Creating Signed Directed Graph Models for Process Plants

by

Claire Palmer

A Doctoral Thesis

Submitted in Partial Fufilment of the requirements

for the award of

Doctor of Philosophy of Loughborough University

January, 1999

Acknowledgements

I would like to thank my supervisor, Dr. P.W.H. Chung, for his guidance, positive criticism and support.

I would also like to thank my colleagues in the Department of Chemical Engineering, especially the members of the Plant Engineering Group, for their help and encouragement.

This work was supported by supported by an EPSRC/ British Gas Postgraduate Studentship.

Abstract

The identification of possible hazards in chemical plants is a very important part of the design process. This is because of the potential danger that large chemical installations pose to the public. One possible route for speeding up the identification of hazards in chemical plants is to use computers to identify hazards automatically. This will facilitate safe plant design and will avoid late design changes which can be very costly to implement.

Previous research at Loughborough has concentrated on developing a model-based approach and an analysis algorithm for automating hazard identification. The results generated have demonstrated the technical feasibility of the approach. This approach requires a knowledge-base of unit models. This library of models describes how different plant equipment behaves in qualitative terms.

The research described in this thesis develops a method for creating and testing the equipment models. The model library was previously achieved by an expert writing the models in a format that could be directly used by the system described above. An engineer unfamililar with the system would find this difficult. An alternative method would have been to use an intermediary (a knowledge engineer) to gather information from the engineer and convert it into the system format. This would be expensive. Both methods would take up a lot of the engineer's time. An engineer should be able to enter information personally in order to maintain efficiency and avoid information loss through the intermediary. A front end interface has been built to the system which enables an expert to enter information directly without needing to understand details of the application system. This interface incorporates ideas from the knowledge acquisition field in order to produce a tool that is simple to use.

Unit-based qualitative modelling can lead to incorrect or ambiguous inference. The method developed here identifies situations where ambiguities may arise. A new modular approach is presented to overcome this type of problem. This method also presents a technique to verify that the models created are both complete and correct.

Table of Contents

1. Introduction

The research described in this thesis develops a method for creating and testing equipment models for process plants. The models are necessary for the modelling of fault propagation. Many applications utilise fault propagation modelling to identify and assess hazards or to diagnose faults. A computer-aided modelling tool, Equipment Model Builder, has been constructed to demonstrate this method. This introduction begins by considering why a method is needed. Further sections provide an overview of the project and describe the structure of this thesis.

1.1. Motivations

Modelling fault propagation requires the representation of process variable deviations and the causes and effects of these deviations in qualitative terms. This section will begin by briefly describing the concept of qualitative models, why they are useful and explaining why they are difficult to construct. The need for a method to aid an engineer in constructing these models is considered.

Qualitative modelling does not require detailed quantitative information (which may be difficult or expensive to acquire or simply unnecessary) in order to describe a physical system. Numerical methods give precise answers but if a broader outlook is required then qualitative reasoning can provide the solution to a whole class of problems. As well as providing a description of the physical system, qualitative modelling can be used to explain how the system functions. As qualitative models do not refer to numeric values, precise values are abstracted into symbolic ones. For example, a change in a value for a process variable such as flow might be depicted qualitatively as *increasing*.

Qualitative model construction is recognised as a difficult task (Schut and Bredeweg, 1994). However, no guidelines exist as to how to build the models. Consequently there are no tools to aid in signed directed graph model construction. Many applications such as those described in section 2.3. require a good library of component models. To create a model library for one of these applications requires an engineer to write the models in a format that can be directly used by the application program. An engineer who is unfamiliar with the system would find this difficult. An alternative method would be to use an intermediary (a knowledge engineer) to gather information from the expert and convert it into the application system format. This would be

expensive and time consuming. Both methods would take up a lot of the engineer's time. The engineer should be able to enter information directly in order to maintain efficiency and avoid information loss through the intermediary.

Models constructed using these methods are normally for a particular purpose only and are difficult to reuse and verify. Much of the experience gained about how to construct the models (i.e. methodologies) will be lost once a project is completed. In all forms of modelling errors will occur due to omissions, mistakes and lack of knowledge. These errors will need to be identified and corrected. Model building also requires knowledge of what is important to model and what approximations and abstractions to make. Currently this is subjective and will result in non-uniform models, making them difficult to update, compare and detect errors in. If the assumptions for a model are not explicitly stated then there is a likelihood that the model may be inappropriately applied.

A method is needed which enables an engineer to enter information directly without needing to understand details of the model format or the application. The engineer should be informed which is the most important part of the model to build first or if any parts of the model are missing.

The equipment models considered by this thesis use the signed directed graph representation. This representation is described in section 2.1.2. Signed directed graphs are a form of qualitative model.

1.2. Project Overview

This thesis develops a method to construct signed directed graph models simply and correctly. A computer-aided modelling tool, Equipment Model Builder, has been built to demonstrate this method. Three main contributions have been made by this thesis. These are:

1. Creating a front end interface enabling an engineer to enter modelling information directly;
2. Developing a novel modular approach which provides a methodology to remove a specific type of ambiguity which arises when unit models are combined;
3. Developing verification techniques for the models created.

A front end interface has been developed. By following the method provided by the interface an engineer is able to create models directly without being required to know exact details of model structure. In order to fulfil the requirement that the method should be simple to use and aid the engineer in building correct models ideas from the knowledge

acquisition field have been incorporated into the interface. This results in a tool for constructing signed directed graph models which is simple to use.

Knowledge acquisition tools are a type of modelling tool although they are not described as such. This is because they have been developed in a different field. Knowledge acquisition tools provide aids to allow the expert to achieve a structured input of information without knowledge of the internal format of the expert systems that these types of tools are acquiring information for. Knowledge acquisition tools are surveyed in chapter 3.

In order for a model to be correct it must be free from ambiguities. This thesis considers ambiguities due to multiple causal paths which occur when unit models are combined to form a system. Problems can occur when one path has a contradictory effect compared to another. The result of the addition of the effects of these paths cannot be determined unambiguously. A novel modular approach is developed which provides a methodology to remove the ambiguities caused by multiple causal paths.

Verification is necessary to detect modelling errors which may give rise to wrong results when the models are utilised. A series of verification techniques for signed directed graph models is described.

The three parts of the method overlap. The user interface also forms part of the module and verification tools. Verification is a feature of many knowledge acquisition tools.

The models created by Equipment Model Builder are utilised by the QUEEN system (Chung, 1993) described in section 2.4. This is to show that the models created are sufficient to be of use. Equipment Model Builder should not be regarded as application specific. The tool takes as input a description of a plant and a library of plant models. A modified plant description and a file of unit models occurring within the plant are created. This information forms the output for QUEEN.

1.3. Layout of Thesis

Chapter 2 considers the modelling of fault propagation in process plants using signed directed graph and functional equation representations. These two representations are equivalent. How ambiguities may arise within the qualitative models created is described. Applications of the representations are discussed. The expert system QUEEN, which provides a common set of procedures for these applications, is described. Related work which considers how qualitative models are created is detailed.

Chapter 3 surveys knowledge acquisition tools in order to determine their desirable features in relation to qualitative modelling.

Chapter 4 gives an overview of Equipment Model Builder. How the tool creates output for QUEEN is detailed. The tool's user interface is described.

Chapter 5 explains the modular approach developed and how it is implemented.

Chapter 6 describes the verification techniques used by Equipment Model Builder.

Chapter 7 presents two case studies to assess the method implemented by the tool.

The final chapter describes the contributions made by this thesis. Limitations of the work are considered and possibilities for future work discussed.

2. Fault Propagation

This chapter will discuss the modelling of fault propagation in process plants. Fault propagation determines the effects of a causal fault upon the process variables of a given plant. Modelling fault propagation requires representing the propagation of input process variable deviations into output process variable deviations and the causes and effects of these deviations in qualitative terms. Computer-aided qualitative modelling allows well-defined models to be produced, facilitating model re-use and validation. It by-passes the process of manual modelling thus leading to some saving of time. This thesis will only examine computer-aided qualitative modelling.

Qualitative modelling comes from the field of qualitative physics and is concerned with developing a variety of models for physical phenomena and engineered systems (Falkenhainer and Forbus, 1990). A model is an abstract representation of a physical system, providing knowledge about a system, its parts and their relationships. A model should provide an adequate description of the system but only take into account those aspects that are relevant to the required perspective. The most appropriate model for a given application should be generated and used.

Quantitative numeric methods have traditionally been used for detailed design. However, for many engineering tasks numerical methods are not always required or suitable. Qualitative reasoning is able to provide statements about the possible behaviour of dynamic systems, emphasising a causal explanation of the behaviour derived from a structural description. Numerical methods give precise answers but if a broader outlook is required then qualitative reasoning can provide the solution to a whole class of problems. Qualitative reasoning does not require detailed quantitative information (which may be difficult or expensive to acquire or simply unnecessary) in order to describe a physical system. As well as providing a description of the physical system, qualitative reasoning can be used to explain how the system functions. As qualitative models do not refer to numeric values, precise values are abstracted into symbolic ones. For example, a change in a value for a process variable such as flow might be depicted qualitatively as *increasing*. Within the process engineering field qualitative reasoning has been applied to alarm analysis (Andow and Lees, 1975), fault diagnosis (Wilcox and Himmelblau, 1994), fault tree synthesis (Lapp and Powers, 1977) and hazard identification (Catino and Lyle, 1995; Larkin et al., 1997).

This chapter will discuss representations used for fault propagation. How ambiguities may arise within the qualitative models created is described. Applications of the representations are discussed. A general purpose tool which provides a common set of procedures for these applications is described. The penultimate section discusses related work which considers how the qualitative models are created. The chapter concludes by stating the type of ambiguities which this thesis will consider. A new methodology is proposed which overcomes this type of ambiguity.

2.1. Representations

Fault propagation may be represented in a number of ways. Some forms of representation are functional equations, program rules, reliability block diagrams, influence graphs, signed directed graphs, logical expressions, truth tables, fault trees, event trees and bond graphs. It is possible to map between some of these representations (Aldersey et al., 1991). The representations considered here are functional equations and signed directed graphs because they are most widely used in the modelling of fault propagation within process plants. It will be shown later that these two representations are equivalent. Process plant are built by connecting a set of smaller units together to perform the required functions. How the unit descriptions are combined so the fault propagation behaviour of the whole plants can be analysed will be described.

2.1.1. Functional Equations

Functional equations have been developed by Lees and co-workers (Andow and Lees, 1975; Parmar and Lees, 1987; Hunt 1992) and used to create plant unit models. The types of functional equation considered here are:

(i) propagation equations;

(ii) event statements.

Propagation equations describe the relationship between the output variable of a unit and the input and other output variables of the unit. An example of a propagation equation is:

$$L = f(Q\,in, -Q\,out)$$

This signifies that the level L increases if the inlet flow 'Q in' increases or the outlet flow 'Q out' decreases, and vice versa. In fault propagation modelling it is necessary to consider two way propagation of flow. For example, a leak will cause an increase in flow

upstream and a decrease in flow downstream. Two way propagation of flow is modelled using, by convention, the flow variable Q and the pressure gradient variable G. The resulting propagation equations are:

1. G in = f(Q in, Q out)
2. Q out = f(G in, G out)

Equation 1 indicates that the inlet pressure gradient 'G in' increases if the inlet flow 'Q in' increases or if the outlet flow 'Q out' increases, and vice versa. Equation 2 indicates that the outlet flow 'Q out' increases if the inlet pressure gradient 'G in' increases or if the outlet pressure gradient 'G out' increases, and the reverse. Both equations are used in the unit model to propagate flow in both directions.

Propagation equations describe how a fault propagates. Event statements describe how a deviation is initiated or terminated. Initial event statements model the way basic faults in units affect the variables propagating out of the unit. They take the form:

Initial fault: variable deviation

For example, the following states that the outlet flow 'Q out' will be low if there is a partial blockage:

partial blockage: Q out low

A terminal event represents the termination of a variable deviation. Terminal events are usually undesired events or hazards. A terminal event statement takes the form:

Variable deviation: terminal event

For example, the following states that if the pressure 'P out' is high there is overpressure of the unit:

P out high: overpressure

2.1.2. Signed Directed Graphs

Wide use has been made of signed directed graphs in modelling fault propagation (for example, Lapp and Powers, 1977; Iri et al., 1981; Shiozaki et al., 1985; Kramer and Palowitch, 1987; Chung, 1993; Larkin et al., 1997). A signed directed graph (SDG) consists of an influence graph with labelled arcs. An influence graph contains the variables in a physical system which are depicted as nodes. These are connected by arcs to

reflect the influence the variables have on one another. An arc from a node X, to another node Y, indicates that a change in the variable X will cause a change in the variable Y:

$$X \xrightarrow{+/-} Y$$

An influence is defined as a causal relation between two variables or simply how one variable affects the other. Several variations of signed directed graph exist in which the arcs may be labelled differently. A convention commonly used is to label each arc of the graph with a sign '+' or '-'. The sign '+' indicates a positive influence, i.e. Y will increase if X is increased and Y will decrease if X is decreased. The sign '-' indicates a negative influence, i.e. Y will decrease if X is increased and Y will increase if X is decreased. '+' and '-' can be shown as '+1' and '-1' as in the work of Kohda and Henley (1987) who also use '0' to indicate what they call nullification or no influence between the variables.

Some workers also use '+10' to denote a very large positive influence and '-10' to denote a very large negative influence (Lapp and Powers, 1977; Andrews and Brennan, 1990; Chang and Hwang, 1992). These large deviations are defined as being beyond the capacity of the system to rectify. Larkin et al. (1997) found the use of the two signs '+' and '-' not rich enough to represent all of the possible relationships between process variables. They have developed a representation allowing the relationship between variables to be more explicitly specified using a code of extended signs. An arc with code 'N' is represented in the SDG as:

$$X \xrightarrow{N} Y$$

Table 2.1 gives the interpretations for 'N'.

Extended sign value (N)	Interpretation
+	high X → high Y low X → low Y
-	high X → low Y low X → high Y
++	high X → high Y
- -	high X → low Y
+++	low X → low Y
- - -	low X → high Y

Table 2.1 Interpretation of Coded Arcs

To model fault propagation the basic SDG representation is extended by the addition of causes (of deviations) and adverse consequences. The cause and consequence nodes are linked into the deviation network. Cause nodes represent the faults (failure

modes) of units. Consequence nodes represent potentially hazardous events arising from causes or deviations. Using this extended representation the node 'X' of the SDG shown could be a process variable deviation or a fault. The node 'Y' could be a process variable deviation or a consequence.

SDG's are an equivalent representation to functional equations. The nodes in an SDG which represent process variable deviations are equivalent to propagation equations. For example, assume that 'Q in' represents flow at an inlet port of a unit and 'Q out' flow at an outlet port. The propagation equation 'Q out pump = f(Q in pump)' could interchange with the partial signed directed graph:

$$\text{pump,out,flow} \xrightarrow{+} \text{pump,in,flow}$$

Those nodes in the SDG which represent faults can be modelled by initial event statements. For example, 'pump,partial blockage: Q out pump low' is equivalent to:

$$\text{pump,partial blockage} \xrightarrow{-} \text{pump,flow,out}$$

Nodes representing consequences can be modelled by terminal event statements. For example, assume that 'P out' represents pressure at the outlet port of a unit. The statement 'P out pump high: pump,overpressure' may be modelled by:

$$\text{pump,pressure,out} \xrightarrow{+} \text{pump,overpressure}$$

2.1.3. Plant Models from Unit Models

To construct a model for a whole plant can be very time consuming. However process plants are built by connecting together a set of smaller units to carry out the required functions. The behaviour of each of these types of units can be modelled generically so that it will apply to any plant in which the unit is used. The assumption is made that the causal relationships within an item of equipment are independent of the context in which the equipment is used in the plant. By combining the unit models the behaviour of the whole plant can be analysed. This unit-based approach is widely used (Chung, 1993; Vaidhyanathan and Venkatasubramanian, 1995; Catino et al., 1991). The SDG and functional equation representations can be used to create unit models.

For the SDG representation each unit model consists of a mini-SDG. The mini-SDG shows how a change in one process variable affects another variable in the same unit. Deviations occurring in the unit can be propagated to other units via inport and outport connections. An SDG for a complete plant is created by joining together the appropriate mini-SDGs based on the plant topology. Consider the simple fluid flow system displayed in figure 2.1. A partial SDG model of the system may be created from its constituent

mini-SDG's as shown in figure 2.2. The mini-SDG's for the pipe, pump and heater are separated by dashed lines. Only arcs for the process variable 'flow' and one initiating fault arc are given.

