in exactly the context that is of interest. If the current context is not a version context used by the current version then the Refinement Manager endeavours to find an appropriate context at the same level.
2. There are no Palm displays explicitly associated with version contexts. Instead, the display for the parent structural context is used. This is because version contexts should only really be used to hold slot values, rather than to define new schemata or links.

The second of these differences means that when one asks to go to context `pl1-v1-plant1-1` it is possible to re-use the existing Palm instance that was looking at the `plant1` context. This is done automatically but the Palm title banner is changed to avoid confusion.

Figure 6.22 shows the result of calling "Goto" on context `pl1-v1-plant1-1`. In addition, it shows the use of Coconut to fill in slot values for the `tol-feed` schema in this context. This is the best editor to use since it automatically offers one all the slots that can be inherited by the schema concerned. In addition, one can easily invoke Coconut on any schema in a Palm display.

Suppose that one uses the Coconut on Figure 6.22 to put the value 6.0 (bar) into the pressure slot of the `tol-feed` schema. This may result in some automatic constraint checks within the current context. In addition, however, suppose that the "pressure" slot is involved in at least one Propagating-relation. In this case, the new slot value will be automatically copied to all other version contexts which belong to the current version and contain the `tol-feed` schema. Thus one may also get automatic constraint checks in other version contexts. For example, this may result in the checking of the constraint in Figure 6.17, which would be found to be violated. One would therefore get

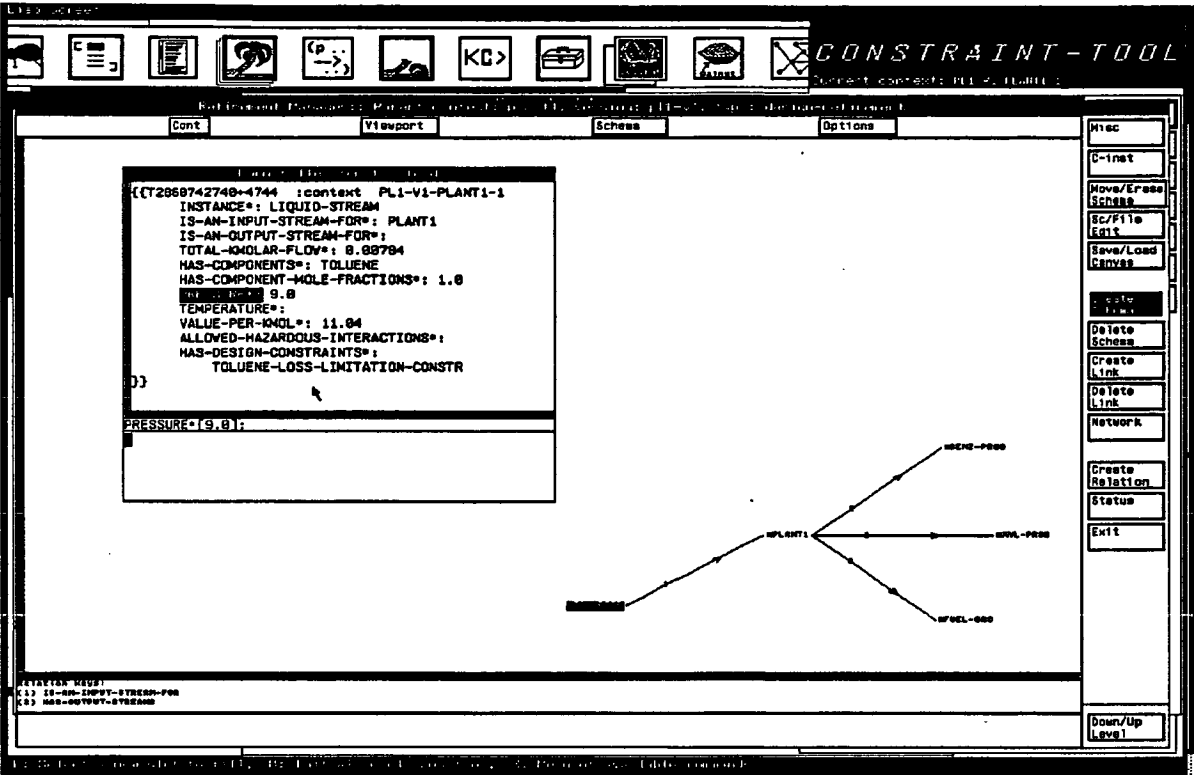


Figure 6.22: Using Palm and Coconut to fill in the tol-feed schema

the following constraint violation message if this constraint was set up for automatic checking:

Constraint Violation

Violation of constraint TOL-FEED-PRESSURE-CONSTR when applied to schema TOL-FEED in the context PL1-V1-PLANT1-FEED1-1.

The constraint description was as follows:

This constraint just checks that the pressure of the feed stream is high enough to not have to pump it into this section. This is an example only.

Type Any Character To Continue

In addition, there will probably be several constraint checks which can't be completed because there is not enough information. If one did nothing about these then one would get LISP errors from type mismatches; for example, by trying to use the value NIL in an arithmetic comparison. To avoid this the Constraint System watches out for missing information and may pop-up a warning message as follows:

Missing Information
Not enough information to apply constraint STREAM-OF-LIMITS-CONSTR-2 to schema TOL-FEED in the context PL1-V1-PLANT1-FEED1-1
Type Any Character To Continue

Both of these behaviours can be changed to print a different message or to take some entirely different action. For example, if one gets tired of reading pop up messages about missing information then the behaviour could be changed to write them to a file instead. This is accomplished by resuming Walnut and clicking mouse-m-1 on the "C/D Options" icon. This creates an instance of Coconut which allows one to edit the options for automatically checking constraints. An example edit is shown in Figure 6.23.

The options in this edit only apply to the automatic checking of constraints but there is a corresponding editor for options concerned with explicit constraint checks. In addition, the option change only applies from the context in which the Coconut was invoked. In Figure 6.23 the user is editing options in the plant1 context so the changes apply throughout the plant1 context tree. However, they do not apply to the heating-circuit tree or the anthra-plant tree. Furthermore, one can even make the change applicable to just the context in which it was made, rather than also to the ones below it. Thus one really can partition constraints to behave differently for different types of design problem.

Now suppose that one needs to make several changes to the design at once. For example, one might have obtained a set of flow values for the stream schemata in Figure 6.22, perhaps from a simulation package. Whilst one is filling in these values the design is known to be in an inconsistent state but after completion all constraints should be

```

DESIGN-CONSTRAINT-CHECKING-DEMON editor: context = PLANT1
((T2868742748+4756 :context PLANT1
  DEMON-VIOLATION-ACTION* DEMON-CONSTRAINT-VIOLATION-WARNING-FN
  DEMON-MISSING-INFO-ACTION*
    DEMON-CONSTRAINT-WRITE-MISSING-INFO-TO-FILE
  ACCESS*: ADD-VALUE ADD-VALUES NEW-VALUE NEW-VALUES REPLACE-VALUE
    REPLACE-ANY-VALUES REPLACE-VALUE-IF CSHEMA
  INCLUSION-FN*: DESIGN-CONSTRAINT-INCLUSION-FN
  EXCLUSION-FN*: DESIGN-CONSTRAINT-EXCLUSION-FN
))
nil, fn/S or (:queue fn/S) DEMON-CONSTRAINT-VIOLATION-WARNING-FN:

```

Figure 6.23: Editing automatic checking options in the plant1 context

satisfied. Consequently, instead of performing an automatic check as soon as a value is inserted, one needs to place it on a queue for later attention. This can be done by editing a further constraint option called *checking-action*. For implementation reasons this does not appear on the editor already shown in Figure 6.23. Instead, one must edit the ‘explicit checking options’, by resuming Walnut and clicking mouse-1-1 on the “C/D Options” icon. One can then change the checking-action to “:queue”, as shown in Figure 6.24.

The changes made within this editor interact with contexts in the same manner as did the editor shown in Figure 6.23. For a complete description of the options that appear in these figures the reader is once again referred to the Constraint Tools Manual [112].

Having initiated queueing of all automatic checks one is now ready to insert the new slot values. One way of doing this is to use the Palm instance in Figure 6.22 to invoke Coconuts on the stream schemata. Alternatively, one could make the changes by typing LISP expressions into a Kc Listener viewport. The latter is probably quicker for an experienced user when there are many changes to be made to many different schemata. However, one nearly always has both a graphical method and the LISP alternative

```

Constraint options editor; context = PLANT1
{{T2868742740+4762 :context PLANT1
  CHECKING-ACTION*: T
  VIOLATION-ACTION*: CONSTRAINT-VIOLATION-WARNING-FN
  MISSING-INFO-ACTION*: CONSTRAINT-MISSING-INFO-WARNING-FN
  CURRENT-CHECKING-QUEUE*:
  CURRENT-VIOLATION-QUEUE*:
  CURRENT-CHECKING-DEMON*: DESIGN-CONSTRAINT-CHECKING-DEMON
  CHECKING-QUEUES*:
  VIOLATION-QUEUES*:
  CHECKING-DEMONS*: DESIGN-CONSTRAINT-CHECKING-DEMON
  MISSING-INFO-SPY-POINTS*: (GET-VALUE LAMBDA (VAL) (NULL VAL))
                           (GET-VALUE-IF LAMBDA (VAL) (NULL VAL))
                           (TRACE-PATH-AND-RETURN-ANSWERS LAMBDA
                           (VAL)
                           (EQUAL VAL :MISSING-INFO))
}}
[. nil or :queue[T]:
:queue

```

Figure 6.24: Editing explicit checking options in the plant1 context

available.

After making the changes one can then examine the queue to see what constraint checks were requested. To do this one must resume Walnut and click mouse-1-1 on the "Checking/Violation Q's" icon. One is then prompted as to whether one wishes to "Examine", or to "Fire" the queue. Suppose that the former is selected. One would then get a pop-up message such as the following:

```

CONSTRAINT CHECKS ON QUEUE

((PL1-V1-PLANT1-1
  (('XYL-PROD 'STREAM-REVERSE-FLOW-CONSTR
    'PL1-V1-PLANT1-1)
  ('XYL-PROD 'XYL-PRODUCT-RATE-CONSTR
    'PL1-V1-PLANT1-1)
  ('FUEL-GAS 'STREAM-REVERSE-FLOW-CONSTR
    'PL1-V1-PLANT1-1)
  ('FUEL-GAS 'FUEL-GAS-LIMITATION-CONSTR
    'PL1-V1-PLANT1-1))))

Type Any Character To Continue

```

This only shows automatic checks that were requested in the `pll-v1-plant1-1` context because that is where the “Examine” command was called from. If it was called from the `plant1` context, however, one would see all the checks that were requested as a result of the changes just made. Similarly, one can fire all the checks at once or just those for a subset of the context tree.

Having examined the constraint checking queue, one may now desire to further change the behaviour before running any checks. For example, one may decide that there are too many checks to deal with by pop-up messages. Consequently, one may want to write violation messages to a file instead. Unfortunately, the checks on the queue always use the violation action that was in place when they were requested. Thus one cannot change their behaviour after the fact. One would therefore have to consider this possibility before queueing the checks and then ensure that the violation function had sufficient flexibility.

This concludes the description of how to edit version `pll-v1` and to change constraint checking behaviour whilst doing so. At any time one can switch to editing a different version or go back to editing structure. The Refinement Manager will always keep track of which context it should be looking at for a particular design level, under the current edit mode.

Here is the `benz-prod` stream under version `pll-v1`:

```
{( BENZ-PROD
  INSTANCE: LIQUID-STREAM
  IS-AN-OUTPUT-STREAM-FOR: LIQUIDS-SEP-SECT PLANT1 PLANT1-SEP-SECTION
  IS-SIDO-FIRST-OUTLET-STREAM-FOR: DIST-COL-1
  IS-SIDO-SECOND-OUTLET-STREAM-FOR: CONDESATE-SPLITTER
  TOTAL-KMOLAR-FLOW: 0.00319
  HAS-COMPONENTS: BENZENE
  HAS-COMPONENT-MOLE-FRACTIONS: 1.0
})
Type Any Character To Continue
```

The first four of these slots were filled in within structural contexts. Thus they will be

the same under every version, at a given design level, though more links may exist at lower levels of the tree. In contrast, the last three of the slots may vary between versions but will be the same throughout a given version (i.e. in every context belonging to that version). Thus there is indeed a strong distinction between data that is filled in within structural and version contexts.

6.2.3 Saving all changes

Before leaving this example it remains to be shown how one can save the results of a Knowledge Craft session. The ability to do this makes the Constraint Tools System somewhat different to other research products, such as PIP [48], which assume that design will be done in only one or two sessions. This assumption is limiting because design is typically a long, ongoing process. Thus one requires sophisticated and efficient ways of handling the large volume of incoherent data that builds up. The Constraint Tools System does not completely solve this problem; indeed, it is really rather inefficient. However, it does at least provide a way of experimenting with the use of Knowledge Craft as a tool for continuous use during a design.

Saving \$root-context information

The \$root-context will probably contain several schemata hierarchies that are defined by a user or an application developer for a particular domain. These can be edited during a Knowledge Craft session using Palm and other workbenches. They can then be saved using the new Save Manager that has been provided. This facility allows one to use a schema restriction to specify the contents of a file; for example, one can specify that a particular file should hold all the schemata on a given hierarchy. In addition, one can also perform a save interactively or by the use of a LISP function.

For the examples described in this chapter, the author used the interactive feature to set up the files initially and then wrote a LISP function to save them all at once. From then on, all the prototypical schemata were saved by typing the following expression into a Kc Listener viewport:

(ant-save-root-context)

This function is defined in Appendix E. From this one see that the user or developer has to do a bit of work to set up his own design environment. However, all the necessary information can be found in the complete version of the Constraint Tools Manual, which describes the new Save Manager in detail.

In a real company environment it seems likely that a system developer would be required to set up and manage a company specific version of the Constraint Tools System. In this way the end user could avoid having to understand some of its more flexible features. Indeed, perhaps the end-user's view of the system would be much like the view which he already has of the operating system he uses. An operating system such as UNIX is extremely complicated but an end-user only needs to understand a fraction of it to get on with his work. However, what he does require are support personnel who can tailor his environment to suit his own needs. One can envisage a similar relationship between the end users of a design environment and the systems personnel to support it.

There is a further question here, which is concerned with who is capable of changing, or allowed to change, which types of knowledge. One strategy would be to have specialist personnel who make all changes to prototype information (i.e. \$root-context information). Thus a design team would be restricted to creating instances of pre-defined schemata. A similar strategy could be taken with constraint definition, so that a designer is never required or allowed to write a constraint.

These restrictions have not been enforced in this thesis work, since the aim was to provide a tool that was easy enough to be used directly by process designers. This would allow them to create and attach additional constraints on a plant specific basis, thus capturing "one-off" pieces of information as well as background constraints. Unfortunately, experience has shown that the resulting tools are probably too complicated to be used by non specialist personnel. In reality, therefore, many layers of system development would have to exist between a domain independent design environment and a design team in a particular company. In addition, some of these layers might intentionally have to restrict the flexibility of the initial system.

Saving Constraint System Changes

The information managed by the Constraint System is easily saved by resuming Walnut and clicking mouse-1-1 on the "Save/Clean" icon. However, one is first asked to confirm the save, by means of a pop-up menu such as the following:

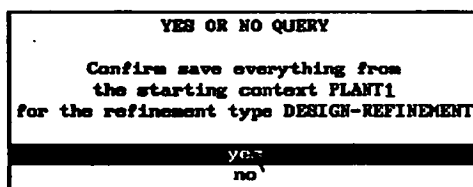
YES OR NO QUERY	
Confirm perform total constraint system save, from the context \$ROOT-CONTEXT, with save-sub-contexts set to T	
yes	
no	

This menu displays the current context, in which one invoked the save, as well as the current value of the *save-sub-contexts* constraint switch. If this switch has the value T then the system will consider saving the current context and all its descendents, otherwise it will just consider saving the current context. The switch value is easily changed using Walnut.

The Constraint System maintains a record of all changes that have been made to its information within the current session. It uses this to avoid resaving a file unless it will definitely contain some changed information. Two files are used for each context, their names being generated automatically. These files can contain LISP expressions as well as schemata definitions. Using these the system can faithfully recreate most of the Constraint System data types, such as queues and automatic constraint checks, in the next Knowledge Craft session. See the Constraint Tools Manual [112] for more details.

Saving the refinement tree

The schemata on the refinement tree are saved by resuming the Refinement Manager and clicking mouse-1-1 on the "Save/Switches" icon. First one is asked for the type of save that one wishes to perform. The options are to save everything, to just save the structural contexts, to just save a particular version or to just save the current context. One is then asked to confirm the save by a pop up menu such as the following:



Which actual schemata are saved depends upon information in the refinement prototype, which in the current example is called design-refinement. In particular, a schema restriction is used to define which schemata are interesting to a given type of refinement. For the current example one is interested in the the five hierarchies mentioned on the list of Fixed-menus in Figure 6.2. Consequently, the system will save any schema that is an instance of a prototype on one of these hierarchies. This definition is easily changed; for example, if one wants to create and manage an additional hierarchy.

6.3 An Example for the design of Batch Plants

The Constraint Tools System has recently been evaluated by use during a large undergraduate design project. The aim was to use it right from the early stages of the design, which requires the representation of the initial design assumptions that affect the economic viability of the project. The evaluation was carried out by an undergraduate student who was unfamiliar with Knowledge Craft and the Constraint Tools System. From this, it was hoped to learn:

- The type of background representations that are needed for process engineering.
- The types of things that one can usefully say about a design.
- How difficult it is for a naive user to pick-up the concepts involved and to learn how to use a complex but highly interactive system such as Knowledge Craft.

This evaluation was conducted during 1989/1990 and written up as a technical report [56]. The final conclusions were somewhat disappointing, particularly from the point of view of system complexity and user friendliness. Partly this was due to a number of “bugs” and efficiency problems that it should be possible to iron out. However, there still remains the serious question of who is capable of using such a complex piece of software. Really the AI-toolkits have still to find their place in the overall software spectrum. In particular, more work needs to be done on “downloading” an application to a less complex and flexible form.

The under-graduate design project concerned a batch plant, whose purpose was to upgrade an existing anthraquinone product to 1:5 and 1:8 dihydroxyanthraquinones. All the examples up to this point had involved continuous plants so some representation changes were necessary to start things off. For example, direct analogies were made between *streams* and *hold-ups*, and between *systems* and *processes*. This allowed the batch plant problem to be managed in the same way as previous examples for continuous plants. In particular, it was still possible to display design networks in a hierarchical manner, using Palm workbenches. However, this time it was the processes which were refined down the context tree, eventually being split into parts small enough to require no further elaboration.

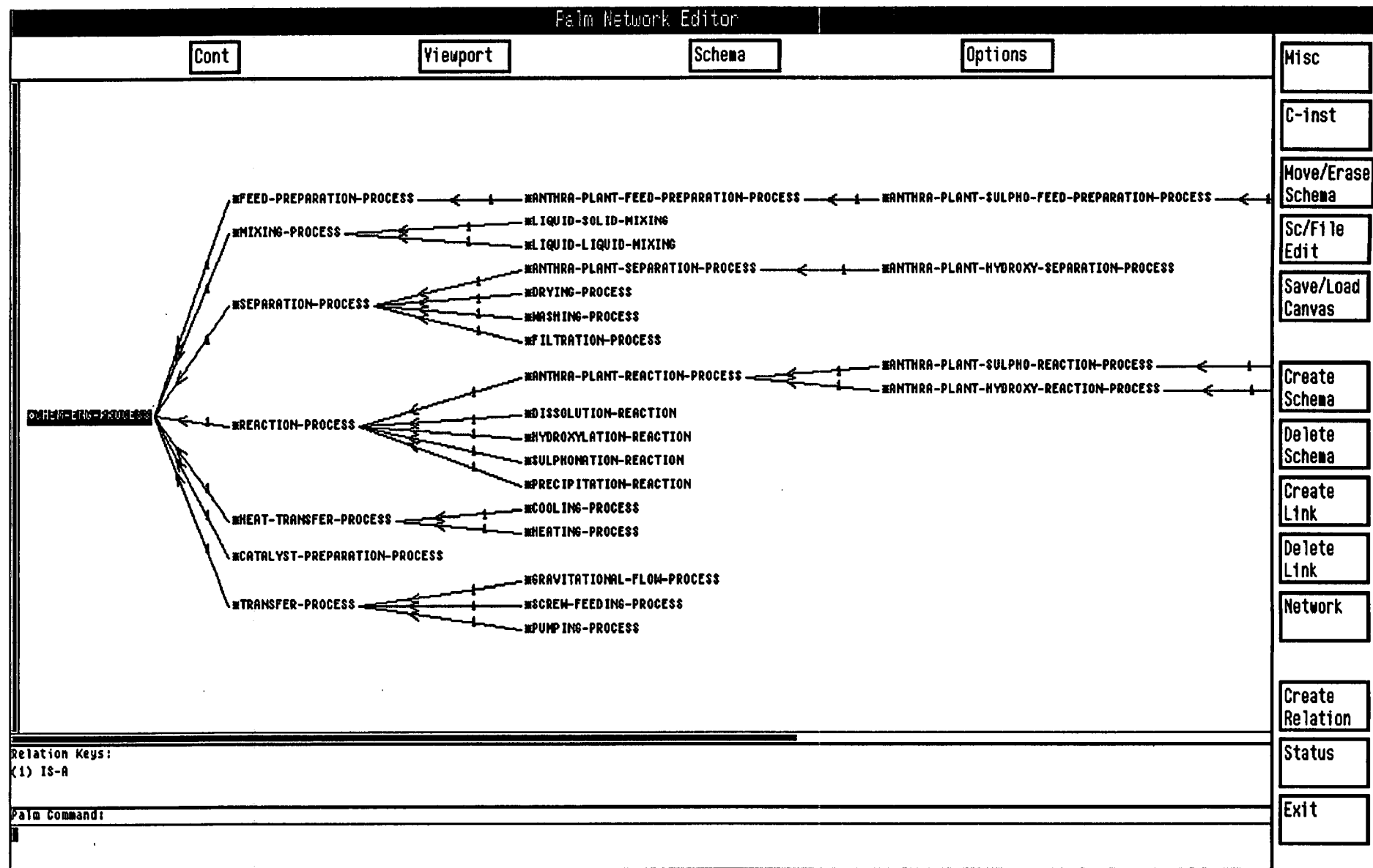


Figure 6.25: The chem-eng-process hierarchy

For example, a complex process called disulphonation is eventually split into separate mixing, heating, reaction and transfer processes. To do this one needs to define a hierarchy of prototypical processes, such as that shown in Figure 6.25. This hierarchy deals with unit operations, rather than with the physical items of equipment used to carry out those operations. Indeed, it is better if items of equipment do not get mentioned until far down the context tree. In this way, one can delay decisions about scheduling and vessel allocation. This was taken to the extreme in the evaluation project, since an attempt was made to only allocate processes to equipment within version contexts. This would then allow one to use different versions to explore alternative production schedules which use the same equipment. However, there were some problems with this approach, mainly for implementation reasons.

Figure 6.26 gives an example of a Palm display solely concerned with hold-ups and processes. The associated context is called 'new-sulphonation', which can be located on the tree already displayed in Figure 6.8. Figure 6.26 also shows a Fixed-menu which lists the plant specific constraints attached at this level.

Obviously the major difference with representing a batch plant, rather than the steady state design of a continuous plant, is the need to represent time and sequence. This is achieved by a 'time interval' definition of hold-ups and processes. In particular, each hold-up or process is not considered to be continuously active but rather exists for a definite length of time at periodic intervals. This means that a hold-up does not persist through several processes but rather is a volume of matter which exists in a certain state, in a certain location, for a certain interval of time. Correspondingly, a process takes a number of input hold-ups, perhaps at different times, and converts them into a number of *different* output hold-ups. This conversion takes place even if the process simply moves the volume of matter to a different location; it is still an entirely different hold-up. Thus a hold-up begins to exist as soon as a process has finished creating it and ceases to exist as soon as a new process starts to use it⁵. This is represented by introducing the following slots:

⁵This representation actually still needs a lot of development; for example, there is the question of how to characterise what is in a vessel from the moment that filling begins to the moment when it ends. It cannot be a hold-up as just defined because it is not in a steady state. However, one still needs to know when its there for the purpose of phrasing constraints.

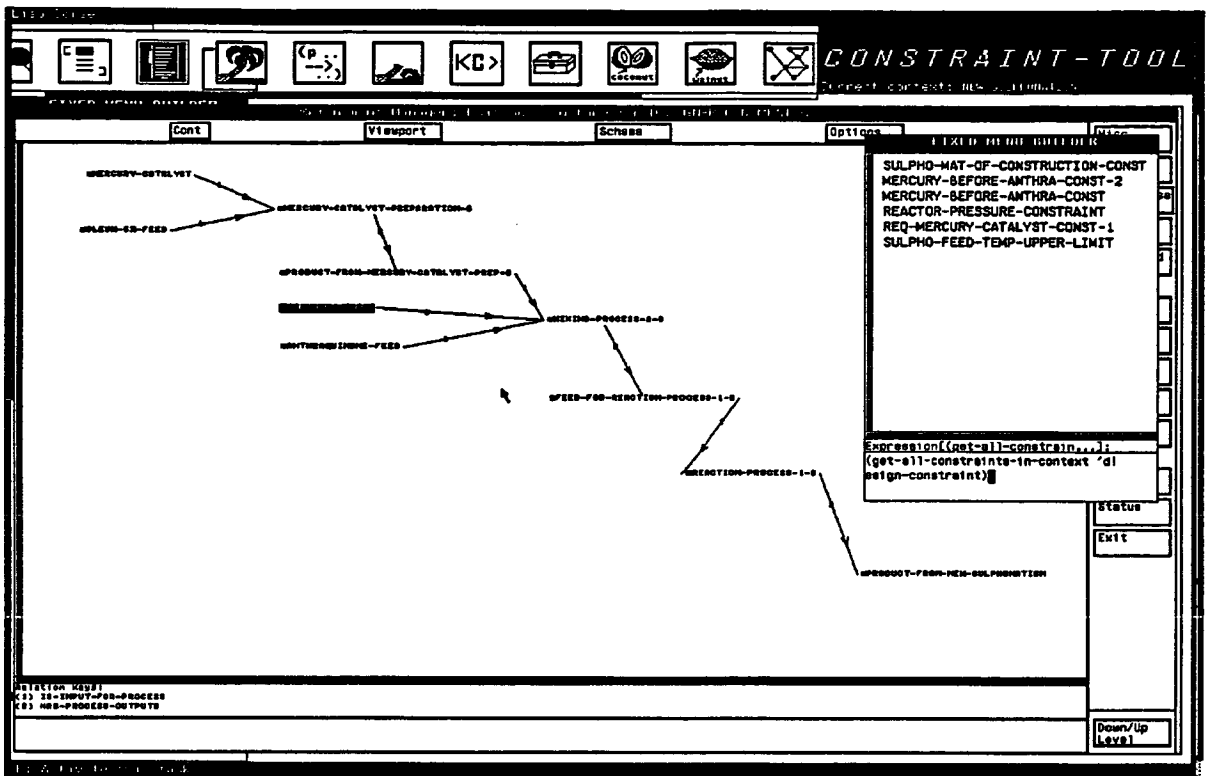


Figure 6.26: A Palm display just showing hold-ups and processes

initial-start-time which defines the time at which the hold-up or process first appears.

duration which defines how long the hold-up or process exists once it is active.

cycle-time which defines the time from the initial-start-time to the next time at which the hold-up or process appears.

From this information one can deduce all the time intervals, from the initial-start-time to infinity, for which a particular hold-up or process is active. This allows one to phrase constraints that account for time and sequence. For example, one certainly wants to prevent the allocation of the same hold-up to more than one vessel or of conflicting processes to the same vessel during overlapping time intervals. In addition, one may

want to place restrictions on the order in which processes occur. All these can be done using these extra pieces of information.

Figure 6.27 presents an example of a plant specific constraint of this nature. The problem is to ensure that the "mercury-catalyst-preparation-b" process is always done before the "mixing-process-2-b" process. To do this one must check that the constraint is satisfied when the processes first occur and that the processes have the same cycle time; the latter test ensuring that the time interval between the processes cannot change. Thus one is able to phrase and check the constraint without having to run a batch simulation for a large number of time steps. This is in contrast to the normal methods of batch scheduling using optimisation techniques (e.g. see [52]). To be fair, however, this is a specialised case; since it will not always be possible to phrase allocation or scheduling constraints in this manner.

```
{mercury-before-anthra-const
  is-design-constraint-for: mercury-catalyst-preparation-b
  description:
    "This checks that the anthraquinone is always added
    AFTER the mercury catalyst. If this is not done, the
    acid group substitutes beta instead of the required alpha."
  external-form:
    "(and
      (< #!~|initial-start-time|?
        #!mixing-process-2-b|initial-start-time|?)
      (= #!~|cycle-time|?
        #!mixing-process-2-b|cycle-time|? ))"
  local-slots-involved: initial-start-time cycle-time
  is-inverse-constraint-for: mercury-before-anthra-const-2
  })
```

Figure 6.27: An example scheduling constraint

This concludes the discussion of this example. For more information the reader is referred to the original report [56].