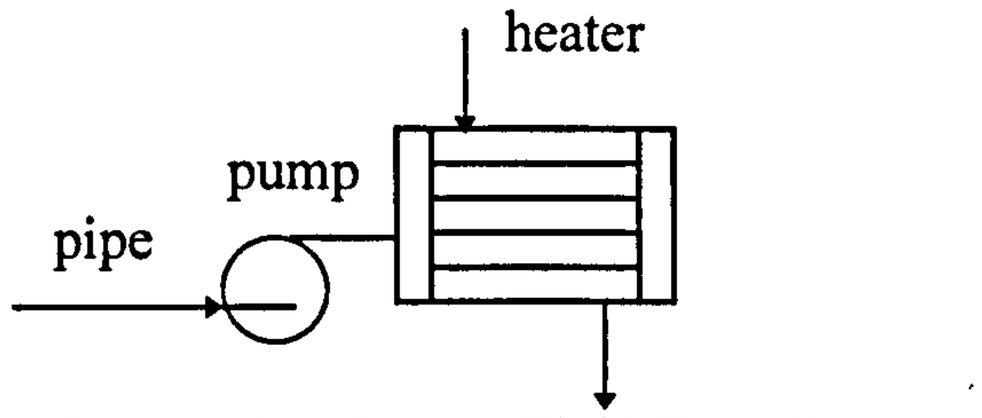


Figure 2.1 A Simple Fluid Flow System

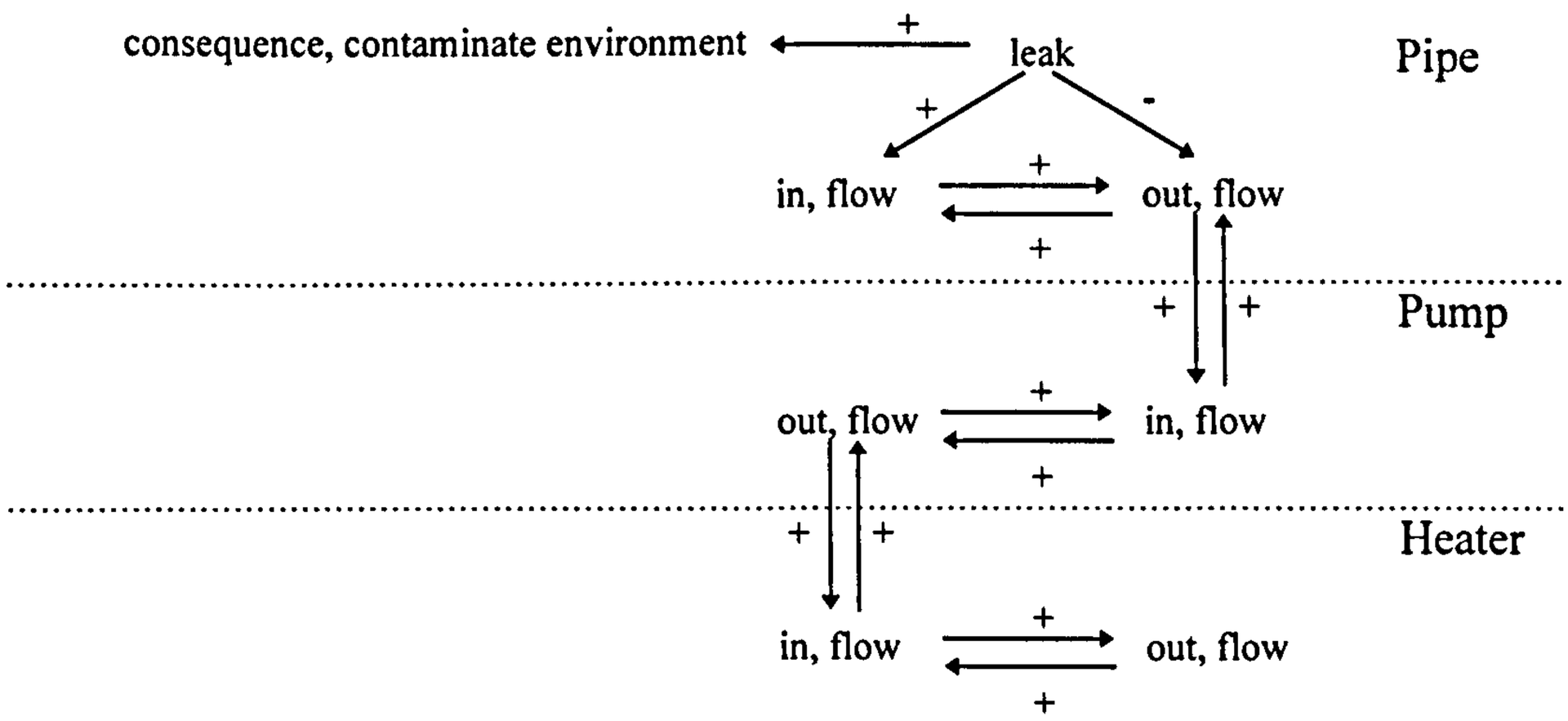


figure 2.2 A Partial Signed Directed Graph of a Simple Fluid Flow System

The mini-SDG's are joined together to create the model of the simple fluid flow system by linking:

'pipe,out,flow' to 'pump,in,flow', 'pump,out,flow' to 'heater,in,flow';

'heater,in,flow' to 'pump,out,flow', 'pump,in,flow' to 'pipe,out,flow'.

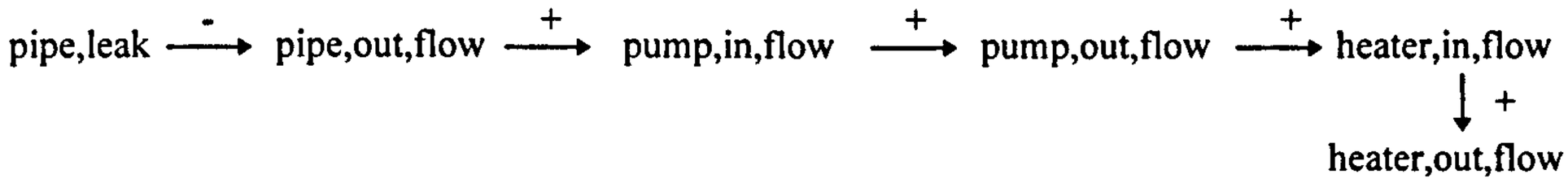
By similar reasoning functional equation unit models may be linked together with propagation equations to create a process plant model. The linking propagation equations model inport and outport connections between the constituent units of the plant. The equivalent linkages for the simple fluid flow system would be:

Q out pump = f(Q out pipe), Q in heater = f(Q out pump),

Q out pump = f(Q in heater), Q out pipe = f(Q in pump).

How any one variable affects another can be found by identifying an acyclic path within the plant model between the two variables. An acyclic path is a path where all the nodes within the SDG (or statements in the case of functional equations) appear only once.

For example, how the leak in the pipe in the simple fluid flow system shown in figure 2.1 affects the out flow of the heater is given by the following acyclic path:



The product of all the signs in the path is '-'. This means that the fault 'leak' will cause a decrease in 'out flow' of the heater. If no path exists between any two variables then the variables are independent.

To generate a plant model in this way requires a library of unit models of all the different types of units that are used in the plant. Individual plant units may be recognised by creating instances of the unit models and assigning a unique identifier to each instance.

2.2. Model Ambiguities

Although the use of qualitative models has many advantages, reasoning based on qualitative models can lead to ambiguous results. For example, arithmetic operations such as addition and subtraction cannot be represented unambiguously. The overall effect of combining a variable that is increasing with one that is decreasing can be an increase, decrease or no effect. This sub-section describes current approaches to resolve the ambiguities which can arise in qualitative models. The approaches fall into three categories:

1. those due to the initial and final effects within feedback;
2. those due to unrealistic scenarios;
3. those due to multiple causal paths.

A path is a directed series of nodes and arcs in the SDG. All the approaches in category 1 construct an SDG for the whole process plant. The unit-based approach is not used.

2.2.1. Ambiguities due to Initial and Final Effects

Feedback occurs when the effects of change in a system return in time to influence the source of the change. Feedback causes ambiguities by introducing opposing effects on variables which cannot be resolved qualitatively. The initial response of a variable to a fault (failure mode) is caused by the direct effects of the fault. These direct effects are transmitted on acyclic paths within the SDG between the fault and the variable under consideration. The final response will take account of all paths through the SDG between

the fault and the variable. Feedback may cancel the initial response, outweigh it or be insufficient to outweigh or cancel the initial response.

An example of feedback is given by Oyeleye and Kramer (1988). The effect of a slow partial blockage in the outlet pipe of an open tank (shown in figure 2.3) is described. A partial SDG model for the tank is given in figure 2.4. Only arcs for the process variables 'flow' and 'level', and the fault 'partly blocked' are shown.

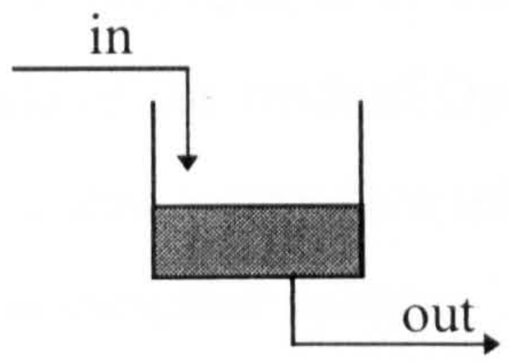


Figure 2.3 An Open Tank

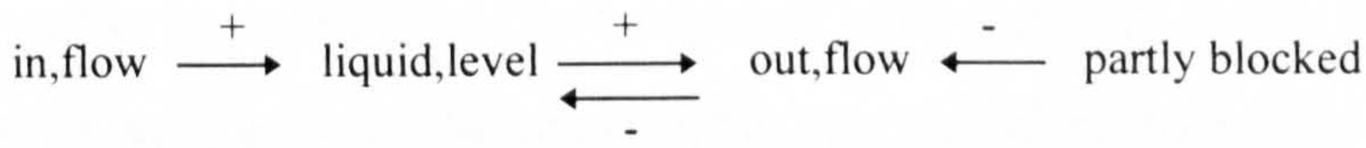
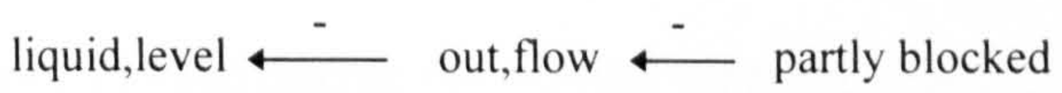


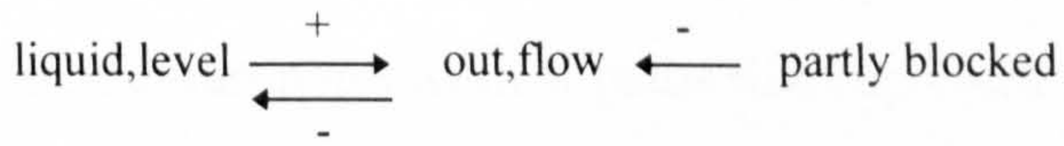
Figure 2.4 A Partial Signed Directed Graph for an Open Tank

The reasoning of Oyeleye and Kramer is shown below, although in practice the outcome depicted is unlikely.

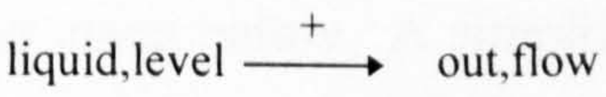
The initial response is described by the arcs:



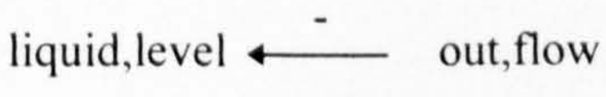
The slow partial blockage in the outlet pipe of the tank causes 'out flow' to decrease. This leads to 'level' increasing. The final response corresponds to the path:



The increased 'level' causes 'out flow' to increase and return to its original value. In reality for 'out flow' to return to its original value would require 'level' to increase to a great extent. The tank is unlikely to be big enough to allow 'level' to increase to this extent. The tank would overflow before 'level' reached this value. Whether 'out flow' can return to equilibrium will depend upon the relative sizes of the tank and the partial blockage. These sizes will determine if the arc



is of the same order of magnitude as the arc



and hence if the two competing influences will cancel out, allowing 'out flow' to return to its original value. In practise the negative influence will be almost always be larger than the positive and equilibrium will not be reached. Oyeleye and Kramer assume the partial blockage is sufficiently small to allow equilibrium to be reached. However it is debatable if this will ever occur.

Approaches to deal with ambiguities due to initial and final effects are those of Oyeleye and Kramer (1988), Rose and Kramer (1991), Chang and Yu (1990) and Fanti et al. (1993). Rose and Kramer build on the work of Oyeleye and Kramer. Fanti et al. refine the work of Rose and Kramer. These approaches rely on analysis algorithms or filtering rules to remove the ambiguities or incorrect values.

Each approach will be described in turn. How it may applied to the example given in figure 2.3 to determine the final response is detailed. In this example qualitative reasoning is unable to predict the final response with certainty as the relative magnitudes of the influences acting upon the arcs within the causal paths of the system cannot be depicted. These relative magnitudes are needed to determine whether or not the system will reach equilibrium. The correct final response is assumed to be that described by Oyeleye and Kramer.

2.2.1.1. The Approach of Oyeleye and Kramer

Oyeleye and Kramer (1988) derive the signed directed graph for a system from a dynamic process model. The initial response to a fault is traced via the direct effects within the SDG. To model final system responses an extension to the SDG, the ESDG (extended signed directed graph) is developed. The EDSG includes the final system response and minimises ambiguities. Extended signed directed graphs are similar to signed directed graphs but include in addition non-physical paths that represent inverse and compensatory responses due to feedback. An inverse response occurs when the final sign of a process variable in its steady state is opposite to its initial deviation. Compensatory response is when the variable returns to its nominal steady state value after an initial deviation. Inverse variables and compensatory variables are variables that respectively exhibit inverse and compensatory response to a fault due to feedback.

To create the ESDG inverse variables and compensatory variables are located from the system topology using the rules given below. A simplified rule-set is given. To view the full set see Oyeleye and Kramer (1988). The necessary conditions for a variable to display inverse response are:

1. The variable is located in a feedback loop.
2. The path due to feedback between the fault and the variable should contain a positive cycle or self cycle. A positive cycle is a path with the same initial and terminal nodes. The product of the signs on the arcs of this path sum to '+'. Variable self cycles are derived from the process equations.

The conditions for a variable to show compensatory response are:

1. The variable is located in a feedback loop.
2. The path due to feedback between the fault and the variable should:

   (a) contain an integrating variable;

   (b) not have a cycle containing all the variables in the path due to feedback.

An integrating variable is one with no (zero) self cycle.

ESDG arcs are created to explain the behaviour not accounted for in the SDG. After identification of the inverse variables and compensatory variables, ESDG arcs are constructed that 'jump' over each string of adjacent inverse variables or compensatory variables. The sign of the arc is the product of the signs of the arcs on the path between the origin and termination of the ESDG arc.

Using these rules the following ESDG may be created to model the tank shown in figure 2.3.

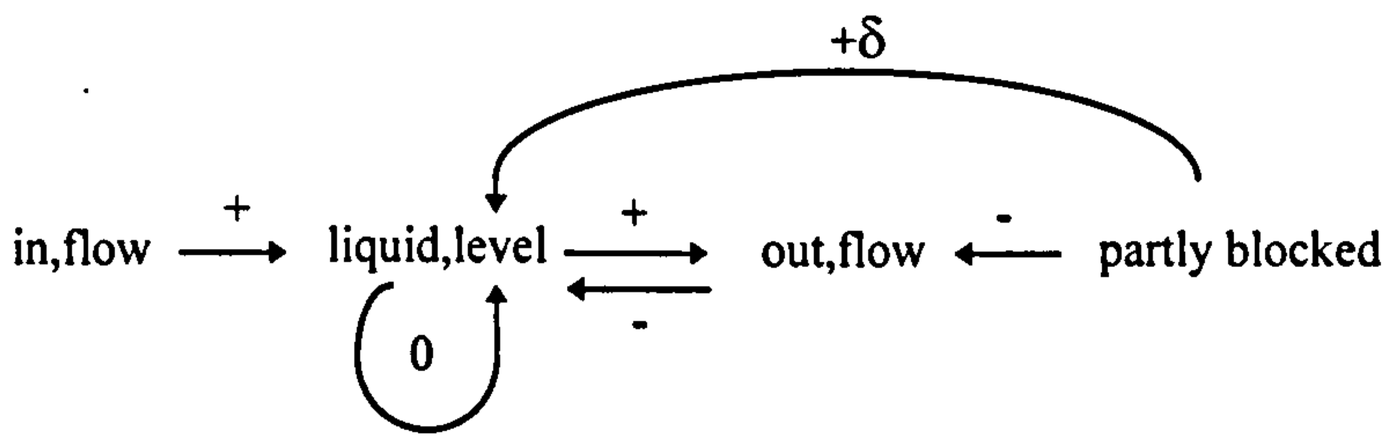
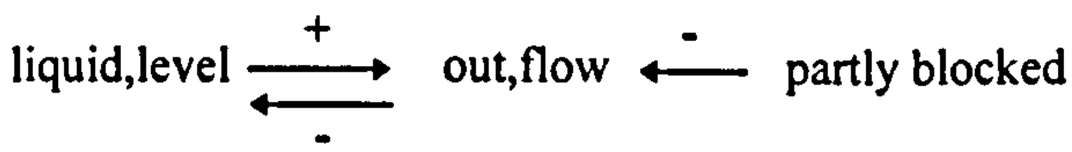


Figure 2.5 ESDG for an Open Tank

There is a feedback path between the fault 'partly blocked' and the variable 'out,flow' consisting of 'liquid,level'. 'liquid,level' is an integrating variable as it has a zero self cycle. This means that 'out,flow' is a compensatory variable with respect to the fault 'partly blocked' and will return to its equilibrium value. The δ function shown in figure 2.5 reflects the net sign of the paths between the fault 'partly blocked' and 'liquid, level' in the SDG. It represents the effect of adding the two acyclic paths:



The ESDG created enables the final response of the system to be predicted.

2.2.1.2. The Approach of Rose and Kramer

To improve the analysis speed and reduce the memory requirements of the Oyeleye and Kramer method led Rose and Kramer (1991) to develop a method implemented as a qualitative simulator called QUAF. QUAF inspects the system topology to predict the initial and final responses of the process variables to a deviation initiated by a fault. The following is a simplified outline of QUAF:

Consider the effect of a deviation initiated by a fault on a direct path between two nodes.

1. The initial response for each variable on the path is the sign of the deviation multiplied by the net sign of the shortest non-cyclic path from the fault to the variable.
2. If the variable examined in step 1 is not found in a feedback loop the final response to the fault is the qualitative sum of the direct effects. If the variable is in a feedback loop then:

    (a) If there is no integrating variable within the path due to feedback between the fault and the variable examined in step 1 and

        (i) the path does *not* contain a positive cycle or self cycle then the final response does not cancel the initial response.

        or

        (ii) the path contains a positive cycle or self cycle then the final response is ambiguous.

    or

    (b) If there is an integrating variable within the path due to feedback between the fault and the variable examined in step 1 the feedback cancels the initial response and the system returns to its original state.

Applying QUAF to the tank shown in figure 2.3 there is one acyclic path from the fault 'partly blocked' to the variable 'out,flow'.

out,flow ⟵ partly blocked

The initial response given by the effect of this path is a decrease in 'out,flow'. There is a feedback path which consists of 'liquid,level'. As previously shown 'liquid,level' is an integrating variable as it has a zero self cycle. Utilising section 2(b) above QUAF predicts that the initial response is cancelled and 'out,flow' returns to its original value.

2.2.1.3. The Approach of Chang and Yu

Chang and Yu (1990) criticise the Oyeleye and Kramer approach for its lack of modularity. When a fault initiates a deviation in a system it will move towards a new steady state by going through several states. Chang and Yu claim that by using states to depict the transition of a system response the diagnostic system becomes modular. They identify the states of a system depending on whether the compensatory variables have returned to their original state. States are obtained from the process model. When a variable returns to its original state its corresponding node is deleted and a new SDG is constructed. A number of SDG's representing the states of the system are constructed to describe the response to a fault.

Ambiguities are further reduced by using steady-state analysis to check the consistency of the fault propagation. This requires an expert's judgement or a quantitative process model. Chang and Yu proceed to consider ambiguities pertaining to control loops. In control loops the true fault origin may be eliminated within a diagnostic system during transient. To overcome this problem the controlled variable is expressed in its velocity form.

Applying Chang and Yu's method to the tank shown in figure 2.3 one compensatory variable, 'liquid,level' is identified. As only one compensatory variable is found this indicates that for this simple system there are no intermediate states between the initial response and the final steady state response. To model the final response the node 'liquid,level' is deleted from the system, leaving 'in,flow' = 'out,flow'. The fault 'partly blocked' does not affect 'in,flow'. The response of 'in,flow' to the fault is none. It follows that the final response of 'out,flow' to the fault 'partly blocked' is none.

2.2.1.4. The Approach of Fanti et al.

All of the previous methods reduce the number of ambiguities but do not eliminate them. Fanti et al. (1993) follow on from Rose and Kramer's work by applying constraints to the method developed by Rose and Kramer to resolve the ambiguities. A mass conservation constraint is applied to the process as a whole to remove false paths. A program needs to be written to identify situations where the constraint can be applicable to resolve ambiguities.

Considering the example given in figure 2.3 the overall mass balance at the steady state requires that 'in,flow' = 'out,flow'. The fault 'partly blocked' does not affect 'in,flow', therefore the final response of 'out,flow' to this fault is none.

2.2.2. Ambiguities due to Unrealistic Scenarios

On the basis of qualitative modelling alone in some cases it is impossible to decide whether or not a particular hazard scenario is realistic. An example of this is given by McCoy and Rushton (1997). A cooler (shown in figure 2.6) is described. A partial SDG model for the cooler is given in figure 2.7.

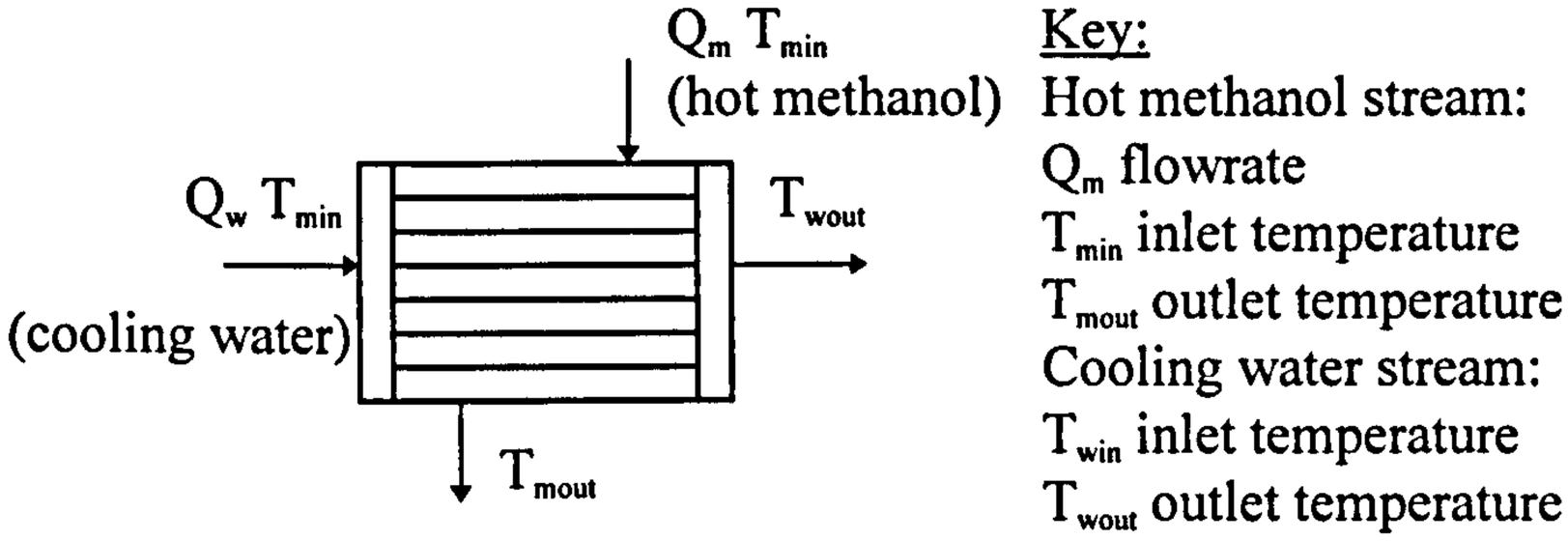


Figure 2.6 A Methanol Cooler

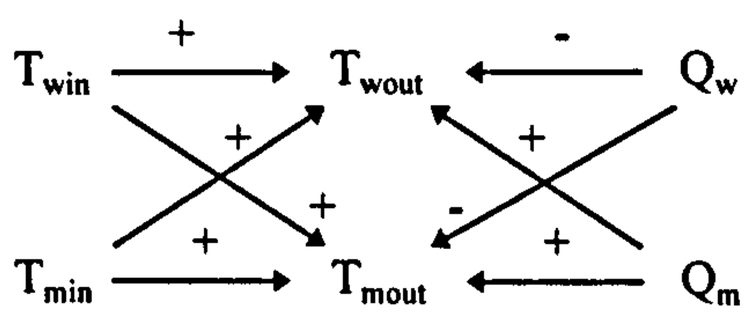


Figure 2.7 A Partial SDG of a Cooler

The SDG model predicts that high flow or low temperature of the coolant will cause a lower process fluid temperature. In the complete model this is linked to the possibility of the process fluid freezing. However, consider an application where the process fluid is methanol which has a freezing point of -94^{O}C, and the coolant is water which freezes at 0^{O}C. This means that the cooling water would never be able to cause the methanol to freeze as it would have ceased to flow before reaching the freezing point of methanol.

McCoy and Rushton (1997), Vaidhyanathan and Ventkatasubramanian (1996) and Srinivasan et al. (1997; 1998) suggest approaches to deal with this type of ambiguity by using additional quantitative knowledge to filter out the ambiguities. These methods are unit-based. McCoy and Rushton encode the additional quantitative knowledge in a set of conditions attached to the relevant arcs in the SDG. These conditions are tested for whenever the qualitative reasoning system considers the validity of a path through the graph. A similar method is employed by Vaidhyanathan and Ventkatasubramanian. Srinivasan et al. perform an analysis using quantitative models on those parts of the plant which cannot be qualitatively modelled without ambiguities. The quantitative models use a state-transition representation of the system. Each state has a set of differential and

algebraic equations associated with it describing the system in that state. When certain logical conditions are satisfied transitions between states are triggered. For safety verification the process is considered unsafe when the process variables take values in undesirable ranges. How those areas of the plant requiring quantitative analysis are identified is not specified.

These types of ambiguities are not considered further by this thesis as they are the subject of another research project at Loughborough University.

2.2.3. Ambiguities due to Multiple Causal Paths

For many plant structures the combination of generic process units provides an efficient method of creating a plant model. However for some plant structures combining unit models together may not result in a correct plant model. Combining units can lead to ambiguities or incorrect model behaviour. The type of ambiguities being discussed here are due to multiple paths between pairs of nodes and hence multiple paths of influence between the nodes. Problems can occur when one path has a contradictory effect compared to another. The result of the addition of the effects of these paths cannot be determined unambiguously. The simple plant loop shown in figure 2.8 is used to illustrate the problem with multiple paths. A partial SDG built up from unit SDGs is shown for the plant loop in figure 2.9. Only arcs for the process variables 'flow' and 'level', and only two initiating faults are shown.

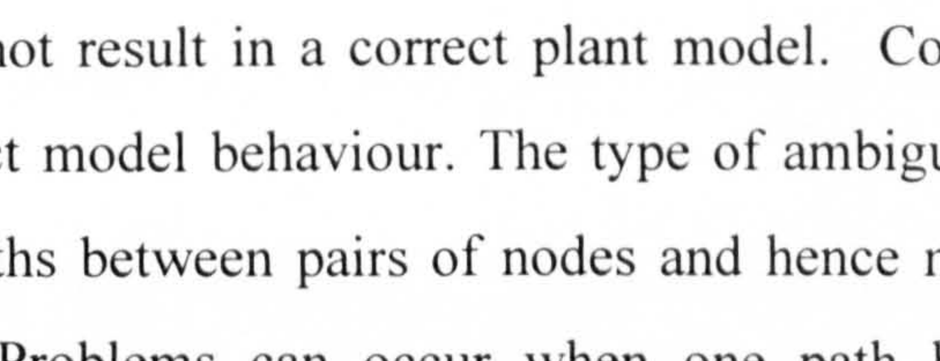


Figure 2.8 A Simple Plant Loop

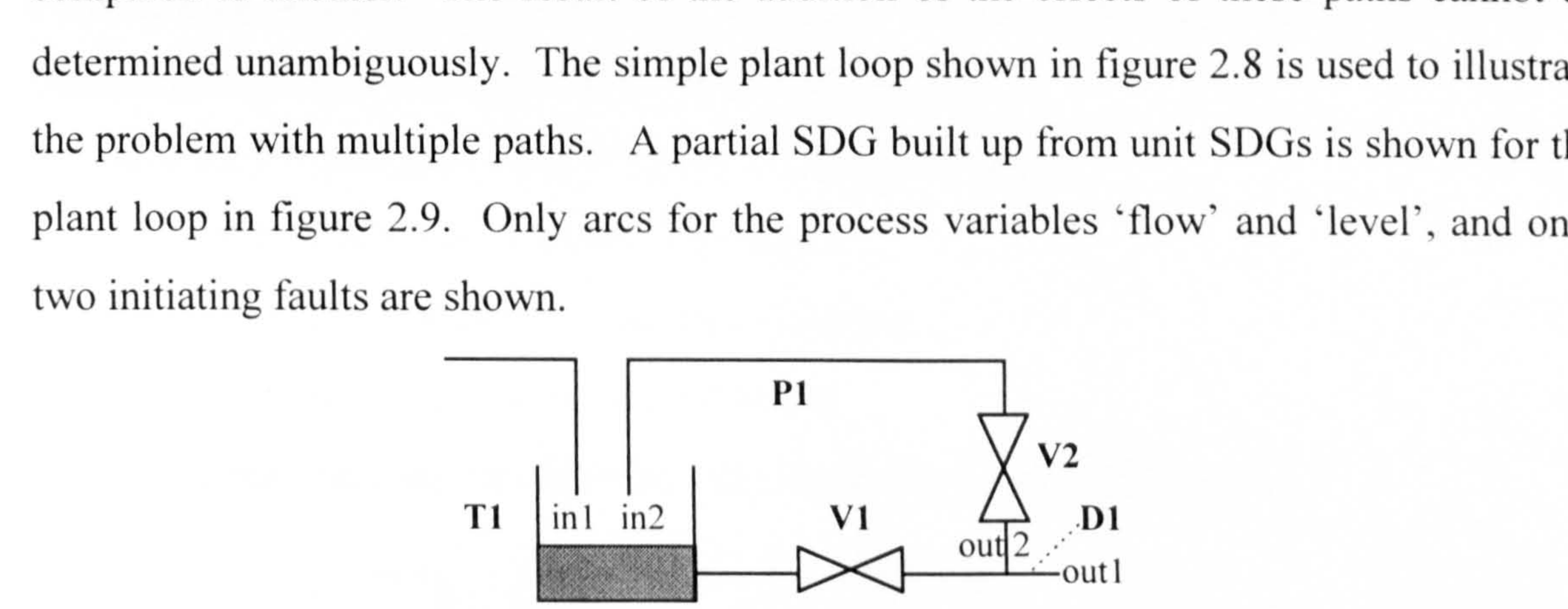
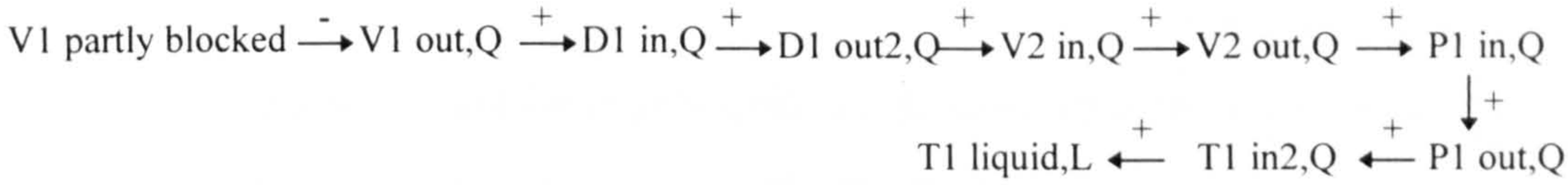
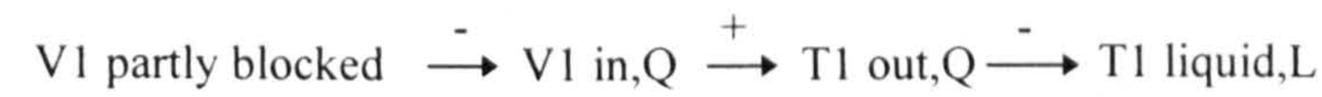


Figure 2.9 Partial Signed Directed Graph for a Simple Plant Loop

The loop consists of units V1, D1, V2, P1 and T1. Consider a partial blockage occurring in V1 and its effect on the level of T1. Two possible paths may be traced through the plant SDG. The first path is:



The effect of a partial blockage in V1 according to this path is a decrease in level of T1. The second path is:



For this path the effect is an increase in level of T1.

The qualitative analysis results in two contradictory paths with the second path having the correct influence.

In order to deal with ambiguities caused by multiple paths a heuristic that is commonly used is that when there is more than one acyclic path through the SDG the shortest path is used. This heuristic has been used as it is applicable to many cases including the example above. However cases may arise:

1. where the shortest path does not have a correct influence;
2. where all the paths have a correct influence;
3. where none of the paths have a correct influence.

Examples of these cases will be detailed below.

For the first case, consider the plant shown in figure 2.10 below.

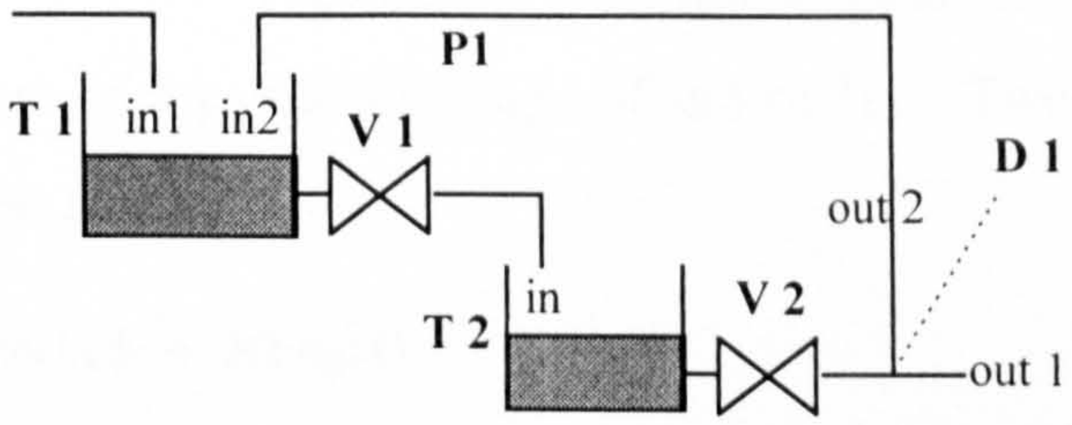
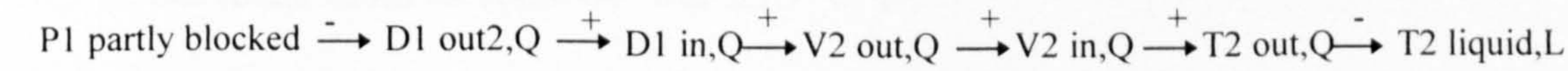


Figure 2.10 A Two Tank System with a Recycle

A SDG similar to that given in figure 2.8 may be drawn by combining the unit models (not shown here). Consider the effect of a partial blockage of P1 on the level of T2. Two possible paths may be traced through the SDG. The shortest path is:



This would result in an increase in level of T2. The alternative path is:

P1 partly blocked $\xrightarrow{-}$ P1 out,Q $\xrightarrow{+}$ T1 in2,Q $\xrightarrow{+}$ T1 liquid,L $\xrightarrow{+}$ T1 out,Q $\xrightarrow{+}$ V1 in,Q $\xrightarrow{+}$ V1 out,Q $\xrightarrow{+}$ T2 in,Q $\xrightarrow{+}$ T2 liquid,L

The effect of this path would correctly lead to a decrease in level of T2. In this case the shortest path heuristic leads to an incorrect inference.

A case where all the paths have the correct influence (case 2) will now be looked at. Returning to the simple plant shown in figure 2.8 for case 2, consider a leak occurring in P1. The shortest path is:

P1 leak $\xrightarrow{-}$ P1 out,Q $\xrightarrow{+}$ T1 in2,Q $\xrightarrow{+}$ T1 liquid,L

The alternative path gives:

P1 leak $\xrightarrow{+}$ P1 in,Q $\xrightarrow{+}$ V2 out,Q $\xrightarrow{+}$ V2 in,Q $\xrightarrow{+}$ D1 out2,Q $\xrightarrow{+}$ D1 in,Q $\xrightarrow{+}$ V1 out,Q $\xrightarrow{+}$ V1 in,Q $\xrightarrow{+}$ T1 out,Q $\xrightarrow{-}$ T1 liquid,L

Both paths are correct and result in a decrease in the level of T1.

An example of case 3 where none of the paths have the correct influence will be discussed.

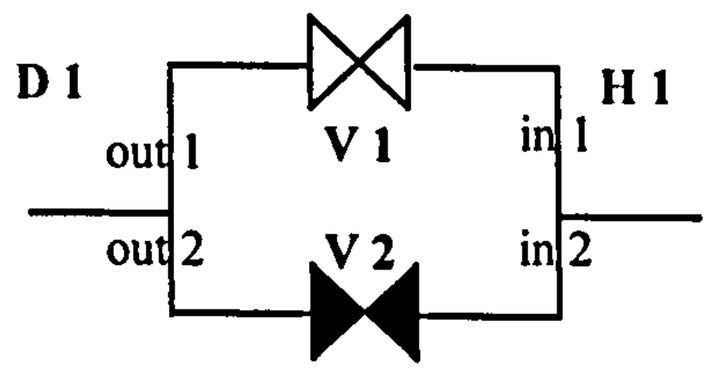


Figure 2.11 A Simple Bypass Line

Consider the effect of a partial blockage of in2 of H1. Two paths will be found in the SDG. The shortest path is:

H1 in2, partly blocked $\xrightarrow{-}$ H1 out,Q

This indicates that 'out flow' of H1 would decrease if 'H1 in2' were blocked. The second path is:

H1 in2, partly blocked $\xrightarrow{+}$ H1 in1,Q $\xrightarrow{+}$ H1 out,Q

This would result in an increase of 'out flow' of H1.