6.4 Summary

This chapter has presented extended examples of how the Constraint Tools System may be used in a design situation. It began by showing how one can edit background knowledge, which is stored in the \$root-context. Whilst doing this it discussed five hierarchies that have been created for the process engineering domain. It was careful, however, to limit the intellectual claims made about these classifications. Indeed, they were shown to be primarily for implementation purposes, rather than being based on a task independent view of process engineering concepts. This was partially justified by discussing the difficulty of phrasing constraints in isolation i.e. without the rest of the design system in place. In particular, the discussion centred on the problem of being able to make up arbitrary slot names; and the resultant problem of updating the values in such slots when design changes occur. This is referred to by Struthers [99] as the problem of 'arbitrary tokenism'.

The chapter then went on to show how one can edit the refinement hierarchy for a continuous plant example. This time the discussion was concerned with contexts below the \$root-context; and primarily with instance schemata rather than with prototypes. Firstly it was demonstrated that one can traverse and display structural contexts using Palm workbenches and Fixed-menus. There was then a discussion of some example constraints and how one can use contexts to partition their application and automatic checking.

The discussion then turned to the distinction between structural and version contexts. This was shown to be very important as it allows one to explore alternative parameterisations of the same basic structure. Furthermore, the behaviour of the Refinement Manager changes subtly according to whether one is in structural or version edit mode. This was demonstrated during a description of how one can also use the Refinement Manager and Palm to edit version contexts. In addition, there was a brief description of the use of Propagating-relations to automatically propagate data between all the contexts associated with the same version. It was also shown that constraint checking behaviour can be changed dynamically; for example, in order to write violation messages to a file instead of popping them up on the screen.

The description of the continuous plant example finished by showing that one can save all the changes made during the current Knowledge Craft session. This distinguishes this work from projects such as PIP [48], which assume that a preliminary design will be done in only one or two sessions.

The final section in this chapter briefly described the evaluation of the Constraint Tools System on a project for the design of a batch plant. For this case study it was necessary to invent a representation that was analogous to the use of systems and streams for a continuous plant, but which could also capture time and sequence. This was successful but the overall evaluation was somewhat disappointing. Partly this was due to a number of teething and efficiency problems. However, it also appears that the system is simply too complex for non-specialist personnel. This has implications for the original intention, which was to allow a designer to write project specific constraints, as well as using ones that have been pre-defined. In reality, therefore, it appears that a company would have to designate a special team to develop and maintain a design environment appropriate to its needs. Thus there would be many layers of system development between a domain independent tool and the eventual end user. Furthermore, some of these layers would intentionally have to restrict the flexibility of the initial system.

This concludes the current chapter but the reader is reminded that Appendix F has more to say on the definition of Propagating-relations and the use of CRL-OPS rules. For the main part of this thesis, however, the discussion will now turn to a comparison of the 'modelling' aspects of the Constraint Tools System with the features provided by some recent AI-based design & modelling environments.

“When I use a word,” Humpty Dumpty said, in a rather scornful tone, “it means just what I choose it to mean - neither more nor less.”

Alice's Adventures in Wonderland

Chapter 7

Decomposition and Refinement in other Modelling Environments

This thesis work has explored many representational questions concerned with how to partition knowledge, represent part-whole relationships and explore design alternatives. These issues are also addressed by some new, AI-based modelling environments, which were briefly described in Chapter 1. It is now worth expanding on this previous discussion, since the Constraint Tools System has now been discussed at some length. In particular, the discussion is now able to turn to the common representational themes between all these environments. Whilst doing this, the reader should bear in mind that this work has not been directly concerned with modelling. Thus the issues may be approached from a rather different perspective. For example, because this work is concerned with requirement constraints, rather than with equations, it has not had to be quite so concerned with how to maintain the consistency of inherited information. This has resulted in a rather different treatment of contextual information than that in systems such as MODEL.LA.

7.1 MODEL.LA

MODEL.LA (pronounced 'modella') is a modelling environment created by George Stephanopoulos' group at MIT. It is described in two large internal reports [38, 39] written in 1989. The biggest contribution of this work is a clarification of the 'semantics' of many of the representational constructs that this thesis has discussed. For example, the authors clearly define what they consider to be the minimum set of semantic relationships needed to represent process design. In addition, they discuss their own view of the nature of contexts and the communication that is needed between them. Within Knowledge Craft one would probably approach these problems rather differently but their work gives us a good basis for discussion. Unfortunately, the papers are not so strong in showing how information is accessed in MODEL.LA or how the system would be used in a design situation.

MODEL.LA is built on top of the AI-Toolkit KEE, which obviously provides the functions needed to access schemata, slots and values. However, although the papers describe the decomposition relationships in some detail they do not even mention the problem of mapping a piece of data between the several locations that need it. For example, they describe how to decompose an object called 'jacketed-cstr' into a jacket, a vessel and some 'ports' to transfer heat or material to the outside world. Furthermore, they show that the vessel and jacket are both constrained by heat-balance equations which are directly attached to these separate objects. However, they do not describe how each of these equations gets information from the object they are attached to, let alone from the other components of the overall model. This is a pity as it is exactly the sort of problem that the Constraint Tools System tries to address using Propagating-relations and the Path-Spec grammar.

Clearly the authors have constructed a solution to this problem or else their system just wouldn't work. Unfortunately, their failure to present it means that one cannot assess their representation by using the ground rules that were established in Chapter 4. Instead therefore, the following discussion will concentrate on the meaning of their semantic relations and the communication of contextual information.

7.1.1 Semantic Relationships

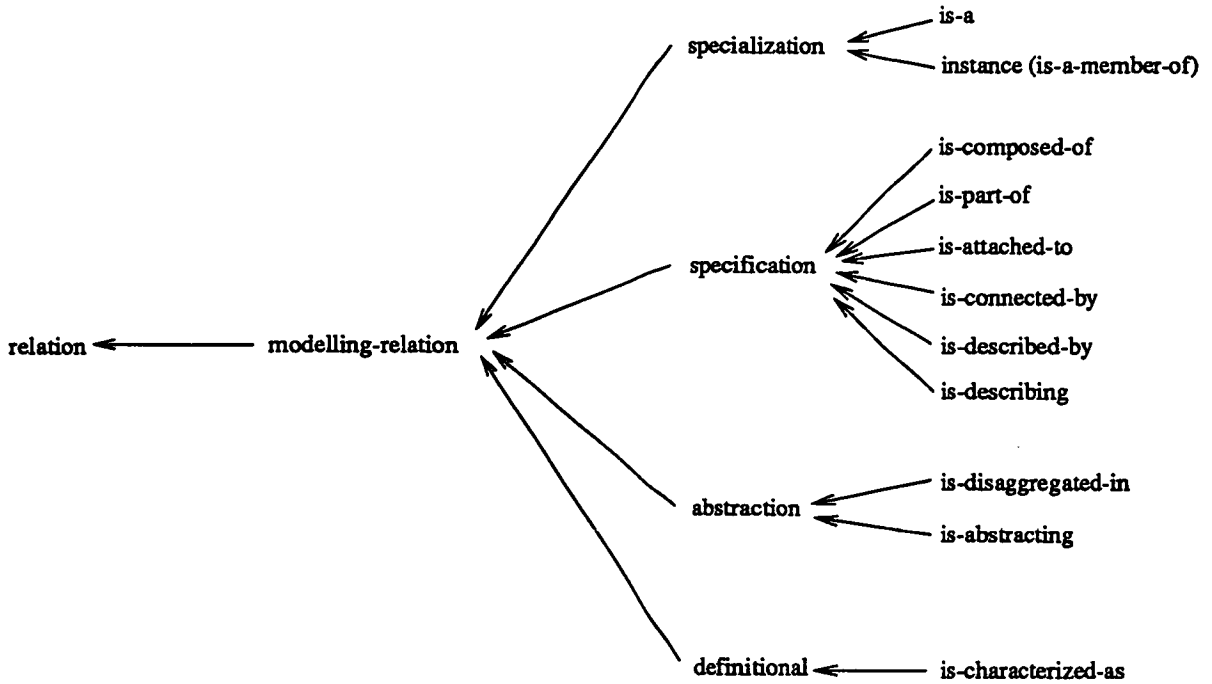


Figure 7.1: MODEL.LA's 11 semantic relationships

Figure 7.1 shows a small hierarchy that encompasses MODEL.LA's 11 semantic relationships. Of these, only the two specialization relations are explicitly represented in the Constraint Tools System as it stands, but most of the others could be easily included using Knowledge Craft's existing relation facilities. However, some difficulties may arise from the strong distinction that MODEL.LA makes between the "is-composed-of" link and the "is-disaggregated-in" link. The first of these is used to relate an overall model to its component parts, which is similar to the use of Propagating-relations to link together model components (e.g. the hot and cold sides of a heat exchanger). The second is used to relate two descriptions of the same system, at different levels of detail. This is therefore similar to the use of contexts to link together design refinements. On the surface, therefore, a mapping exists between the use of decomposition and refinement in the two systems. However, MODEL.LA makes a much stronger distinction between

the meaning of the two links, and places restrictions on their use accordingly. In particular, the 'is-composed-of' link is only ever used for modelling, rather than being used for more general decomposition purposes. Indeed, the components defined using this link exist solely for the purpose of modelling their parent system. That is, they are wholly dependent upon the parent and can have no meaningful existence if it is destroyed. By virtue of this dependency MODEL.LA knows exactly when such objects are needed and can create them all automatically via a model instantiation procedure. In contrast, within the Constraint Tools System there are currently no instantiation routines, which makes it more awkward for the end-user in this respect.

Figure 7.2 shows how one can impose some structure on the use of Propagating-relations by linking them into the hierarchy shown in Figure 7.1. Note, however, that there will be difficulties with the characterization of the abstraction relations because they are often not explicitly represented. Indeed, in most cases they are just implicit in the relationships between contexts, as recorded by the Refinement Manager. Such information is available at any time, but it only expresses the mapping from a schema in one context to the child context that refines it, rather than explicitly recording links between schemata in the two contexts. The implication, therefore, is that the refinement is made up of all the schemata which are newly created in the child context. Thus one must 'browse' the context in order to explicitly find the disaggregation objects. In addition, it will be necessary to filter out the schemata which have been pulled into this context from higher up the tree. Thus within the Constraint Tools System one does not have to explicitly create a link between a system and each component of its refinement; but this is at the expense of having to do more sophisticated reasoning to find out exactly what these components are.

The question now arises as to what is the effect of the relation structure that has just been imposed? Firstly, it could prevent the specification of inconsistent definitions, by the inheritance of domain checks, range checks and constraints. For example, one might force any 'is-connected-by' link to begin at a plant item and end at a stream. Secondly, one can use the semantic relations to answer queries about the current design network. For example, one might want to know all the points at which a distillation column called T-1 should connect to the rest of the design. This could be achieved by considering all the Propagating-relations which are descended from the "is-connected-by" prototype.

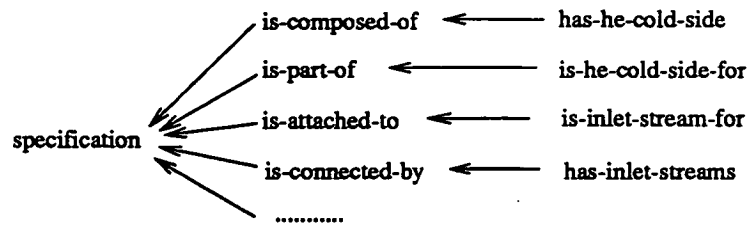


Figure 7.2: Linking Propagating-relations into MODEL.LA's relation set

One can then use this information to check the system specification.

MODEL.LA extends the flexibility of such queries by defining the relations to obey the laws of *transitivity* and *generalized transitivity*. The first allows deductions such as the following:

If A is-connected-by B and B is-connected-by C then A is-connected-by C

The second kind of law extends this, as in the following example:

```

IF
    system-A is-composed-of (system-B system-C ...)
    & system-B is-described-by (var-1 ...eqn-1 ..)
    & system-C is-described-by (var-c1 ...eqn-c1 ..)
    & .....
THEN
    system-A is-described-by (var-1 .. eqn-1 .. var-c1 ...eqn-c1 ..etc.)
  
```

Similar behaviour can be provided by using Knowledge Craft's transitivity grammar to full advantage. For example, Figure 7.3 gives a partial definition of the "is-connected-by" relation.

```

{{is-connected-by
  is-a: relation
  inverse: is-attached-to
  transitivity:
    (list (repeat (path is-disaggregated-in) 0 1)
          (repeat (path is-composed-of) 0 1)
          (repeat (or (step is-connected-by T)
                      (step feed T)
                      (step input-convective-ports T)
                      .....etc.)
                  1 inf))
}}

```

Figure 7.3: A possible definition of the is-connected-by relation

7.1.2 Multifaceted Modelling

MODEL.LA is said to allow for the multifaceted modelling of process systems by providing the following facilities:

- Multiple viewing of process models in terms of structure, topology and behaviour.
- Disaggregation of abstract models to more detailed ones and aggregation of detailed descriptions to more abstract ones.
- Contextual¹ description of alternative models for the same process.
- Controlled flow of information among the models at various levels or in various contexts, and detection of modelling conflicts.

¹Note, however, that one must be careful when people talk about “contexts” because there are so many different meanings for the word. Indeed, Knowledge Craft’s use of the term is rather implementation specific and not what is generally meant. Perhaps “Worlds” would have been a better term for Knowledge Craft’s versioning mechanism.

These features of MODEL.LA are built on top of KEE-worlds, which is the equivalent of Knowledge Craft's context mechanism. One can therefore anticipate many similarities between these aspects of MODEL.LA and the features provided by the Refinement Manager described in this thesis. This section will discuss the extent to which this is true.

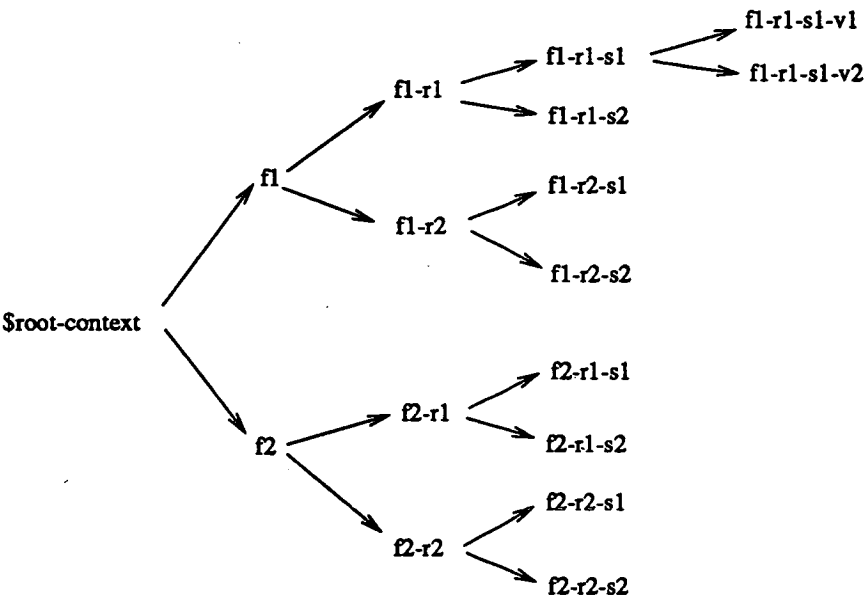
Both MODEL.LA and the Constraint Tools System use a hierarchical context mechanism to allow for the representation of alternative models or design refinements. However, the two systems differ in the extent to which data is automatically mapped and propagated. In particular, the Refinement Manager does this in order to achieve the multiple inheritance of contexts. This allows it to distinguish between structural and version contexts, which then allows one to consider alternative parameterisations of the same basic structure. In contrast, within MODEL.LA information is only allowed to go down the tree; nor do they allow for the multiple inheritance of contexts. The implication of this is that a complete design is not spread across the whole tree, as in the case of the thesis work described in previous chapters, but instead has to reside along a single branch. Thus each time one creates a new branch one is effectively considering a complete alternative design. This is very much the Jim Douglas approach to refinement [48], in which the whole design is taken through a series of successive levels.

This approach is unfortunate because it becomes very difficult to consider a large number of design alternatives. Most importantly, it will necessitate the replication of contexts which are holding virtually identical structural information. To see this, suppose that one has a design situation in which the following alternatives have been found to exist:

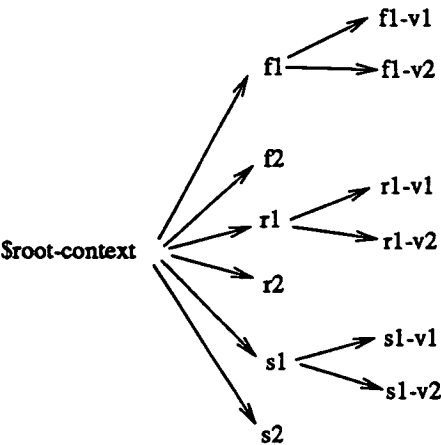
Feed section: 'f1' or 'f2',

Reactor section: 'r1' or 'r2',

Separation section: 's1' or 's2'.



(a) The context tree using the MODEL.LA approach.



(b) The context tree using the Refinement Manager approach.

Figure 7.4: Alternative uses of a context tree

Figure 7.4 then shows the alternative context trees that arise from the two approaches. The context names on these trees should be taken as reflecting the choices that have so far been made, and consequently the information that is contained within them. For example, in context 'f1' on tree (b) one has only chosen between the separation section alternatives and deferred commitment on the other sections. On the other hand, in context 'f1-r1-s1' on tree (a) one has completely specified the design by choosing alternatives f1, r1 and s1. In addition, the context trees show the exploration of two alternative parameterisations of the structure 'f1-r1-s1'. These two versions are represented by the contexts which end in 'v1' and 'v2' on both trees. On tree (a) there is only one of each such context because the structural alternatives have already been chosen in the parent context (i.e. 'f1-r1-s1'). However, in tree (b) there is a separate context for each piece of structure that appears in each version. These contexts still end in either 'v1' or 'v2' so one can still find all the structural choices that have been made for a particular version.

The context trees in Figure 7.4 can now be compared using criteria such as ease of management, duplication of information and ease of expansion. Firstly then, one should notice that tree (a) branches more than tree (b) and has several more contexts. Indeed, the method for constructing tree (a) is combinatorial, so an additional alternative for any of the sections would have necessitated the creation of half as many contexts again. Nor does this problem go away when one progresses to considering more trivial design decisions. For example, note that 's1' is duplicated in its entirety in 4 separate contexts, in order to explore all combinations of s1 with other parts of the design. This is unfortunate because all of these s1 contexts must be branched at the same time. That is, if the s1 choice is split into two further alternatives then two new contexts are needed for each appearance of s1. Thus, even for trivial design decisions, such as whether or not to use a thermosyphon reboiler within s1, one must still create eight new contexts! In contrast, for tree (b) one would simply create two new contexts to represent the two alternatives. These are made children of the single 's1' context.

This difference arises because tree (b) allows one to specify internal structural information as being essentially independent of the alternatives chosen for the upstream and downstream sections. For example, one can say that the separation task is essentially the same whether the reactor is pre-heated by a furnace or by some other means. Of

course, this only applies to structural information - the actual operating parameters of a system, including the physical size of its components, are very much dependent upon what happens elsewhere in the plant. This is allowed for by the distinction between structural and version contexts.

This is not to say that the approach taken in the Refinement Manager can completely escape all combinatorial problems. Indeed, if one wishes to explore every possible combination down to the level of putting in numbers, then the Refinement Manager will be worse off than MODEL.LA. This is because many different contexts are used to represent a single version and data must be propagated between them. The assumption, therefore, is that not every possible combination needs to be explored fully but one should have complete freedom of when to stop. This is achieved by providing for a "pick and mix" approach, which can get just as complicated as one wants it to. Furthermore, because alternative sections are explored using totally different contexts it should be easier to play around with design numbers and model individual sections in isolation.

There is a downside to this, however, as one will sometimes want to protect against the misuse of structural information that *does* depend upon upstream or downstream alternatives. For example, suppose that one has a design in which there are several alternative catalysts that might be used in the reactor. Furthermore, suppose that one of these catalysts just happens to be poisoned by benzene (an example only!). In this case one could phrase a constraint that under this alternative benzene is not allowed into the reactor. This might do the job but it relies upon the successful detection and prevention of operational errors. Instead, therefore, one might want to phrase a more general principle that benzene must not be used as a mass separating agent in conjunction with the catalyst. Unfortunately, this constraint would have to be created at a higher level, since it references data across the context tree. In this case it would then apply to all the other catalysts as well. Thus it is difficult to phrase constraints that are specific to a particular combination of structural alternatives.

There are a few other differences between the use of contexts within MODEL.LA and the Refinement Manager that it is worth mentioning. Firstly, data is not propagated automatically within MODEL.LA. Instead, the user must explicitly request for a partic-

ular piece of data to be obtained from a particular context. MODEL.LA does perform some data mapping, however, which makes this request easier. Nevertheless, the experience gained during this thesis work suggests that some automatic data propagation is essential for a usable system.

Secondly, when a refinement is performed in MODEL.LA the original system is replaced completely. MODEL.LA must then do some work to recreate existing semantic links. In contrast, within the Refinement Manager a child context is simply created and the new system defined within it. The original system still exists in the parent context (with all its links intact) and will be copied down to the new context if necessary. This simplifies the representation and allows one to maintain link consistency whilst exploring alternative refinements (as shown in Figure 6.16 in the last chapter). However, remember that this approach may be over-simplified because the Refinement Manager is not performing modelling. It can therefore allow the data further down the context tree to be the proper union of all the data higher up. It may be difficult to take a similar approach within a modelling environment such as MODEL.LA.

Finally, within MODEL.LA abstraction is allowed to go down the tree as well as up, whereas the Refinement Manager constructs a tree that is monotonic in detail. Indeed, it would not make sense for the Refinement Manager to abstract downwards because it does not regard the context tree as a progression in time. That is, the upper levels are not discarded as time progresses and the design develops. However, this does cause problems because one may sometimes have to split a context into two, in order to create a choice point that was not originally anticipated. This remains a rather difficult problem within Knowledge Craft.

7.2 ASCEND

ASCEND 3 [78] is an equation oriented modelling environment created by Art West-erberg's group at Carnegie Mellon University. Its main component is a modelling language with a clear and concise syntax and some object oriented features. The system is written in PASCAL but a C version is being prepared. As a modelling system

it is probably the cleanest of the ones that are described here.

The most basic structure in ASCEND is called a 'model', which describes a set of equations by the use of local variables. Often these variables will have a declared type, which is used for error checking. In addition, the notion of type is generalised in an object oriented fashion. For example, the variables can be declared to have a type which is itself a model. In this case equations will also be inherited from this other model. Furthermore, the variables local to the other model can be accessed by use of the appropriate pathname. Finally, one can establish correspondences between the variables local to the two models. All these declarations can be done using the following set of operators:

Refines: This operator is the equivalent of the 'is-a' link in Knowledge Craft in that it provides for inheritance and specialization down a model hierarchy. For example, the following defines the 'two-phase-stream' model to be a child of the 'stream' model:

```
MODEL two-phase-stream REFINES stream;
...
END two-phase-stream;
```

The inheritance is strictly additive so a structure or value assignment cannot be overwritten down the tree, only added to. This is also the approach that this thesis work has taken with constraints. However, no steps have been taken to enforce it for other types of information.

Is-a: This is the equivalent of the 'instance' link in Knowledge Craft, though in ASCEND instance names only have to be unique within a local model definition. Other models then access the data in these instances by the use of a pathname. For example:

```
dist-col-1.feed could represent the feed stream to a distillation column.
dist-col-1.feed.temp could represent the temperature in that stream.
```

If one replaces the use of a '.' by the use of a '|' then the similarity of this language to the previously described Path-Spec grammar should be obvious. Indeed, a

description of this ASCEND feature was the main inspiration for the author's work on this utility. Note, however, that specifications written using the Path-Spec grammar can branch to return a list of values, perhaps from different places. In this sense it is more flexible than its counterpart in ASCEND.

Are-alike: This operator is used to declare two variables to be of the same type. To allow this they must be coercible, which in ASCEND means that they must be instances of models which are on the same branch of a hierarchy. The higher instance will then be coerced to be of the same type as the lower one. In addition, any further coercions will affect them both. For example:

```
MODEL flowsheet1;
  sep1 IS-A separation-system;
  sep2 IS-A distillation-system;
  ... ..
  sep1, sep2 ARE-ALIKE;
                                (* Both now of type distillation-system *)
  ... ..
  sep2 IS-REFINED-TO binary-distillation-system;
                                (* Both now of type binary-distillation-system *)
  ... ..
END flowsheet1;
```

Is-refined-to: This operator is used to change the type of an instance i.e. to link it to a different parent. An instance can only ever move down a model hierarchy or else one could not enforce strictly additive inheritance.

Are-the-same: This operator is used to *merge* variables or structures so that equivalence is directly expressed through 'pointers'. For example, if one merges two variables, X and Y, then the final set of equations will use a single variable to represent either of them, rather than including $X=Y$ as an additional equation. One can also merge two structures, which will result in the merging of all their corresponding attributes. Furthermore, any substructures that they reference will also be merged, in a top-down fashion. If two structures are not of the same type then the 'are-alike' operator is automatically applied before the merge. If some of

the variables have already been assigned values then the merge routine will check for conflicts. The merge operator is probably ASCEND's most powerful feature.

One can compare ASCEND's use of the merge operator with the use of Propagating-relations described in this thesis. In particular, in both cases an object is split into component parts and data is mapped between them. However, in the Constraint Tools System data can be automatically propagated, whereas in ASCEND data mapping only occurs when the equation set is instantiated. Indeed, apart from when performing simple type checks ASCEND only uses data when the equation set is initialised and solved. In contrast, the Constraint Tools System is much more active so a user may see an immediate effect when he makes a change.

To see the similarity between the use of Propagating-relations and the use of the merge operator let us further consider the ASCEND definition of a two phase stream:

```
MODEL stream;
  nc is-a integer;
  f is-a flow;
  p is-a pressure;
  t is-a temperature;
  ...
END stream;

MODEL two-phase-stream REFINES stream;
  v, l is-a stream;
  nc, v.nc, l.nc ARE-THE-SAME;
  t, v.t, l.t ARE-THE-SAME;
  p, v.p, l.p ARE-THE-SAME;
  ...
END two-phase-stream;

MODEL flowsheet2;
  s1 is-a two-phase-stream;
```



```

...
END flowsheet2;

```

The instantiation of flowsheet2 will then result in the creation of one instance of 'two-phase-stream' and two instances of 'stream'. However, the 't', 'p' and 'nc' attributes of these three objects have been declared equivalent. Thus whenever one assigns a value to an attribute of one of the streams one automatically assigns the same value, to the same attribute, in the other two streams as well. Similar behaviour can be achieved using Propagating-relations by defining the objects shown below:

```

{{equilibrium-stream-proprel
  is-a: propagating-relation
  source-slots: temperature pressure nc
  mappings: (:source :source)
}}

;; -----

{{has-vapour-stream
  is-a: equilibrium-stream-proprel
  .....}}

{{has-liquid-stream
  is-a: equilibrium-stream-proprel
  .....}}

;; -----

{{two-phase-stream
  is-a: stream
  has-vapour-stream:
  has-liquid-stream:
  .....}}

```

As already described in the last section, however, the user must currently create all instances himself.

In conjunction with its 'refines' operator, ASCEND's merge operator can also be useful for the representation of design alternatives. Indeed, this can be done in a way that is highly analogous to the use of contexts within the Refinement Manager. For example, here are some ASCEND models that might represent the context tree from Figure 7.4(b):

```
(* The top level of the problem *)
```

```
MODEL flowsheet3;  
  feed is-a feed-section;  
  reac is-a reactor-section;  
  sep is-a separator-section;  
  ... ..  
  feed.out, reac.in ARE-THE-SAME;  
  reac.out, sep.in ARE-THE-SAME;  
  ... ..  
END flowsheet3;
```

```
(* The first level down *)
```

```
MODEL f1 REFINES flowsheet3;  
  (* This alternative preheats the furnace feed *)  
  heater1 is-a heater;  
  furnace1 is-a furnace;  
  ... ..  
  heater1.in, feed.in ARE-THE-SAME;  
  heater1.out, furnace1.in ARE-THE-SAME;  
  furnace1.out, feed.out ARE-THE-SAME;  
  ... ..
```

```
END f1;
```

```
MODEL f2 refines flowsheet3;
```

```
    (* This alternative doesn't preheat the furnace feed *)
```

```
    furnace2 is-a furnace;
```

```
    ... ..
```

```
    furnace2.in, feed.in ARE-THE-SAME;
```

```
    furnace2.out, feed.out ARE-THE-SAME;
```

```
    ... ..
```

```
END f2;
```

```
MODEL r1 refines flowsheet3;
```

```
    ... ..
```

```
END r1;
```

```
..... etc. ....
```

```
(* Constructing the versions *)
```

```
MODEL v1;
```

```
    f1-v1 refines f1;
```

```
    r1-v1 refines r1;
```

```
    s1-v1 refines s1;
```

```
    ... ..
```

```
    f1-v1.feed.out, r1-v1.reac.in ARE-THE-SAME;
```

```
    r1-v1.reac.out, s1-v1.sep.in ARE-THE-SAME;
```

```
    ... ..
```

```
END v1;
```

Thus it appears that ASCEND can represent design alternatives in a way that is comparable to the approach described in this thesis. However, one should note that the ASCEND 'contexts' and objects created in this way cannot be reasoned about in an explicit manner. ASCEND may therefore run into trouble because its inheritance is

strictly additive. In particular, at the detailed levels of the refinement tree one has no way of excluding more approximate models from higher up. This is bound to lead to equation sets which are inconsistent or overspecified.