As V2 is closed a blockage of 'H1 in2' can exert no influence on the out flow and both paths above are incorrect. This example also illustrates that problems with ambiguities and incorrect model behaviour are not constrained to recycle loops.

It can be seen that this problem will only occur in the unit-based approach as if the plant were being modelled as a whole any effects of the fault 'H1 in2, partly blocked' would not be included as the overall plant is viewed. With the unit-based approach, although the individual unit models are correct, the combination of these unit models to form a plant model do not necessarily result in a correct model. The unit models are generic and may not be able to encompass behaviour which occurs in some configurations such as those given above.

The method of Fanti et al. (described in section 2.2.1.4.) may be used to eliminate some of the ambiguities resulting from multiple causal paths. Considering the example given for case 3, mass conservation requires that 'D1 in,Q' = 'H1 out,Q'. The fault 'H1 in2, partly blocked' does not affect 'D1 in,Q'. It follows that the final response of 'H1 out,Q' to the fault 'H1 in2, partly blocked' is none. However this approach is currently limited to looking at the effects of deviations in pressure, flow and level.

The algorithm of Rose and Kramer (see section 2.2.1.2.) may be used to remove ambiguities resulting from multiple causal paths under limited circumstances. When the limited circumstances do not exist Rose and Kramer use the shortest path heuristic to resolve ambiguities. These circumstances occur when feedback takes place within one or more of the multiple causal paths. This does not appertain to any of the cases described in this sub-section. It might be expected that feedback would occur within the causal paths of the examples shown in figures 2.8 and 2.10 in a similar way to which Oyeleye and Kramer (1988) describe feedback occurring within the system shown in figure 2.3 (section 2.2.1.). However, neither of the tanks in figures 2.8 and 2.10 is assumed to be of sufficient size to allow tank 'level' to increase to the extent that it is able to increase 'out flow' of the tank, therefore no feedback takes place. None of the other approaches which examine ambiguities due to initial and final responses may be utilised to consider ambiguities due to multiple causal paths. Oyeleye and Kramer and Chang and Yu use additional quantitative knowledge to resolve the ambiguities when multiple causal paths are encountered.

In the literature no one has tried to solve the problem caused by ambiguities due to multiple causal paths in a systematic way. Applications using qualitative models of process plants either use ad hoc rules to deal with these problems when they are

discovered (Larkin et al., 1997) or resort to additional quantitative knowledge to resolve the problem (Vaidhyanathan and Venkatasubramanian,1996). All applications that claim correct results must use one of these methods.

2.3. Applications

Many of the methods used to identify and assess hazards or to diagnose faults involve tracing the paths by which faults propagate through the plant. It follows that fault propagation is a common feature of these methods. The representations described in this chapter have been used by computer aids for:

- fault tree synthesis;
- HAZOP emulation;
- diagnosis.

The following sub-sections will consider these methods. Examples of tools which apply them will be discussed. The majority of these tools are constructed to analyse continuous process systems. Exceptions which consider batch systems are noted.

2.3.1. Fault Tree Synthesis

A fault tree is a graphical representation of the logical relationship between a specific undesirable event and its initiating or causal events. The faults are propagated backwards from the specific undesirable event. The specific event is termed the 'top event'. Fault trees have been used for the identification and assessment of hazards and fault diagnosis. A fault tree may be used as a design tool to identify failure paths at an early stage. The intention is that by using the fault tree method the design will evolve to eliminate or reduce the probability of the significant failure mechanisms. A number of codes have been developed to synthesise fault trees automatically. This sub-section will discuss fault trees constructed from functional equations (Kelly and Lees, 1985; Hunt, 1992; Shafaghi et al., 1984a, 1984b) and SDG's (Lapp and Powers, 1977; Shaeiwitz et al., 1977, Allen and Rao, 1980; Allen, 1984; Kumamoto and Henley, 1986; Andrews and Morgan, 1986; Andrews and Brennan, 1990; Chang and Hwang, 1992).

2.3.1.1. Constructing Fault Trees from Functional Equations

The FAULTFINDER code of Kelly and Lees (1985) and Hunt (1992) is unit-based. A plant unit is modelled using either of the functional equation types described in sub-section 2.1.1. From the unit model mini-fault trees are constructed for the unit. For each output from the unit there are at least two mini-fault trees, one for the deviation 'high' and

one for the deviation 'low'. The top event of a mini-fault tree is a deviation of a unit output.

The following example illustrates the creation of mini-fault trees for a simple pipe model. Consider the propagation equation

T out = f(T in)

and the initial event statements

external heat: T out high

external cold: T in low

The propagation equation signifies that the outlet temperature 'T out' increases if the inlet temperature 'T in' increases. The reverse is also true, i.e. the outlet temperature 'T out' decreases if the inlet temperature 'T in' decreases. The initial event statements state that the outlet temperature 'T out' will be high if there is an external heat source and low if there is an external cold source. The resultant mini-trees are shown below.

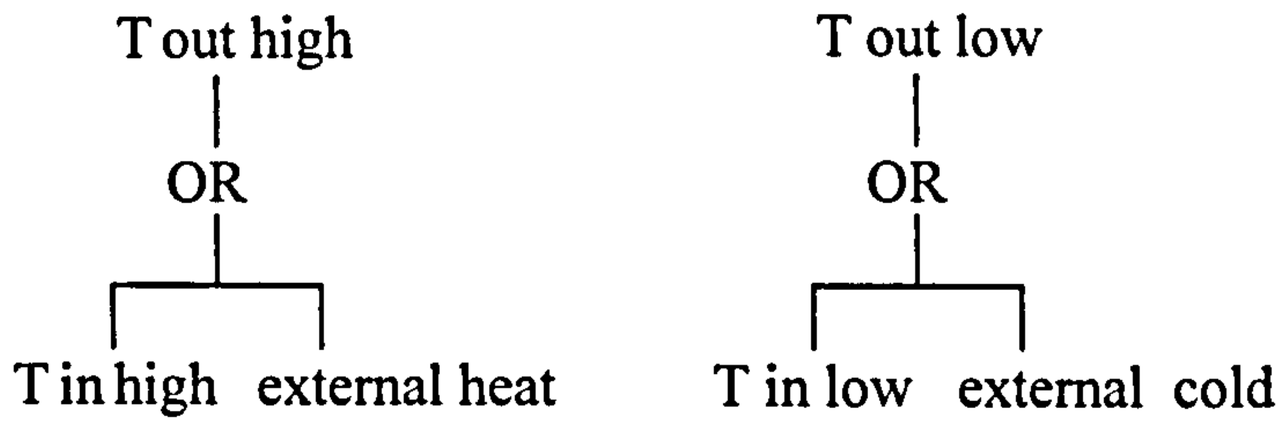


Figure 2.12 Outlet Temperature Mini-trees for a Pipe

Separate mini-fault trees are used for top events. These are constructed from terminal event equations.

The mini-fault trees form the building blocks for fault tree synthesis. Construction of a fault tree begins by nominating a top event. The appropriate top event model is retrieved first. Then the unit mini-trees which have as their top event the causes of the variable deviations which are the causes of the top event are retrieved. The causes of the top events in the set of mini-trees retrieved are found by using further mini-trees. Construction continues until the base events in the mini-trees are basic failures or the plant boundaries are reached.

An example of fault tree synthesis will be described. Assume a pipe model contains the terminal event statement:

T out high: expansion

'expansion' is nominated as the top event of the fault tree. This event has the variable deviation cause 'T out high'. The initial causes of 'T out high' are found in the pipe unit

where the top event occurs and are given by the mini-fault tree shown on the left hand side of figure 2.12. The causal events of 'T out high' are 'T in high' or 'external heat'. The 'external heat' event is a basic failure and is developed no further. The event 'T in high' does not have any causes in the pipe model. However, as 'T in high' occurs in the inlet of the pipe it must have propagated there from the next unit upstream. The appropriate mini-fault tree is retrieved from this unit.

Two types of consistency check are performed: series consistency and parallel consistency. Series consistency means that an event cannot be caused by itself or its obverse. For example, assume the two propagation equations:

$G_1 = f(Q \text{ in}, Q \text{ out})$

$Q_2 = f(G \text{ in}, G \text{ out})$

are being used to model flow in a pipe model. G is the pressure gradient variable and Q the flow variable. Section 2.1.1. explains what these equations signify. The fault tree developed from these equations is given below.

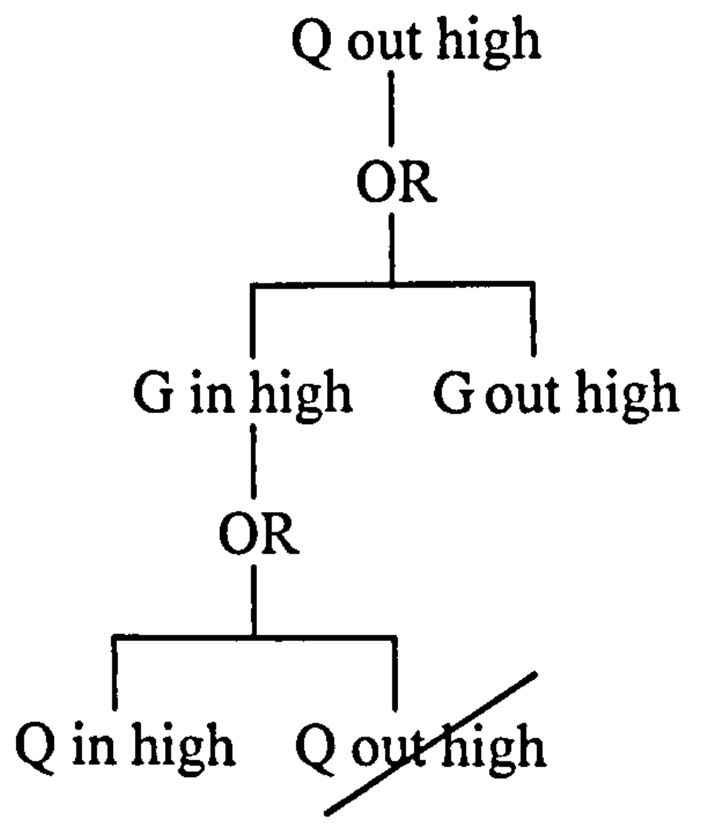


Figure 2.13 Fault Tree for High Flow

The second occurrence of 'Q out high' is deleted as it violates series consistency.

An event may also be inconsistent with certain faults. Assume the following two initial event statements are added to the fault tree given in figure 2.13.

leak from a high pressure environment: Q out high

leak to a low pressure environment: G in high

The resulting fault tree is shown below.

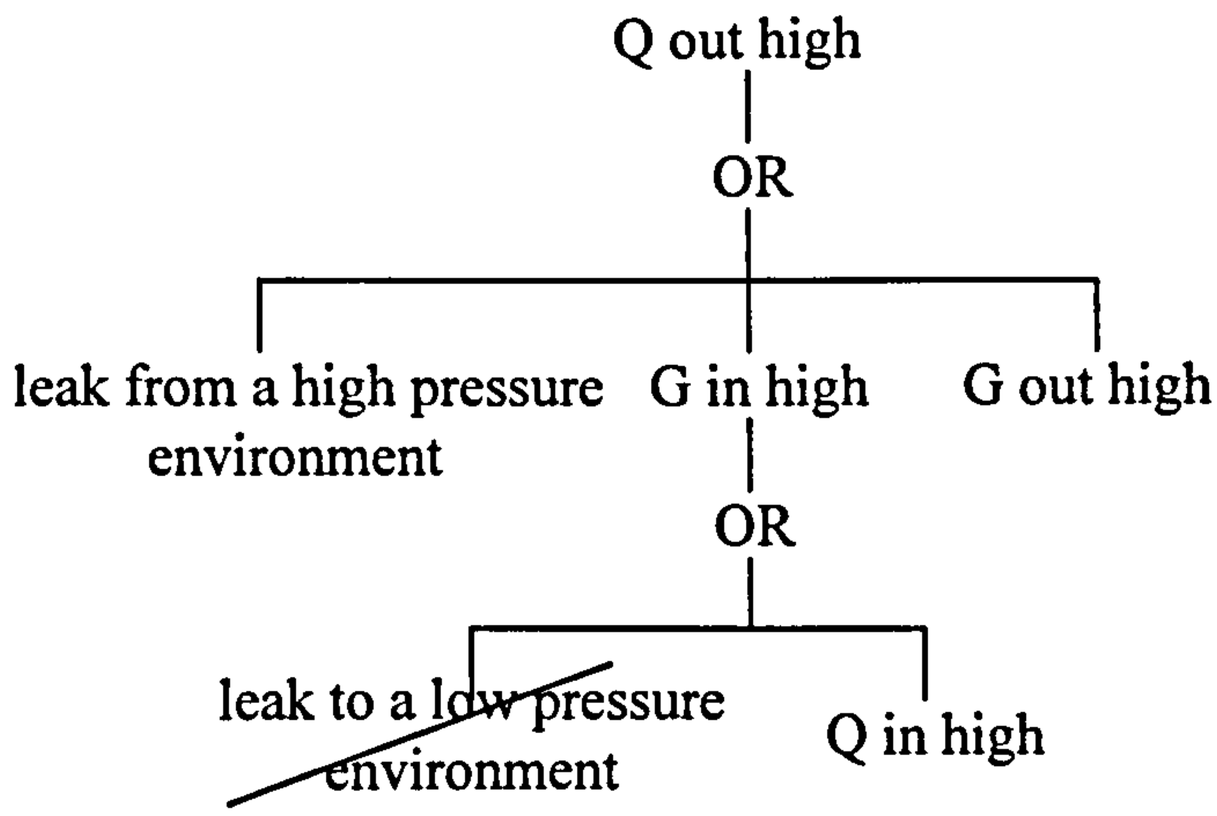


Figure 2.14 Fault tree for High Flow with Additional Events

The basic fault 'leak from a high pressure environment' forms part of the mini-fault tree for 'Q out high'. The fault 'leak to a low pressure environment' forms part of the mini-fault tree for 'G in high'. However, the fault 'leak from a high pressure environment' is inconsistent with the fault 'leak to a low pressure environment'. The consistency checks note that 'leak to a low pressure environment' is added to the fault tree after 'leak from a high pressure environment' and consequently the fault that 'leak to a low pressure environment' is deleted.

Series consistency is checked as the tree is developed. Parallel consistency is the consistency of events in one branch of the tree with events in other branches of the tree under an AND gate. This process can only be carried out once the entire tree has been synthesised.

Control loops, trip loops and divider/header combinations are treated as special cases. Information about what loops the plant contains is provided as the part of the input. The program is provided with an algorithm to effect identification of divider/header combinations. For control loops and trip loops use is made of templates (Shafaghi et al., 1984a; 1984b) which impose a structure on the tree at the point in tree development where the loop is encountered. Special models are used for divider/header combinations.

2.3.1.2. Constructing Fault Trees from Signed Directed Graphs

The first fault tree synthesis code using signed directed graphs was developed by Lapp and Powers (1977). The fault tree is constructed by selecting the node for the top event of interest. The tree is developed for the causes of this node. For a given node, the causes are the inputs to this node. Causes which violate consistency are deleted. The tree is developed down to the basic events. Special operators are used to handle control loops.

Lapp and Powers approach and the following approaches which build on their work are not unit-based.

Lapp and Powers algorithm has been extended by Shaeiwitz et al. (1977) so that it may be applied to batch processes. The arcs of the SDG's are made conditional on system events so that successive behaviour within the batch system can be modelled.

Further work on Lapp and Powers algorithm has considered control loops (Allen, 1984; Andrews and Brennan,1990; Chang and Hwang,1992). Allen looks at cascaded control loops. Andrews and Brennan put forward more operators to deal with complex nested control loops. Chang and Hwang advance further operators to deal with differing types of control loop structure and also consider process loops. They develop an algorithm to automatically identify control and process loops within the system SDG.

Lapp and Powers use '+10' to denote a very large positive influence and '-10' to denote a very large negative influence within the SDG. These large influences are assigned to arcs which propagate deviations that exceed the ability of control loops to cancel. This feature is criticised by Allen and Rao (1980) for complicating the task of SDG preparation in the presence of multiple control loops. In such instances all the control loops that can act upon a given deviation must be identified before assigning influences to the SDG arcs. Allen and Rao restrict the influences which may act upon an arc to '+', '0' and '-'. Faults initiating deviations which exceed the ability of control loops to handle are listed as circumstances under which the control actions can fail.

Lapp and Powers algorithm is further refined by Kumamoto and Henley (1986) who carry out disturbance analysis prior to fault tree synthesis to reduce the amount of manual interaction needed to create a correct fault tree.

None of these approaches can cope with all of the types of ambiguities which may occur in process plants. Some manual input is needed to build a correct fault tree.

2.3.2. Hazard Emulation

This sub-section will discuss automated hazard identification by emulation of hazard and operability studies (HAZOP). The HAZOP technique is considered as this is a widely recognised analysis technique (CIA, 1977). HAZOP is very time consuming. To overcome this there have been attempts to automate HAZOP using computer technology. (Parmar and Lees, 1987; Vaidhyanathan and Venkatasubramanian, 1995;1996; Larkin et al., 1997; Leone, 1996; Kuo et al., 1997).

This sub-section will begin by considering how a HAZOP study is carried out. The basic methodology which is used by all applications undertaking automated HAZOP is described. Further details of some of these applications are given. The applications discussed use the functional equation or SDG representations considered in this thesis.

2.3.2.1. HAZOP

The HAZOP technique applies guide words (which relate to process deviations) to plant diagrams on a line by line basis to identify causes and consequences of the process deviations. The list of guide words includes NONE, LESS, MORE, PART OF etc. Deviations are derived from a combination of guide words and process variables, e.g. if the guide word is MORE and the variable is 'level', then the combination of the two gives MORE LEVEL. The approach taken is outlined in figure 2.15 (taken from Jefferson et al., 1995). A conventional HAZOP can be applied to both batch and continuous plants.