This problem would also apply to Constraint Tools System in its current state. However, because one can obtain explicit information about the objects stored within a given context it should be possible to selectively filter out information that comes from higher up the tree. Thus there is at least the possibility of achieving multi-level reasoning and consistency at the same time. This really highlights the difference between ASCEND and the other systems that are being described; namely, that ASCEND is *just* a modelling environment so it could be hard to use in a wider design context. In contrast, the other systems are not just aimed towards the needs of a modelling specialist but also towards the process designer himself.

7.3 CLAP

CLAP (Combined Logic and Procedures) is an object-oriented toolkit developed by members of Jack Ponton's group at Edinburgh. Its most significant feature is that, in contrast to all the other systems that are being described here, its core language is Prolog, not LISP. This means that unification and backtracking can be used in the definition of objects and procedures, which is a powerful facility. In addition, it has an in-built tool for equation-based modelling which is broadly comparable to that of ASCEND. CLAP has an advantage, however, because it does not require strictly additive inheritance in the definition of models. Thus one can consistently construct models at different levels of detail. On the other hand, CLAP has neither a merge operator nor the facilities to construct a hierarchical context tree. It would therefore be difficult to extend it with a general refinement mechanism, such as that described in this thesis.

Within CLAP, equations are obtained and expanded in a top-down fashion, often beginning with a general conservation law. For example, one might start with a general representation of the overall heat balance and then specialise it according to how many

streams enter or leave the system. One would then expand the component parts according to the system being modelled. This will probably require one to find additional expressions, such as for the heat transfer within a heat exchanger. Furthermore, when one has such an expression one may want to be able to choose between supplying a value for the overall heat transfer coefficient or using the individual equations for the tube side and shell side coefficients. A user can do this within CLAP as he has complete control over how far an expression or term is expanded.

This gets around a classic inflexibility problem in traditional flowsheeting tools, which is that the user has the value for a compound term but the simulation program won't accept it. Instead it demands more detailed information in order to calculate the value itself. A likely scenario for this problem is with the simulation of existing plants, in which case one can often measure compound data, such as viscosity, easier than the stream compositions that are needed to calculate it. It appears that ASCEND is not designed to solve this problem since all possible equations are included in the final set i.e. expansion is complete. The user would thus have to construct different models for different expansions. This may be possible using the 'refines' operator described already but one could never expand an expression interactively.

Thus CLAP addresses some of the problems of hierarchical modelling, although it does not directly provide for the exploration of alternative design refinements. Neither does it address the mapping and propagation issues discussed in this thesis. Indeed, to do so would probably go against the original design of the system, as its author did not strongly approve of the use of demons and inheritance. CLAP therefore provides an example of a software philosophy rather different to that of the LISP based AI-toolkits. This is further discussed by Struthers [99] in his Ph.D. thesis. In addition, Hutton [41] has been applying the toolkit to the problem of process design.

7.4 ModAss

ModAss [67] is yet another modelling environment constructed using an AI Toolkit. This time it is based on Knowledge Craft so one might expect some similarities with

this thesis work. Indeed, ModAss provides a graphical tool for the multi-level viewing of a design, which is somewhat similar to the Refinement Manager. However, ModAss does not use contexts, or provide any other means of exploring alternatives, so the similarity is really just surface level.

The most notable features of ModAss are its integration with MACSYMA², which is said to be complete, and its use of a supervisory process³. This supervisor is based on the AKORN D blackboard system [66], which has already been described in Chapter 5. It is particularly intended for error checking but also performs some "well defined" programming tasks to tidy up loose ends. Unfortunately, few details are given in the paper, so it is difficult to further compare ModAss with the other systems described here.

7.5 Summary

This chapter has compared the Constraint Tools System with some recent AI-based modelling environments. Mostly it concentrated on the use of decomposition and refinement, as these are the areas in which there is much overlap. By virtue of this it has talked more about MODEL.LA and ASCEND rather than CLAP or other systems.

It began by discussing MODEL.LA and showing how many of its relation facilities could be fairly easily provided using existing Knowledge Craft features. It gave the authors credit, however, for clarifying the 'semantics' of the different relation classes. It then talked about their method of multifaceted modelling, which appears to require that the whole design is refined along the same branch, rather than being spread across the context tree. The disadvantages of this approach were discussed in terms of the duplication of information and the difficulty of expansion of the tree. In particular, the discussion noted that the tree is combinatorial so it is difficult to maintain all possible alternatives. The main reason for this was seen to be that one cannot specify the internal structure of a section as being independent of which alternatives are chosen

²MACSYMA is a LISP-based package for algebraic manipulation.

³The supervisor is run as a separate process using the multi-tasking facilities of SYMBOLICS COMMON LISP. This allows it to be continuously active and to transparently share data with other lisp processes. A similar facility is provided by the latest version of SUN COMMON LISP (3.0).

for the rest of the plant.

Next the discussion turned to ASCEND and showed how one could use Propagating-relations to implement their notion of *merge*. On the other hand, it also showed how their 'merge' and 'refines' operators could be used to simulate the use of contexts by the Refinement Manager. Thus there is some interesting common ground due to the power of the merge operator. However, there are also large differences in philosophy. In particular, in the Constraint Tools System data can be automatically propagated, whereas in ASCEND data mapping only occurs when the equation set is instantiated. Indeed, apart from when performing simple type checks ASCEND only uses data when the equation set is initialised and solved. In contrast, the Constraint Tools System is much more active so a user may see an immediate effect when he makes a change. Furthermore, in ASCEND one cannot reason about 'objects' and 'contexts' in an explicit manner. This may cause problems for hierarchical modelling since there is no way of overcoming strictly additive inheritance. Such problems are further discussed in the section on CLAP, which considers the top-down expansion of equations. Finally, there is a description of ModAss which is necessarily brief, as no more details were available.

Thus there are some interesting similarities and differences in the use of decomposition and refinement in current AI research projects. In addition, there are many issues of philosophy which remain to be fully explored. Paramount among these are questions concerning the use of strictly additive inheritance. Much interesting work remains to be done in this area. The final chapter will now make some recommendations as to the specific directions that this should take.

A conclusion is the place where you got tired of thinking
Arthur Bloch

Chapter 8

Overall Summary and Conclusions

This chapter concludes the main part of this thesis by presenting an overall summary and then making some recommendations about the direction of future research efforts. For the most part, the summary of each chapter is just a repetition of its concluding paragraphs. However, as the thesis has been so wide-ranging it was felt that it would be useful to have these compiled into a single section, which now follows.

8.1 Overall Summary

Chapter 1

This chapter began by defining the main area of interest as being design-stage Loss Prevention. However, it was quickly shown that this area cannot really be tackled without thinking about the wider context of the design problem. In particular, it was shown that one must provide a design structure to control the application of the “thousands of little checks” that are needed to ensure the validity of a design. There was then a discussion of the differences between *discrete* and *incremental* approaches to Loss Prevention. The conclusion was that the ideal approach would be the near-continuous assessment of design quality, in order to provide for inherent safety, together with discrete formal design reviews, which may eventually be reduced to just performing

an audit function. To facilitate this one must be able to reason about the design at many different levels of detail, perhaps even in the complete absence of numerical information.

The discussion then progressed to the notion of Integrated Process Design, which could provide the structure that was deemed to be necessary. Past attempts at solving this problem have concentrated on database technology. This has many shortcomings because of a basic mismatch between the data types and transactions needed for engineering and those used by the business community. In contrast, more recent work, particularly that of Struthers [99], has taken a knowledge based, rather than a databased approach. This time the data types may be adequate but it is the traditional database issues which are neglected, such as efficiency and concurrency control. There is therefore much benefit to be had from a synthesis of database and knowledge based technology. The work in this thesis does not address this issue but there is much database research which does. In addition, this thesis partly follows on from the work of Struthers by considering the problem of representing designer's intention. This is part of what he calls the problem of "maintaining the integrity of knowledge".

This chapter concluded with a review of previous attempts to apply AI techniques to the problems of Loss Prevention. This also included an initial discussion of some attempts to provide AI-based design and modelling environments.

Chapter 2

This chapter described a rule-based Prolog program which tried to copy the simulation side of HAZOP reasoning exactly. The central concepts were the use of a local, qualitative analysis and the use of rules to represent causality. Unfortunately, this combination was capable of only limited success. In particular, the approach ran into difficulties on problems with feedback of any kind, which occurs in all but the simplest of cases.

To understand the failings of the approach the underlying assumptions were formalised as three basic premises. Each of these were then invalidated by the presentation of counter examples. For example, problems were shown to arise from the use of a device-

centred viewpoint, rather than examining the flowsheet as a whole. A possible solution to this is the use of 'less obvious' rules, which model wider sections of plant. In addition, it was shown that further problems arise from the use of fixed, discrete levels to represent qualitative magnitude. In particular, it proves difficult to reason about maximum and minimum variable values in a general and consistent manner. In contrast, such concepts are easily represented using qualitative simulation because it allows one to distinguish between the direction in which a variable changes and its qualitative magnitude at a certain point in the reasoning process.

Chapter 3

This chapter discussed the similarities and differences between top-down and bottom-up methods of hazard identification. To do this it described in some detail a bottom-up technique called qualitative simulation and a top-down one called 'automatic fault-tree synthesis'. The main emphasis was on the former, as this is a novel AI technique not much explored in chemical engineering. Compared with Chapter 2 the central concept was now *constraint satisfaction*; indeed, causality is only used to help with the efficiency of this process. This allows one to resolve many of the feedback problems identified earlier. In addition, qualitative simulation provides a well formulated representation, which makes it easier to reason about limiting values and relative magnitudes. Unfortunately, severe efficiency problems were found with the technique.

The chapter began by discussing a range of flowsheet situations that one may be interested in modelling. It was shown that some of these examples can be adequately modelled in a 'quasi-steady-state' manner, but for others it is necessary to take proper account of the actual history of a transition. In addition, there was a discussion of one example where it is not possible obtain a unique solution without recourse to numerical arguments. These distinctions were seen to be useful in the next two sections, which discussed the previous work of two sets of AI researchers. The first was the work of De Kleer and Brown [20], which is quasi-steady-state, and the second was that of Kuipers [55], which is more truly dynamic. For the work described in this thesis it was chosen to extend the method of De Kleer and Brown, by providing a means of reasoning about binary relational facts. This significantly increased the inferencing power of the approach but turned out to be an extremely costly improvement. In addition, some

serious problems were identified, such as how to represent composition in a binary split. These were felt to severely limit the applicability of qualitative simulation in any of its current forms.

This is unfortunate, since the later sections of the chapter demonstrated the central importance of qualitative representation to a number of loss prevention problems. For example, the work on automatic operating procedure synthesis has identified the close relationship between modelling and planning. This leads to the conclusion that one cannot separate a planning algorithm from the modelling representation that it uses. Similarly, fault diagnosis is now using qualitative simulation techniques to enhance the robustness of the diagnostic rules; in particular, by making them less brittle to changes in flowsheet structure. Finally, fault-tree synthesis relies upon qualitative knowledge to define the connections between the discretised events on the fault-tree.

The remaining sections of the chapter described some representational differences between qualitative simulation and the techniques used for automatic fault-tree synthesis. Most of these seemed to stem from the failure of the latter to distinguish between 'events' that represent changes in continuous process variables (e.g. $\text{del}(p1) = +$), those that represent value ranges of those variables (e.g. $\text{Fin} > 0$) and those that represent device states or device state transitions (e.g. "pump stopped" or "pump stops"). It was felt that qualitative simulation could provide some valuable input here but the form it might take was not discussed in detail.

Chapter 4

This chapter began by discussing the different uses that are made of the term 'constraint'. The distinctions were primarily concerned with the way in which a constraint is used. In particular, distinctions were drawn between the use of constraints for numerical simulation, optimisation, constraint satisfaction or just as procedural checks upon a data or knowledge base. This thesis work was shown to be concerned with constraints of the last type, despite the fact that they cannot be used in *generative mode*. This was justified by a discussion of the fundamental trade-off between expressiveness and efficiency, as identified by Levesque and Brachman [64]. In addition, there was a discussion of some other pragmatic issues that should be addressed when comparing

two forms of knowledge representation. The chapter concluded by listing the features that the Constraint Tools System is intended to provide.

Chapter 5

This chapter began by discussing Knowledge Craft at the most general level and how it can be integrated with software written in conventional languages. It then reviewed some previous chemical engineering research using AI toolkits. This discussion was mostly devoted to the Design Kit system, produced by George Stephanopoulos' group at MIT, as this was a good way of introducing the notions of inheritance, multiple inheritance and the separation of topology from function. These were shown to be powerful concepts but they can cause some difficulties with data sharing and consistency. Similar problems were also seen to arise from the representation of abstract systems as schemata in their own right, which is useful for constraint representation and other purposes. Lastly, the chapter also discussed hierarchical design, which encounters similar difficulties due to the need to share information between different design levels.

The bulk of the chapter was devoted to an abstract description of the three main components of the Constraint Tools System:

1. The Constraint System, which is used to create constraints and control their application.
2. The Propagating-relation System, which is used to specify mappings in data sharing situations and to automatically update equivalent values.
3. The Refinement Manager, which uses Knowledge Craft contexts to represent hierarchical design.

In each case the discussion began with a description of existing Knowledge Craft features that are useful for the area concerned. This set the scene for the subsequent description of the new software; which was vital in understanding how it all ties together. In addition, there was a description of how the user interacts with Knowledge Craft's graphical environment, in a manner that can be called 'pseudo multi-tasking'.

Chapter 6

This chapter presented extended examples of how the Constraint Tools System may be used in a design situation. It began by showing how one can edit background knowledge, which is stored in the \$root-context. Whilst doing this it discussed five hierarchies that have been created for the process engineering domain. It was careful, however, to limit the intellectual claims made about these classifications. Indeed, they were shown to be primarily for implementation purposes, rather than being based on a task independent view of process engineering concepts. This was partially justified by discussing the difficulty of phrasing constraints in isolation i.e. without the rest of the design system in place. In particular, the discussion centred on the problem of being able to make up arbitrary slot names; and the resultant problem of updating the values in such slots when design changes occur. This is referred to by Struthers [99] as the problem of 'arbitrary tokenism'.

The chapter then went on to show how one can edit the refinement hierarchy for a continuous plant example. This time the discussion was concerned with contexts below the \$root-context; and primarily with instance schemata rather than with prototypes. Firstly it was demonstrated that one can traverse and display structural contexts using Palm workbenches and Fixed-menus. There was then a discussion of some example constraints and how one can use contexts to partition their application and automatic checking.

The discussion then turned to the distinction between structural and version contexts. This was shown to be very important as it allows one to explore alternative parameterisations of the same basic structure. Furthermore, the behaviour of the Refinement Manager changes subtly according to whether one is in structural or version edit mode. This was demonstrated during a description of how one can also use the Refinement Manager and Palm to edit version contexts. In addition, there was a brief description of the use of Propagating-relations to automatically propagate data between all the contexts associated with the same version. It was also shown that constraint checking behaviour can be changed dynamically; for example, in order to write violation messages to a file instead of popping them up on the screen.

The description of the continuous plant example finished by showing that one can save all the changes made during the current Knowledge Craft session. This distinguishes this work from projects such as PIP [48], which assume that a preliminary design will be done in only one or two sessions.

The final section in this chapter briefly described the evaluation of the Constraint Tools System on a project for the design of a batch plant. For this case study it was necessary to invent a representation that was analogous to the use of systems and streams for a continuous plant, but which could also capture time and sequence. This was successful but the overall evaluation was somewhat disappointing. Partly this was due to a number of teething and efficiency problems. However, it also appears that the system is simply too complex for non-specialist personnel. This has implications for the original intention, which was to allow a designer to write project specific constraints, as well as using ones that have been pre-defined. In reality, therefore, it appears that a company would have to designate a special team to develop and maintain a design environment appropriate to its needs. Thus there would be many layers of system development between a domain independent tool and the eventual end user. Furthermore, some of these layers would intentionally have to restrict the flexibility of the initial system.

Chapter 7

This chapter compared the Constraint Tools System with some recent AI-based modelling environments. Mostly it concentrated on the use of decomposition and refinement, as these are the areas in which there is much overlap. By virtue of this it talked more about MODEL.LA and ASCEND rather than CLAP or other systems.

It began by discussing MODEL.LA and showing how many of its relation facilities could be fairly easily provided using existing Knowledge Craft features. It gave the authors credit, however, for clarifying the 'semantics' of the different relation classes. It then talked about their method of multifaceted modelling, which appears to require that the whole design is refined along the same branch, rather than being spread across the context tree. The disadvantages of this approach were discussed in terms of the duplication of information and the difficulty of expansion of the tree. In particular, the discussion noted that the tree is combinatorial so it is difficult to maintain all possible

alternatives. The main reason for this was seen to be that one cannot specify the internal structure of a section as being independent of which alternatives are chosen for the rest of the plant.

Next the discussion turned to ASCEND and showed how one could use Propagating-relations to implement their notion of *merge*. On the other hand, it also showed how their 'merge' and 'refines' operators could be used to simulate the use of contexts by the Refinement Manager. Thus there is some interesting common ground due to the power of the merge operator. However, there are also large differences in philosophy. In particular, in the Constraint Tools System data can be automatically propagated, whereas in ASCEND data mapping only occurs when the equation set is instantiated. Indeed, apart from when performing simple type checks ASCEND only uses data when the equation set is initialised and solved. In contrast, the Constraint Tools System is much more active so a user may see an immediate effect when he makes a change. Furthermore, in ASCEND one cannot reason about 'objects' and 'contexts' in an explicit manner. This may cause problems for hierarchical modelling since there is no way of overcoming strictly additive inheritance. Such problems are further discussed in the section on CLAP, which considers the top-down expansion of equations. Finally, there is a description of ModAss, which is necessarily brief as few details were available.

If you can look into the seeds of time and say, which grain will grow, and which will not, speak to me then.

Shakespeare, 'Macbeth'

8.2 Future Directions

This brings the discussion up to the present chapter, so it is now time to make some final recommendations about future research directions. These will be divided according to the three research classes that have already been identified.

Loss Prevention

The methodology of Loss Prevention is already much developed but there is plenty of scope for research with respect to specific tools. In some cases this need only involve simple graph searching procedures, such as could be easily constructed in Prolog. An example of such a problem might be the design of pressure relief systems, which could require one to find all the pressure sources that can be 'seen' at a given point in a flowsheet. On the other hand, some tools will be very complicated and require much more sophisticated techniques. For example, the discussion in Chapter 3 suggested the possibility of using qualitative simulation to support automatic fault-tree synthesis. Indeed, it identified the central importance of qualitative reasoning to many Loss Prevention problems, such as the synthesis of safe operating procedures or fault diagnosis. However, it also identified many problems with 'pure' qualitative simulation. The author therefore recommends that such techniques are pursued, but only in the context of methods that provide additional means of taming complexity and ambiguity.

Engineering Databases

One of the things to consider is whether or not to aim for a production version of the Constraint Tools System. This may be tempting but one must remember the difficulty of phrasing constraints in the absence of a complete design environment. Furthermore, the previous technical discussions have shown that there are a large number of issues to be sorted out before one can create even a prototype for such an environment. Nevertheless, one might be able to take some of the ideas from this thesis and work them into existing database systems such as Prodas. Initially this could just result in an extension of the idea of type checking but later on one should try to carry information between successive design levels. Probably this will have to be done on a company specific basis, since current process engineering databases do not appear to provide generic features for mappings of this kind.

A specific company will have an existing project structure and probably a large set of recommended design programs. In contrast, academics are typically working in a vacuum, with no concrete design process to focus upon. One should therefore expect some partial solutions to be achieved within specific companies before the successful creation of a company independent system for truly integrated design. Indeed, one might question whether the latter is ever achievable, given the secrecy with which each company regards its design procedures.

This does not mean that one can never achieve the goal of integrated design, just that a generic tool will have to leave many features open, such as the organisation of the design teams and the communication between them. This is already the way that databases and communication systems seem to work i.e. by providing partial solutions that are tailored to the needs of a particular company. Furthermore, one can expect future generations of databases, which may have a substantial AI content, to also follow this philosophy. It is therefore imperative that future researchers keep abreast of the work being done on the integration of AI and database technology. Only then can they understand enough about the nature of the tools to generate the ideas and 'theories' behind using them for process engineering design.

Knowledge Representation

Some of the more interesting technical discussions in this thesis have focussed around the notions of decomposition, refinement and design alternatives. These have turned out to be common concepts among systems as different in style as MODEL.LA, The Constraint Tools System and ASCEND. Hopefully the development of these concepts will continue, perhaps with more clarification of their underlying semantics. For example, there is a need for further understanding of the implications of strictly additive inheritance and the separation of structural and parametric information. A particularly interesting question is how to give the user control over whether or not the internal structure of a plant section is regarded as being dependent upon the choices made elsewhere in the plant. Perhaps truth maintenance systems will play a role here.

Another major aspect to pursue is the integration of the many different areas that people are currently working on. At first this should be done at the functional level i.e. by the integration of tasks tackling different aspects of the design problem. For example, one could bring together Design and Loss Prevention by integrating equation-based modelling with some of the constraint tools described in this thesis. This would allow a much better definition of the concepts needed for process engineering, which is necessary to avoid the problem of arbitrary tokenism. At a second level one might attempt the integration of more generic tools, such as the four different types of constraint that have already been identified. Finally, one may even consider the integration of software languages and paradigms; for example, the integration of unification with the use of vectors or matrices. CLAP and the AI-toolkits provide an excellent way of experimenting with all three of these integration levels, although they may not turn out to be the best vehicles for the ultimate end products.

This brings the discussion onto the question of what software to use in the future? There is no single answer to this as it depends upon many issues and criteria; such as cost, availability, functionality, portability and reliability. All these factors contribute, though their relative weightings will vary according to who is considered to be the 'primary customer' and how the 'end product' is defined. It is therefore difficult to make any specific recommendations as it is really a matter to be decided by an overall research strategy. Nevertheless, it is worth noting that there are certainly benefits to

be had from using a diversity of software 'paradigms'. In particular, because the use of different software can encourage rather different perspectives of the same problem. This is clearly seen in the difference between ASCEND and the systems based on AI toolkits. There is a caveat, however, which is that there is little point in describing similar features at a purely surface level. Instead one must describe the detail and philosophy of a representation and be willing to dig deep into such descriptions to extract the essential ingredients. Knowledge representation really is the name of the game.

Man is not weak - knowledge is more than equivalent to force.
The master of mechanics laughs at strength
Samuel Johnson

Bibliography

- [1] J Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11), November 1983.
- [2] P.K. Andow. The use of expert systems to improve process safety. *Proceedings of I. Mech. E. One Day Symposium on AI Applications in the Process Industries*, 1988.
- [3] P.K. Andow. Difficulties in fault-tree synthesis for process plant. *IEEE Transactions on Reliability*, R-29:2-9, April 1980.
- [4] P.K. Andow and M. Galluzzo. Process plant safety and artificial intelligence. *Proceedings of CEF-87 (The Use of Computers in Chemical Engineering)*, April 1987.
- [5] D.G. Austin and G.V. Jeffreys. *The Manufacture of Methyl Ethyl Ketone from 2-Butanol*. I.Chem.E., Rugby, 1979.
- [6] S.H. Booth and M.C. Jones. Use of rule based programming for operability studies. *Annual Research Meeting - University of Nottingham*, 9th April 1987.
- [7] J. Branch. Interacting with computers: Expert systems. *Chemistry and Industry*, 9:321-324, 5th May 1986.
- [8] J. Branch. Towards integrated process design. *Processing*, pages 37-39, March 1985.
- [9] M.L. Brodie. Future intelligent information systems: AI and Database technologies working together. J. Mylopoulos and M.L. Brodie, editors, *Readings in Artificial Intelligence and Databases*. Morgan kaufmann Publishers, Inc., 1989.
- [10] M.L. Brodie and F. Manola. Database management: A survey. J. Mylopoulos and M.L. Brodie, editors, *Readings in Artificial Intelligence and Databases*. Morgan kaufmann Publishers, Inc., 1989.
- [11] L.L. Bucciarelli. An ethnographic perspective on engineering design. *Design Studies*, 9(3):159-168, July 1988.
- [12] A.R. Bunn and F.P. Lees. Expert design of plant handling hazardous materials. *Chem. Eng Res Des*, 66, Sept. 1988.
- [13] Carnegie Group, Inc. *Knowledge Craft Reference Manual*, 1986.

- [14] D.H. Cherry, J.C. Grogan, G.L. Knapp, and F.A. Perris. Use of data bases in engineering design. *CEP*, pages 59–67, May 1982.
- [15] J. Chowdury and R. Caudwell. Thinking machines and the CPI. *Chemical Engineering*, 89(19):45–50.
- [16] CISHC (Chemical Industry Safety and Health Council). *A Guide to Hazard and Operability Studies*. Chemical Industries Association, London, 1977.
- [17] J. Craft. The impact of CAD and Database techniques in process engineering. *I.Chem.E. Symposium Series*, 92, Process Systems Engineering '85, 1985.
- [18] D.T. Dalle-Molle. *Qualitative Simulation of Dynamic Chemical Processes*. PhD thesis, University of Austin, Texas, May 1989.
- [19] D.T. Dalle Molle, B.J. Kuipers, and T.F. Edgar. Qualitative modeling and simulation of dynamic systems. *Computers and Chemical Engineering*, 12(9/10):853–866, October 1988.
- [20] J. De Kleer and J.S. Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1983.
- [21] M. Dohnal. Naive models in chemical and system engineering. *Proceedings of CEF-87 (The Use of Computers in Chemical Engineering)*, April 1987.
- [22] J. Douglas. *Conceptual Design of Chemical Processes*. McGraw-Hill, New York, 1987.
- [23] B. Drabble. Planning and reasoning with processes. Technical report, Artificial Intelligence Applications Institute, Edinburgh, 1989.
- [24] B. Drabble. *Intelligent Execution Monitoring and Error Analysis in Planning Involving Processes*. PhD thesis, University of Aston, Birmingham, U.K., July 1988.
- [25] M.A. Duran and I.E. Grossman. A mixed-integer nonlinear programming algorithm for process system synthesis. *A.I.Ch.E. J.*, 32, 1986.
- [26] H.A. Duxbury and M.L. Preston. The process systems contribution to process safety. *FOCAPO - proceedings*, 1987.
- [27] C. Eastman. Automated space planning. *Artificial Intelligence*, 4:41–64, 1973.
- [28] G.H. Ferguson. The knowledge problem in artificial intelligence with special reference to Hazard and Operability studies. Master's thesis, Loughborough University of Technology, Loughborough, U.K.
- [29] R.E. Filman. Reasoning with worlds and truth maintenance in a knowledge-based programming environment. *Communications of the ACM*, 31(4), April 1988.
- [30] K.D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

- [31] N.R. Foulkes, M.J. Walton, P.K. Andow, and M. Galluzzo. Computer-aided synthesis of complex pump and valve operations. *Computers and Chemical Engineering*, 12(9/10):1035–1044, October 1988.
- [32] R.H. Fusillo and G.J. Powers. A synthesis method for chemical plant operating procedures. *Computers and Chemical Engineering*, 11(4):369–382, 1987.
- [33] R.H. Fusillo and G.J. Powers. Operating procedure synthesis using local models and distributed goals. *Computers and Chemical Engineering*, 12(9/10):1023–1034, October 1988.
- [34] R.H. Fusillo and J. Powers, G. Computer-aided planning of purge operations. *A.I.Ch.E. J.*, 34(4):558–566, April 1988.
- [35] M. Galluzzo and P.K. Andow. Expert systems in chemical engineering. *Proceedings of CEF-87 (The Use of Computers in Chemical Engineering)*, April 1987.
- [36] I. Hamlet and J. Hunter. Representing time and reasoning with it. *Proceedings of IEE Colloquium - Qualitative Modelling in Diagnosis and Control*, January 1988.
- [37] D.J. Harris and M.C. Jones. Expert system shells in process safety assessment. *Annual Research Meeting - University of Nottingham*, 9th April 1987.
- [38] G. Henning, H. Leone, and G. Stephanopoulos. Model.la a modelling language for process engineering. Part 1: The formal framework. Technical report, Laboratory for Intelligent Systems in Process Engineering, Department of Chemical Engineering, MIT, Boston, Massachusetts, March 1989.
- [39] G. Henning, H. Leone, and G. Stephanopoulos. Model.la a modelling language for process engineering. Part 2: Multifaceted modeling of processing systems. Technical report, Laboratory for Intelligent Systems in Process Engineering, Department of Chemical Engineering, MIT, Boston, Massachusetts, March 1989.
- [40] M. Hofmeister, L. Halasz, and D.W.T. Rippin. Knowledge-based tools for batch processing systems, 1988.
- [41] D. Hutton. *Knowledge Based Flowsheet Modelling for Chemical Process Design*. PhD thesis, University of Edinburgh, Department of Chemical Engineering, UK, To be submitted, December 1990.
- [42] D. Hutton, J.W. Ponton, and A. Waters. AI applications in process design, operation and safety. *Accepted for publication in The Knowledge Engineering Review.*, 1990.
- [43] I.Chem.E. *A First Guide to Loss Prevention*. I.Chem.E., Rugby, 1977.
- [44] I.Chem.E. *Nomenclature for Hazard and Risk Assessment in the Process Industries*. I.Chem.E., Rugby, 1985.
- [45] P. Jackson. *Introduction to Expert Systems*. Addison-Wesley Publishing Company, USA, 1986.

- [46] A. James. Selection criteria for an integrated CAE system. *I.Chem.E. Symposium Series*, 92, Process Systems Engineering '85, 1985.
- [47] J.K. Kingston. A technical overview of Knowledge Craft. *Airing Newsletter, AI Applications Institute, Edinburgh*, 3 & 4, October 1987 & Feb 1988.
- [48] R.L. Kirkwood, M.H. Locke, and J.M. Douglas. A prototype expert system for synthesizing chemical process flowsheets. *Computers and Chemical Engineering*, 12(4):329-343, April 1988.
- [49] T.A. Kletz. Hazop and Hazan - notes on the identification and assessment of hazards. *I.Chem.E. Workshop Modules*, 1983.
- [50] T.A. Kletz. *What Went Wrong? : Case Histories of Process Plant Disasters*. Gulf Pub. Co., Book Division, Houston, 1985.
- [51] T.A. Kletz. Eliminating potential process hazards. *Chemical Engineering*, April 1985.
- [52] E. Kondili, C.C. Pantelides, and R.W.H. Sargent. A general algorithm for scheduling batch operations. *Submitted for publication to Computers and Chemical Engineering*, 1989.
- [53] M.A. Kramer and B.L. Palowitch. A rule-based approach to fault diagnosis using the signed directed graph. *A.I.Ch.E. J.*, 33(7):1067-1078, July 1987.
- [54] M. Krishnamurthy and H.J. Efstathiou. Monitoring and fault diagnosis. *Proceedings of IEE Colloquium - Qualitative Modelling in Diagnosis and Control*, January 1988.
- [55] B. Kuipers. Qualitative simulation. *Artificial Intelligence*, 29:289-338, 1986.
- [56] E. Lakie. Evaluation of a tool for the representation and checking of design constraints by its continuous use during a 4th year design project. Technical report, Department of Chemical Engineering, University of Edinburgh, Edinburgh, April 1990.
- [57] R. Lakshamanan and G. Stephanopoulos. Synthesis of operating procedures for complete chemical plants -1. Hierarchical, structured modeling for nonlinear planning. *Computers and Chemical Engineering*, 12(9/10):985-1002, October 1988.
- [58] R. Lakshamanan and G. Stephanopoulos. Synthesis of operating procedures for complete chemical plants -2. A nonlinear planning methodology. *Computers and Chemical Engineering*, 12(9/10):1003-1022, October 1988.
- [59] H.E. Lambert. Comments on the Lapp-Powers 'computer-aided synthesis of fault-trees'. *IEEE Transactions on Reliability*, R-28:6-11, April 1979.
- [60] S.A Lapp and G.J. Powers. Computer-aided synthesis of fault-trees. *IEEE Transactions on Reliability*, R-26:2-11, April 1977.

- [61] H.G Lawley. Operability studies and hazard analysis. *Chem. Eng. Prog.*, 70(4):45, April 1974.
- [62] H.G Lawley. Size up plant hazards this way. *Hydrocarbon Processing*, April 1976.
- [63] F.P. Lees. *Loss Prevention in the Process Industries*. Butterworths, London, 1980.
- [64] H.J. Levesque and R.J. Brachman. A fundamental tradeoff in knowledge representation and reasoning. R.J. Brachman and H.J. Levesque, editors, *Readings in Knowledge Representation*. Morgan kaufmann Publishers, Inc., 1985.
- [65] K. Lien, G. Suzuki, and A.W. Westerberg. The role of expert systems technology in design. Technical report, Engineering Design Research Centre, Carnegie Mellon University, Pittsburgh, September 1986.
- [66] K.M. Lien. A framework for opportunistic problem solving. *Computers and Chemical Engineering*, 13(4/5):331-342, 1989.
- [67] K.M. Lien, P.E. Wahl, C. Sorlie, and T. Hertzberg. Integration of expert systems and quantitative computation in chemical engineering design - Presented at CHISA-90, Prague. Technical report, Laboratory of Chemical Engineering, The Norwegian Institute of Technology, N-7034 Trondheim, Norway, August 1990.
- [68] J.A. Liles, D.H. Lott, and M.L. Preston. Process definition to plant design - integration at last? *Proceedings of the Third International Symposium on Process Systems Engineering (PSE '88)*, Sydney, August 1988.
- [69] M.O. Locks. The fail-safe feature of the Lapp & Powers fault tree. *IEEE Transactions on Reliability*, R-29(1), April 1980.
- [70] G.A. Martin-Solis. *Fault-tree Synthesis for Real Time and Design Applications on Process Plant*. PhD thesis, University of Loughborough, 1978.
- [71] G.A. Martin-Solis, P.K. Andow, and F.P. Lees. Fault tree synthesis for design and real time applications. *Transactions of the I.Chem.E*, 60:14-25, 1982.
- [72] R.L. Motard. Computer horizons in process-systems engineering - part 2. *Chemical Engineering*, 7th March 1983.
- [73] K. Niida, I. Koshijima, and T. Umeda. Diagnosis-kit: Progress report on knowledge aquisition, knowledge representation, and inference. *Process Systems Engineering '88 - proceedings*, September 1988.
- [74] E. O'Shima. Sequencing and scheduling of plant operations. *FOCAPO - proceedings*, 1987.
- [75] O.O. Oyeleye and M.A. Kramer. Qualitative simulation of chemical process systems: Steady state analysis. *A.I.Ch.E. J.*, 34(9):1441-1454, September 1988.

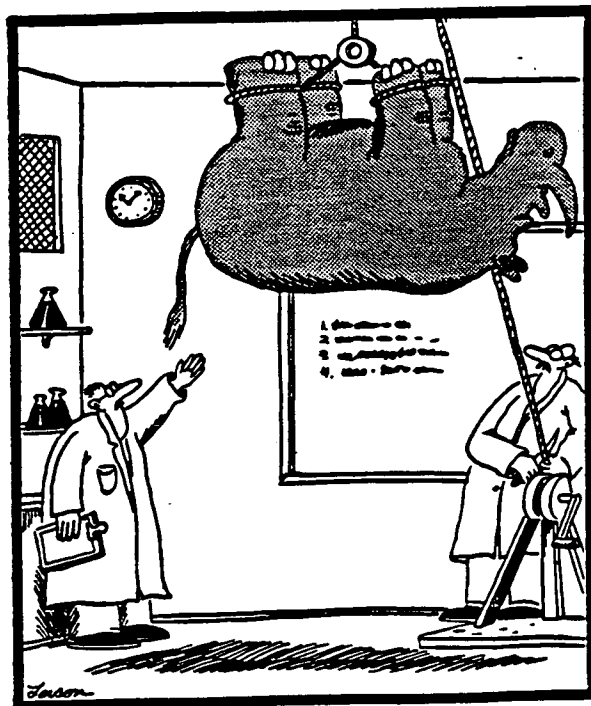
- [76] H. Ozog and A.D. Little. Hazard identification, analysis and control. *Chemical Engineering*, 18th February, 1985.
- [77] N. Piccinini and G. Levy. Ethylene oxide reactor: Safety according to operability analysis. *Can. J. Chem. Eng.*, 62, Aug 1984.
- [78] P.C. Piela. *ASCEND - An Object Oriented Environment for the Development of Quantitative Models*. PhD thesis, Carnegie Mellon University, Pittsburgh, 1989.
- [79] J.W. Ponton, D. Hutton, G. Jones, and N. Skilling. A demonstration 'intelligent' physical properties data system. *Conference Proceedings: CHEMPOR'89 - Lisbon, Portugal*, September 1989.
- [80] P.E. Preece and M. Stephens. PROCEDE - Opening windows on the design process. *CIPE-89*, 1989.
- [81] M.L. Preston. Integrated process plant design. J.J. Siirola, I.E. Grossmann, and G. Stephanopoulos, editors, *Foundations of Computer-Aided Process Design (FOCAPD 1989)*, Austin, Texas, 1990. Cache Corporation.
- [82] R. H. Rand. Computer algebra in applied mathematics: An introduction to MACSYMA. *Pitman Research Notes in Mathematics*, 94, 1984.
- [83] G.V. Reklaitis. Progress and issues in computer-aided batch process design. *Conference Proceedings: Foundations of Computer-Aided Process Design - Snowmass Village, Colorado*, July 1989.
- [84] J.R. Rivas and D.F. Rudd. Synthesis of failure-safe operations. *A.I.Ch.E. J.*, 20(2), March 1974.
- [85] J.R. Roach and F.P. Lees. Some features of and activities in Hazard and Operability (HAZOP) studies. *The Chemical Engineer*, October 1981.
- [86] P. Ross. *Notes for Lecture Course - Representation and Inference Part 2*. University of Edinburgh, Department of Artificial Intelligence, Edinburgh, 1986/87.
- [87] E. Sacks. Hierarchical inequality reasoning. Technical Report MIT/LCS/TM-312, MIT Lab. for Computer Science.
- [88] R.W.H. Sargent. Process design - what next? J.J. Siirola, I.E. Grossmann, and G. Stephanopoulos, editors, *Foundations of Computer-Aided Process Design (FOCAPD 1989)*, Austin, Texas, 1990. Cache Corporation.
- [89] A. Shafaghi, P.K. Andow, and F.P. Lees. Fault tree synthesis based on control loop structure. *Chem. Eng. Research and Design*, 62:101-110, March 1984.
- [90] J.J. Siirola. The future of product and process design. *Conference Proceedings: Foundations of Computer-Aided Process Design - Snowmass Village, Colorado*, July 1989.

- [91] B.C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT, Cambridge, 1982.
- [92] T. Smithers, A. Conkie, J. Doheny, B. Logan, K. Millington, and Ming Xi Tang. Design as intelligent behaviour: An AI in design research programme. Technical report, Department of Artificial Intelligence, University of Edinburgh, November 1989.
- [93] D. Sririam. A bibliography of Knowledge-Based Expert Systems in engineering. Technical report, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, USA, December, 1984.
- [94] D. Sririam et al. Knowledge-Based Expert Systems: An emerging technology for CAD in chemical engineering. Technical report, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, USA, December, 1984.
- [95] G.L. Steele. *Common Lisp - The Language*. Digital Press, 1984.
- [96] G. Stephanopoulos. The scope of artificial intelligence in plant-wide operations. *FOCAPO Proceedings*, 1987.
- [97] G. Stephanopoulos. Artificial intelligence and symbolic computing in process engineering design. J.J Sirola, I.E. Grossmann, and G. Stephanopoulos, editors, *Foundations of Computer-Aided Process Design (FOCAPD 1989)*, Austin, Texas, 1990. Cache Corporation.
- [98] G. Stephanopoulos, J. Johnston, T. Kriticos, R. Lakshmanan, M.L. Mavrovouniotis, and C. Siletti. Design-kit: An object-orientated environment for process engineering. *Computers and Chemical Engineering*, 11(6):655-674, 1987.
- [99] A. Struthers. *A Knowledge Based Approach to Process Engineering Design*. PhD thesis, University of Edinburgh, Department of Chemical Engineering, UK, 1990.
- [100] E. Subrahmanian, G. Podnar, B. Elm, and A.W. Westerberg. Towards a shared computational support environment for engineering design. Technical report, Engineering Design Research Centre, Carnegie Mellon University, Pittsburgh, April 1989.
- [101] J. Suokas, P. Heino, and I. Karvonen. The development of an Expert System to support Hazop analysis, April 1987.
- [102] J.R. Taylor. An algorithm for fault-tree construction. *IEEE Transactions on Reliability*, R-31(2):137-146, June 1982.
- [103] M. Tomita, K. Hwang, and E. O'Shima. Automatic synthesizer of operating procedures for chemical plant by use of fragmentary knowledge. *Journal of Chemical Engineering of Japan*, 22(4):364-373, 1989.
- [104] S. Tomita, K. Hwang, and E. O'Shima. On the development of an AI-based system for synthesising plant operating procedures. *CIPE-89*, 1989.

- [105] Y. Tsuge, H. Matsuyama, and C. McGreavy. Qualitative simulator for estimation of plant behaviour during abnormal situations. *CIPE-89*, 1989.
- [106] R.D. Turney. Designing plants for 1990 and beyond: Procedures for the control of safety, health and environmental hazards in the design of chemical plant. *Trans. Instn Chem. Engrs.*, 68, Part B., February 1990.
- [107] T. Umeda, T. Kuriyamam, E. O'Shima, and H. Matsuyama. A graphical approach to cause and effect analysis of chemical processing systems. *Chem. Eng. Sci.*, 35:2379-2388, 1980.
- [108] R.E. Valdes-Perez. Spatio-temporal reasoning and linear inequalities. *AI Memo, MIT AI Laboratory*, May 1986.
- [109] V. Ventkatasubramanian and S.H. Rich. An object-orientated two-tier architecture for integrating compiled and deep-level knowledge for process diagnosis. *Computers and Chemical Engineering*, 12(9/10):903-922, October 1988.
- [110] A. Waters, P.W.H. Chung, and J. Ponton. Representing safety constraints upon chemical plant designs. S. Vadera, editor, *Expert System Applications*. Sigma Press, Wilmslow, England, 1990.
- [111] A. Waters and J. Ponton. Qualitative simulation and fault propagation in process plants. *Chemical Engineering Research and Design*, 67:407-422, July 1989.
- [112] A.J. Waters. The Constraint Tools System - A Knowledge Craft Extension for Hierarchical Design. Technical report, Chemical Engineering Dept., Edinburgh University, Kings Buildings, West Mains Road, Edinburgh, UK, 1990.
- [113] T. Weatherill and I. Cameron. Preliminary hazop studies using expert systems, August 1988.
- [114] G.L. Wells. *Safety in Process Plant Design*. George Godwin Ltd, London, 1980.
- [115] A.W. Westerberg, P.C. Piela, E. Subrahmanian, G. Podnar, and B. Elm. A future computing environment for preliminary design. J.J Sirola, I.E. Grossmann, and G. Stephanopoulos, editors, *Foundations of Computer-Aided Process Design (FOCAPD 1989)*, Austin, Texas, 1990. Cache Corporation.
- [116] P.H. Winston and B.K.P. Horn. *LISP*. Addison-Wesley Publishing Company, USA, 1981.
- [117] P. Winter and S.M. Rosen. Process engineering databases. *Chemical Engineering*, pages 65-67, July 1986.

Part III

Appendices



Testing whether or not rhinos land on their feet

Appendix A

Some qualitative rules for bottom-up reasoning

This appendix gives an English description of some of the ‘rules’ that were used in the simple HAZOP program¹. We partition this description according to the use of the rules and the types of plant item they apply to. Most of these rules were derived from examples in the HAZOP literature. We have no illusions about their real utility; they are presented for example purposes only.

A.1 Causes of a Deviation

Causes in a line

If there is a valve on a line then we assume that it may be closed accidentally giving rise to the deviations ‘full-back-pressure’ and ‘no-forward-pressure’. However, if it is a control valve then we first ask the user whether or not he wants to explore deviations caused by instrument air failure. If the answer is yes then we generate the appropriate deviations, according to whether the valve is specified as “fail-open” or “fail-closed”. A further extension might be to check whether a manual valve is normally locked open or locked closed. All valves are also taken to be capable of partial blockage, thus causing the deviations ‘high-back-pressure’ and low-forward-pressure’.

If there is a bypass on a line then we assume that it may be open accidentally thus resulting in the deviations ‘low-back-pressure’ and ‘high-forward-pressure’. Finally, if the line is very long then we assume that external heating is a possible cause of ‘high-temperature’. The rationale here is that an external source of heat, such as a fire, is much less likely to be noticed on a long pipe run, particularly if it is not inspected regularly.

¹As already mentioned in Chapter 2, this work was conducted during the author’s under-graduate honours year so it pre-dates the work of the Ph.D. proper.

At Start or End points

For each start or end point the user is asked if he wishes to propagate deviations from this point. If the answer is yes then he is asked about each deviation in turn, as it arises. The forward propagating deviations are given as options at start points and the backward propagating ones at end points. These options are necessary to allow the HAZOP of a large plant to be split into several smaller studies.

In Tanks and Liquid Pressure Vessels

Tanks and liquid pressure vessels are assumed to contain a liquid whose vapour pressure increases significantly with temperature. External heating is therefore taken to be a cause for 'high-temperature', 'high-forward-pressure' and 'high-back-pressure'. Tanks require some means of filling the space that is left when they are emptied. Usually this is by the use of a nitrogen blanket; in which case, loss of nitrogen pressure is taken to be a possible cause of 'low-back-pressure' and 'low-forward-pressure'. In addition, the contents of a tank may freeze, blocking off the outlet. This is taken to be a possibility if the melting point of the tank liquid is above -10 Degrees Centigrade. The resulting deviations are 'no-forward-pressure' and 'full-back-pressure'.

At Pumps

Pumps are responsible for increasing pressure and causing flow. Consequently, the deviations that are attributable to pumps are the result of disruptions to this intent. Poor pump performance is taken to be a possibility, thus causing 'low-forward-pressure'. Also, we assume that motor failure can occur, which causes 'full-back-pressure' and 'no-forward-pressure'. Finally, seal leakage may result in a loss of process fluid. This may be immediately harmful to nearby personnel. In addition, if it is not detected then it may worsen and eventually lead to the process deviations 'low-forward-pressure' and 'low-back-pressure'.

At Heat Exchangers

Heat exchanger tubes may get fouled or blocked, particularly if the tube-side fluid polymerises. Unfortunately it is difficult to deduce if this is a likely scenario, so we just ask the user. If he replies yes then we consider the deviations 'high-back-pressure' and 'low-forward-pressure', which come from the increased pressure drop in the tubes. In addition, the reduced heat transfer results in 'high-temperature' in the hot stream leaving the exchanger and 'low-temperature' in the cold stream. A second possibility is that a heat exchanger tube bursts. This is particularly likely if there is a large pressure difference between shell-side and tube-side, or if the fluids are very corrosive. Consequently, we define a critical pressure difference at which this is taken to be a possibility. The result will be opposite pressure deviations on the two sides of the exchanger. The critical pressure difference was arbitrarily set at 6 Bara.

A.2 Rules for the Propagation of a Deviation

Propagation through a Tank

Tanks are generally used as buffer storage for liquids, so we should not expect flow deviations to be propagated unconditionally. However, a process operator can only be expected to respond to a changing tank level if he is made aware of what is taking place. This consideration leads to the following propagation rules:

1. The deviations 'low-temperature' and 'high-temperature' propagate unconditionally from the inlet to the outlet.
2. If the tank doesn't have a low level alarm then the flow deviations 'no-forward-pressure' and 'low-forward-pressure' propagate straight from the inlet to the outlet.
3. If the tank doesn't have a high level alarm then the flow deviation 'high-forward-pressure' propagates straight from the inlet to the outlet.
4. Similarly, if the tank doesn't have a high level alarm then the deviations 'high-back-pressure' and 'full-back-pressure' propagate straight from the outlet to the inlet. The reasoning here is that without a high level alarm the tank will be left to fill up completely. Thus the liquid will back-up and exert pressure on the upstream fluid. In reality, whether or not this actually happens depends upon the actual design of the tank. For example, the tank may just burst or overflow.
5. The deviation 'low-back-pressure' is not propagated through a tank because the pressure above the liquid surface should not be affected by a fall in liquid level (assuming the tank is vented properly).

Through a Pump

Pumps have virtually no material hold-up and so cannot fail to propagate all forward and backward acting deviations directly. The advantages of thinking in terms of pressure, not flow, are demonstrated quite well when dealing with pumps. To see this, suppose that a control valve fails open upstream of a pump, which should effectively cause high flow. To propagate this deviation we know that a greater pressure will be required for the downstream section. At the same time, however, we must have higher flow through the pump. This appears to be in contradiction to the characteristic behaviour of pumps, in which flow is *inversely* proportional to pressure head. However, this problem is solved when we realise that the pressure *into* the pump has also been increased by the control valve failure. Thus we can have an increase in the downstream absolute pressure at the same time as having a decrease in the pressure head across

the pump. This argument is more easily expressed in propagation rules if we say that the valve failure causes the deviation ‘high-forward-pressure’ rather than the deviation ‘high-flow’.

Through a Heat Exchanger

Heat exchangers will generally have no buffer capacity and so all forward and backward deviations will propagate directly from inlet to outlet, on the same side of the exchanger. For example, the result of ‘high-forward-pressure’ at the inlet to the hot side is ‘high-forward-pressure’ from the outlet for the hot side. In addition, however, the hot and cold sides of an exchanger interact, so there are extra propagation paths across the exchanger. For example, a reduction in the hot stream flow into an exchanger means that it leaves at a lower temperature (due to a reduction in its heat capacity flowrate). At the same time, the log mean temperature difference across the exchanger has decreased, which means that less heat is transferred to the cold side. Less heat is put into the exchanger and so both streams come out colder. This behaviour, and its corollaries, are described by the interactions shown in Table A.1. Note that this table does not itself allow for the possibility of competing effects. That is, the table is only correct if the elements marked with a “/” can be guaranteed to be undeviated. This is not the case where there are feedback loops in a process.

Hot stream in	Cold stream in	Hot stream out	Cold stream out
no-forward-pressure	/	—	low-temp
low-forward-pressure	/	low-temp	low-temp
high-forward-pressure	/	high-temp	high-temp
/	no-forward-pressure	high-temp	—
/	low-forward-pressure	high-temp	high-temp
/	high-forward-pressure	low-temp	low-temp
high-temp	/	high-temp	high-temp
low-temp	/	low-temp	low-temp
/	high-temp	high-temp	high-temp
/	low-temp	low-temp	low-temp

Table A.1: Interaction table for propagation across a heat exchanger

At a Binary Split or a Binary Junction between two lines

A split is taken to be the divide of a single process stream into two. These two may be of unequal magnitude; it may even be possible that one of the streams is used only very occasionally. Figure A.1 shows four example uses of a binary split.

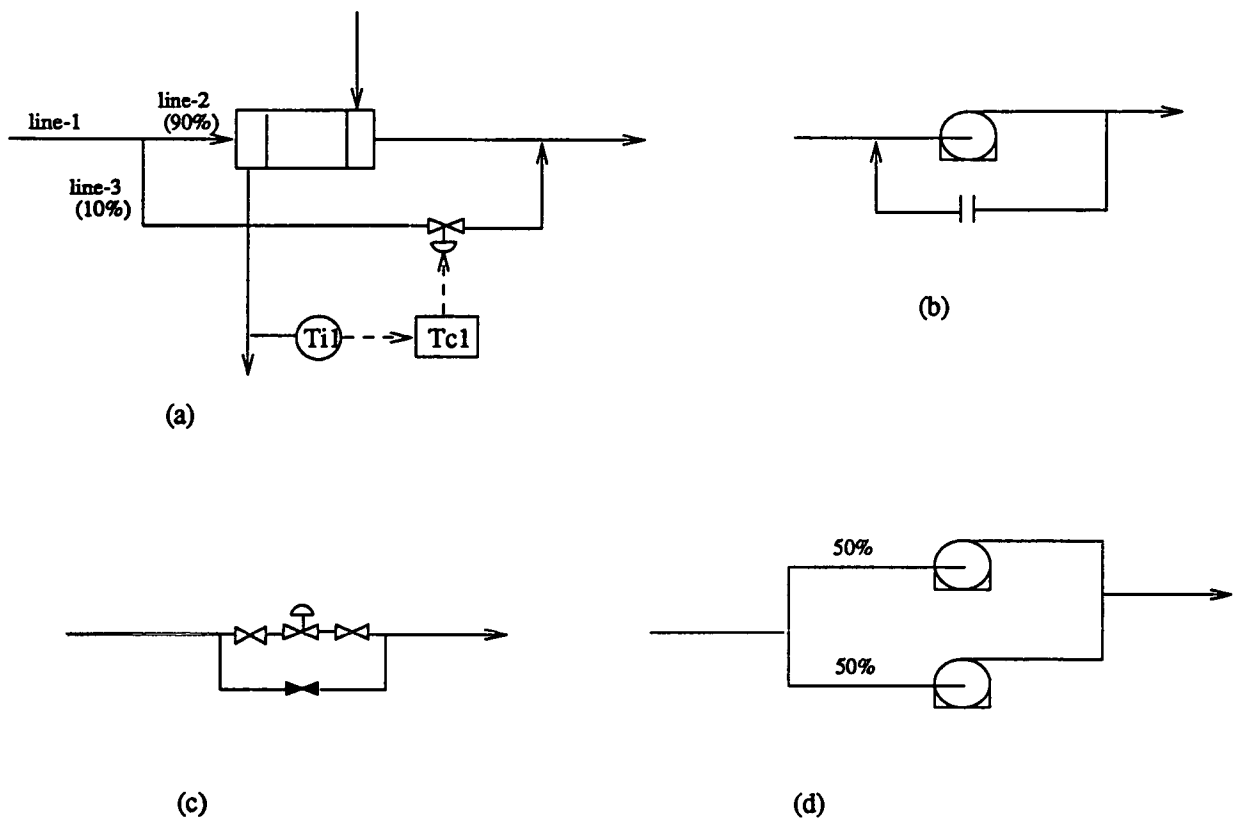


Figure A.1: Example uses of a binary split

The first example in this figure is that of a fine temperature control on a heat exchanger, which works by bypassing anything up to 10% of the heating medium. The streams therefore split in a ratio of about 90:10. The second example involves an orifice plate kickback loop around a pump. In normal operation the resistance of the kickback line is so much higher than that of the downstream section that it takes virtually no flow. However, if the downstream path is accidentally closed then the pressure head across the pump shoots up. The kickback line now takes just enough flow to safely relieve the pressure. This line of the split is therefore dormant until called into action by a downstream deviation. The third example is that of a split which never has flow in both branches unless the bypass is open accidentally. Finally, the fourth example involves two pumps operating in parallel, so the split is now 50:50.

Consider what happens if the control valve in Figure A.1(a) fails shut. The immediate deviation is 'full-back-pressure' in line 3 but how is this propagated at the split? Clearly the result is not 'full-back-pressure' in line 1 because the closed line took only 10% of the

flow anyway. At worst, the result will be 'high-back-pressure' in line 1, with the total flow being lowered to between 90% and 100% of its original value. On the other hand, if one of the pumps in Figure A.1(d) failed then the result will be a more significant increase in the upstream pressure. Finally, if the exchanger tubes in Figure A.1(a) became blocked then the flow in line 1 may be reduced by up to 90% of its original value. These examples suggest that maybe we could relate the effect of pressure deviations at a split to the proportion of the total flow normally carried by the line concerned.

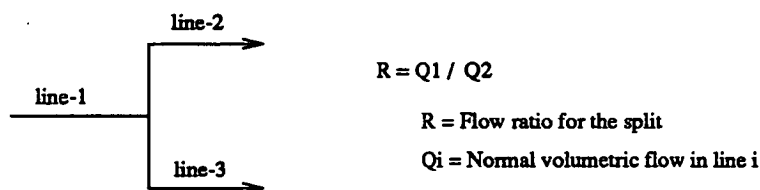


Figure A.2: Definition of flow ratio (R) in a Binary Split

To formalise this consider the split in Figure A.2. Forward acting deviations (high-temperature etc.) are obviously propagated directly from line 1 to both lines 2 and 3. However, to decide how to propagate backward acting deviations we define the flow ratio (R), which characterises the split. The backward propagation rules are then as follows:

1. If $R > 80\%$ then the result of full-back-pressure, high-back-pressure or low-back-pressure in line 2 is the same deviation in line 1.
2. If $R > 80\%$ then another result of full-back-pressure in line 2 is high-forward-pressure in line 3.
3. If $20\% < R < 80\%$ then the result of full-back-pressure in line 2 or 3 is high-back-pressure in line 1.
4. If $R < 20\%$ then switch lines 2 and 3 in order to apply rules (1) and (2).

Obviously these rules are somewhat arbitrarily defined. For example, we may ask how we chose the critical flow ratios to be 20% and 80%. This is similar to the problem of deciding how to discretise the continuous variables in our domain. In particular, in both cases the resolution is limited by the number of levels that we choose. In addition, we should note that these rules only use the *normal* flow ratio for a split and so cannot adequately capture the behaviour of the splits in Figures A.1(b) and (c). For example,

if we used these rules for a kickback loop then we would deduce that a closure of the downstream section causes full-back-pressure at the pump outlet. Unfortunately, this is exactly the situation that the kickback loop protects against i.e. we would not be modelling its behaviour correctly. Such problems are also discussed elsewhere in this thesis.

The concepts involved in the propagation rules for a binary junction are entirely analogous to those just outlined for a binary split. However, this time it is the backward acting deviations which propagate directly and the forward acting ones which require the consideration of flow ratios. The reader should be able to deduce these analogous rules for himself so we shall not list them here.

A.3 Rules for the Consequences of a Deviation

External Consequences at a Tank

To deduce the external consequences of flow deviations at a tank we can use similar reasoning to that which we used for the propagation of deviations through a tank. For example, if the tank doesn't have a high level alarm then the deviations high-forward-pressure, high-back-pressure and full-back-pressure may persist long enough for the tank to fill up completely. The possible consequences are then "tank spillage" or "tank rupture", though it is difficult to be more precise without quantification of the maximum pressure that is being exerted. Similarly, if the deviation 'no-forward-pressure' occurs at the tank outlet then the contents may drain reasonably quickly. This could lead to collapse of the tank unless the venting system was designed for such a rapid change in level.