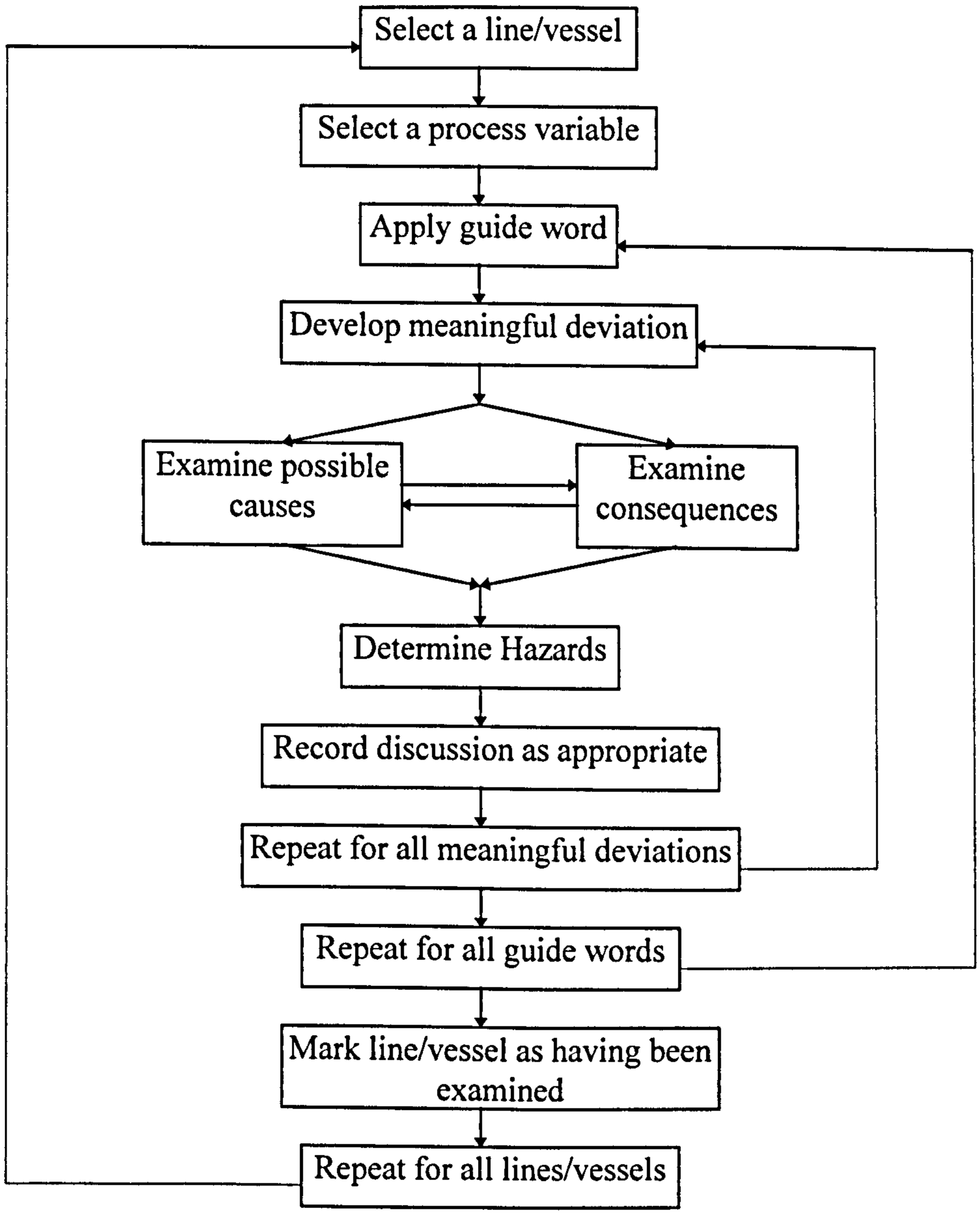


Figure 2.15 Basic Methodology for Conventional HAZOP

An example of a HAZOP study report is given in table 2.2. The plant analysed is the simple fluid flow system shown in figure 2.1. For the purposes of this example the whole system is assumed to comprise a line. For simplicity only results for the pipe and pump units of the system are given. The lack of a specific guide word for a unit in the table implies that no causes or consequences have been identified for the deviation described by this guide word for the unit.

Unit	Guide Word	Deviation	Cause	Consequence
pipe	more	more flow out	pump leak	contaminate environment, loss of material
	less	less flow out	pipe leak, pump partly blocked	contaminate environment
	none	no flow out	pump completely blocked	
	reverse	reverse flow out	pump fails	
	more	more temperature out	pipe external heat	
	less	less temperature out	pipe external cold	
pump	none	no flow in	heater blocked	overheating
	less	less flow out	heater partly blocked	
	more	more pressure out	heater blockage	possible rupture

Table 2.2 HAZOP Report for a Simple Fluid Flow System

2.3.2.2. The Basic Methodology

All the applications considered in this section apply automated HAZOP to continuous plants. The applications are unit-based. The HAZID system (Parmar and Lees, 1987) uses the functional equation representation. The other applications described (Vaidhyanathan and Venkatasubramanian, 1995;1996; Larkin et al., 1997; Leone, 1996; Kuo et al., 1997) use the SDG representation.

The knowledge required by the applications to perform a HAZOP study can be separated into two types:

1. generic knowledge;
2. plant specific knowledge.

The generic knowledge consists of a library of models of standard continuous plant units, e.g. pumps, valves, tanks etc. Each unit model describes the behaviour of process variables within the unit, failure modes (faults) and the consequences associated with the faults and deviations. The plant specific knowledge consists of the plant piping and instrumentation diagram and the process material properties. The process material properties needed vary depending on the application. Examples of material properties which may be required are: toxicity, flammability, and freezing and boiling points at various temperatures and pressures; the normal physical state of the material; the intended role of the material in the plant (e.g. reactant, product, impurity).

The generic knowledge and process specific knowledge bases are combined to perform the reasoning behind the HAZOP analysis. The piping and instrumentation diagram provides information as to what units comprise a plant and their connectivities. From this information a plant model describing the plant under analysis can be created by reference to the unit models in the unit library.

To emulate HAZOP all the faults that will lead to a particular deviation and all the consequences associated with the deviation must be explored. The inference engines of all the applications described here follow the method used in conventional HAZOP studies. The plant is examined systematically line by line. For each of the principal process variables potential deviations, their causes and consequences are considered. Figure 2.16 illustrates the basic methodology used by all the applications to emulate conventional HAZOP.

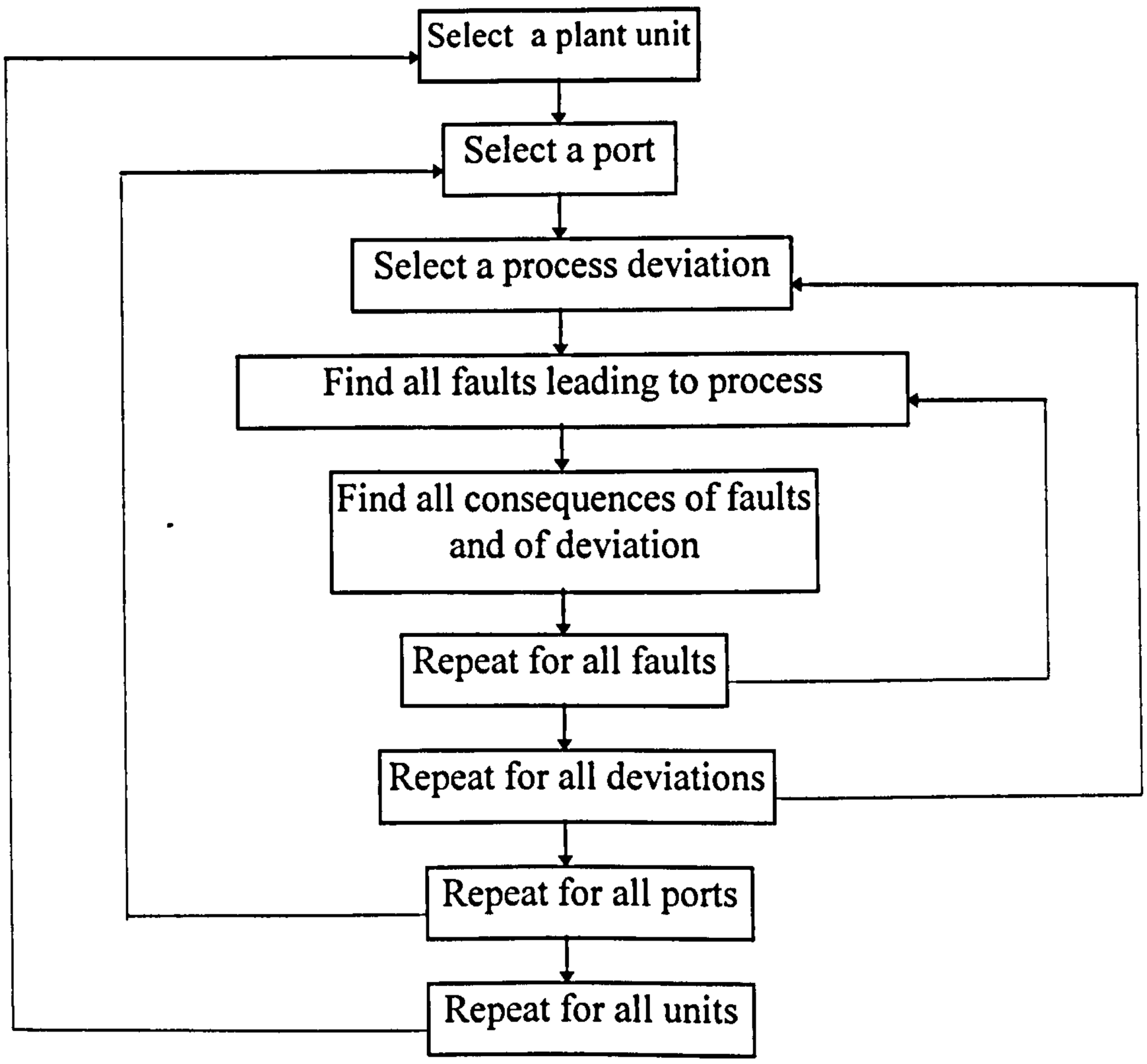


Figure 2.16 Basic Methodology Used by Applications to Emulate HAZOP

In a conventional HAZOP, the HAZOP team leader will decide the order in which the lines will be studied. The applications consider the units in the order in which they appear in the model of the plant being analysed. Each unit has number of inlets and outlets which are referred to as ports. These ports form the connections between unit models. The inlet port of one unit model is connected directly to the outlet port of another unit model. The effects of deviations are considered at all the inlet and outlet ports of the relevant unit. The propagation paths of the deviations are traced through the plant model to determine the faults and consequences.

2.3.2.3. Hazard Emulation Applications

All the applications described use the same basic methodology to emulate HAZOP. This sub-section will begin by discussing the HAZID system (Parmar and Lees, 1987). HAZID uses the functional equation representation. Other applications which use the SDG representation will then be described.

HAZID takes a block diagram of the plant to be analysed as plant specific knowledge input. The block diagram is constructed by a conversion of the piping and instrumentation diagram. For each unit in the block diagram, a unit model is specified,

drawing on the systems unit model library. Within the unit library the models have a hierarchical structure. Thus a pump is a sub-class of the class of pressure raisers and inherits certain characteristics. A model generation program exists to allow additional models to be configured by the user and added to the library.

The fault propagation through the units is modelled in terms of statements similar to Prolog clauses (Bratko, 1990). The initial and terminal events modelled by the functional equations are, respectively, the causes and consequences which the hazard identification procedure seeks to discover. In addition to the unit models there are models for the process fluids and the materials of construction. These are used to allow consequences to be made conditional on the existence of a particular process fluid or material susceptibility.

The hazard identifier program receives as input tables describing the topology of the block diagram and listing the library models to be used for each unit. The program works through the system line by line. For each line, every parameter deviation is considered in turn and causes and consequences are generated from the initial and terminal event statements in the unit models. Heuristics are used to reduce the large number of potential faults generated.

The HAZOPExpert system approach described by Vaidhyanathan and Venkatasubramanian (1995;1996) and co-workers (Srinivasan et al., 1997; 1998) appears to be broadly similar to that used by HAZID. In this approach and in the following applications described in this sub-section the SDG representation is used. HAZOPExpert is implemented in G2 which is marketed by Gensym. G2 is supported by a strong graphical user interface. This is used in HAZOPExpert to specify the piping and instrumentation diagram and to define new unit models for the system's model library. A fluid model is used to enhance the elimination of potential hazards.

Larkin et al. (1997) build on the work of Chung (1993). Chung developed the QUEEN system. QUEEN can be used as an engine to emulate various forms of hazard identification (described further in section 2.4.). The system of Larkin et al. consists of a HAZOP engine, AutoHAZID, and a number of supporting modules. The most important modules are a graphical tool, a unit model library, a model creation tool, a fluid property package, an applications programming interface and a database.

The graphical tool is a graphical user interface which allows a user to specify a process plant. In the unit model library, as in the two applications described above, each model exists within a hierarchy. The model creation tool enables a user to build a new

unit model by defining the number of chambers, inlets, outlets, phases present, failure modes, the consequences of failure modes and the consequences of deviations. Chambers are distinct internal spaces. The model creation tool is a simple text tool. The information needed to build up the elements of a unit model is elicited from the user through a question and answer session. This is the only application in the literature which provides a user with guidance on creating unit models which use the SDG representation. However the model creation tool is not fully developed. It only allows vessel models to be created. The tool also lacks flexibility. The highly structured question and answer approach does not allow the user to change the model during creation if a piece of information has been overlooked, or to edit an existing model. In addition, the user is unable to see what information the model under construction contains. The lack of these features reduces ease of use of the tool.

The fluid property package uses quantitative data to determine whether a particular scenario is feasible (see McCoy and Rushton, 1997, section 2.2.2.). The applications programming interface provides a set of functions through which the modules can communicate with each other. The database stores information for the graphical tool, e.g. icon shapes and process descriptions created using the graphical tool.

Leone (1996) defines a high level language to define unit models. This language translates into an object-oriented representation. An instance of a processing unit consists of instances of the classes VarRelationship, Consequence, Port and Compound. Instances of VarRelationship comprise SDG's representing the unit behaviour in normal and faulted modes. The Compound object contains hazard knowledge concerning the chemicals used or produced in the process, e.g. flammability, corrosivity, toxicity etc. A graphical user interface is provided to allow new unit models to be specified.

The approach of Kuo et al. (1997) differs to those described above in that fault and event trees are created as intermediate representations. Fault trees are described in section 2.3.1. An event tree is a graphical structure in which faults are propagated forward from an event. A system SDG is obtained by connecting component SDG's corresponding to all the units in the system. Loops are searched for using the method of Chang and Hwang (1992), see section 2.3.1.2. A deviation is selected and a guide word is determined. A fault tree is constructed using the selected deviation as the top event. The contents of the fault tree are generated as 'causes'. Next, an event tree is constructed corresponding to a minimum cut set. The contents of the event tree are generated as 'consequences'.

Larkin et al. (1997), using the method of McCoy and Rushton (1997), and Vaidhyanathan and Venkatasubramanian (1995;1996) consider ambiguities due to unrealistic scenarios. Kuo et al. (1997) use special operators to overcome some of the ambiguities caused by multiple causal paths within loops. None of the other applications mention ambiguities.

2.3.3. Diagnosis

When a failure occurs in a complex system the operator may be overwhelmed by the large amount of complex information obtained from the plant. However an efficient and accurate diagnosis of the problem must be made to avoid any loss of product, potential hazard or unnecessary plant shut-down. Due to this difficulty several methods of automated diagnosis have been proposed. Diagnosis determines the root causes of a process disturbance from observable deviations from the normal range of behaviour in real time. The first two applications discussed use the functional equation representation (Andow and Lees, 1975; Martin-Soils et al., 1977). The other applications described here use the SDG representation. Only the first two applications are unit-based.

Early work on diagnosis for process plants was carried out by Andow and Lees (1975). They create an alarm data structure of a plant from unit models using the functional equation representation. The plant flow diagram is converted into a block diagram of linked unit models. This representation is converted into a network of interacting process variables, basically an SDG of the plant. The network is then reduced to the network of process variables on which there are alarms. This network is used to provide the operator with information about the relations between process alarms. The network is created on an off-line computer and stored in the process computer. The network is used to analyse the alarms as they occur in real time.

An alternative method to analyse alarms has been developed by Martin-Solis et al. (1977). The input data for the system is composed of the plant topology and mini-fault tree models of the plant units. The mini-fault trees are derived from unit models represented by functional equations via the method already described in section 2.3.1.1. The data is stored in the process computer without prior processing. This data is used to construct the fault tree used to provide the information for alarm analysis in real time.

Iri et al. (1979) were the first workers to propose the use of SDG for diagnosis. SDG's are used to model the alarm structure of a plant. Special symbols are used to

represent controlled deviations. The origin of failure is found by searching through the SDG. Umeda et al. (1980) extend the work of Iri et al. to model a batch process.

More efficient search algorithms for diagnosing faults using the SDG representation which require less computer time are presented by Shiozaki et al. (1985), Kokawa et al. (1983) and Qian (1990). Kokawa et al. introduce failure propagation probabilities and failure rates of the plant equipment to prioritise candidate faults. Qian builds on the work of Kokawa et al. Frames are used to represent the nodes of the SDG. This allows the failure propagation network to be described in more detail. Heuristics are presented to allow the effect of loops to be considered.

Other workers have considered new algorithms to further improve the efficiency of the diagnostic search and to reduce the number of unresolvable hypotheses generated (Kramer and Palowitch,1987; Nam et al., 1996; Wilcox and Himmelblau, 1994). To improve search efficiency Kramer and Palowitch convert the SDG modelling the plant into a set of logical rules. The hypotheses are reduced by incorporating knowledge from numerical simulation or operating experience into the rules. Nam et al. propose a methodology to automatically construct the logical rules. Search efficiency is improved by an SDG partitioning method for on-line fault diagnosis. Wilcox and Himmelblau propose the possible cause and effect graph (PCEG) which is a modified SDG. The PCEG limits the statements that can be used to describe the root cause of a fault, based on material and energy balances. This reduces the size of the search space and the number of spurious solutions generated. Approaches to reduce the number of ambiguities inherent when using qualitative reasoning are discussed in section 2.2.

Ulerich and Powers (1988) propose an alternative method to searching the SDG for the origin of failure. Fault trees are developed from the plant SDG using the method of Lapp and Powers (1977), see section 2.3.1.2. The causal events in the fault tree are verified with real-time data. From this a fault-detection tree is derived. Fault diagnosis is accomplished with cutsets formed from the fault-detection tree.