At a Pump

If the flow is blocked downstream of a working pump then we say that the pump has been deadheaded. Within our terminology this is represented by the deviation 'full-back-pressure' at the pump outlet. This will then result in the outlet to the pump being subjected to the full delivery pressure that the pump can achieve. For a reciprocating pump this will almost immediately lead to system failure due to overpressurisation. For a centrifugal pump the consequences are not quite so severe but we might still get a seal leakage due to overheating of the pump contents. This would probably then lead to a fire.

We also have a problem if the flow is blocked upstream of a working pump, which is represented by the deviation 'no-forward-pressure' at the pump inlet. This time there is a large reduction in pressure, which can cause cavitation (boiling) within the pump. This effect can damage a pump in a fairly short space of time, though it does not

present an immediate external hazard. It can also be caused by high-temperature at the pump inlet.

At a Heat Exchanger

The only external consequence considered for a heat exchanger is that if flow on the cold side is stopped then this fluid will overheat and may foul or polymerise in the tubes. However, because this result depends upon the particular fluid being heated we first ask the user for confirmation before recording it.

At Start or End Points

Forward acting deviations which reach end points, and backward acting ones which reach start points, are taken as their own consequences. That is, we record their existence so that we can propagate the deviations to connected flowsheets at a later time.

Appendix B

More Detail on the Qualitative Simulation Program

B.1 Notation for Chapter 3

Variables and process units:

fi	mass flow in stream 'i'
Pi or pi	pressure in stream 'i'
Ti	temperature in stream 'i'
xi	mass-fraction in stream 'i' (binary mixture)
del(X)	the sign of the change in variable X
[X]	the sign of the value of variable X
rvi	relief valve 'i'
set-point(X)	the set point of relief valve or control loop X
A	the open area of a relief valve
vi	manual valve 'i'

Subscripts:

h	hot-side (of a heat exchanger) e.g. fh = mass flow of hot stream into exchanger
c	cold-side (of a heat exchanger)

B.2 Code and Algorithms

As we have already described in Chapter 3, the program is split into two stages which are both applied to each new state. The first stage finds the *intra-state behaviour* i.e.

the behaviours that may lead to a state transition. The top goals for this are described in Figures B.1 & B.2. The second stage finds all the device state transitions that might occur as a result of any of these behaviours. The simulation then branches to repeat the process on each new system state. The top goals for the second stage are shown in Figures B.3 & B.4.

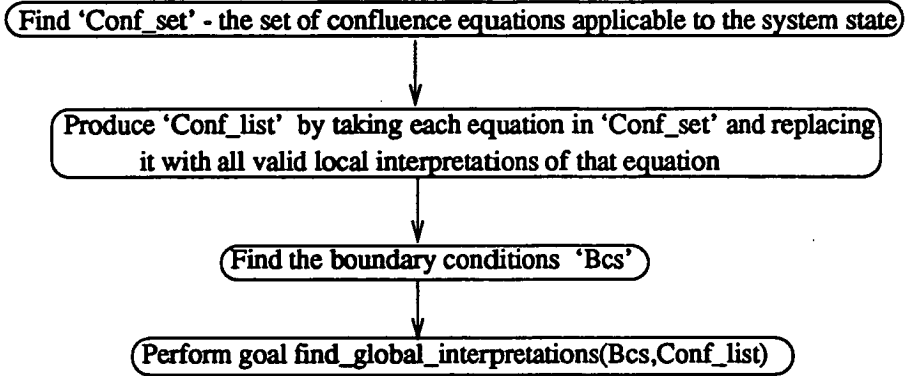


Figure B.1: The top goals for finding all valid global interpretations

Figure B.2 is a semi-Prolog representation of a recursive Prolog goal. Note the use of the 'fail' point after a valid 'global interpretation' has been found. This allows us to backtrack to the 'choice point' shown. We then work forwards again to find other valid solutions. This use of backtracking makes it difficult to represent the goals on a conventional sequential flowchart.

The overall philosophy of the goals in Figure B.2 is that we have only found a valid global interpretation when we have satisfied all the confluences concerned. To ensure this we work through the list of confluences, choosing one 'local interpretation' from each. Each time we make a choice we include the new bindings within the partial interpretation so far built up. Thus we start with just the boundary conditions but finish with bindings for every variable mentioned in the confluences. However, before we include a choice we first test the new bindings for conflicts with ones already made. If there is an inconsistency then the choice is rejected and we choose again from the same confluence. If we run out of choices for a confluence then we backtrack to a previous choice point and try again. When we do this Prolog will automatically unwind the partial interpretation to remove the bindings that no longer apply. The code for all this is fairly simple because the algorithm can be naturally expressed in Prolog and because inconsistency can be proven by a strict equality test.

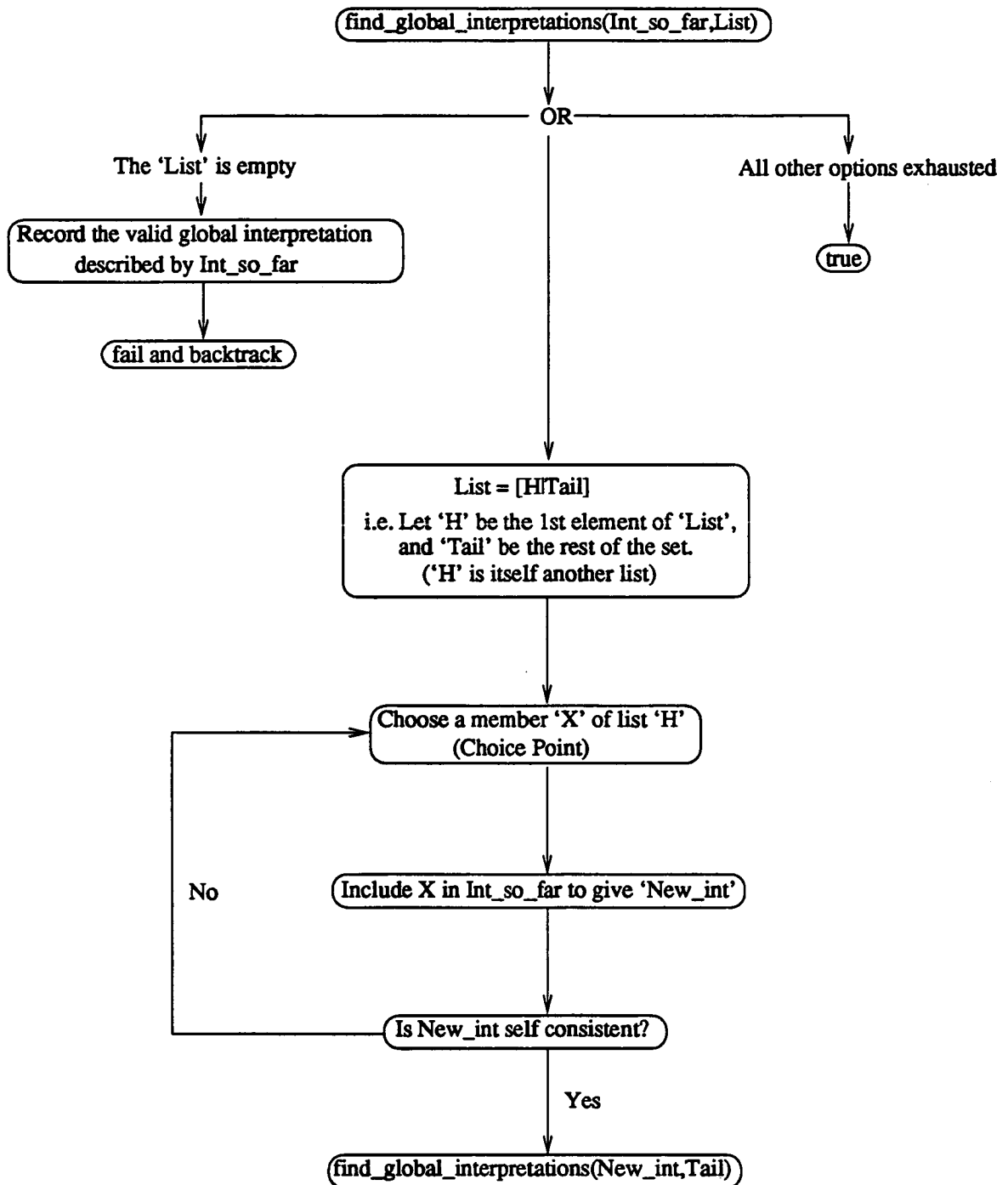


Figure B.2: A semi-Prolog description of the goal 'find_global_interpretations'

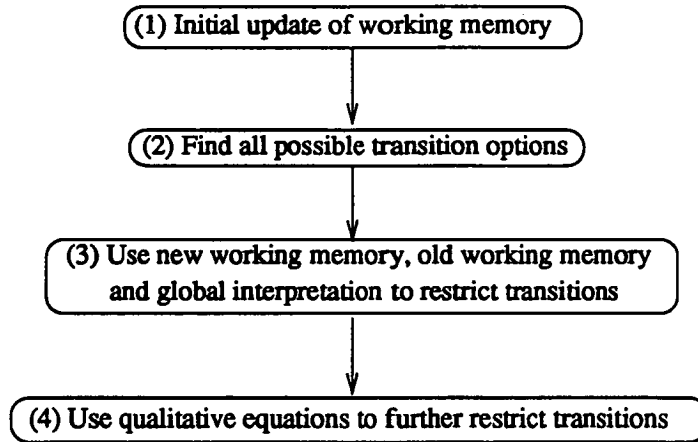


Figure B.3: The top goals for finding state transitions

Whenever we set up a new state we create new facts of the following form:

```

device_states(State,List)
boundary_conditions(State,Bcs)
in_list_within_state(State,In)
out_list_within_state(State,Out)
  
```

where the variable 'State' is just a tag that is unique for each state, 'List' is the set of device states, 'Bcs' is the list of boundary conditions that apply, 'In' is the union of all the 'Infacts' for the devices and states concerned and 'Out' is the union of all the 'Outfacts'¹. The facts are restricted to binary relational form, as already described in Chapter 3. They give us an initial working memory for the system state, which we can then update according to different global interpretations.

¹In case the reader has forgotten, Infacts must be true and Outfacts must be false *within* the state (though not necessarily at the state boundaries).

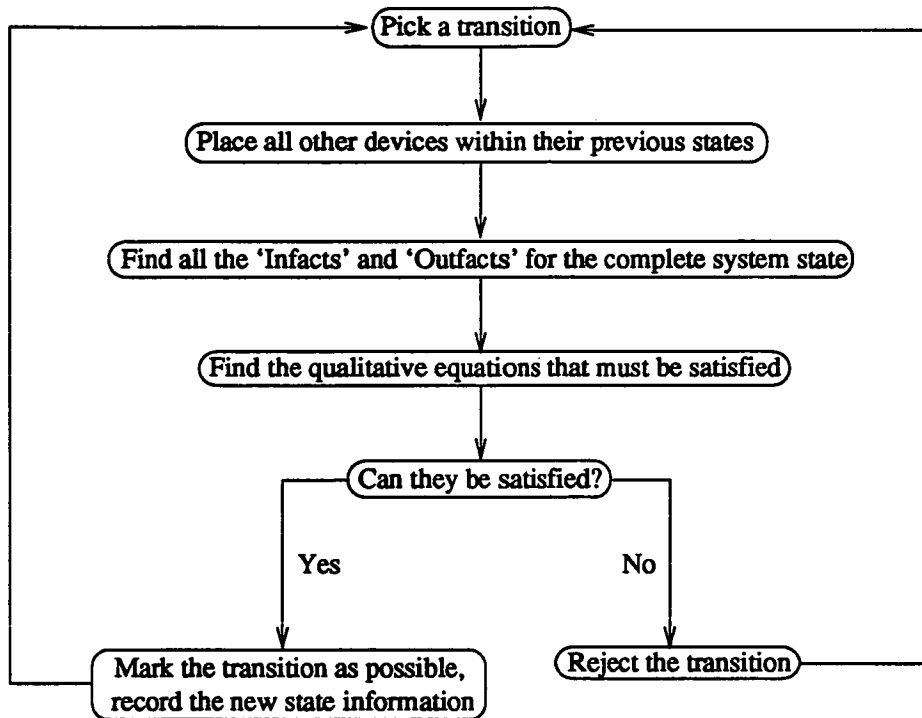


Figure B.4: Using qualitative equations to further restrict transitions

Stage one on Figure B.3 is the first goal which uses this new working memory. Its main purpose is to generate new facts which we know:

must_still_apply
 must_now_not_apply
 must_still_not_apply
 must_now_apply

For example, if we knew that $X > 10$ was true when the state was first entered, and $\text{del}(X) = +$ under the interpretation we are considering, then this fact 'must_still_apply'. However, if $X = 10$ was the original fact then $X > 10$ 'must_now_apply' and $X = 10$ 'must_now_not_apply'.

Stage two on Figure B.3 uses the unit models described later to find all the changes that could possibly occur. For a relief valve in the 'fluttering' state this would return the transitions:

fluttering -> open, fluttering -> closed

Stage 3 of the same figure then uses the information from stage 1, plus other information, to rule out some of these transitions. For example, suppose we are modelling a relief valve in the fluttering state and $\text{del}(A) = +$ under the global interpretation that we are currently considering. Now, when we entered this state we would have known the 'Infact' $A = 0$, so now we know that $A > 0$. This makes it impossible to satisfy the 'Trans_infact' $A = 0$, which must be true for the relief valve to transition back to the closed state. Thus we can rule out this transition as impossible.

Stage 4 on Figure B.3 also filters out impossible transitions but this time we use the qualitative equations, rather than just the facts. This stage is very time-consuming but is necessary if we wish to solve problems such as that in Figure 3.1(d), which we have already discussed in Chapter 3. We redescribe stage 4 on Figure B.4, from which the reader can see that only one transition is allowed at a time. All other devices are therefore taken as remaining in their previous state.

The algorithm to test whether a set of qualitative equations can be satisfied is highly analogous to the algorithm for confluence solution shown in Figure B.2. However, there are two important differences:

1. We are only trying to prove that a set of facts is *consistent* with a set of qualitative equations, not trying to find all possible solutions. Thus we only need to find one global interpretation to satisfy our goal. However, to prove inconsistency we do still have to perform an exhaustive search.
2. Checking inconsistency now involves the use of rewrite rules to 'chain' through all possible derived facts, rather than just the strict equality test we could use before.

For this second reason, finding inter-state behaviour is typically much slower than finding intra-state behaviour.

B.3 Unit Models

This section will display some unit models in a natural language format. In addition, we shall describe the Prolog goals needed to represent the models for our qualitative simulation program. We shall do this by discussing the Prolog terms used to represent selected parts of these models.

B.3.1 Modelling a Relief Valve

Figure B.5 defines the variable names for a prototypical relief valve. Figure B.6 then uses these names to give a natural language description of the complete relief valve model. This model regards a relief valve as having three possible states. We discuss the second of these in more detail to show how we represent a model using Prolog.

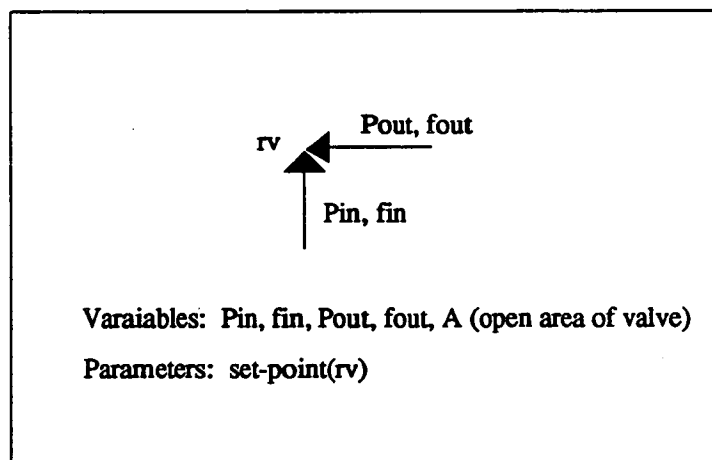


Figure B.5: Defining the variables for the relief valve model

Prolog representation of the Fluttering state

The first thing that we need to define is the topology of the device concerned. This is done by a term of the following form:

device_ports(Dev,List)

State = Closed

Confluences : (1) $\text{del}(\text{fin}) = 0$, (2) $\text{del}(\text{fout}) = 0$, (3) $\text{del}(A) = 0$

Relational equations : None

Infacts : $\text{fin} = 0$, $\text{fout} = 0$, $A = 0$, $(\text{Pin} - \text{Pout}) < \text{set-point}(\text{rv})$

Transitions :

Closed -> Fluttering :

Trans-infacts : $\text{fin} = 0$, $\text{fout} = 0$, $A = 0$, $(\text{Pin} - \text{Pout}) = \text{set-point}(\text{rv})$

State = Fluttering

Confluences :

1. $\text{del}(\text{fin}) + \text{del}(\text{Pout}) = \text{del}(A) + \text{del}(\text{Pin})$; the pressure balance.
2. $\text{del}(\text{fout}) = \text{del}(\text{fin})$; the mass balance.
3. $\text{del}(A) + \text{del}(\text{Pout}) = \text{del}(\text{Pin})$; force balance on spring.
4. $\text{del}(\text{fin}) = \text{del}(A)$; an additional confluence

Relational equations : None

Infacts : $\text{fin} = 0$, $\text{fout} = 0$, $A = 0$, $(\text{Pin} - \text{Pout}) = \text{set-point}(\text{rv})$

Transitions :

Fluttering -> Closed :

Trans-infacts : $\text{fin} = 0$, $\text{fout} = 0$, $A = 0$, $(\text{Pin} - \text{Pout}) < \text{set-point}(\text{rv})$

Fluttering -> Open :

Trans-infacts : $\text{fin} > 0$, $\text{fout} > 0$, $A > 0$, $(\text{Pin} - \text{Pout}) > \text{set-point}(\text{rv})$

State = Open

Confluences : Same as for the Fluttering state.

Relational equations : None

Infacts : $\text{fin} > 0$, $\text{fout} = \text{fin}$, $A > 0$, $(\text{Pin} - \text{Pout}) > \text{set-point}(\text{rv})$

Transitions :

Open -> Fluttering :

Trans-infacts : $\text{fin} = 0$, $\text{fout} = 0$, $A = 0$, $(\text{Pin} - \text{Pout}) = \text{set-point}(\text{rv})$

Figure B.6: The Representation of a Relief Valve

which says that a device of type 'Dev' has the stream connection points listed in 'List'. The order in which these connection points are given should correspond to that used by the other goals which describe the unit model. In addition, the names used should correspond to those used when describing an actual simulation problem. This will be further described where necessary. The term for the relief valve model is as follows:

```
device_ports(relief_valve,[in,out]).
```

Next we shall define the confluences which have to be satisfied by any valid global interpretation of the Fluttering state. The confluences are described by terms of the following form:

```
confluence_eqn(Num,Dev,State,Device_instance,Types,Vars,Eqn)
```

where:

Num is just a number used to refer to the confluence. The confluence numbers only need to be unique within a given state.

Dev is the type of device the confluence applies to.

State is the state of the device for which the confluence must be satisfied.

Device_instance gets bound during a simulation run to the name of the actual instance being modelled.

Types declares the types of the variables in each sub list of **Vars**. For example, in the first `confluence_eqn` goal below **Fin** is a mass_flow and **Pin** is a pressure.

Vars defines the variables used within **Eqn** to represent the confluence. The sub lists of 'Vars' should appear in the order defined by the 'Ports' described in the 'device_ports' goal. For a relief valve this order is:

```
[inlet_stream_variables,outlet_stream_variables].
```

In addition, the sub lists should all be the same length as the 'Types' list. Thus the variables are defined by using the Types and Vars arguments, not by virtue of their names. That is, a variable such as 'Fi' only represents a mass flow if the Types argument says so, rather than because it begins with the letter "F".

The actual symbols for the variables are obtained by a goal called 'instantiate_stream_vars'. By virtue of Prolog unification this instantiation will also instantiate the same variables within the confluence equation, which is returned in the 'Eqn' argument.

Eqn is the confluence, which is represented in the form:

$$[\text{Lhs\_pos}, \text{Lhs\_neg}, \text{Rhs}]$$

where:

Lhs_pos = The terms that are positive when brought over to the left hand side

Lhs_neg = The terms that are negative when brought over to the left hand side

Rhs = The sign of the right hand side (+, 0 or - but usually just 0)

The first **confluence_eqn** goal therefore represents the following confluence:

$$\text{del}(\text{Fin}) = \text{del}(\text{area}(\text{Dev})) + \text{del}(\text{Pin}) - \text{del}(\text{Pout})$$

The confluences for the Fluttering state of a relief valve can then be defined as follows:

/ (1) is the pressure balance. */*

```
confluence_eqn(1,relief_valve,fluttering,Dev,Types,Vars,Eqn):-
    Types=[mass_flow,pressure],
    Vars=[[Fin,Pin],[_,Pout]],
    Eqn=[[del(Fin),del(Pout)],[del(area(Dev)),del(Pin)],0].
```

/ (2) is the mass balance. */*

```
confluence_eqn(2,relief_valve,fluttering,_,Types,Vars,Eqn):-
    Types=[mass_flow],
    Vars=[[Fin],[Fout]],
    Eqn=[[del(Fout)],[del(Fin)],0].
```

/ (3) is the force balance on the spring. */*

```
confluence_eqn(3,relief_valve,fluttering,Dev,Types,Vars,Eqn):-
    Types=[pressure],
    Vars=[[Pin],[Pout]],
    Eqn=[[del(area(Dev)),del(Pout)],[del(Pin)],0].
```

/ (4) is an extra confluence to remove the ambiguity of behaviour after the valve has opened. */*

```
confluence_eqn(4,relief_valve,fluttering,Dev,Types,Vars,Eqn):-
    Types=[mass_flow],
    Vars=[[Fin],[_]],
    Eqn=[[del(Fin)],[del(area(Dev))],0].
```

Now we might define the relational equations that must be satisfied within the current state. However, as there aren't any for the Fluttering state of a relief valve we shall describe their representation in a later section, when we talk about modelling a simple resistance element.

Next we define the facts that we know to be true or false within the current state. This is done by a term of the form:

facts(Dev,State,Device_instance,Types,Svars,Dvars,Infacts,Outfacts)

where:

Dev, State, Device_instance & Types are the same as in the **confluence_eqn** goal described above.

Svars is the same as the **Vars** argument in the **confluence_eqn** goal described above.

For example, in the **facts** goal below, **Fin** is the mass_flow for the 'in' connection point, **Pin** is the pressure for the same point, **Fout** is the mass_flow 'out' and **Pout** is the pressure 'out'.

Dvars is used to represent parameters of the device. Each element of **Dvars** is a two membered list of the form:

[Parameter,Variable_name]

where **Variable_name** is used within **Infacts** and **Outfacts** and **Parameter** refers to a device attribute. For example, **[setpt,S]** will bind the variable **S** to the **setpt** (set point) of the device being considered. The attributes are set in the input file for a problem, which should contain facts such as the following:

device(rv1,relief_valve,[setpt=10, ...etc.]).

This defines the parameters for a relief_valve called 'rv1'.

Infacts returns a list of facts which are known to be true within the **State**.

Outfacts returns a list of facts which are known to be false within the **State**. The variables mentioned in the **Infacts** and **Outfacts** are bound using **Svars, Dvars & Types**.

The **facts** goal for the Fluttering state of a relief valve is as follows:

```
facts(relief_valve,fluttering,Dev,Types,Svars,Dvars,Infacts,Outfacts):-
    Types=[mass_flow,pressure],
    Svars=[[Fin,Pin],[Fout,Pout]],
    Dvars=[[setpt,S]],
    Infacts=[Fin=0,Fout=0,area(Dev)=0,Pin-Pout=S],
    Outfacts=[].
```

The only remaining thing to do now is to define the device transitions that might occur from this state. This is done by a goal of the form:

```
device_state_transition(Num,Dev,Old_state,New_state,Inst,Svars,Dvars,Info)
```

where:

Num is an identification number, similar to that used for confluences.

Old_state & **New_state** are the device states before and after the transition respectively.

Dev, **Svars** & **Dvars** are similar to as described for the **facts** goal above.

Inst is the same as **Device_instance** in the **facts** goal described above.

Info defines some facts and rules which must be true for the transition to take place. These are used to filter out impossible transitions. This argument returns a list of the form:

```
[Infacts,Outfacts,Rules]
```

where:

Infacts is a list of facts which must be true at the transition point.

Outfacts is a list of facts which must be false at the transition point.

Rules is a list of relational equations which must apply while the transition is taking place.

The variables mentioned in **Infacts** and the **Outfacts** are bound as already described for other goals. Similarly, the representation of the relational equations is the same as will be described later for the **qualitative_eqn** goal.

The transitions that can possibly occur from the Fluttering state of a relief valve are defined as follows:

```
device_state_transition(1,relief_valve,fluttering,closed,Dev,Svars,Dvars,Info):-
    Svars=[Types,Vars],
        Types=[mass_flow,pressure],
        Vars=[[Fin,Pin],[Fout,Pout]],
    Dvars=[[setpt,S]],
    Info=[Infacts,Outfacts,Rules],
        Infacts=[Fin=0,Fout=0,area(Dev)=0,(Pout-Pin)>(-S)],
        Outfacts=[],
        Rules=[].

device_state_transition(2,relief_valve,fluttering,open,Dev,Svars,Dvars,Info):-
    Svars=[Types,Vars],
        Types=[mass_flow,pressure],
        Vars=[[Fin,Pin],[Fout,Pout]],
    Dvars=[[setpt,S]],
    Info=[Infacts,Outfacts,Rules],
        Infacts=[Fin>0,Fout>0,area(Dev)>0,Pin-Pout>S],
        Outfacts=[],
        Rules=[].
```

B.3.2 Modelling a Simple Resistance Element

The model for a simple resistance element has already been displayed in Figure 3.8, so we need not reproduce it here. In addition, the Prolog representation of this model is much the same as that just described for a relief valve. However, we do now have some relational equations to define, which we did not have before. This is done by a goal of the following form:

```
qualitative_eqn(Num,Dev,State,Device_instance,Types,Vars,Eqn)
```

where:

Num, Dev, State, Device_instance, Types & Vars are the same as already described above for the **confluence_eqn** goal.

Eqn is the relational equation, which is represented in the same way that we represented a confluence in the **confluence_eqn** goal described above. The only difference is that for a confluence the basic terms were of the form 'del(X)' whereas for a

relational equation they are of the form '[X]'. For example, the `qualitative_eqn` goal below represents the following relational equation:

$$[\text{Fin}] - [\text{Pin} - \text{Pout}] = 0$$

The reader is referred to Chapter 3 for a description of what this really means.

There is only one relational equation for the Working state of a simple resistance element. This is defined as follows:

```
qualitative_eqn(1,resistance_element,working,_,Types,Vars,Eqn):-
    Types=[mass_flow,pressure],
    Vars=[[Fin,Pin],[_,Pout]],
    Eqn=[[Fin],[(Pin-Pout)],0].
```

B.3.3 Modelling a Split, a Junction and a Line Blockage

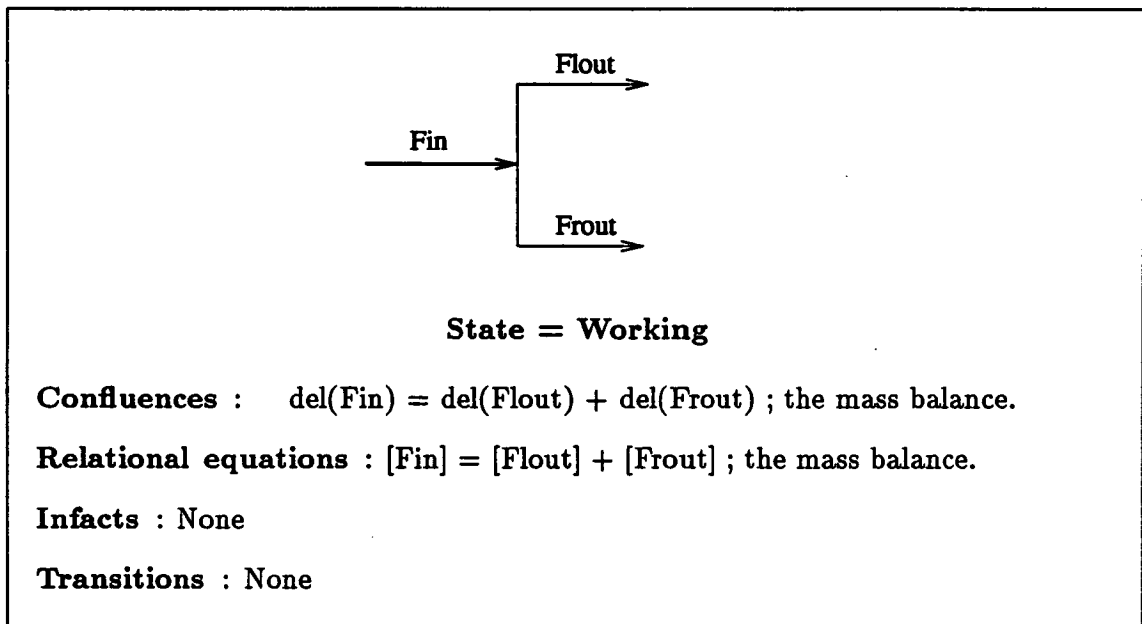


Figure B.7: The Representation of a Binary Split

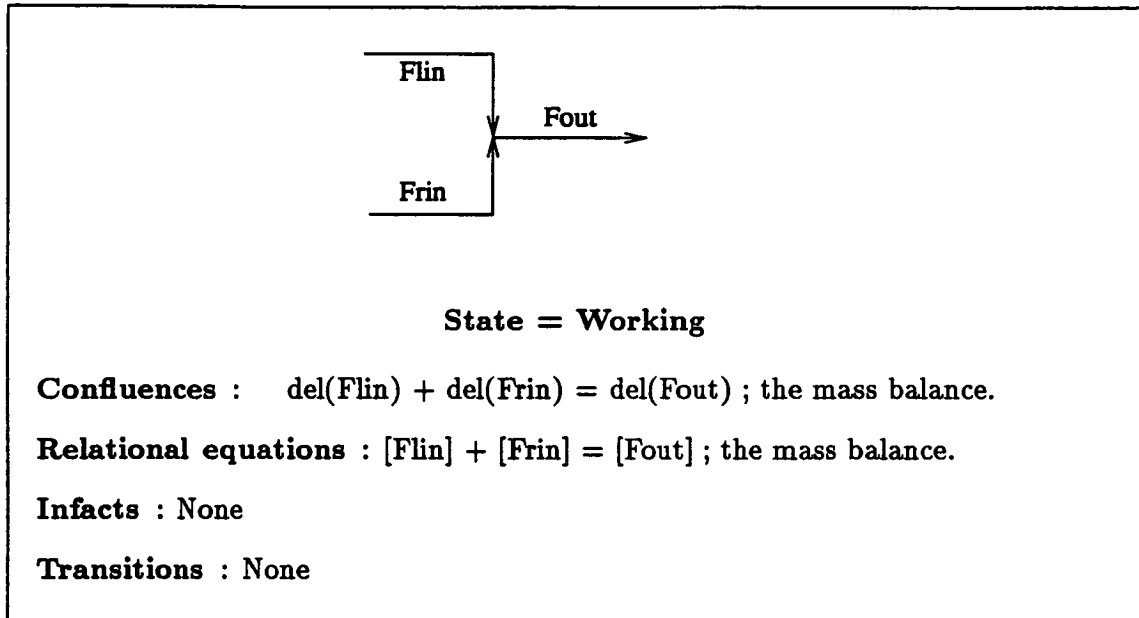


Figure B.8: The Representation of a Binary Junction

The models for the remaining devices that can currently be used within our simulation program are shown in Figures B.7, B.8 & B.9. The Prolog representation of these models mostly uses the goals already described in the sections above. However, we provide one additional goal which can be used to *merge* some of the stream variables around a device. By doing this we are declaring the variables to be equivalent. They will then be *bound* to the same symbol. This is much more efficient than using facts to assert the equivalence of different symbols.