A method for distributed fault diagnosis has been developed by Mohindra and Clark (1993). The diagnostic search is conducted by a parallel scheme in which every sensor and controller is a smart node participating in the search for possible explanations of observed anomalies.

Tarifa and Scenna (1997) look at providing an explanation for the diagnostic. All the potential process faults are found by looking at FMEA (Failure Modes and Effects Analysis) and HAZOP analysis of the process. A qualitative simulator uses the SDG of

the process to find the set of all the symptoms caused by each potential fault. These sets of symptoms are compiled into if-then rules, one for each potential fault. Potential faults are ranked using fuzzy logic. The qualitative simulation allows an explanation of fault diagnosis to be given.

Multiple fault diagnosis is considered by Vedam and Venkatasubramaniam (1997). All previous work described in this chapter uses the single fault assumption. Vedam and Venkatasubramaniam use a knowledge base to screen out physically impossible root nodes. Computational complexity is reduced by assuming that the probability of occurrence of a multiple fault scenario decreases with an increasing number of faults involved.

The emphasis of many workers in this subject has been on reducing the search time for diagnosing the faults. Other approaches consider a variety of facets, e.g. distributed fault diagnosis (Mohindra and Clark 1993), providing explanations for the fault (Tarifa and Scenna 1997) and multiple fault scenarios (Vedam and Venkatasubramaniam 1997). The only approaches to consider ambiguities are those of Kramer and Palowitch (1987) and Wilcox and Himmelblau (1994). These workers propose methods to reduce the number of spurious solutions generated.

2.4. QUEEN - A General Purpose Tool

This chapter has shown that the functional equation and SDG representations have been utilised by a range of applications employing a variety of methods to identify and assess hazards. It can be seen that it would be possible to construct a general purpose tool using these basic representations to undertake several methods. This statement assumes that the methods either all use unit modelling or non-unit based modelling. This sub-section describes the QUEEN system (Chung, 1993). QUEEN provides a general front-end for fault propagation modelling in activities such as hazard identification, fault diagnosis, alarm analysis and fault tree synthesis. QUEEN uses the unit-based modelling approach.

QUEEN is described in some detail. This is because the tool developed to carry out the method described in this thesis, Equipment Model Builder, constructs models for QUEEN. It should be noted that Equipment Model Builder could easily be modified to construct models for other applications utilising unit-based SDG or functional equation models. It should not be regarded as an application specific tool.

QUEEN performs qualitative analysis of the effects of process deviations in continuous process plants. A signed directed graph is used to determine how deviations

propagate, i.e. to represent the causal relationships. QUEEN takes a file describing the topology of a plant as input and generates the complete signed directed graph from a library file of models describing individual units that are commonly found in process plants. The library of models describing individual units is currently built by an engineer writing these models in the QUEEN format.

2.4.1. The Unit Model Structure

Each unit model is defined as a frame. The basic structure is shown below.

```
frame (unit_name isa parentname,
[
        inports info [...],
        outports info [...],
        unitports info [...],
        attributeName is ...,
        propLinks info [...],
        conditionLinks (include) exclude [...]
]
).
```

Each model within the unit model library exists within a hierarchy which supports the inheritance of characteristics between models. When a unit (child) is defined as being of a particular type (parent), the child model inherits all the information contained within the parent model which is added to its own information. For example, the units 'open tank' and 'blanketed vessel' may inherit information from a parent unit, 'vessel'. The parent unit contains information applicable to all its children. A child model contains information of relevance to itself but not to other children of its parent.

The model is composed of a declaration frame which contains a number of optional slots, each of which defines a particular type of information. The structure of a slot is: 'slot_name keyword value(s)'. There are four keywords which can be used in slots: *is*, *include*, *exclude* and *info*. The *is* keyword is used with attributes and denotes that a default value is being assigned to an attribute. The *include*, *exclude* and *info* keywords are used to define either structural or propagation information for the model. The keyword *include* corresponds to the addition of information to a model. The keyword *exclude* causes information already held by a model to be deleted. *info* corresponds to the overwriting of information held by a model. Existing model information is present as a result of inheritance.

The inports, outports and unitports slots specify the names of the process inlet, outlet and internal ports of a unit. Internal ports are used to refer to distinct regions within a unit, for example the liquid and vapour phases in a tank.

Attributes are used to define properties of process units. A user can give an attribute a value so that it becomes more specific to a particular unit in the plant under consideration. The value assigned to an attribute in a unit model is the default value.

The propLinks slot is used to add SDG propagation arcs to the unit model. More information about the propLinks arcs is given below. The conditionLinks slot is used to add or overwrite information within the model based on the value of an attribute. An example structure of a conditionLinks slot is:

```
frame (unit_name isa parent_name,
[propLinks info
        [       ---
                ---
        ],
conditionLinks info
        [
                [attributeName is X,
                        [propLinks include
                                [       ---
                                        ---
                                ]
                        ]
                ],
                [attributeName is Y,
                        [propLinks include
                                [       ---
                                        ---
                                ]
                        ]
                ],
        ]
).
```

This means if the value assigned to attributeName matches with X then add then the appropriate information to the model for attribute X. If the value assigned to attributeName matches with Y then add the appropriate information for attribute Y.

A unit model defines qualitatively:

- The propagation of deviations through a unit;
- The faults occurring within a unit and their effect on the process variables (fault initiation);
- The consequences of deviations and faults within a unit (fault termination).

An example unit model for a pipe is:

```
frame(pipe isa unit,
        [  inports info  [ in ],
           outports info  [ out ],
           propLinks info [

           %propagation

            arc([in,pressure],+,[out,pressure]),
            arc([out,pressure],+,[in,pressure]),
            arc([in,temp],+,[out,temp]),
            arc([in,flow],+,[out,flow]),
            arc([out,flow],+,[in,flow]),

           %faults
           arc([fault,'partly blocked'],-,[out,flow]),
           arc([fault,['leak into vacuum system', vacuum]],+,[in,pressure]),

            %consequences resulting from faults
            arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
            arc([fault,'leak to environment'],+,[consequence, 'loss of material']),

            %consequences resulting from deviations
            arc([deviation,[morePressure,in]],+,[consequence,'possible rupture'])
           ]
           ]
).
```

This model says that *pipe* is a sub-class of *unit*, i.e. it inherits the attributes and the default values associated with a *unit*. A pipe has one inport called *in* and one outport called *out*. The propLinks slot stores a list of arcs that define the mini signed directed graph related to a pipe. The information contained within the propLink arcs (i.e. the plant unit's faults and deviations and their resulting consequences) is specific to the individual plant unit.

It can be seen from the example pipe unit model that there are four types of propLinks arc:

(i) deviation linked to deviation, e.g. ([in,flow],+,[out,flow]);

(ii) fault linked to deviation, e.g. ([fault, 'partly blocked'],-,[out,flow]);

(iii) fault linked to consequence,

e.g. ([fault,'leak to environment'],+,[consequence, 'loss of material']);

(iv) deviation linked to consequence,

e.g. ([deviation,[morePressure,in]],+,[consequence, 'possible rupture']).

A deviation linked to deviation arc defines the effect of a deviation of one process variable in the unit on another. The syntax of such an arc is 'arc ([port1,variable1], influence, [port2,variable2])'. This thesis applies the influences defined by Larkin et al. (1997, see section 2.1.2.) to the QUEEN models. Note that for arcs containing the process

variables 'noFlow' and 'reverseFlow' influences can only be valid from the sub-set of '++' and '- -'.

A fault linked to deviation arc defines the effect of a possible fault on a process variable in the unit. The syntax is 'arc ([fault, 'fault descriptor', influence, [port,variable])'. In a fault linked to deviation arc the influence may take the value '+' indicating that the fault causes an increase in the variable or '-' which indicates that the fault causes a decrease in the variable.

The syntax of a fault linked to consequence arc is 'arc ([fault, 'fault descriptor'], +, [consequence, 'consequence descriptor'])'. The influence of a fault linked to consequence arc is always '+'.

A deviation linked to consequence arc defines the possible consequence of a process variable deviation within the unit. The syntax of this arc type is 'arc ([deviation, [deviation label, port]], +, [consequence, 'consequence descriptor'])'. The deviation label is a keyword specifying the type of deviation, e.g. moreTemperature or lessFlow. The influence of a deviation linked to consequence arc is always '+'.

The unit models constructed are generic. However, some of the faults and consequences within a unit model may not occur within a certain plant, depending on its operating conditions and the properties of the process fluids. The use of physical conditions allows the feasibility of a fault or consequence to be determined. The syntax of the fault and consequence descriptors is altered when a physical condition is present to '['descriptor', physical condition]'. For example, consider the arcs from the pipe model '([fault,['leak into vacuum system', vacuum]],+,[in,pressure])' and ([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]])'. If on resolution of the condition the fault or consequence is found not to be valid, it is excluded from the output report.

2.4.2. The Plant Description

The plant description file for QUEEN can be partially generated from a CAD system or constructed using a text editor. The units in the plant are specified using 'instance' statements. The following sample description shows that a outlet unit is connected to the outport of a pipe, the pipe is connected to the outport of a valve and that the valve is connected to the outport of pump. The plant described is illustrated in figure 2.17.

```
instance (tail1 is a outlet,
        [outports info
                [ ]
        ]
).

instance (pipe1 isa pipe,
        [outports info
                [out is [tail1,in]]
        ]
).

instance (valve1 isa valve,
        [outports info
                [out is [pipe1,in]],
                aperture is open
        ]
).

instance (pumpJ1 isa pump,
        [outports info
                [out is [valve1,in]]
        ]
).
```

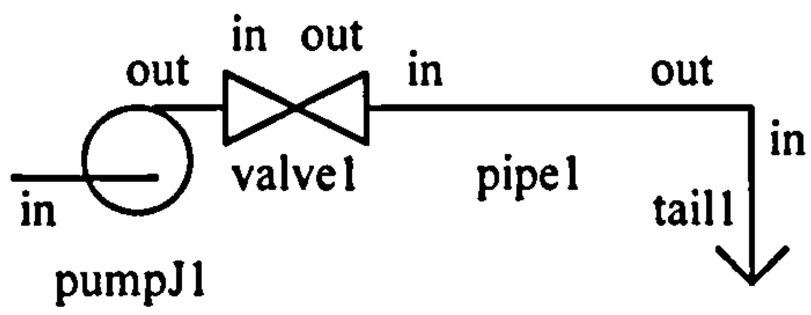


Figure 2.17 A Sample of a Plant Section

This example shows the use of instance statements to represent a specific pump, valve, pipe and outlet occurring in a plant. Compare the frame statement describing a unit model of a pipe given in section 2.4.1. with the instance of pipe1 in the plant.

The valve unit instance, valve1, is assigned an attribute 'aperture is open'. This means that information from the matching conditionLinks slot of the generic unit valve model will be added to 'valve1'. The outlet unit instance, tail1, obviously is not connected to any plant units. This is shown by '[ ]'.

2.4.3. Facilities Provided

QUEEN provides a number of commands to analyse how a plant behaves. A user may examine a list of all the paths between any two given process variables, a list of effects due to the propagation of a deviation through the paths and the effect one process variable has on another process variable given a deviation. Other commands analyse control loop behaviour and construct fault trees.

A driver, CHEQUER (Jefferson et al., 1995), has been developed for QUEEN to emulate conventional HAZOP. Like the other applications undertaking HAZOP described in this chapter, CHEQUER follows the method used by conventional HAZOP. CHEQUER uses a fluid library to filter its output as certain faults and consequences are only applicable given particular fluid properties. For example, the possibility of fire or explosion requires that the fluid be flammable. Fluids present in the plant are specified within the constituent units of the plant model using physical conditions.

QUEEN has only been developed to handle continuous processes so far. This reduces the number of guide words and process variables that need to be handled by CHEQUER. For process plant the main process deviations that can be generated from the combination of guide word and variable are:

HIGH/LOW FLOW

NO FLOW

REVERSE FLOW

HIGH/LOW PRESSURE

HIGH/LOW TEMPERATURE

HIGH/LOW LEVEL

HIGH/LOW CONCENTRATION.

Further work on QUEEN and CHEQUER has resulted in the system of Larkin et al. (1997) described in section 2.3.2.

2.5. Related Work

None of the applications described in this chapter, with the exception of Larkin et al., (1997), consider how to build the models which they employ. The method for creating models provided by Larkin et al. is not easy to use. A graphical user interface is provided by Vaidhyanathan and Venkatasubramanian (1995;1996) and Leone (1996) to allow model information to be input. Models will be created for the other applications with a text editor. The emphasis of the work within the applications described is placed upon constructing the inference engine. The work discussed in this section considers model building but does not use the functional equation or SDG representations. Tools which construct unit-based models and provide an interface to help the user build new unit models are described. These constraints are chosen because this thesis looks at providing

a method for building unit models. Two approaches fall within this category, those of Price et al. (1997; 1998) and Schut and Bredweg (1993; 1994; 1995).

Price et al. (1997; 1998) develop a system called FLAME which undertakes automated Failure Mode and Effect Analysis (FMEA) of electrical systems. FLAME takes as input circuit descriptions from a design tool. FLAME verifies that its component (unit) library contains descriptions of the operation and failure modes for each type of component (unit). The unit models are combined with circuit/function links and with the intended circuit functions and used to analyse the safety of the design. Functionality is modelled by functional labels that describe what the system is intended to do. These labels are held in a database along with risk priority numbers which describe the importance of each failure. The FLAME system calculates significance values for the effect of each failure. When the effects are ambiguous or when FLAME is unable to decide on the significance values of complex effects the entry is left blank for the engineer to fill in. An interface, Component Builder, exists to specify specialised units. The units are of the same types that engineers employ during design. Component Builder consists of a set of forms which provides an engineer with a guide as to what information is required to build a unit model.

Schut and Bredeweg (1994) divide qualitative model construction into two main subtasks: specification and debugging. Model specification is decomposed further into specifying basic building blocks and constructing compound building blocks out of the basic blocks. They describe a graphical user interface that supports a knowledge engineer in defining the models. The functionality and layout of the interface are based on a task analysis and an explicit assignment of tasks to user and system.

Model debugging is decomposed into a diagnostic task and a repair task (Schut and Bredeweg, 1993). A qualitative simulator is used to derive the expected behaviour of the device. The knowledge engineer is required to point out the discrepancy between predicted and expected behaviour. A technique is described which identifies and removes irrelevant detail from the qualitative model. Irrelevance is defined as existing in two forms: superfluous (unused) model behaviour and behaviour that is overly detailed. Rules are used to specify the conditions under which the model behaviour is either relevant, overly detailed or superfluous. Further work looks at eliminating incorrect derivatives (Schut and Bredeweg, 1995). The user identifies one or more derivatives that are incorrect and the derivatives that are expected. Hypotheses are generated that eliminate the

incorrect derivative and realise the expected derivative. These candidates for model adaptation are presented to the user who decides whether the hypothesis is accepted or not.

2.6. Conclusions

Fault propagation considers the propagation of deviations of process variables initiated by a causal fault. The process variable deviations may be represented qualitatively. This chapter considers fault propagation using the functional equation and SDG representations. Using qualitative modelling means that ambiguities may arise. Current approaches to resolve ambiguities fall into three categories:

1. those due to the initial and final effects within feedback;
2. those due to unrealistic scenarios;
3. those due to multiple causal paths.

Many of the applications described in this chapter do not consider ambiguities. The applications that do take ambiguities into account (for example, Lapp and Powers (1977), Chang and Hwang (1992), Kuo et al. (1997), Rose and Kramer (1991), Oyeleye and Kramer (1988), Chang and Yu (1990) and Fanti et al. (1993)) use specific rules to cope with ambiguities occurring in certain models. Further work is needed to resolve the problems caused by ambiguities in applications undertaking fault propagation. This thesis will consider ambiguities due to multiple causal paths.

There are two possible approaches for dealing with this type of ambiguity. One way would be to develop an analytical method to resolve some or all of the ambiguities which occur when unit models are combined. This could result in a large amount of work being generated each time a new plant model is created. Combining an analytical method with the existing unit models would cause model clarity to be lost as the existing unit models may describe incorrect behaviour when they are combined. The user would have to refer to the analytical method to understand the behaviour of the process plant.

The approach considered in this thesis to overcome ambiguities due to multiple causal paths is a modular modelling approach. It considers the influence exerted by a plant module (e.g. a recycle loop) as a whole on the other plant units. The plant module forms a complete new unit. Creating a module removes the multiple causal paths which lead to ambiguities. The advantage of this method is that the user is able to see clearly what influences the new models are describing. The modules created may be reused in other process plants. This modular approach is detailed in chapter 5.

This chapter has described applications using the functional equation and SDG representations for fault tree synthesis, hazard emulation and diagnosis. A general purpose tool providing a common set of procedures for these applications and related work which discusses the creation of qualitative models have also been described. The majority of the applications using functional equations and SDG's concentrate on construction of the inference engine. How the fault propagation models used are created is not considered. There is a need for a method to allow an engineer to create models simply and correctly. A method is proposed in chapter 4.

3. Knowledge Acquisition Tools

Knowledge acquisition deals with the structuring of expertise for expert systems. Knowledge acquisition tools enhance the knowledge acquisition process for gathering and structuring knowledge. They allow experts to develop their own knowledge bases. Some tools do not require an intermediate knowledge engineer to structure and encode the knowledge. The expert should only be required to have basic computer literacy. Hence knowledge acquisition tools reduce the human effort. Ideally experts will be more efficient when they are entering the knowledge directly into a tool than when having to teach developers all the knowledge relevant for a system (Neale, 1988; Eriksson, 1994).

Many knowledge acquisition tools consist of workbenches or toolsets, for example AQUINAS (Boose and Bradshaw, 1988), ETS (Boose, 1986), PATHFINDER (Cooke and McDonald, 1988) and KRITON (Diederich et al., 1988). This means that they comprise several tools that undertake different knowledge acquisition techniques (see later for a description of these techniques). Diederich et al. (1988) justify the use of workbenches with the observation that "no single acquisition method will be powerful enough to overcome the so called knowledge acquisition bottleneck in knowledge engineering." Workbenches allow one knowledge acquisition technique to complement another in order to elicit more information or to enable better structuring of information.