The goal to represent equivalence is of the following form:

`equivalent_vars(Dev,Stream_vars)`

where:

Dev is the type of device that this information applies to.

Stream_vars is a list which defines the variables that are to be merged. Each member of Stream_vars is a two membered list of the form:

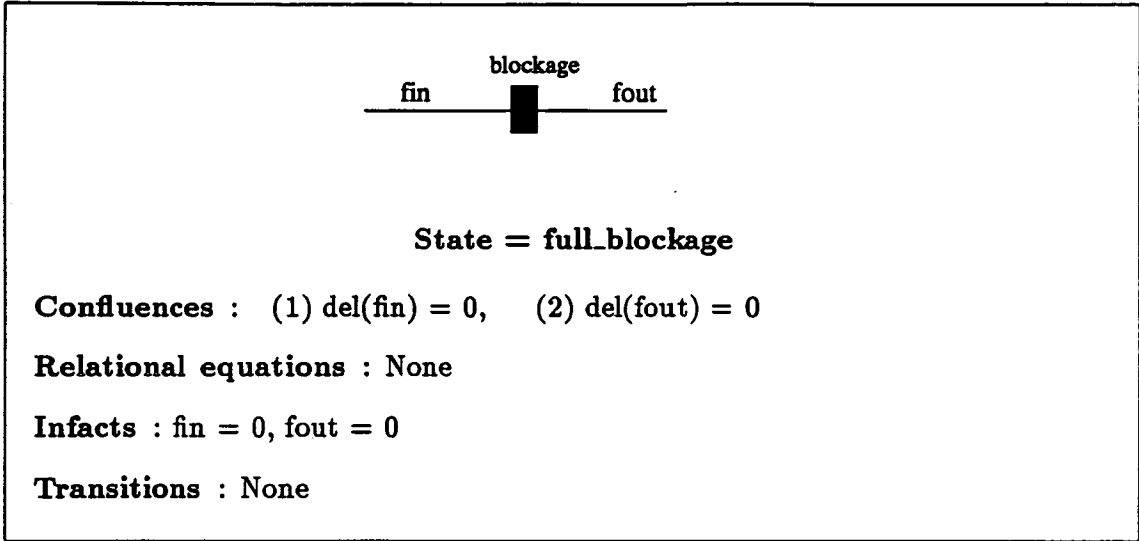


Figure B.9: A Pseudo Unit to Represent a Complete Line Blockage

[Port,Type]

in which 'Port' defines the stream connection point and 'Type' says which variable from that stream we are interested in. For example, [in,pressure] means the pressure of the inlet stream into a binary split.

Using this goal we can declare the three stream pressures around a binary split to be equivalent. That is, we can specify that a split does not cause a pressure drop. The term to do this is as follows:

```
equivalent_vars(split, [[in,pressure], [leftout,pressure], [rightout,pressure]]).
```

This term will automatically be used each time we initialise an item of type 'split'. This means that the three stream pressures around a split are always represented by the same symbol. In addition, if we had several splits in series then the merge routine would chain through the devices and make all the stream pressures equivalent. This procedure is guaranteed to terminate, even if there is a circular path of merges.

B.4 An Example Run

In this section we shall describe the input and output to the program for an example run. The input is written in Prolog but Figure B.10 depicts it diagrammatically for the sake of clarity. We have labelled the streams and devices on this figure so that the reader can match it to the Prolog representation.

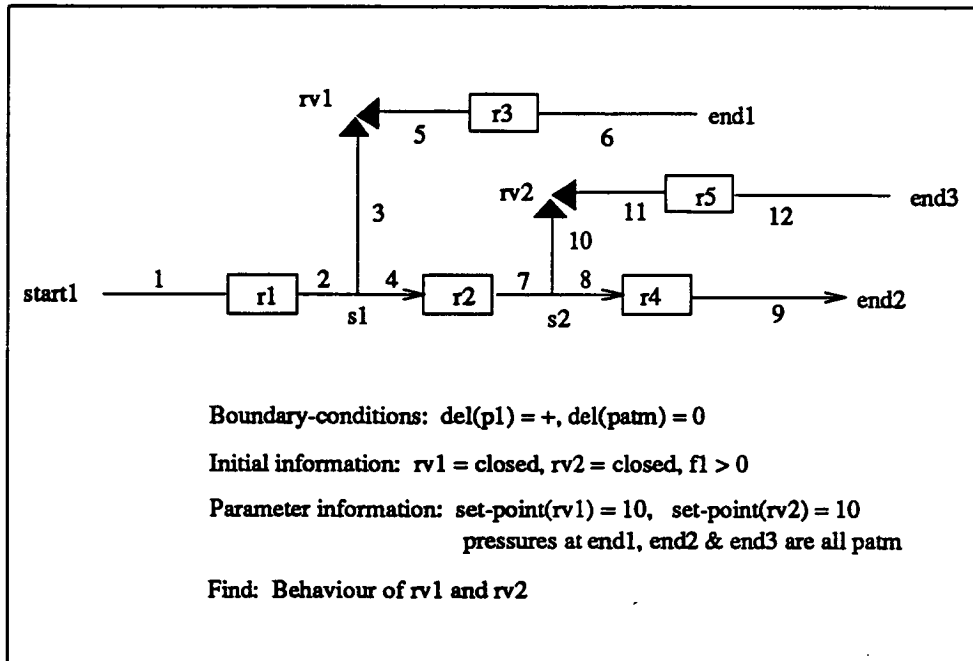


Figure B.10: An example input problem

The input problem is described as follows:

```
boundary_conditions([[pressure,1]='+',del(patm)=0]).
```

```
initial_in_list( [ ([mass_flow,1]>0) ] ).
```

```
initial_out_list(□).
```

```
stream(1,start1,[r1,in]).
```

```
stream(2,[r1,out],[s1,in]).
```

```
stream(3,[s1,leftout],[rv1,in]).
```

```
stream(4,[s1,rightout],[r2,in]).
```

```

stream(5,[rv1,out],[r3,in]).
stream(6,[r3,out],end1).
stream(7,[r2,out],[s2,in]).
stream(8,[s2,rightout],[r4,in]).
stream(9,[r4,out],end2).
stream(10,[s2,leftout],[rv2,in]).
stream(11,[rv2,out],[r5,in]).
stream(12,[r5,out],end3).

```

```

device(r1,resistance_element,[]).
device(r2,resistance_element,[]).
device(r3,resistance_element,[]).
device(r4,resistance_element,[]).
device(r5,resistance_element,[]).

```

```

device(s1,split,[]).
device(s2,split,[]).

```

```

device(rv1,relief_valve,[setpt=10]).
device(rv2,relief_valve,[setpt=10]).

```

```

device_states([0],Devstates):-
    Devstates= [r1=working,s1=working,
                rv1=closed,r2=working,
                r3=working,s2=working,
                rv2=closed,r4=working,
                r5=working
                ].

```

```

pre_named_vars([[6,pressure,patm],[9,pressure,patm],[12,pressure,patm]]).

```

Typically these goals would be stored within a file, in which case the problem definition is loaded by using the normal Prolog 'consult' predicate. We can then run the simulation by asking Prolog to answer the query:

```

analyse_network.

```

The result of this request will be a state transition description of the complete solution space. Figure B.11 displays this solution diagrammatically for the problem that we are

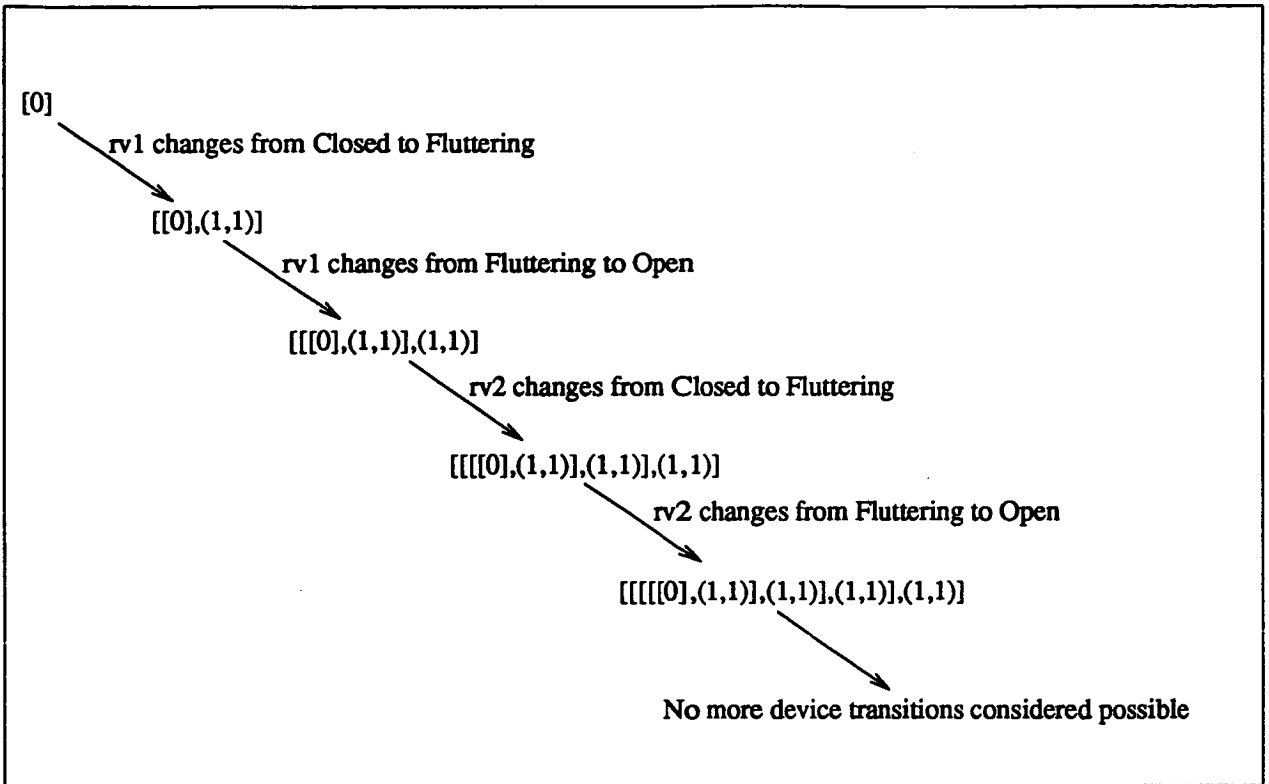


Figure B.11: A state transition diagram for the example problem

considering. Each node on the graph uses a unique tag to represent a system state. The format of this tag is as follows:

$[\text{Parent\_tag}, (\text{Interpretation\_number}, \text{Transition\_number})]$

where:

Parent_tag is the complete tag for the predecessor of this state. The starting state has the tag [0].

Interpretation_number is a number given to the global interpretation that led to this new state. When we solve the confluences for a new system state then we may find that several interpretations arise from the specified boundary conditions i.e. we don't have a unique solution. In this case these interpretations are numbered sequentially, starting at 1. We then branch the simulation to consider

device transitions that are caused by any of these. Note that the **Interpretation_number** is not unique as we always restart numbering from 1 for each new state.

Transition_number is a number similar to the **Interpretation_number**. This time we are branching the solution because the same interpretation could result in several different device state changes. Again, we always restart the numbering from 1 so the numbers are not unique.

For our example problem the solution does not branch at either stage because we are able to use relational algebra to rule impossible transition sequences. This would not have been the case if the set-point of rv1 had been greater than that of rv2, because then we could not have deduced which relief valve would open first.

The state tags are used by the program output to describe the state transitions that occur. For each state we output information in the following form:

Starting Pathname = <State tag>

The starting device states were:

<Device states>

The boundary conditions were: <Boundary conditions>

The following interpretations and transitions
were considered possible:

<Transitions and Interpretations>

where <Transitions and Interpretations> is a list, each element of which is in the following format:

Pathname = <New pathname>

The interpretation considered was:

<Interpretation>

The device <Device> changed state from <Old state> to <New state>

Within the program output the stream variables are referred to by names which are automatically created during a problem initialisation phase. For example, 'f1' and 'p1' are the mass_flow and pressure within stream '1', respectively.

We are now in a position to present the output exactly as it comes from the program:

The following types of stream variables are being used :

mass_flow

pressure

OUTPUTTING RESULTS

The facts known at the start were

```
f1>0    f2=f1    f3=0    f5=0
area(rv1)=0    p2-p5<10    f7=f4    f6=f5
f10=0    f11=0    area(rv2)=0    p7-p11<10
f9=f8    f12=f11
```

```
/* Note that only f1>0 was user supplied, the rest of the facts were
   deduced from the state information. */
```

Starting Pathname = [0]

The starting device states were:

```
r1=working    s1=working    rv1=closed    r2=working
r3=working    s2=working    rv2=closed    r4=working
r5=working
```

The boundary conditions were: del(p1)= + del(patm)=0

The following interpretations and transitions
were considered possible:

Pathname = [[0],(1,1)]

The interpretation considered was:

```
del(f1)= +    del(f10)=0    del(f11)=0    del(f12)=0    del(f2)= +
del(f3)=0    del(f4)= +    del(f5)=0    del(f6)=0    del(f7)= +
del(f8)= +    del(f9)= +    del(p1)= +    del(p11)=0    del(p2)= +
del(p5)=0    del(p7)= +    del(patm)=0    del(area(rv1))=0
del(area(rv2))=0
```

The device rv1 changed state from closed to fluttering

Starting Pathname = [[0],(1,1)]

The starting device states were:

```
r1=working    s1=working    rv1=fluttering    r2=working
r3=working    s2=working    rv2=closed    r4=working
r5=working
```

The boundary conditions were: del(p1)= + del(patm)=0

The following interpretations and transitions
were considered possible:

Pathname = [[[0],(1,1)],(1,1)]

The interpretation considered was:

```
del(f1)= +    del(f10)=0    del(f11)=0    del(f12)=0    del(f2)= +
del(f3)= +    del(f4)= +    del(f5)= +    del(f6)= +    del(f7)= +
del(f8)= +    del(f9)= +    del(p1)= +    del(p11)=0    del(p2)= +
del(p5)= +    del(p7)= +    del(patm)=0    del(area(rv1))= +
del(area(rv2))=0
```

The device rv1 changed state from fluttering to open

Starting Pathname = [[[0],(1,1)],(1,1)]

The starting device states were:

r1=working s1=working rv1=open r2=working
r3=working s2=working rv2=closed r4=working
r5=working

The boundary conditions were: del(p1)= + del(patm)=0

The following interpretations and transitions
were considered possible:

Pathname = [[[0],(1,1)],(1,1)],(1,1)]

The interpretation considered was:

del(f1)= + del(f10)=0 del(f11)=0 del(f12)=0 del(f2)= +
del(f3)= + del(f4)= + del(f5)= + del(f6)= + del(f7)= +
del(f8)= + del(f9)= + del(p1)= + del(p11)=0 del(p2)= +
del(p5)= + del(p7)= + del(patm)=0 del(area(rv1))= +
del(area(rv2))=0

The device rv2 changed state from closed to fluttering

Starting Pathname = [[[0],(1,1)],(1,1)],(1,1)]

The starting device states were:

r1=working s1=working rv1=open r2=working
 r3=working s2=working rv2=fluttering r4=working
 r5=working

The boundary conditions were: del(p1)= + del(patm)=0

The following interpretations and transitions
 were considered possible:

Pathname = [[[[[0],(1,1)],(1,1)],(1,1)],(1,1)]

The interpretation considered was:

del(f1)= + del(f10)= + del(f11)= + del(f12)= + del(f2)= +
 del(f3)= + del(f4)= + del(f5)= + del(f6)= + del(f7)= +
 del(f8)= + del(f9)= + del(p1)= + del(p11)= + del(p2)= +
 del(p5)= + del(p7)= + del(patm)=0 del(area(rv1))= +
 del(area(rv2))= +

The device rv2 changed state from fluttering to open

Starting Pathname = [[[[[0],(1,1)],(1,1)],(1,1)],(1,1)]

The starting device states were:

r1=working s1=working rv1=open r2=working
 r3=working s2=working rv2=open r4=working
 r5=working

The boundary conditions were: del(p1)= + del(patm)=0

The following interpretations were considered possible:

del(f1)= +	del(f10)= +	del(f11)= +	del(f12)= +	del(f2)= +
del(f3)= +	del(f4)= +	del(f5)= +	del(f6)= +	del(f7)= +
del(f8)= +	del(f9)= +	del(p1)= +	del(p11)= +	del(p2)= +
del(p5)= +	del(p7)= +	del(patm)=0	del(area(rv1))= +	
del(area(rv2))= +				

No device state changes were considered possible.

It took 400.02 CPU secs to analyse the network given in file out/10

The goal succeeded

Appendix C

A list of Example Loss Prevention Constraints

This appendix lists some example constraints which have been derived from the loss prevention literature (e.g. from the HAZOP examples mentioned in Chapter 2). All the constraints are intended to prevent undesirable situations. However, their actual form makes some assumptions about company policy and the rest of the design or plant. Consequently, not all the constraints will apply to every project. Indeed, some of them may even be redundant or inconsistent when taken together. Allowing such flexibility is the essence of our approach.

The reader should consider these constraints from the point of view of the different representations they require. In Chapter 6 we have already shown how some of them can be implemented using our Constraint Tools System.

Global Constraints

Prevent the following everywhere in the plant (list very incomplete):

- Build up of explosive mixtures.
- Unexpected presence of flammable mixtures.
- Release of toxic materials except to a “safe sink”.
- Undesired polymerisation.
- Excessive corrosion.
- Ice formation.
- Undesired contamination.
- Removal of safe-guards/contingency action.

- Violation of area classifications (e.g. not installing spark proof equipment in a zone 1 area).

Local Constraints

The following constraints are listed under the type of equipment they apply to:

Tank:

- Fit low-level and high-level alarms.
- Use inert venting gas for liquids above flash point.
- Never overfill tanks.
- Always vent tanks.
- Install vacuum breaker
- Include mixing where necessary.

Buffer-tank:

- Fit high-level and low-level alarms.
- Don't allow tank to empty completely in 'normal' operation.

Pressure-vessel:

- Ensure adequate pressure relief.
- Design for total vacuum.

Furnace:

- Never overheat tubes.
- Never allow the furnace to be lit when there is no flow in the tubes.

Burner:

- Never allow gas flow when the flame is out.
- Never allow gas flow when the air flow is zero.

pump:

- Don't allow cavitation (boiling).
- Don't 'deadhead' pumps (i.e. pump running but no flow because of downstream blockage).
- Don't have pump running but loss of suction (upstream blockage).
- Always include a kickback loop for a pump (ensures satisfaction of last two constraints).
- Prevent reverse flow while pump is electrically connected (becomes a dynamo).
- Prevent reverse flow through pump (more general).
- Ensure adequate relief of full pump pressure.
- Design connected pipe-work and vessels for full pump delivery pressure.

Compressor:

- Prevent liquid entering compressor (even small droplets)

Heat exchanger:

- Pass dirtiest fluid through the tubes.
- Prevent polymerisation.

Reactor:

- Prevent build up of dangerous compositions (e.g. by loss of temperature, flame or one of the reactants).
- Prevent loss of cooling.

Vent-system:

- All inlet points must have unrestricted path to vent header.

Relief-valve:

- Don't try to pass two-phase flow if only sized for gas relief.
- Relief pressure must be between the operating pressure and the upper design pressure of the system being protected.

- All valves between a relief valve and the vessel being relieved must be locked open.

Bursting-disc:

- Don't try to continue normal operation with a ruptured disc.

Conduit:

- Prevent back-flow (generally applicable).

Some more complex examples

These examples show cases where a safety device was prevented from performing its intended function. As such, they do not directly describe constraints but rather situations where constraints were violated. It is left as an exercise for the reader to deduce the constraints that we need to enforce, in order to prevent repetition of these accidents.

1. A trip system was designed to activate if the pressure at the inlet to a pump was low. However, a filter was installed BETWEEN the pressure indicator and the pump inlet. This meant that when the filter became blocked the pump suction was lost but the trip not activated.
2. An extra line was installed into a vessel which meant that the relief valve was undersized for the maximum possible flow.
3. The removal of a flow restriction plate into a vessel meant that again the relief valve was undersized for flow.
4. Unexpected backflow into a vessel meant that yet again the relief valve was undersized for flow.
5. Two vessels were relieved by the same relief valve. Unfortunately, an isolation valve was installed on the line between them. Inevitably it was closed accidentally, thus isolating one vessel from its source of relief.
6. Steam was passed up the stack from a low temperature relief system to disperse the effluent. Ice formed, blocking the stack.

Most of these accidents were attributable to plant modifications. However, similar errors could just as easily have arisen during design. They are typical of the types of

problems that should be picked up by a HAZOP study. Indeed a central concept of performing HAZOP studies is to consider the *designer's intention* for pieces of equipment and how that intention can be violated.

Appendix D

Extensions to existing Knowledge Craft Tools

The result of this project is a set of tools which extend the facilities of Knowledge Craft. These are programmed in a domain independent manner so they could be used by any task for which they appeared to be appropriate. However, they are, of course, aimed at the process engineering design domain in which we are interested. Some of the tools are merely extensions to existing Knowledge Craft features, such as additional commands for The Palm Network Editor. Some of these extensions are described in this appendix. Other tools, which provide entirely new knowledge representation modules, have already been described in the main text of this thesis.

D.1 Using Palm for Drawing Networks

New commands have been added to Palm to allow it to be used for drawing networks, as opposed to hierarchies. An example new Palm is shown in Figure D.1(a). The new commands are as follows:

1. A command for moving a schema in the Palm display and redrawing the links to it.
2. A command for erasing a schema from the Palm display without actually deleting the schema itself.
3. A command for creating an instance schema i.e. it first prompts for a schema prototype before running the Palm create-schema command. The new schema is instance linked to the prototype.
4. Commands for saving a Palm display to a file and for recreating it on a different invocation of Palm, perhaps in a different Knowledge Craft session.

(b)

Small Palm Editor

Options/ Misc/ Print-sc/C-Inst Coco/P-Edit C-Schema/C-Link Del-schema/Del-link Sc/File Edit

Type in name of schema for PRINT SCHEMA or mouse-r-1 to select from fixed-menu[BURSTING-DISC]:

FIXED MENU BUILDER

CENTRIFUGE
DRIER
FILTER
VESSEL
HE-HOT-SIDE
HEAT-EXCHANGE-DEVICE
HE-COLD-SIDE
SIMPLE-DISTILLATION-SYST
DISTILLATION-SYSTEM
BURSTING-DISC
RELIEF-POINT
RELIEF-VALVE
SAFETY-EQPT
PRESSURE-VESSEL
BUFFER-TANK
BOILER
LIQUID-CONTAINER
TANK
DISTRIBUTION-EQPT
FURNACE
HEATER
CENTRIFUGAL-PUMP
PUMP
RECIPROCATING-PUMP
PRIME-MOVER
COMPRESSOR
PLANT-ITEM
CONTROL-EQPT

(c)

(a)

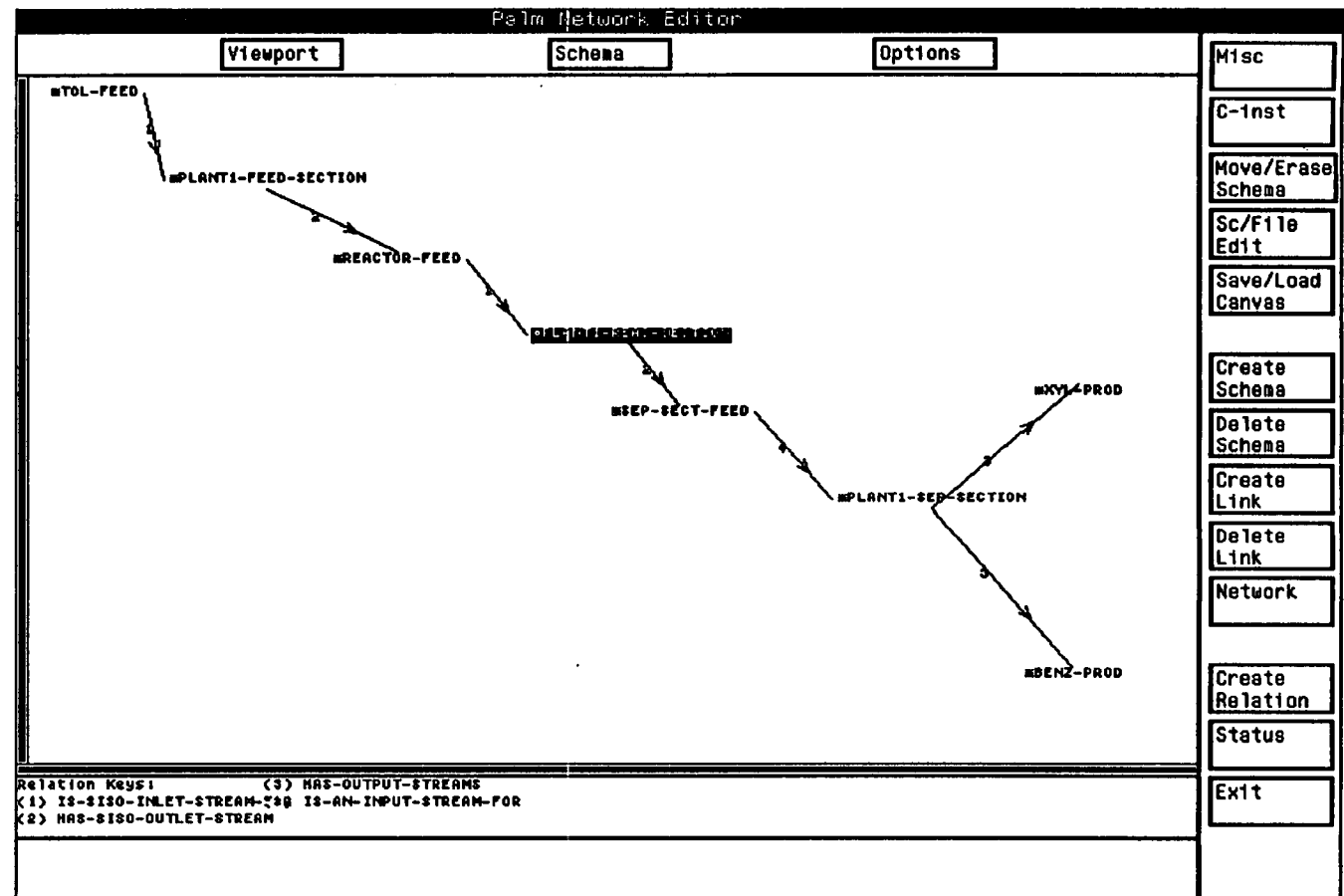


Figure D.1: (a) An extended Palm, (b) a Small Palm and (c) a Fixed-menu

It was possible to create these commands because of the transparency of the Knowledge Craft system. In particular, we were able to look at example instances of Palm and work out how the schema positions and links were represented. We were also able to simulate mouse clicks in the main Palm viewport, thus using the standard Palm create-schema command to redraw the display. This has a few disadvantages; for example, it takes about 65 seconds to draw a full screen network, as opposed to the 15 seconds it normally takes Palm to draw a full-screen hierarchy. In addition, we are using information which is not in the Knowledge Craft manual and may be less static than properly supported features. In other words, a future version of Knowledge Craft may represent Palm instances differently, which would necessitate redoing some of our work. Nevertheless, as the alternative was to completely recreate Palm from scratch we still believe that we are ahead.