Ways of classifying knowledge acquisition tools are those of Musen (1989), Boose (1989a; 1989b and 1990) and Eriksson (1994). Musen (1989) uses a taxonomy based on the terms and relationships that a given tool uses to establish the semantics of a user's entries. Musen calls these relationships conceptual models and distinguishes three types:

1. models based on the data-structures required by the expert systems e.g. 'if ..then' rules;
2. models based on a generic problem-solving method;
3. models based on a set of application tasks to be performed e.g. OPAL (Musen et al., 1988) presents a model of this type.

Musen further classifies the generic problem-solving methods into explicit and implicit methods. In an explicit method the way in which knowledge for a generic task is used to arrive at a problem's solution is modelled. Implicit methods shield the user from specific details of the problem-solving method. Musen admits that some knowledge acquisition tools may not fit into this taxonomy.

Boose (1989a, 1989b and 1990) uses a similar classification method to Musen but groups the problem-solving methods together with methods based on application tasks, i.e.

categories (2) and (3) above. Boose justifies this by arguing that "most problems are strongly linked to certain types of problem-solving methods. Consequently, certain types of domain knowledge and possibly control knowledge should be acquired to build the corresponding knowledge-base."

Eriksson (1992 and 1992a) defines four major tool types:

1. conceptualisation tools;
2. task specific knowledge acquisition tools;
3. domain oriented knowledge acquisition tools;
4. refinement tools.

Conceptualisation tools present a picture to the expert of the way the domain knowledge elicited is structured although none of them build a domain model. They appear to comprise the same category as tools modelling Musen's implicit methods although Eriksson makes no claims as to problem-solving methods. Conceptualisation tools utilise psychological techniques. Some tools are based on psychological theories, others on computing theories. Task specific knowledge acquisition tools are the same as tools using Musen's explicit methods. Domain oriented tools are based on a conceptual domain model i.e. perceptions about the data on which they operate. For example, spreadsheet programs conceptualise their data as rows and columns of interdependent numbers. Domain oriented knowledge acquisition tools adopt conceptual models that reflect recurring domain tasks to be performed. The methods by which they are performed are not depicted in the tool. These tools can be seen to be the same as those in Musen's class (iii). Refinement tools aim at improving an existing knowledge base. These categories are not mutually exclusive. Also some of the tools may not fit into any of the categories. Eriksson's scheme is not totally definitive but is only intended as a guide.

For brevity this thesis does not classify the whole field of knowledge acquisition tools. The discussion is limited to looking at the applications of the knowledge acquisition tools from two angles: the tasks the tools undertake and the domains they undertake those tasks in. Some tools are generic in that they can be applied to several domain areas. These will be referred to as 'tools for generic tasks'. The second category of tools, the 'domain oriented tools', is usually applied to specific application areas, for example to develop medical protocols for cancer treatment. There is an overlap between these categories. Tools for generic tasks are the same as tools using Musen's explicit methods. Eriksson's term "task specific knowledge acquisition tools" is not used as the tasks

referred to are generic. Domain oriented tools are the same as Musen's class of tools using conceptual models based on a set of application tasks.

This chapter begins by discussing tools for generic tasks and domain oriented tools. Metatools are described to show where current research is leading. Finally, knowledge acquisition tools are surveyed. The survey considers their desirable features in relation to qualitative modelling.

3.1. Tools for Generic Tasks

Generic tasks may be divided into analysis tasks and synthesis tasks. Analysis tasks involve identifying sets of objects based on their features. Synthesis tasks require a solution to be built up from component pieces or sub-problem solutions. Analysis and synthesis tasks can be further broken down into sub-problem areas. For example, diagnosis is a sub-problem of an analysis task. It consists of inferring system malfunctions from observables. Design is a sub-problem of a synthesis task. It consists of configuring objects under constraints. A problem may require a combination of both analysis and synthesis tasks. It will not be attempted to break down the tasks further as sources disagree and contradict about which area the sub-problems should be classified in (Eriksson 1992a; Eriksson and Larses, 1992; Boose, 1989a, 1989b and 1990; Clancey 1985).

A generic task may be used as a building block to which a particular knowledge acquisition problem may be matched. The problem-solving method for the generic task is already provided by the tool. The method is used to describe the task. The user must conceptualise the entered knowledge in terms of this problem-solving method.

Analysis tasks use the heuristic classification problem-solving method. In heuristic classification problem features are mapped onto solution features. The solution features are refined down through a hierarchy of solutions into specific solutions. In this method a solution to the problem is selected. For example, MOLE (Eshelman, 1988) asks the expert to list relevant events (hypotheses and symptoms) for its application area. MOLE then requests knowledge which may explain the symptoms. Knowledge which helps differentiate which hypothesis is the most likely explanation of a symptom is also requested. MOLE uses the knowledge elicited to build a network of associations where each node is an event and the links represent explanatory relationships.

In synthesis tasks the solution to a problem cannot be pre-enumerated. Reasons for this might be that there are no pre-existing links for mapping problem descriptions to

solutions directly or that there are too many potential solutions to enumerate. If a solution to a problem cannot be pre-enumerated the solution must be constructed. This comprises the heuristic construction method. SALT (Marcus, 1988) is one of the few knowledge acquisition tools to use this method. The expert provides procedures for obtaining initial values for pieces of a design, procedures for obtaining any constraints on these values and revisions for violated constraints. The expert may supply this knowledge in any convenient order. Thus SALT takes a bottom-up approach to building the knowledge base, exploiting the knowledge that experts find easiest to provide. SALT checks that all pieces of design mentioned by the expert have associated procedures and that all constraints that might be violated have remedies. This propose and revise process continues until no more additions can be made to the design.

Using the above definitions it is now possible to choose tools using an appropriate problem-solving method for the task in hand. For example, MORE (Kahn, 1988), MOLE (Eshelman, 1988), TDE (Kahn et al., 1988) and YAKA (McDermott, 1988) are examples of tools using methods specific to analysis problems. The problem-solving methods used are specialised forms of heuristic classification. SALT (Marcus, 1988), on the other hand, employs a method to solve a synthesis problem.

3.2. Domain Oriented Tools

In domain oriented tools the relationships within the application area itself are used to govern access to an expert system's knowledge base. Defining a particular application area requires that users apply the explicit terms and relations of a generic task model to a specific task instance. Users are insulated from implementation details of representing knowledge permitting information to be entered in more familiar terms.

Given the power of domain oriented tools relatively few have been developed. The reason could be that a large amount of effort is required to develop this type of tool. The narrow field to which the resulting tool can be applied may not justify this laborious task. Domain oriented tools are useful in application areas where multiple related knowledge bases are required. In these areas each task may be viewed as an instantiation of a common model. The tools provide knowledge structures specialised to the experts field. This means the experts will have a better understanding of the knowledge structure being used by a tool. Eriksson (1992b) claims that this more specialised support will lead to a better acceptance of the knowledge acquisition tool among experts.

Examples of domain oriented tools are: P_{10} (Eriksson, 1992b) specialised to the application area of protein purification planning; OPAL (Musen et al., 1988) specialised to creating medical protocols for cancer treatment; ALF-A (Eriksson and Larses, 1992) specialising in troubleshooting laboratory equipment; Student (Gale, 1988) for the domain of statistical data analysis and MU (Gruber and Cohen, 1988) specialised to the application area of prospective diagnosis. KNACK (Klinker, 1988) is the only example of a domain oriented tool in the literature that is not domain specific. KNACK requires that it is possible to document the task with a report. The conceptual domain model of KNACK can be customised for a particular domain. KNACK also comes within the category of tools for generic tasks. KNACK is applicable to design tasks. Although not originally intended as one, KNACK can be regarded as a metatool.

3.3. Metatools

Recent research in knowledge acquisition is turning towards metatools. These are defined as ''automated methods for creating tools'' (Eriksson and Musen, 1993) or as ''tools that enable developers to generate new domain-oriented knowledge acquisition tools from high level descriptions'' (Eriksson, 1993). Metatools are intended to be used by a knowledge engineer to build a knowledge acquisition tool for use by an expert. The proponents of metatools claim that they can be used to reduce significantly the work required to implement domain-oriented tools. Metatools are designed to ''custom-tailor'' new knowledge acquisition tools to meet the requirements of a particular domain. Most of the work on metatools has been carried out by Eriksson and Musen (1992 and 1993) and their classification scheme will be used as it is the only one currently in existence.

The metatools are classified in terms of their high level descriptions or metaviews. A metaview is the conceptual model that a metatool presents to its users. Three metaviews are defined: the method-oriented view; the abstract-architecture view and the ontological view. The method-oriented view ''lets the developer instantiate a model of a problem-solving method by mapping the concepts and relationships used by the method to the appropriate terms in the application domain - in effect creating a template for a generic method'' (Eriksson and Musen, 1993). The abstract-architecture view allows developers to instantiate and combine subcomponents into specifications of knowledge acquisition tools, independently of the method. These are then used to instantiate target tools. The ontological view describes terms and relationships shared by both the problem-solving

method and the knowledge acquisition tool. The ontological view bridges the method-oriented view and the abstract-architecture view.

PROTEGE (Eriksson and Musen, 1992) is a metatool that supports a method-oriented view. PROTEGE was abstracted from OPAL (Musen et al., 1988). Two examples of metatools that support the abstract-architecture view are DOTS (Eriksson,1993) and SIS (Kawaguchi et al., 1991). DOTS is designed to enable development of tools whose knowledge acquisition techniques utilise graphical knowledge editing by domain experts. SIS can be used to create interview-oriented knowledge acquisition tools - i.e. those that conduct a question and answer dialogue. DASH (Eriksson and Musen, 1993) supports an ontological metaview. DASH uses the relationships among classes defined in the ontology for generating the dialogue structure of the knowledge acquisition tool.

3.4. Features of Knowledge Acquisition Tools

This project has surveyed twenty knowledge acquisition tools and desirable features have been identified. Any new knowledge acquisition tool to be built should possess most of these features in order to be functional. Representative examples of knowledge acquisition tools are surveyed rather than attempting to include all existing tools. Features of the tools are identified and what techniques the individual tools use to achieve these are presented. This survey was carried out in order to show the desirable features of knowledge acquisition tools in relation to qualitative modelling. Only the aspects of the tools relevant to this survey will be expanded upon. No attempt will be made to describe or summarise the way individual tools work.

The desirable features identified are:

(i) a differentiation facility;

(ii) verification testing;

(iii) an explanation facility;

(iv) the ability to rank information;

(v) a user interface.

Not all of the knowledge acquisition tools have all of these features but a ''good'' tool should exhibit most of them. The features are interlinked, e.g. an explanation facility may show how information is differentiated. The key features are detailed in the sub-sections below and examples of techniques used by the tools to provide these features are given.

3.4.1. Differentiation

A differentiation facility is the ability to classify components as separate entities. It is required to distinguish between information segments. For example, tools eliciting knowledge within the medical field might need to differentiate patient age, type of drug required, dosage etc. Tools eliciting knowledge on chemical plants might need to differentiate between units of equipment, faults occurring within those units and the resulting deviations and consequences. It may be necessary to differentiate between rules created in knowledge acquisition tools in order to show that no duplication is occurring. Within tools using heuristic classification, differentiation is used to seek symptoms to distinguish among diagnosable events. In a similar way within tools for solving synthetic problems attributes are used to differentiate between values. Some differentiation techniques used by knowledge acquisition tools will now be described.

The triadic elicitation technique consists of presenting problem solutions (elements) in groups of three and asking the expert for attributes (traits) two of them share which distinguish them from the third. For example, taking a process plant as an application area, the expert might be presented with three valves: a check valve, a relief valve and a control valve and initially asked to distinguish the check valve and the relief valve from the control valve. The expert might reply that the check and relief valves do not have a port to receive signals. The expert might then be asked if there are any other distinguishing features. Further traits might be elicited by asking the expert to distinguish the control valve and the relief valve from the check valve and the control valve and the check valve from the relief valve. AQUINAS, ETS, KITTEN (Shaw and Gaines, 1988) and KRITON are examples of tools which use triadic elicitation.

Another technique, card-sorting, consists of preparing cards, each one bearing the name of a concept. The concepts presented should be at the same knowledge level in the domain, so meaningful sorts can be accomplished. For example, in the process engineering domain, it would be inappropriate to sort cards representing 'sulphuric acid plant' and 'pipe'. An initial set of cards is chosen, possibly as the output of some other knowledge acquisition tool. The expert is allowed to add or remove cards. The expert sorts the cards into piles according to any criterion he chooses, with the groups being different values of the criterion. There should also be a group 'unknown' else the knowledge elicited will be restricted to that which is known by the expert with respect to all the cards. The sort is recorded. The task is repeated for all other dimensions along

which the expert perceives the concepts to vary. CATO (Major, 1991) uses a computerised form of card-sorting.

Using symptoms associated with hypotheses provides another type of differentiation. A symptom is any event or state whose observation may lead to the acceptance of an hypothesis. Of particular importance are symptoms that uniquely identify an underlying problem. This technique is used by MORE (Kahn, 1988) and can be explained as follows:

"MORE pursues this differentiation strategy when it identifies a pair or triple of hypotheses for which there is no differentiating symptom. A symptom (S) is said to differentiate one hypothesis (H_1) from another (H_2) when there is a path over one or more links from H_1 to S and no path from H_2 to S. For triplets, a symptom differentiates H_1 when it is associated with H_1 but not with either of the other two hypotheses.

When MORE finds a pair or triplet without a differentiating symptom, it will ask the knowledge-base developer to provide a symptom associated with one hypothesis but not the others."

In a related manner SALT uses attributes to differentiate between values in order to determine which ones to retrieve from its data base. Attributes are used within MORE to distinguish between symptoms. The symptom attributes are represented in the MORE structuring as conditions attached to the link joining the hypothesis and the symptom.

Existing knowledge may be used to differentiate between hypotheses. This is employed by MOLE (Eshelman, 1988). Given the knowledge that event E_1 explains event E_2, it can be inferred that the presence of E_1 is likely to lead to E_2, or alternatively, that the absence of E_2 tends to rule out E_1. The user can be asked to confirm this. MOLE seeks further information for distinguishing between two explanations in a similar way to MORE.

In tools undertaking heuristic classification a different differentiation strategy is required if a symptom is found to be directly linked to two hypotheses. Within MORE this strategy is called path differentiation. MORE asks the expert for an event caused by H_1 and not H_2 that in turn causes S. MORE represents such an event as a symptom with a link to H_1 and S. MORE also employs a strategy which its creators call path division which is similar to path differentiation in that it seeks to elicit intermediate events on

causal pathways. However, the strategy differs in that the intermediate symptomatic event sought (S_2) is expected to have a greater probability of occurrence, given the hypothesis (H), than the original event (S_1). The failure to observe the intermediate event provides stronger evidence that the hypothesis has not occurred than the failure to observe the original event. S_2 is represented as a symptom with a link to H and S_1.

3.4.2. Verification

Several definitions exist in the literature of verification. Andert (1992) defines verification as "substantiation that the system does the job right. Verification requires determination through formal confirmation and/or proof that the system is correct, complete, and consistent within itself and its specification". By contrast Hoppe and Meseguer (1993) state that verification "checks a knowledge-based system against the specifications generated by its totally formalizable requirements. It is performed by checking and not by proving knowledge-based system properties". Gupta (1993) defines verification as "a process by which the requirements, specifications, and design of the system are tested and evaluated for accuracy and consistency."

For clarity, verification will be defined here as ensuring that the internal structure of each model is complete, correct and internally consistent. Verification should also ensure that the model's behaviour is plausible. This means that the model should function correctly. This overlaps with some definitions of validation. For example, Gupta defines validation as "the functional accuracy or correctness of the system's performance." Ensuring that the internal structure of the model is correct will go some way towards ensuring that the model functions correctly. Some of the methods used for verification may also be applicable to validation. There is an overlap between verification of a model and validation. However complete validation may only be achieved by testing the model in the environment in which it is going to be used.

Verification consists of a number of methods which will vary depending on the system being verified. The verification methods must ensure that the model is complete, correct and consistent. In order for a model to be consistent it must be correct and concise.

A field of work exists on the verification of expert systems. The verification methods used by knowledge acquisition tools form part of this work. A brief overview of the verification methods employed by this field will be given. This will include verification methods used by knowledge acquisition tools. Examples of techniques individual knowledge acquisition tools use to perform these methods will be given. Only

methods used to verify the expert systems knowledge base will be discussed. Methods used to verify the inference mechanism will not be mentioned as these are not relevant to the thesis.

To provide a structure for this overview two categories of verification as defined by O'Keefe and O'Leary (1993) are used: domain dependent verification and domain independent verification.

3.4.2.1. Domain Dependent Verification

Domain dependent verification matches meta-knowledge from the domain to ensure compatibility within existing knowledge. The verification methods employed by knowledge acquisition tools are domain dependent. As the tools accumulate knowledge about the domain they use this knowledge to verify new information gained as part of the acquisition process.

Four methods of verification have been identified in the knowledge acquisition tools surveyed. These are:

1. checking for missing information;
2. preventing conflicting information;
3. preventing invalid information;
4. looking for duplicated information.

These methods can be used to improve the completeness, correctness and conciseness of a model. It is unreasonable to expect the expert to be able to remember, or to be aware of, all the information used to create a complex system. In order to ensure that knowledge acquired is complete information missing is determined. It is quite possible when dealing with complex information that the expert will make a mistake. To prevent this kind of incorrect information methods exist to prevent conflicting and invalid information. So that the knowledge gathered is concise duplicated information is tested for. A user may duplicate an object by using a different name to one already entered or by entering the same object twice thus creating redundant information. Techniques for performing the verification methods will be given. A tool may use the same technique for more than one of the verification methods.

One technique utilised by knowledge acquisition tools to look for missing information is to provide an active demon continually testing for the missing information. In KRITON (Dierderich et al., 1988) this demon is called the ''watcher''. The ''watcher'' controls KRITON's intermediate knowledge representation. Every object in the

intermediate representation is placed in a taxonomic organisation. The "watcher" looks for missing links and sends a message to the user if any are found i.e., if information on inheritance paths and part-of relations is missing. Hence KRITON continually checks the knowledge as it is acquired. For example, the user may have generated several objects. The "watcher" will check that each object's relations to the other objects in the domain are specified. Further details of exactly how the "watcher" finds the missing knowledge are unclear. Continuous checking will have the disadvantage that error messages will be sent indicating information is missing which otherwise would be filled in by the expert in due course as knowledge acquisition continued. This could be very annoying to the expert and disrupt the natural flow of information. The technique of utilising an active demon to look for missing information is also used by AQUINAS. In a similar manner to the "watcher" the AQUINAS dialog manager applies a set of rules when there are missing pieces of knowledge to recommend to the expert to employ a tool from its workbench to complete the knowledge base (Kitto and Boose, 1988).