D.2 Using Palm to Simplify Knowledge Craft Programming

We have added other commands to Palm which allow it to be used for actually programming within Knowledge Craft. In particular, our new version, in combination with our New Save Manager (described below), is very useful for constructing command hierarchies and graphical workbenches. The features are as follows:

1. A command for editing a schema within the SUN COMMON LISP Editor. The schema definition is written to an edit buffer, in a format which can be re-evaluated when you have finished. This is useful for making many changes to the schema at the same time. In addition, three buffers are used so that up to 3 schemata can be edited at once. This makes it easy to yank text between them.
2. A command to search the Palm display for a schema name which contains a given sub-string. The display is scrolled so that this schema is at the centre.
3. A new relation which can be used to display all the schemata connected with a command hierarchy on a single tree.
4. A new Coconut command which ensures that inherited values are not cached into the schema being edited. The existing Palm-Coconut interface results in all inherited slots and values being cached into the edited schema regardless of whether or not the slots are changed.
5. Interfaces to fixed-menus, as described below.

Using these commands we can easily display the hierarchy for an existing workbench, such as Palm itself, and copy or change it as we desire.

D.3 Small Palm

Palm provides access to a lot of useful commands but most of them can only refer to schemata displayed within the main Palm viewport. This makes it rather awkward to use for one-off operations on schemata you don't currently have displayed in an instance of Palm. In addition, because of the efficiency problems already described elsewhere, it is not advisable to open too many Palm instances at once. This causes problems because we will have to use the same instance of Palm in several different contexts.

To get around these difficulties we have created a new task called Small Palm, which provides access to a subset of Palm facilities. Its main advantages are the following:

- Schema names can be typed in or selected from fixed-menus, rather than having to be selected with the mouse.
- It can be used in more than one context since it doesn't have a display that needs to be kept consistent with the context.
- It doesn't create garbage irretrievably, which Palm unfortunately does.

A typical Small Palm instance is shown in Figure D.1(b).

D.4 The Fixed-menu Builder

A new task called the Fixed-menu Builder has been provided. It allows items to be set-up on a viewport which acts as a permanent menu. Other tasks can interface to this by the addition of simple-commands which all call the same function. An example Fixed-menu is shown in Figure D.1(c).

An item is selected from a Fixed-menu by simply clicking mouse-l-1 in the menu viewport. This will cause the item text to be stored as a string in a buffer. This text is then potentially available for use with any command, in any other task.

Suppose that a command is waiting for you to type in a string, such as a schema name. If this command has been set-up in a certain way then clicking mouse-r-1 will make the text from the Fixed-menu buffer the default input for this command. You can then accept this default, by hitting the "Return" key, or you can yank it into the command buffer to edit in the normal way.

All the new workbenches which we have created interface to Fixed-menus like this. In addition, most of the existing Knowledge Craft workbenches have been altered so that

they also do so. As an example refer to Figure D.1(c) and suppose that we want to place the “heater” schema in the Palm display. To do this we will run the “create-schema” command in Palm, which will prompt for the schema name. We then click mouse-l-1 on the “heater” text displayed in the Fixed-menu. This temporarily resumes the Fixed-menu but then pauses it again straight away. We now simply click mouse-r-1 to make “heater” the default value for the create-schema command, and then hit “Return” to accept the default. We could later move directly to the Small Palm workbench and print the “heater” schema in a similar manner. The default value persists until we select another item from a Fixed-menu.

Extra commands are provided to scroll Fixed-menus, search them for items matching a specified sub-string and so on. See the complete version of the Constraint Tools Manual [112] for more details.

D.5 The SUN COMMON LISP Editor

The SUN COMMON LISP Editor is an EMACS editing system that sits behind Knowledge Craft. If you ask to go into the editor then the Knowledge Craft root viewport is completely covered by the editor viewport until you exit. From the editor you can edit files and load them back into Knowledge Craft. In addition, you can just reload a single function definition, which speeds up the edit-load-debug cycle. However, when a LISP error occurs within Knowledge Craft you cannot simultaneously have access to the editor and the LISP debugger. Consequently, we have found it easier to edit files within separate UNIX windows and just reload the whole file of functions.

We do use the editor, however, to edit schema definitions. This is useful when we wish to make many changes to a schema at once or to copy information between schemata. Neither of these can be easily done within P-edit but can be done using our extensions. This facility can be accessed from our extended version of Palm, or from Small Palm.

We also use the editor as a temporary buffer when we write schemata definitions to a file. That is, we first write all the definitions to the buffer and then write the whole buffer to the file. This means that if we get an error whilst writing the definitions we haven't corrupted an existing file.

D.6 Graphics and the Command System

Knowledge Craft graphics are used to create viewports, icons, boxes, circles etc.. If you know what you wish to draw then it is relatively easy to use the graphics system, though

it can be a little 'buggy'. Thus it is easy to draw graphs and networks; indeed, a general graph drawing tool has been constructed during work outside the scope of this thesis. However, there is no "paintbox" or bitmap editor, so it is difficult to use the system as a drawing tool. Murray Laing, a member of our group, has solved part of this problem by writing a C program that converts SUN bitmap files into Knowledge Craft bitmap format. Thus we have been able to use the SUN Icon Editor to create Knowledge Craft icons. Another useful extension would be a graphics workbench that allowed the mouse selection and rearrangement of graphics primitives in order to construct composite items.

Another criticism is that there is no Knowledge Craft equivalent to the SUNTOOLS "stuffing" facility, which allows you to pick up text with the mouse and place it down again in any UNIX window. This is unfortunate, since this facility makes it extremely easy to pass information between different windows running different tasks. However, it is probably not possible to allow stuffing within Knowledge Craft because all the text is actually graphics, rather than ASCII characters. In addition, we are using windows within windows so the operating system would not know where to pick up or put down the text. Consequently, this is really a limitation of SUN COMMON LISP windows, rather than Knowledge Craft.

The window in which you invoked Knowledge Craft is called the "terminal-window". This remains a UNIX window and so can be used to stuff information from other UNIX windows. This facility can be used by interrupting Knowledge Craft so that you temporarily return to LISP within the terminal-window. You can then stuff information directly into the LISP interpreter. For example, you could pick up a function definition from a file and load it into the LISP image.

This facility is only of limited use, however, as it cannot transfer text within Knowledge Craft. A more general facility, for use within Knowledge Craft viewports, would be extremely useful. We have already described the "Fixed Menu Builder", which achieves this in part. Unfortunately however, it requires that you explicitly set up items on a menu, instead of being able to pick up any text strings currently displayed in any viewports.

D.7 The Save Manager

Knowledge Craft provides several nice editing facilities, such as the Palm Network Editor and the P-edit schema editor, as well as providing the facilities to easily build and modify some new ones appropriate to your own needs. However, all these features are next to useless if the results of an edit cannot be permanently saved in an efficient and consistent manner. Knowledge Craft allows for two solutions to this problem; namely the database interfaces and the Save Manager. The former of these seems

most appropriate, in the long term, for managing large schemata sets (such as would arise from a complete plant design). Unfortunately, it requires either that you run Knowledge Craft on a VAX or that you buy SQL for a SUN (or some other machine). Neither of these is appropriate for a SUN-based research project or other experimental application. We must therefore take the second route which involves storing schemata in files and updating the files periodically during an editing session.

The Save Manager provided by Knowledge Craft allows you to write a set of schemata to a file but suffers from the following drawbacks:

1. There is no way of describing the contents of a file except by editing in comments after each save.
2. There is no record of what the file should hold; hence there are no checks for drastic errors, such as accidentally deleting schemata during a session.
3. There is no way of including additional terms before and after the schemata definitions, except by editing the file after each save. Such a feature is useful to write comments to the file or to change the environment for the duration of the load. For example, we may want to change some of the Knowledge Craft switches or to perform some actions after the schemata are defined (perhaps to recompile demons or to implement similar features of your own).

These are fairly serious limitations so we considered it worthwhile creating a specialised save manager to tackle them. This uses an information schema to describe the contents of a file, how to obtain the save-set and the structure of the resulting file. The schemata are written to the file in a format suitable for reloading. The file contains the declaration and compilation statements that are needed to recreate relations as they were before. In addition, you can set both local and global Knowledge Craft switches for the duration of the load. You can also set the values of your own switches or introduce your own declaration or compilation statements. These features are quite important in the Constraint Tools System, as the way in which Knowledge Craft defines schemata is quite sensitive to the values of its switches and the order in which schemata and contexts are loaded.

The New Save Manager has both a graphical interface and a programmers interface. The former uses Coconut to provide a slot-filler to perform a save. This allows you to use a schema restriction or a LISP expression to specify the schemata that should be saved in a file. In addition, it tells you which schemata are missing from when the file was last saved and which have been added. Thus you can inspect a save for unexpected changes, before writing the file. Figure D.2 shows us using this facility to save all the schemata that implement the Walnut workbench. The programmers interface can be used by an application to save files non-interactively. The Constraint System and the Refinement Manager both use this facility.

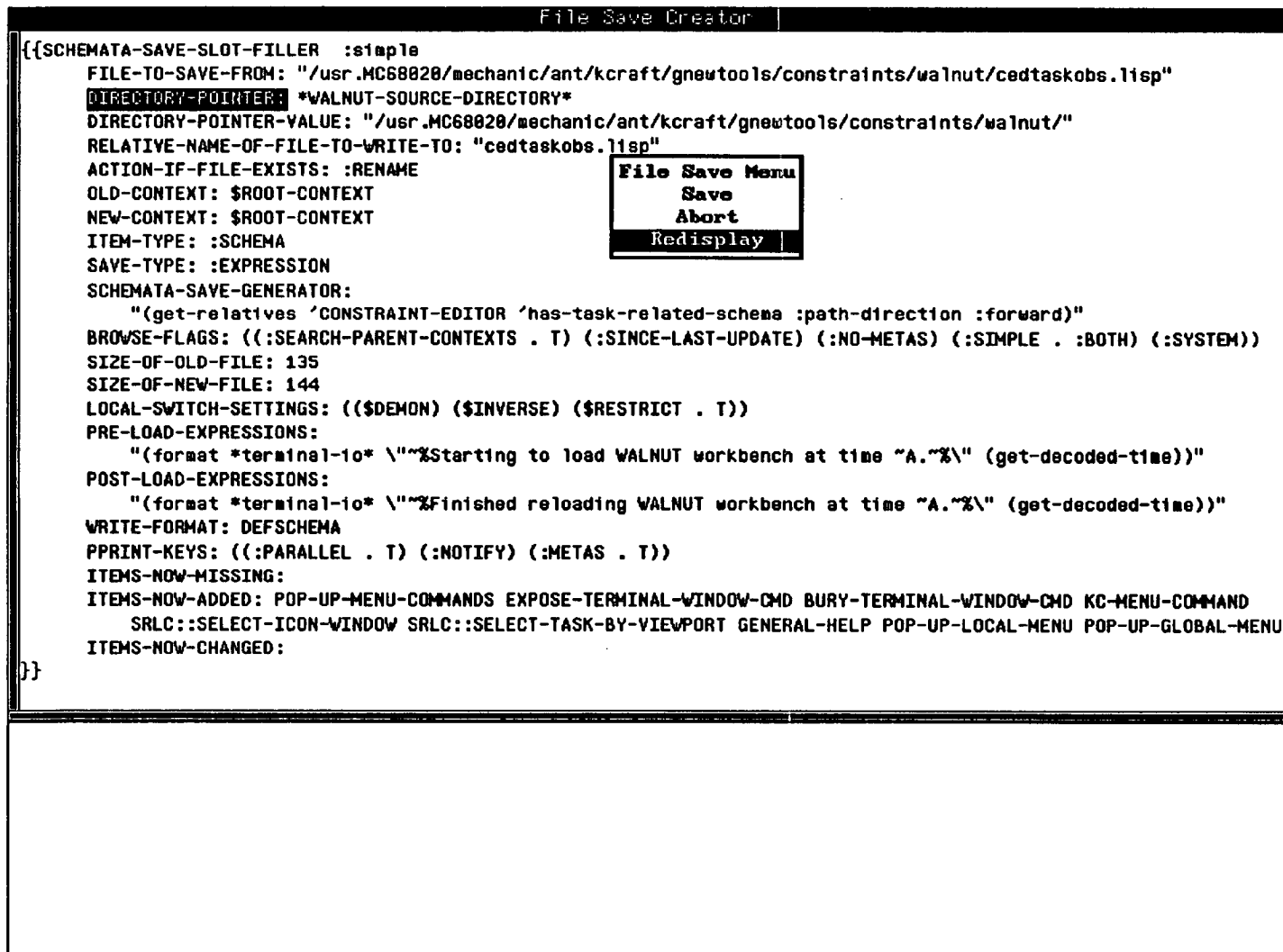


Figure D.2: Interactively using the New Save Manager

Appendix E

File Used to Load the Constraint Tools Examples

#|

This file is used to load the examples of using the Constraint Tools System. The examples themselves have already been described elsewhere.

To load the file type:

```
> (load <filename>)
```

into the kc-listener, where <filename> is the complete pathname for this file (within double quotes). You can then examine the context tree and use the Refinement Manager to see what you have loaded up.

The load takes about 5 minutes.

|#

```
;; ++++++
```

#|

First we define some functions which can be typed into a kc-listener. These resave the files not directly owned by the Refinement System or the Constraint System. Note that you should make sure you invoke a Walnut save from the \$root-context if you wish to save constraint definitions for that context. See the Constraint Tools Manual for more details.

|#

```
(defun ant-save-root-context-file (file)
  (let ((save-info-schema (find-old-save-info-schema file)))
    (call-method save-info-schema 'file-save)))
```

```
;; -----
```

```
(defun ant-save-root-context ()
  (save-proprel-file
    (concatenate 'string *ant-root-context-dir* "relsobs.lisp"))
  (mapc #'(lambda (fil)
    (ant-save-root-context-file
      (concatenate 'string *ant-root-context-dir* fil)))
    '("hierarkobs.lisp"
      "processobs.lisp"
      "topologyobs.lisp"
      "datobs.lisp"))))
```

```
#|
```

To use this function, you type:

```
> (ant-save-root-context)
```

into the kc-listener.

```
|#
```

```
;; ++++++
;; ++++++
```

```
;; ++++++
;; Loading and saving info.....
;; ++++++
```

```
;; The commands below find and load the files needed for these examples.
;; The files which contain info-schemata are described using a directory
;; pointer which is bound to a directory name (e.g. the symbol
;; *ant-root-context-dir*). This enables you to move whole
;; directories very easily by rebinding the pointer. However, it is
;; recommended that you both redefine the pointer and move the directories
;; OUTSIDE of lisp.
```

```
;; Setting directory names....
```

```
(dir-pointer-setq *ant-root-context-dir*
  "/usr/chemeng1/ant/kcraft/main/examples/rootcont/")
```

```
(dir-pointer-setq *elspeth-refinement-dir*
  "/usr/chemeng1/ant/kcraft/main/examples/anthra-plant/design/")
```

```
(dir-pointer-setq *elspeth-constraint-dir*
```

```

"/usr/chemeng1/ant/kcraft/main/examples/anthra-plant/constraints/")

(dir-pointer-setq *ants-refinement-example-dir*
  "/usr/chemeng1/ant/kcraft/main/examples/plant1/design/")

(dir-pointer-setq *ants-refinement-example-constraints-dir*
  "/usr/chemeng1/ant/kcraft/main/examples/plant1/constraints/")

(dir-pointer-setq *heating-circuit-refinement-dir*
  "/usr/chemeng1/ant/kcraft/main/examples/heating-circuit/design/")

(dir-pointer-setq *heating-circuit-constraints-dir*
  "/usr/chemeng1/ant/kcraft/main/examples/heating-circuit/constraints/")

;; -----

;; -----
;; Root Context..... |
;; -----

(mapc #'(lambda (fil)
          (load (concatenate 'string *ant-root-context-dir* fil)))
      '("utils.lisp"
        "misc"
        "relsobs.lisp"
        "root-context-design-constraint-1"
        "root-context-design-refinement-1"
        ))

;; -----

;; We have loaded the basic definition of the "design-constraint" schema
;; but we have NOT yet loaded the $root-context constraint definitions.
;; These will be loaded AFTER their home-schemata, which is the recommended
;; procedure.

;; Turn-off all demon checking because we may get errors from constraint
;; checking demons being mentioned by slots before they have been defined:

(setq *ant-demon-switch* (get-switch '$demon))
(set-switch '$demon nil)

;; -----

(mapc #'(lambda (fil)

```

```

        (load (concatenate 'string *ant-root-context-dir* fil)))
'("hierarkobs.lisp"
  "topologyobs.lisp"
  "datobs.lisp"
  "processobs.lisp"
  "root-context-design-constraint-2" ;; loading constraint definitions
  ))

;; -----
;; Refinements for plant1 Example |
;; -----

(load-refinement-contexts
  'design-refinement '*ants-refinement-example-dir*)

;; -----
;; Constraints for plant1 Example |
;; -----

(load-constraints-contexts
  'design-constraint '*ants-refinement-example-constraints-dir*)

;; -----
;; Refinements for anthra-plant Example |
;; -----

(load-refinement-contexts 'design-refinement '*elspeth-refinement-dir*)

;; -----
;; Constraints for anthra-plant Example |
;; -----

(load-constraints-contexts 'design-constraint '*elspeth-constraint-dir*)

;; -----
;; Refinements for heating-circuit Example |
;; -----

(load-refinement-contexts
  'design-refinement '*heating-circuit-refinement-dir*)

;; -----
;; Constraints for heating-circuit Example |
;; -----

```

```
(load-constraints-contexts
  'design-constraint '*heating-circuit-constraints-dir*)

;; *****

;; Turn-on demon checking as it was before.

(set-switch '$demon *ant-demon-switch*)

;; Clean the kc context tree.

(clean-all-contexts)

;; -----
```

Appendix F

Using CRL-OPS Rules for Partial Simulation

This appendix follows on from the author's early work using Knowledge Craft, which explored the idea of using rules to implement *partial simulation* [110]. The word "partial" is used here to emphasise that the aim is not to provide a simulation tool that is complete in any qualitative or quantitative way but rather one that allows a user to maintain simple dependencies between pieces of design data. For example, the user should be able to write a rule that sets the flow through a valve to zero whenever the valve position is set to closed. In addition, the user should be able to write another rule that updates the flow whenever the valve is opened again. This latter case is more difficult, however, because one might have to check against blockages elsewhere in the plant. The user may therefore decide to forego this option and perform only a "partial" simulation.

There are several issues to address with respect to the use of such simulation rules:

1. Is there any danger of circularity in the rules such that propagation never ceases? This could arise where there are feedback loops in the digraph (influence graph) of the process. Such loops are caused by recycle loops of material, energy or control information.
2. What action should be taken if there is insufficient information to perform the test part of a rule?
3. What strategy should one use to decide which of conflicting rules should fire and in what order?
4. Should one build causality into the rules or let them work in design mode also?
5. Does one need a means for making assumptions and backtracking?
6. If the rules are incomplete, and whilst they are firing, the design may be in an inconsistent state e.g. a valve is marked as closed but the flow through it has not

yet been set to zero. Is this a problem? Should one queue all constraint checking at such times?

In the initial work on this problem a new rule system was created, which represented rules as LISP lists and used demons to implement their automatic firing. This approach was similar to that which is still used for the Constraint System, in that data was accessed by starting from a known point. In contrast, more recent work has explored the use of CRL-OPS rules instead, which access data by pattern matching. Chapter 4 has already discussed the difference between these two approaches.

The first approach will not be described any further here as it is adequately covered by an early paper [110]. Instead, this section will concentrate on the use of CRL-OPS rules. Initially the author was quite interested in this problem, partly as a follow up to his work on qualitative simulation. Indeed, the initial reason for constructing Propagating-relations was so that one could write CRL-OPS rules in a natural and non redundant manner. In particular, it was desirable to be able to separate topology from function; for example, so that one could apply the same rules to any object that has topology, regardless of what its function is. This is facilitated by the use of Propagating-relations to represent the necessary part-whole relationships. This will be demonstrated within the current appendix, which describes how Propagating-relations are defined and how they interact with CRL-OPS. The reader should note, however, that the Constraint Tools System does not actively use CRL-OPS rules but still has many uses for Propagating-relations. For example, chapter 6 has already shown that they are used to propagate data around a refinement tree.

F.1 CRL-OPS and Propagating-relations

CRL-OPS is based on the OPS5 language, which is a common vehicle for creating forward chaining rule bases. However, CRL-OPS has extensions that allow rules to be integrated with the CRL schema language in the following manner:

- The rules can be pattern matched against CRL schemata.
- The action part of the rules can execute any LISP commands; thus they can use CRL functions to edit schemata.

CRL-OPS is a useful extension of OPS5 but still suffers from several limitations. In particular, LISP functions on the left hand side of a rule can only be used as predicates i.e. to test data rather than to generate it. Furthermore, they mustn't have any side-effects and should really only refer to the schema that the relevant condition

element is currently matching. This makes it difficult to integrate CRL-OPS with other knowledge representation utilities, such as a truth maintenance system written in LISP or C. Furthermore, CRL-OPS uses the RETE algorithm to create a network of current matches; and updates this network whenever schemata are altered. To do this easily, a working memory element is created for every "literalized" schema, in every context. Although this means that CRL-OPS rules fire extremely quickly it has the disadvantage of slowing down schema manipulation. In particular, the creation and deletion of contexts can be cripplingly slow. This is very limiting if one tries to use CRL-OPS in conjunction with the Refinement Manager described in earlier chapters.

Nevertheless, CRL-OPS does extend OPS5 by recognising inheritance across is-a and instance links. For example, if a condition element in a rule refers to a class object then that element can be matched by any schema below the class object in an is-a hierarchy. In addition, the attributes of the element will match against inherited slots and values, rather than just against those explicitly contained within the schema being tested. Unfortunately, both these features only work across is-a and instance links, as opposed to across user defined relations. This causes a problem for the representation of part-whole relationships, such as would arise from trying to separate topology and function. To solve it one must use some other means to propagate changes across user defined links. Typically this will involve the use of demons to copy values across. This is one of the major functions of Propagating-relations, as will be described below. In addition, they provide a means of specifying data mappings that could be used in other situations.

F.2 The heating-circuit example

This example is based around the flowsheet in Figure F.1, which consists of a propane heating circuit and a trip system. These together are supposed to prevent the export of propane at a temperature below -10 Centigrade. This purpose is represented by the stream3-constr1 constraint, which is shown in Figure F.2.

The discussion will consider three versions which differ only in the set-point of the trip system. The basic problem is to decide, for each version, whether or not the constraint will actually be violated and whether or not one can write rules, and manage the constraint system, in a way that always returns this answer correctly.

This example actually follows on from those already presented in chapter 6. Indeed, the top level for the problem is the heating-circuit context, which has already appeared on the context tree in Figure 6.8. The design objects are all stored in the context subtree which begins at this top level. The Palm display for this context is shown in Figure F.3. From this the reader can see how one can map from objects on the flowsheet to schema names within Knowledge Craft. In particular, it is worth noting

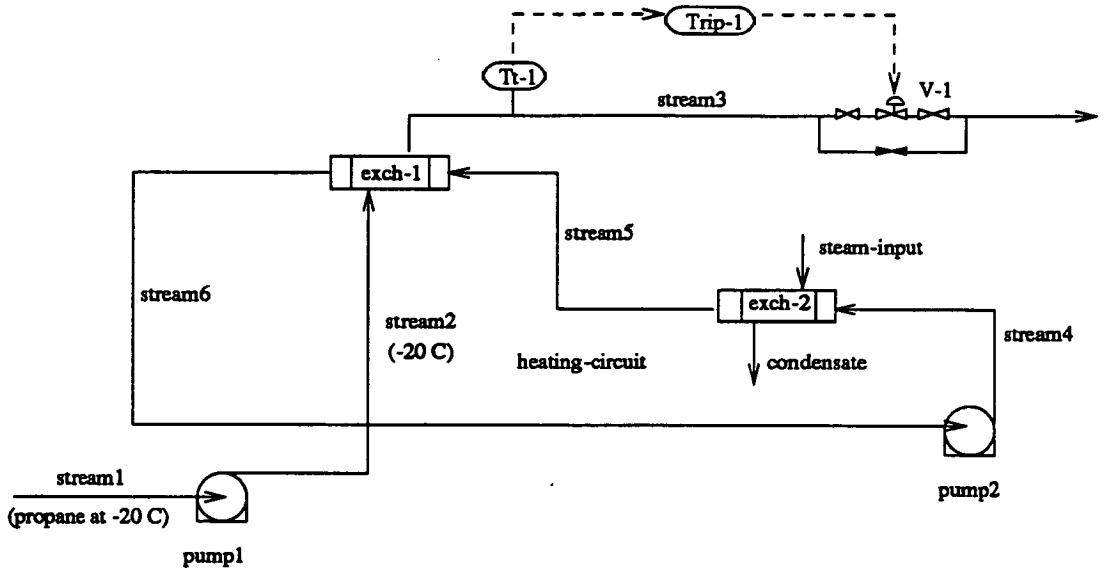


Figure F.1: Propane export plant with trip protection

how some objects, such as the heat exchangers, are split into several schemata. One can then use Propagating-relations to tie the pieces together.

As an example of this consider the two Propagating-relations in Figure F.4 and the three schemata in Figure F.5. Together, these show how data is mapped between the objects that make up a heat exchanger. In particular, note the correspondence between the slots in exch-1 which begin with the prefix “hotout#”, the slots in exch-1-hot-side which begin with the prefix “out#” and the slots in stream6 which contain the basic process data (i.e. temperature, total-kmolar-flow and so on). These correspondences are maintained by the two Propagating-relations, whose “mappings” slots describe how the prefixes match. Where the mapping information refers to the symbol “:source” then this is taken to mean there is no prefix on the slots at this end of the link. For example, the “has-he-hot-side” relation in Figure F.4 contains the mapping:

“(hotout out)”.

This means that the object at the start of this link will have a slot called “hotout#temperature”; and the object at the end the corresponding slot called “out#temperature”.

```

{{stream3-constr1
  is-design-constraint-for: stream3
  description:
    "This constraint checks that you are not exporting
    propane from the plant at a temperature less
    than -10 C. This is to protect the downstream
    section, which is not designed for such duty."
  external-form:
    "(not (and (> #!~|total-kmolar-flow|? 0)
               (< #!~|temperature|? 263)))"
  local-slots-involved: total-kmolar-flow temperature
}}
```

Figure F.2: The main constraint for the propane export example

Similarly, the "has-siso-inlet-stream" relation has the mapping:

"(in :source)".

This time, therefore, the match is between a slot called "in#temperature" and one simply called "temperature". Further examples are given below.

The CRL-OPS rules that have been constructed within this work are listed in Appendix G. In addition, Figure F.6 shows an instance of the CRL-OPS workbench which is being used to display and fire these rules. The rules are listed in the viewport entitled "Production Memory", while the "Trace" viewport gives information about rule firings that have taken place. Most of the rules are simply listed by name but pump-rule-1 is being displayed in more detail. The workbench allows graphical debugging of a CRL-OPS rule set.

Now, the problem that has been posed is to assess the effect of the failure of pump-2. This can be simulated by simply changing the value of that object's status slot to "stopped", in the relevant version context. The rules can then be fired to completion within this context. The rest of this section is devoted to describing the sequence of rule firings that might take place.

The first rule to fire might be pump-rule-1, which can be interpreted as saying:

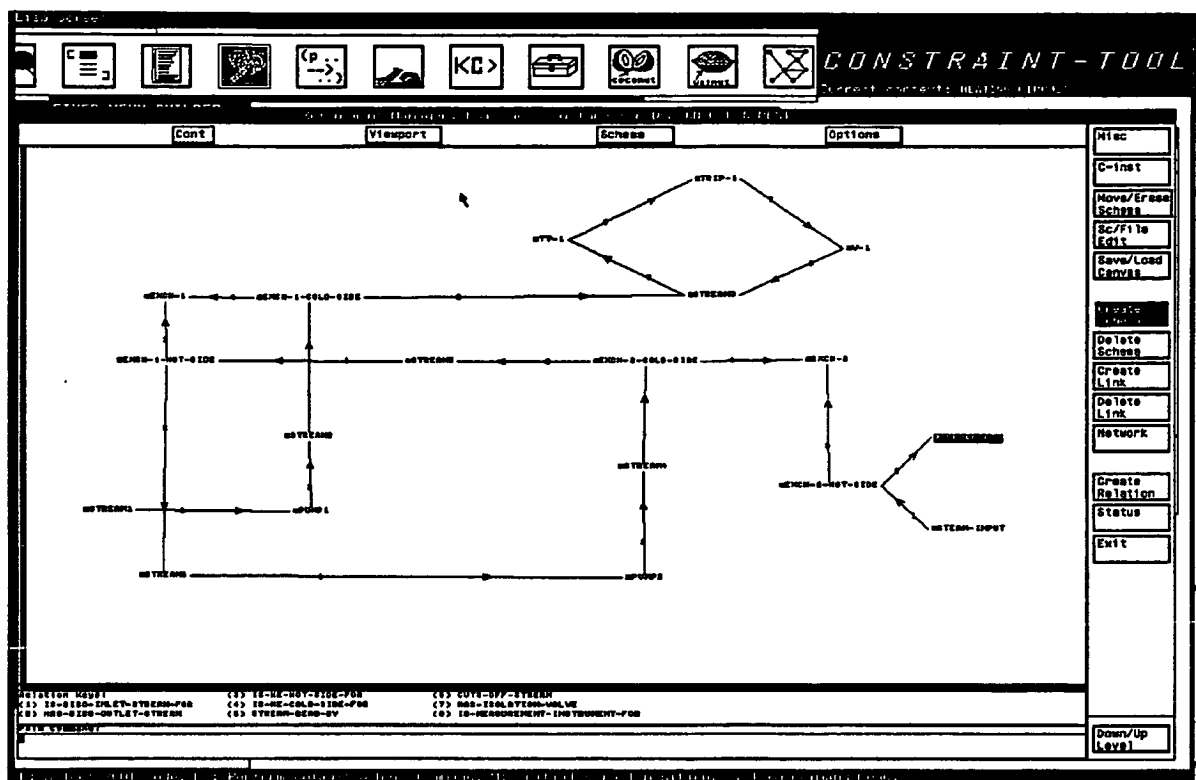


Figure F.3: The Palm network to represent the propane export plant

```

IF
  The status of a pump is "stopped"
  & The in#total-kmolar-flow of the pump has not already been set to 0.0
THEN
  Set the in#total-kmolar-flow of the pump to 0.0

```

As a result of this firing the "in#total-kmolar-flow" slot of pump2 is given the value 0.0. In addition, a Propagating-relation demon will then fire, in order to map this information to the "total-kmolar-flow" slot of stream6, the "out#total-kmolar-flow" slot of exch-1-hot-side and the "hotout#total-kmolar-flow" slot of exch-1. The first of these pieces of information might then cause siso-rule-2 to fire upon exch-1-hot-side. This rule maintains the principle "flow in equals flow out", which means that the "total-

```

(HAS-HE-HOT-SIDE :SIMPLE
  NIL
  :CONTEXT
  $ROOT-CONTEXT
  (IS-A PROPAGATING-SYSTEM-TOPOLOGY-RELATION)
  (INVERSE IS-HE-HOT-SIDE-FOR)
  (MAPPINGS (HOTIN IN)
             (HOTOUT OUT))
  (DOMAIN-PROTOTYPE COOLER)
  (RANGE-PROTOTYPE SISO))
Type Any Character To Continue

```

```

(HAS-SISO-INLET-STREAM :SIMPLE
  NIL
  :CONTEXT
  $ROOT-CONTEXT
  (IS-A SYSTEM-TO-STREAM-REL)
  (INVERSE IS-SISO-INLET-STREAM-FOR)
  (DOMAIN-PROTOTYPE SISO)
  (RANGE-PROTOTYPE STREAM)
  (MAPPINGS (IN :SOURCE)))
Type Any Character To Continue

```

Figure F.4: Example Propagating-relations

kmolar-flow” slot of stream5 becomes set to 0.0. A further round of propagation then maps this value to the “hotin#total-kmolar-flow” slot of exch-1. Numerous other rules may then fire but the one of interest to this discussion is heat-exchange-device-rule-2:

```

IF
  The hotin#total-kmolar-flow of a heat exchanger is 0.0
  & The heat-transferred by the heat exchanger is not already 0.0
THEN
  Set the heat-transferred by the heat exchanger to 0.0

```

This rule will fire upon exch-1 and set its “heat-transferred” slot to 0.0. Heat-exchange-device-rule-5 can then fire upon exch-1:

```
(EXCH-1 :SIMPLE
  NIL
  :CONTEXT
  HC-V1-HEATING-CIRCUIT-1
  (INSTANCE HEAT-EXCHANGER)
  (HAS-HE-HOT-SIDE EXCH-1-HOT-SIDE)
  (HAS-HE-COLD-SIDE EXCH-1-COLD-SIDE)
  (COLDIN#TOTAL-KMOLAR-FLOW 0.16)
  (COLDIN#TEMPERATURE 253)
  (COLDOUT#TOTAL-KMOLAR-FLOW 0.16)
  (COLDOUT#TEMPERATURE 283)
  (HOTIN#TOTAL-KMOLAR-FLOW 0.2)
  (HOTOUT#TOTAL-KMOLAR-FLOW 0.2)
  (HOTIN#TEMPERATURE 323)
  (HOTOUT#TEMPERATURE 303)
  (HEAT-TRANSFERRED 1.0))
```

Type Any Character To Continue

```
(EXCH-1-HOT-SIDE :SIMPLE
  NIL
  :CONTEXT
  HC-V1-HEATING-CIRCUIT-1
  (INSTANCE HE-HOT-SIDE)
  (HAS-SISO-INLET-STREAM STREAM5)
  (HAS-SISO-OUTLET-STREAM STREAM6)
  (IS-HE-HOT-SIDE-FOR EXCH-1)
  (IN#TOTAL-KMOLAR-FLOW 0.2)
  (OUT#TOTAL-KMOLAR-FLOW 0.2)
  (IN#TEMPERATURE 323)
  (OUT#TEMPERATURE 303))
```

Type Any Character To Continue

```
(STREAM6 :SIMPLE
  NIL
  :CONTEXT
  HC-V1-HEATING-CIRCUIT-1
  (INSTANCE LIQUID-STREAM)
  (IS-SISO-INLET-STREAM-FOR PUMP2)
  (IS-SISO-OUTLET-STREAM-FOR EXCH-1-HOT-SIDE)
  (TOTAL-KMOLAR-FLOW 0.2)
  (TEMPERATURE 303))
```

Type Any Character To Continue

Figure F.5: Example schemata maintained by the two Propagating-relations

```
IF
  The heat-transferred by a heat exchanger is 0.0
  & The coldin#total-kmolar-flow of the heat exchanger is > 0.0
  & The coldout#temperature of the heat exchanger is not already
    the same as the coldin#temperature of the heat exchanger
THEN
  Set the coldout#temperature of the heat exchanger to be
    the same as the coldin#temperature of the heat exchanger
```

The eventual outcome of all this is that the "temperature" slot of stream3 becomes set to the temperature of the source stream of propane, i.e. -20 Degrees Centigrade. Thus the rules and demons have propagated the initial disturbance from pump2, around the heating circuit and across the heat exchanger. The point has now been reached at which a constraint violation may be notified, unless the trip system is shown to act successfully. However, what actually happens depends upon the trip set point and the order in which constraint checks and rules are fired.

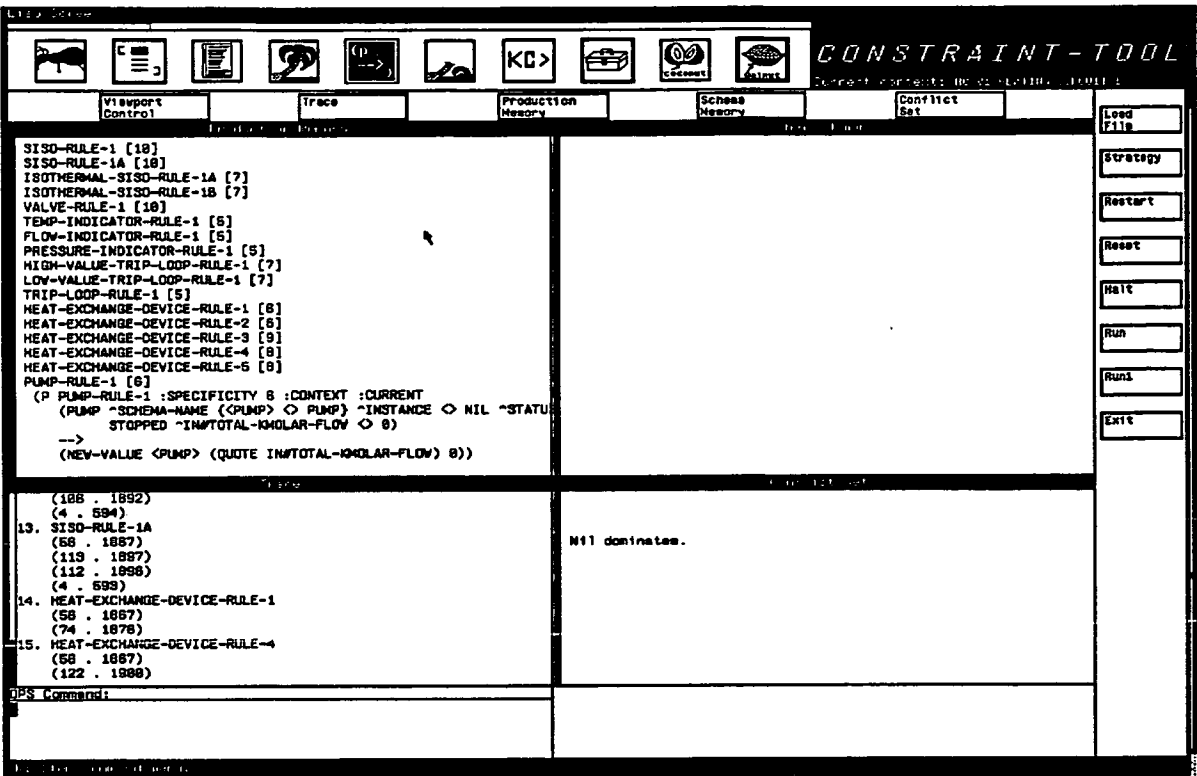


Figure F.6: An example instance of the CRL-OPS workbench

Version hc-v1 has the trip set so that the it should fire when the temperature of stream3 drops below 5 Degrees Centigrade. Consequently, the rule to represent this behaviour may now be activated (low-value-trip-loop-rule-1). However, if one has set-up automatic checking of stream3-constr1 then this may also be activated at this point. This means that whether or not one is informed of a constraint violation depends upon the order of these two firings. If all the CRL-OPS rules fire first then the total-kmolar-flow of stream3 will have been set to 0.0 so the constraint will be satisfied. However, if the constraint is checked before the rules are fired then it will appear to be violated.

One can partially solve this problem by queueing constraint checks until all the rules have fired. This works for version hc-v1, in which the trip is properly set at a temperature greater than the constraint temperature. It also works for version hc-v2, in which the trip is stupidly set at a temperature below the source temperature of the propane. In this case the trip loop rules will never fire, so a violation will be correctly identified when the constraint is finally checked. However, in version hc-v3 the trip is set at a temperature between the constraint temperature and the source temperature of the

propane. In this case the constraint should actually be violated before the trip loop fires. Unfortunately, this will not be demonstrated if all constraint checks are queued until the end, as the trip loop rule will have fired by then. Thus one does not always get the correct answer by simply queueing constraint checks until after all the rules have fired.

This problem arises because the rules are not modelling time and sequence properly but instead just try to set the end points of the simulation regions. For example, they don't model the gradual cooling of the heating circuit, with the consequent firing of the trip system when the temperature of stream3 *first* reaches the trip set point. Instead, they immediately set the temperature of stream3 to -20 Degrees Centigrade, the lowest value that it can reach. This rather haphazard approach is a direct consequence of the desire to avoid performing a detailed qualitative or numerical simulation. Obviously the information that can be obtained from such loosely coupled test-action rules is somewhat unreliable. Unfortunately, Chapter 3 has already shown that many problems also exist with the alternative of using an all-encompassing qualitative scheme.

Within the rest of his Knowledge Craft work the author has shied away from tackling this problem any further because he was more interested in constraints and hierarchical design representations. In addition, because of the previous difficulties with qualitative simulation, the approach taken was to limit the commitment to any single representational system. This has led to the provision of a loose collection of tools that should prove to be useful for different types of design task. The use of CRL-OPS rules for maintaining simple data dependencies could be regarded as one of these available features but really it seems that the pay-off does not justify the restrictions imposed by their use. This appendix has therefore been presented in order to discuss further the issues and problems with using rules, rather than as a suggested way of using the Constraint Tools System.

Appendix G

Example File of Rules To Maintain Simple Data Dependencies

```
;; This file can simply be loaded into Knowledge Craft using the normal
;; LISP "load" function. However, you also need to load in the schemata
;; definitions. The rules assume that all schemata have at least been created,
;; even if the required attributes have not been filled in. That is, most of
;; the rules change attribute values but none of them actually create memory
;; elements. The problem set-up must be performed by means external to this
;; rule set.
```

```
;; ----- ;;
```

```
(strategy MEA)
```

```
;; ----- ;;
```

```
;; NOTE that the literalized schemata must exist in the $root-context. This
;; is a serious drawback within a partitioned/hierarchical system, such as
;; that we are envisaging.
```

```
(schema-literalize stream
      is-iso-inlet-stream-for
      is-iso-outlet-stream-for
      temperature
      pressure
      has-components
      has-component-mole-fractions
      total-kmolar-flow)
```

```
(schema-literalize valve stem-position cuts-off-stream)
```

```

(schema-literalize siso
    has-iso-inlet-stream
    has-iso-outlet-stream
    in#total-kmolar-flow
    out#total-kmolar-flow)

(schema-literalize isothermal-iso
    in#temperature
    out#temperature)

;; in#total-kmolar-flow and out#total-kmolar-flow are
;; literalized from siso.

(schema-literalize measurement-instrument measurement-value)

(schema-literalize temperature-indicator measured#temperature)

(schema-literalize flow-indicator measured#total-kmolar-flow)

(schema-literalize pressure-indicator measured#pressure)

(schema-literalize trip-loop
    status
    trip-set-point
    measurement-value)

(schema-literalize high-value-trip-loop)

(schema-literalize low-value-trip-loop)

(schema-literalize heat-exchange-device
    coldin#total-kmolar-flow
    coldin#temperature
    coldout#temperature
    hotin#total-kmolar-flow
    hotin#temperature
    hotout#temperature
    heat-transferred)

(schema-literalize pump status) ;; other slots are literalized in siso

;; ++++++

```

```

;; -----
;; The rules for a siso |
;; -----

;; ===== siso-rule-1 =====

;;      IF      a new value has just been set for fin
;;      &      fout does not have the same value
;;      THEN     set fout to the same value

(p siso-rule-1 :context :current
  (stream ^schema-name <in> ^instance <> ()
    ^is-iso-inlet-stream-for <siso> ^total-kmolar-flow {<fin> <> ()})
  (siso ^schema-name <siso> ^has-iso-outlet-stream <out>)
  (stream ^schema-name <out> ^instance <> ()
    ^total-kmolar-flow {<> <fin>})
  -->
  (new-value <out> 'total-kmolar-flow <fin>)))

;; ===== siso-rule-2 =====

;;      IF      a new value has just been set for fout
;;      &      fin does not have the same value
;;      THEN     set fin to the same value

(p siso-rule-2 :context :current
  (stream ^schema-name <out> ^instance <> ()
    ^is-iso-outlet-stream-for <siso> ^total-kmolar-flow {<fout> <> ()})
  (siso ^schema-name <siso> ^has-iso-inlet-stream <in>)
  (stream ^schema-name <in> ^instance <> ()
    ^total-kmolar-flow {<> <fout>})
  -->
  (new-value <in> 'total-kmolar-flow <fout>)))

;; -----

;; These siso rules allow data dependencies to be maintained in either
;; direction, according to which data has just been changed. This need for
;; "reversibility" causes problems because CRL-OPS remakes the whole
;; memory element when any data in it is changed. Consequently, to
;; ensure that both rules can fire we have to place the active data in
;; a separate element. This explains the explicit reference to streams
;; in these two rules. This really goes against the philosophy of

```

```

;; propagating-relations as demonstrated by the rest of the rules.

;; Note that using these rules prevents us from having a "has-topology"
;; relation because CRL-OPS only inherits across is-a links...consequently,
;; pumps etc. have to be is-a linked to the siso schema.

;; ++++++

;; -----
;; The rules for an isothermal-siso |
;; -----

;; ===== isothermal-siso-rule-1 =====

;; IF      a new value has just been set for Tin
;;   &      fin > 0   OR   fout > 0
;;   &      Tout does not have the same value as Tin
;; THEN     set Tout = Tin

;; -----

(p isothermal-siso-rule-1a :context :current
  (isothermal-siso ^schema-name {<schema> <> isothermal-siso} ^instance <> ()
    ^in#temperature {<tin> <> ()}
    ^in#total-kmolar-flow > 0
    ^out#temperature <> <tin>)
  -->
  (new-value <schema> 'out#temperature <tin>))

;; -----

(p isothermal-siso-rule-1b :context :current
  (isothermal-siso ^schema-name {<schema> <> isothermal-siso} ^instance <> ()
    ^in#temperature {<tin> <> ()}
    ^out#total-kmolar-flow > 0
    ^out#temperature <> <tin>)
  -->
  (new-value <schema> 'out#temperature <tin>))

;; -----

;; We do not really need both of these rules if we can rely upon the siso
;; rules to keep fin and fout the same. However, some redundancy would probably
;; be necessary if the rule base became large.

```

```

;; ++++++

;; -----
;; The rules for a valve |
;; -----

;; ===== valve-rule-1 =====

;; IF      stem-position = 0
;;   &     flow in relevant stream is not set to zero
;; THEN    set flow in relevant stream to zero

(p valve-rule-1 :context :current
  (valve ^schema-name <> valve ^instance <> ()
    ^stem-position = 0
    ^cuts-off-stream <stream>)
  (stream ^schema-name {<stream> <> stream} ^instance <> ()
    ^total-kmolar-flow <> 0)
  -->
  (new-value <stream> 'total-kmolar-flow 0))

;; ++++++

;; -----
;; The rules for Measurement instruments |
;; -----

;; The first set of rules simply change the name of the slot in which the
;; measured values are stored. This is necessary because we want to use
;; the same trip loop rules for each type of measurement but proprels cannot
;; map to slots of a different type. Thus we use CRL-OPS rules to do the
;; transformation.

;; -----

(p temp-indicator-rule-1 :context :current
  (temperature-indicator ^schema-name <schema>
    ^instance <> ()
    ^measured#temperature {<temp> <> ()}
    ^measurement-value <> <temp>)
  -->
  (new-value <schema> 'measurement-value <temp>))

```

```

;; -----

(p flow-indicator-rule-1 :context :current
  (flow-indicator ^schema-name <schema>
    ^instance <> ()
    ^measured#total-kmolar-flow {<flow> <> ()}
    ^measurement-value <> <flow>))
-->
  (new-value <schema> 'measurement-value <flow>))

;; -----

(p pressure-indicator-rule-1 :context :current
  (pressure-indicator ^schema-name <schema>
    ^instance <> ()
    ^measured#pressure {<pressure> <> ()}
    ^measurement-value <> <pressure>))
-->
  (new-value <schema> 'measurement-value <pressure>))

;; ++++++

;; -----
;; The rules for a Trip Loop |
;; -----

;; First we define the default functions for the trip action and trip reset
;; action:

(defun fire-trip-loop-action (this-schema)
  (new-value #!~|has-isolation-valve|?
    'stem-position #!~|isolation-valve-position|?))

(defun fire-trip-reset-action (this-schema)
  (let* ((fire-position #!~|isolation-valve-position|?)
    (reset-position (if (= fire-position 0) 100 0)))
    (new-value #!~|has-isolation-valve|?
      'stem-position reset-position)))

;; -----

;; And then we define the rules.....

;; ===== high-value-trip-loop-rule-1 =====

```

```
;; IF      meas-var >= set-point
;;      &    loop status is "reset"
;; THEN    Perform the trip action
;;      &    Change loop status to "tripped-out"
```

```
(p high-value-trip-loop-rule-1 :context :current
  (high-value-trip-loop
    ^schema-name {<schema> <> high-value-trip-loop} ^instance <> ()
    ^status reset
    ^trip-set-point {<set> <> ()}
    ^measurement-value >= <set>)
  -->
  (let ((act (get-value <schema> 'trip-action)))
    (funcall act <schema>)
    (new-value <schema> 'STATUS 'tripped-out)) )
```

```
;; This rule is disabled by loop status becoming "tripped-out" so we don't
;; also need to check that the trip action has not already been performed.
```

```
;; ===== low-value-trip-loop-rule-1 =====
```

```
;; IF      meas-var <= set-point
;;      &    loop status is "reset"
;; THEN    Perform the trip action
;;      &    Change loop status to "tripped-out"
```

```
(p low-value-trip-loop-rule-1 :context :current
  (low-value-trip-loop
    ^schema-name {<schema> <> low-value-trip-loop} ^instance <> ()
    ^status reset
    ^trip-set-point {<set> <> ()}
    ^measurement-value <= <set>)
  -->
  (let ((act (get-value <schema> 'trip-action)))
    (funcall act <schema>)
    (new-value <schema> 'status 'tripped-out)) )
```

```
;; This rule is disabled by loop status becoming "tripped-out" so we don't also
;; need to check that the trip action has not already been performed.
```

```
;; =====
```



```

;; -----
;; reset rule for either trip loop type |
;; -----

;; ===== trip-loop-rule-1 =====

;; IF      loop status has just been "reset"
;; &       the trip reset action has not already been performed
;; THEN    Perform the trip reset action

(p trip-loop-rule-1 :context :current
  (trip-loop
    ^schema-name {<schema> <> trip-loop} ^instance <> ()
    ^status reset)
  -->
  (let ((resact (get-value <schema> 'trip-reset-action)))
    (funcall resact <schema>)))

;; This rule can compete with one of the two rules above but is less specific
;; and so they will fire preferentially. This is desirable because it
;; means that if we try to "reset" without first taking the measurement-value
;; within the "legal" value then the trip fires again. This rule will then
;; be disabled because the trip-loop-status is set to "tripped-out".

;; ++++++

;; -----
;; The rules for a heat-exchange-device |
;; -----

;; The first set of rules describe the conditions under which the
;; heat-exchanger is unable to transfer heat:

;; ===== heat-exchange-device-rule-1 =====

;; IF      fcin = 0
;; &       heat-transferred is not already set to 0
;; THEN    set heat-transferred to 0

(p heat-exchange-device-rule-1 :context :current
  (heat-exchange-device
    ^schema-name {<exch> <> heat-exchange-device} ^instance <> ()
    ^coldin#total-kmolar-flow = 0

```

```

        ^heat-transferred <> 0)
-->
    (new-value <exch> 'heat-transferred 0)    )

;; ===== heat-exchange-device-rule-2 =====

;;      IF      fhin = 0
;;      &      heat-transferred is not already set to 0
;;      THEN    set heat-transferred to 0

(p heat-exchange-device-rule-2 :context :current
  (heat-exchange-device
    ^schema-name {<exch> <> heat-exchange-device} ^instance <> ()
    ^hotin#total-kmolar-flow = 0
    ^heat-transferred <> 0)
-->
    (new-value <exch> 'heat-transferred 0)    )

;; We are relying upon siso rules to update cold-inlet-flow if
;; cold-outlet-flow becomes 0.

;; ===== heat-exchange-device-rule-3 =====

;;      IF      Thin = Tcin
;;      &      fhin > 0
;;      &      fcin > 0
;;      &      heat-transferred is not already set to 0
;;      THEN    set heat-transferred to 0

(p heat-exchange-device-rule-3 :context :current
  (heat-exchange-device
    ^schema-name {<exch> <> heat-exchange-device} ^instance <> ()
    ^hotin#temperature {<thin> <> ()}
    ^coldin#temperature = <thin>
    ^hotin#total-kmolar-flow > 0
    ^coldin#total-kmolar-flow > 0
    ^heat-transferred <> 0)
-->
    (new-value <exch> 'heat-transferred 0)    )

;; -----

```

```
;; The next set of rules describe the dependencies between input and output
;; temperatures when the heat transferred is zero:
```

```
;; ===== heat-exchange-device-rule-4 =====
```

```
;; IF      heat-transferred = 0
;;   &     fhin > 0
;;   &     Thout is not already set to Thin
;; THEN    set Thout = Thin
```

```
(p heat-exchange-device-rule-4 :context :current
  (heat-exchange-device
    ^schema-name {<exch> <> heat-exchange-device} ^instance <> ()
    ^heat-transferred = 0
    ^hotin#total-kmolar-flow > 0
    ^hotin#temperature {<thin> <> ()}
    ^hotout#temperature <> <thin>)
  -->
  (new-value <exch> 'hotout#temperature <thin>)))
```

```
;; ===== heat-exchange-device-rule-5 =====
```

```
;; IF      heat-transferred = 0
;;   &     fcin > 0
;;   &     Tcout is not already set to Tcin
;; THEN    set Tcout = Tcin
```

```
(p heat-exchange-device-rule-5 :context :current
  (heat-exchange-device
    ^schema-name {<exch> <> heat-exchange-device} ^instance <> ()
    ^heat-transferred = 0
    ^coldin#total-kmolar-flow > 0
    ^coldin#temperature {<tcin> <> ()}
    ^coldout#temperature <> <tcin>)
  -->
  (new-value <exch> 'coldout#temperature <tcin>)))
```

```
;; ++++++
```

```
;; -----
;; The rules for a pump |
;; -----
```

```

;; ===== pump-rule-1 =====

;;   IF      pump-status = "stopped"
;;   &       fin has not already been set to zero
;;   THEN    set fin = 0

(p pump-rule-1 :context :current
  (pump
    ^schema-name {<pump> <> pump} ^instance <> ()
    ^STATUS stopped
    ^in#total-kmolar-flow <> 0)
  -->
  (new-value <pump> 'in#total-kmolar-flow 0))

;; This rule assumes that the flows will be updated by siso rules as required
;; i.e. we only set the inlet flows in these rules.

;; -----

;; -----
;; Other possible pump rules|
;; -----

;; pump-rule-2
;;   IF      fin = 0
;;   &       pump-status = "running"
;;   &       pressure-head has not been set to the shut-off value
;;   THEN    set pressure-head = shut-off

;; pump-rule-3
;;   IF      delp = 0
;;   &       pump-status = "running"
;;   &       fin has not already been set to maxflow
;;   THEN    set fin = maxflow

;; pump-rule-4
;;   IF      delp = shut-off
;;   &       pump-status = "running"
;;   &       fin has not already been set to 0
;;   THEN    set fin = 0

;; pump-rule-5
;;   IF      fin = max-flow
;;   &       pump-status = "running"
;;   &       delp has not already been set to 0

```

```

;;      THEN      set delp = 0

;; ++++++

;; -----
;; Rules for a heating circuit|
;; -----

;; heating-circuit-rule-1
;;      IF      mass flow of heating source = 0
;;      &      heat transferred to sink is not already 0
;;      THEN      set heat transferred to sink = 0

;; ++++++

;; -----
;; Rules for a feedback-control-loop|
;; -----

;; feedback-control-loop-rule-1
;;      IF      meas-val > set-point
;;      &      gain = +1
;;      &      the manipulated variable has no effect om meas-val
;;      THEN      drive the manipulated variable to its upper limit

;; feedback-control-loop-rule-2
;;      IF      meas-val > set-point
;;      &      gain = -1
;;      &      the manipulated variable has no effect om meas-val
;;      THEN      drive the manipulated variable to its lower limit

;; feedback-control-loop-rule-3
;;      IF      meas-val < set-point
;;      &      gain = +1
;;      &      the manipulated variable has no effect om meas-val
;;      THEN      drive the manipulated variable to its lower limit

;; feedback-control-loop-rule-4
;;      IF      meas-val < set-point
;;      &      gain = -1
;;      &      the manipulated variable has no effect om meas-val
;;      THEN      drive the manipulated variable to its upper limit

;; ++++++

```

```

;; -----
;; Rules for a relief-valve|
;; -----

;; relief-valve-rule-1
;;   IF      stem-position = 0
;;   THEN     set fin = 0 & set fout = 0    ;; streams and normal valves?

;; relief-valve-rule-2
;;   IF      delp <= set
;;   THEN     set stem-position = 0    ;; rule-1 then fires..

;; relief-valve-rule-3
;;   IF      fin = 0 OR fout = 0
;;   THEN     set stem-position = 0    ;; rule-1 then fires!!??

;; relief-valve-rule-4
;;   IF      delp > set
;;   THEN     set stem-position =  positive, fin =  positive & fout =  positive

;; relief-valve-rule-5
;;   IF      stem-position > 0
;;   THEN     set fin =  positive & fout =  positive

;; relief-valve-rule-6
;;   IF      fin > 0
;;   THEN     set stem-position > 0 & fout > 0    ;; rule-5 then fires!!??

;; relief-valve-rule-7
;;   IF      fout > 0
;;   THEN     set stem-position > 0 & fin > 0    ;; rule-5 then fires!!??

;; ++++++

;; -----
;; Rules for a bursting-disc|
;; -----

;; bursting-disc-rule-1
;;   IF      delp < set
;;   &      status = unburst
;;   THEN     set fin = 0 & set fout = 0    ;; streams and normal valves?

;; bursting-disc-rule-2
;;   IF      delp >= set

```

```

;;      &    status = unburst
;;      THEN    set status = burst, fin =  positive & fout =  positive

;; How can we model a bursting-disc as a siso after it has burst?

;; ++++++

;; -----
;; The rules for an isobaric-siso |
;; -----

;; isobaric-siso-rule-1
;;      IF      a new value has just been set for Pin
;;      &      Pout does not have the same value as Pin
;;      THEN    set Pout = Pin

;; isobaric-siso-rule-2
;;      IF      a new value has just been set for Pout
;;      &      Pin does not have the same value as Pout
;;      THEN    set Pin = Pout

;; Perhaps also for an isocomposition-siso?

;; ++++++

```