Looking for missing information may be performed by the expert system which the knowledge acquisition tool acquires knowledge for rather than the tool itself. Iterative refinement via test cases provides a technique which makes use of the expert system to look for missing and incorrect information. The expert inputs a test case and provides an expected set of results. If the results returned by the expert system differ from this, the knowledge acquisition tool tries to determine the source of errors and recommends possible remedies to the expert. The expert corrects the information and the cycle of refinement continues until the two result sets agree. The MOLE knowledge acquisition tool (Eshelman, 1988) is an example of a tool using this technique. The user gives the expert system a test case and it makes a diagnosis. If an incorrect diagnosis is made MOLE tries to determine the source of the error by assuming there is missing information and recommends possible remedies. This may mean adding knowledge or qualifying existing knowledge but sometimes the interpretation previously provided will need to be revised. However, iterative refinements means the system is not checked for completeness until after an initial version is built. This could allow a complex series of errors to develop, each compounding a previous error, making major modifications necessary.

The overwriting technique avoids conflicting information by allowing overwriting of previous specifications. This also prevents duplication. OPAL (Musen et al., 1988) utilises this technique.

Tools acquiring numerical information may use a constraint checking technique to prevent invalid information by assessing the values provided to see if they fall within a predefined range. For example, when pH values are entered into the P_{10} tool the values are constrained to be a real number between 0.0 and 14.0 (Eriksson, 1992b; 1994). KNACK also contains a similar testing technique to this (Klinker et al., 1988).

Duplicated information may be looked for by the overlap of preconditions technique. This technique uses heuristics to check that preconditions on multiple procedures that contribute to the same object are unique to each procedure. This technique is used by SALT (Marcus, 1988). SALT warns the user if overlap of preconditions occurs. TDE (Kahn et al., 1988) and KNACK (Klinker et al., 1988) also provide examples of tools utilising this technique. TDE looks for duplicated information by monitoring for failure-modes which have similar causes and consequences, or those that share the same test and repair procedures. KNACK looks for duplication by testing for synonyms, i.e. whether two designers are using different words to describe the same facts.

3.4.2.2. Domain Independent Verification

Domain independent approaches are based upon the concept of an anomaly. An anomaly is a potential error. It may be an actual error needing correcting or it may be intended.

There are several ways of representing knowledge in an expert system. Some of these are:

1. model-based systems;
2. case-based reasoning systems;
3. rule-based systems;
4. hybrid systems.

Very little work has been done on verifying model-based systems. Verification of case-based reasoning systems will not be described as it is not relevant to this thesis.

Verification methods applied to the other two representations will now be discussed. Most work on verification has been performed on rule-based systems. O'Keefe and O'Leary (1993) divide methods for verifying rule-based systems into three categories: structure, weights and statistical investigations. The majority of methods developed for verifying rule-based systems fall within the structure category. Sources vary slightly as to how verification methods which test for structural anomalies are classified (Andert, 1992; O'Keefe and O'Leary (1993); Tepandi, 1997). To maintain uniformity this thesis will use

the three categories of verification methods previously defined in this chapter. These method categories are: completeness, correctness and conciseness.

Completeness problems include unreferenced attributes. Within rule-based systems an unreferenced attribute can prevent the system from arriving at a conclusion. The omission of a rule will cause one or more attributes to be unaccounted for. The dead-end conclusion check identifies rules with conclusions that can never be utilised. The unreachable condition check identifies rules with conclusions whose conditions can never be met.

To ensure correctness within a rule-base methods are used to check that the same condition is called by the same name and that the same conclusion is called by the same name. The developer may be required to establish lists of attributes and conclusions from which rules can be constructed. Correctness checks also ensure that there are no illegitimate attributes and the structure of the rule-base is not violated. Illegitimate attributes are those values outside a stipulated set pertaining to the application domain. Examples of violations are conflicting rules, subsumed rules and circular rules. Conflicting rules are rules with identical conditions that lead to conflicting conclusions. They are of the form:

If P then Q

If P then R

If Q then not R

Both 'R' and 'not R' are inferred from the input P thus causing a conflict. A rule is subsumed by another rule if both rules have identical conditions and conclusions but the subsuming rule has additional conditions and/or conclusions, e.g.

If P then R

If P and Q then R and S

Circular rules start with some condition and return to the same condition. They are of the form:

If P then Q

If Q then R

If R then P

Checks need to be performed to remove the violations.

To secure conciseness in a rule-base methods detect redundancy. Redundancy occurs when a reasoning chain has a redundant rule. For example,

If P then Q

If Q then R

If P then R

These rules could be simplified to

If P then Q and R

In systems using certainty factors it is important to verify the weights. Verification ensures that each rule supposed to have a weight has one and that is developed alongside the theory on which it is based. Statistical investigations analyse the attributes, conclusions or firing of rules. Any of these rule facets occurring infrequently suggest the presence of an anomaly.

Hybrid systems combine rules with other structures such as frames, procedures and objects to represent knowledge. Verification methods developed for rule-based systems will also be applicable to hybrid systems but further anomalies may arise within hybrid systems which may also need to be considered. Examples will be described for each of the hybrids listed.

O'Keefe and O'Leary (1993) discuss how anomalies may arise within a hybrid system using a frame structure. To ensure correctness of the names and labels given to the frames and their slots and contents they suggest the use of lists to construct the frames from. They also suggest that these lists may be used to test for completeness. Each name on a list should be used in a frame, slot or content, otherwise the system is incomplete. How redundancy can occur within systems using frames is discussed. Frames, slots within frames, contents within frames and connections with other frames may all be redundant. Redundant frames may be determined by comparing the contents of frames with each other. Redundant slots may be found by comparing slots within a frame to each other. Redundant contents may also be found by comparison techniques. Redundant connections may be determined by inspecting the connections to other frames, for a given frame.

Renard et al. (1993) present an algorithm to verify an object-oriented knowledge-base of an expert system which combines procedures with rules. The algorithm converts the procedures into rules which are added to the existing rules to form an equivalent rule set. The equivalent rule set is partitioned into decision subtables. The subtables are checked for redundant, conflicting and missing rules and unreferenced attribute values.

Mukherjee et al. (1997) look at anomalies in a hybrid system combining rules and objects. Anomalies due to subsumption by objects and monitor/rule interaction are considered. Subsumption by objects occurs due to mixed references to classes and

instances in rules. Monitors are side-effect methods attached to attributes in objects that react to the accessing of values. A monitor may interrupt a rule matching or firing.

Mukherjee et al. detect subsumption anomalies by clustering the rule set. Adjacency matrices are used to check for subsumption amongst rules within a cluster. An algorithm is proposed to detect monitor/rule interaction. The algorithm transforms the monitors into rule-like structures. These structures are matched against the rules of the knowledge-base. A successful match indicates a possible anomaly.

3.4.3. Explanation

An explanation facility is needed to provide the user with feedback as to what is occurring within the tool as knowledge is being acquired i.e., it is needed to show the knowledge already elicited and how the knowledge acquisition tool has structured this knowledge. An explanation facility provides an insight into the methods the tool employs to acquire knowledge, for example, why differentiation is occurring within MORE. It is also necessary to tell the expert where information is missing or why it may be incorrect.

An explanation facility may be viewed as the reciprocal of knowledge acquisition in that in the first the knowledge must be 'transferred' to the expert whereas in the second the expert must 'transfer' knowledge to the tool. A consequence of this relationship is that the existing knowledge structures of the tool may also serve as the tool's explanation facility. They may be used to show the information acquired by the tool. This in turn will enable the expert to see where information is missing or if it is incorrect. This technique is used by MOLE (Eshelman, 1988). If, during iterative refinement (see section 3.4.2.1.), the expert indicates that an incorrect diagnosis has been made MOLE responds by listing those knowledge structures which have been used by the expert system to reach that diagnosis. The expert can see if any of those knowledge structures are incorrect.

The knowledge structures may be shown as graphic representations of the knowledge base. The expert may find this difficult to follow if unaccustomed to thinking in this format, i.e. a semantic net. TDE (Kahn et al., 1988) uses this type of explanation technique. OPAL (Musen et al., 1988) shows its knowledge structures as filled-in forms.

3.4.4. Ranking

Ranking in knowledge acquisition tools may be used at two distinct levels.

1. It may be used by the expert to order the knowledge. This information ranking is necessary to establish the relative importance of certain facts. This sub-section discusses the uses of this form of ranking and which techniques are applied for these uses by the tools in the survey.

2. The knowledge acquisition tool developer uses ranking within the tool to make it more user friendly, allowing the expert to locate the most important or urgent information first or to make the most efficient use of time. This will be described in the user interface (section 3.4.5.) as this form of ranking may be provided as a function of that feature.

Two sorts of information have been identified that the expert may need to rank as it is acquired. These are rules and knowledge elements.

The expert may be required to rank rules created by the knowledge acquisition tool to indicate the reliability of the data obtained. Certainty factors may be used to rank rules. MORE (Kahn, 1988) asks the expert to assign positive and negative confidence (certainty) factors to the rules generated. The positive confidence factor represents the significance of a symptom when all the rules conditions are fulfilled. The negative confidence factor represents the measure of disbelief.

Eshelman (1988) argues against certainty factors on the grounds that experts are not very good at specifying their degree of certainty. He found that experts would tend to choose a number near the middle of the range. An alternative technique is to rank explanations in order of preference. The expert indicates his preference, given some piece of evidence, by dividing the possible explanations into those that are more favoured and those that are less favoured. This is utilised by MOLE (Eshelman, 1988).

Ranking may be needed to specify the importance of knowledge pieces or elements relative to each other. In the rating scales technique the expert may rank knowledge by giving each element a rating showing where it falls on a bipolar trait scale with ordinal ratings between 1 and 5. Bipolar means that the scale is arranged between two poles or extremes such as 'hot' and 'cold' or 'easy to use' and 'hard to use'. The rating scales technique is utilised by AQUINAS (Boose and Bradshaw, 1988), ETS (Boose, 1986) and KITTEN (Shaw and Gaines, 1988).

If the user wished to view information already acquired the developer might use ranking within the knowledge acquisition tool to ensure that the most general was retrieved first. For example, in a tool acquiring medical knowledge a list of patients

undertaking a drug trial might be retrieved in preference to individual patient details. A hierarchical-type scheme may be used to retrieve more general information first. OPAL (Musen et al., 1988) is an example of a tool using this. OPAL displays its higher level forms first. Additional forms show more detailed values for elements contained in previous forms.

3.4.5. User Interface

The user should be able to enter information and also to browse, search and edit information already acquired, i.e. a user interface should be provided. Browsing is a passive process. It shows the information within the knowledge acquisition tool in a structured format. Searching consists of actively looking for information. As will be seen later browsing and searching facilities may be integrated. A knowledge acquisition tool should be able to add extra knowledge when it is found to be missing or to alter or delete it if it is found to be contradictory or duplicated, either by the user or upon verification. This means the tool should be able to edit existing information. In some circumstances a user may wish to duplicate an object and edit it to create a new but similar object in order to save effort, e.g. a complete set of information about a butterfly valve might be input quickly into an expert system by copying and altering information on a generic valve. This again requires that the existing knowledge be added to or updated.

There are two types of browsers: list browsers and graph browsers. List browsers present lists of items, possibly sorted according to a criteria, and allow the users to scroll (search) through the list. The items can be selected for certain operations such as edit, rename, copy and delete. P_{10} (Eriksson, 1992b and 1994) utilises this sort of browser. A graph browser displays the relationship between a set of items. The knowledge acquisition tool ALF-A (Eriksson and Larses, 1992) is an example of a tool using a graph browser. By selecting a node and issuing an 'edit' command the user can edit the items that each node represents. The items are edited by form-based editors. Forms are also used to edit properties of the relationships (links) between the nodes. The graph may also be directly edited, e.g. nodes may be moved, copied and deleted, links may be created or deleted. The networks created by TDE (Kahn et al., 1988) may also be edited in this way.

Searching may be achieved by matching a string against the names of knowledge base objects. TDE and SALT are examples of tools using this. In TDE the user is allowed to specify the class of the object to constrain the search. SALT uses a parameter test plus further attribute tests if necessary to narrow the candidate range. TDE also provides an

example of a tool allowing duplication and editing of objects. The failure-modes in the TDE knowledge base may be copied and edited to form new objects.

The expert is far less likely to make mistakes if allowed to enter information in a way that seems natural but this way must be unambiguous. The expert should be able to provide information as it comes to mind or to skip questions whose answers are unknown. Tools such as SALT (Marcus, 1988) require the user to visualise solutions in terms of the problem-solving model which may seem artificial. TDE's creators (Kahn et al., 1988) claim it allows developers to provide information as they wish. OPAL's forms (Musen et al., 1988) provide a model of what knowledge is expected. However, OPAL is highly structured. The user is unable to specify new domain constructs. As it is impossible to anticipate all of the constructs which might be encountered within the domain area this limits the amount of knowledge which may be gathered by the tool.

The user interface should be structured so as to allow the knowledge acquisition tool to make economic use of the expert's time. This means there may be some form of ranking within the tool in order to make it efficient. Ranking within the tool falls into one of three categories:

1. Ranking the application area;
2. Ranking the question order;
3. Ranking knowledge techniques.

Ranking may be necessary to decide which part of the application area to elicit knowledge on first. For example, the tool may be structured so as to allow information on a generic valve to be gathered first. This would mean that less would need to be gathered when a more specialised example of a valve, such as a butterfly valve, was acquired by the tool. This would increase tool efficiency. There are no specific examples of this within the tools surveyed.

Ranking the order in which a knowledge acquisition tool puts questions to the expert may also increase tool efficiency or allow more efficient use of the experts time. The questions may be ranked so as to suggest that a generic part of the application area was worked on first or so as to infer a piece of information whenever possible. This would avoid asking the expert unnecessary questions and hence save expensive expert time.

Heuristics may be used to achieve this. KNACK (Klinker et al., 1988) produces a reporting system called a WRINGER which uses heuristics to select which piece of information to gather next based on previously gathered information. The WRINGER's heuristics are based on the following premises: an inference might make a question

unnecessary, the expert might feel more natural providing information in a certain order, or information gathered in a certain order might reduce the amount of information needed. MOLE also uses heuristics to reduce the number of questions it asks the expert. MOLE's heuristics are based on inference too. For example, if some piece of information partitions the set of hypotheses explaining some symptom into two, those hypotheses to be rejected and those that are viable, and if one subset is considerably larger MOLE will assume the reason directly qualifies the hypotheses in the smaller subset. That is, if R is a reason for favouring H_1 and H_2 over H_3, then MOLE proposes that the reason provides evidence against H_3.

Ranking could be needed to order the manner in which knowledge acquisition techniques are applied by the tool. The various techniques could be applied in order of user preference. In workbench tools, containing several acquisition techniques, the user should be guided as to the most appropriate technique to use. The dialog manager in AQUINAS "examines heuristics to identify a set of alternatives for the given state of the knowledge base and recommends the strategy of greatest potential value, determined by accumulated rule priorities" (Kitto and Boose, 1988).

Directive modelling uses models of the knowledge acquisition process and the system task to suggest what to do next in the knowledge acquisition process. Directive models identify the task that the application system should perform; they discover which knowledge is needed about the application area to perform that task; and they structure the information acquired in a way that mimics the structure of the task at hand. Directive models are used to provide advice as to which tools to use to elicit knowledge about the application area. However, this knowledge is needed to select an appropriate directive model. KEW (van Heijst, 1992) attempts to resolve this problem by using generalised directive models. These are directive models but they describe parts of the problem-solving process at a very general level. Enough structure is provided to guide elicitation of more knowledge. This enables further detail to be added so that the model can be made more specific. KEW uses the generalised directive model to advise the knowledge engineer as to the most appropriate tool to chose from its workbench to elicit the domain knowledge. For example, a task might be identified as classification and its domain as mushrooms. Kew does not yet know how to classify the mushrooms but as the overall task is known it is able to recommend a tool, such as a card-sorting tool (see section 3.4.1), to acquire more knowledge.

3.4. Conclusions

The features of knowledge acquisition tools were identified. Reasons why these features are necessary was discussed. The attempt was made to show that a knowledge acquisition tool is involved with more than the straight and simple transfer of knowledge from the expert into a knowledge-based system.

Knowledge exists in different forms. Providing a workbench which integrates knowledge acquisition tools using single techniques overcomes some of the tools' individual limitations. Domain oriented tools focusing on the task domain were found to be a powerful type of knowledge acquisition tool. This approach allows specialised tool support to be provided, leading to better understanding of, and acceptance for, the knowledge acquisition tools among experts. However the cost of developing such tools is high, considering their restricted fields of application. Metatools enable developers to generate new domain oriented knowledge acquisition tools from high-level descriptions. These metatools can simplify the task of developing domain oriented knowledge acquisition tools and will be able to reduce significantly the amount of work needed to implement these tools. No existing metatool is complete in the sense that all conceivable knowledge acquisition tools can be specified in it. Further work is required to improve the generality of metatools.

Desirable features of knowledge acquisition tools were identified as:

(i) a differentiation facility;

(ii) verification testing;

(iii) an explanation facility;

(iv) the ability to rank information;

(v) a user interface.

Some differentiation techniques used by knowledge acquisition tools are the triadic elicitation technique, the card-sorting technique and using symptoms associated with hypotheses.

Verification was defined as ensuring that the internal structure of a model is complete, correct and internally consistent. A brief overview of the verification methods used by expert systems was given. The verification methods used by knowledge acquisition tools form part of this work. Verification techniques used by knowledge acquisition tools are domain dependent. The tools were found to use demons or the expert system itself to look for missing information. Duplicated information was looked for by the overlap of preconditions technique, monitoring for similar causes and consequences

and testing for synonyms. The domain independent verification techniques used by expert systems were described. Among the techniques used by expert systems were checks for dead-end conclusions, unreachable conditions and conflicting, subsumed and circular rules.

Knowledge acquisition tools provide an explanation facility by displaying their knowledge structures as lists, networks or forms. Ranking techniques used are certainty factors, order of preference and rating scales. A user interface is provided to allow the user to enter information and to browse, search and edit information already gathered. Browsers may be provided as list browsers or graph browsers. The user interface may be ranked to make economic use of the experts time. Heuristics may be used to achieve this ranking.

4. Equipment Model Builder

This thesis develops a method to construct signed directed graph models for process plants simply and correctly. A computer-aided modelling tool, Equipment Model Builder, has been constructed to demonstrate this method. The models created by Equipment Model Builder are equipment unit models. These models are utilised by the QUEEN system (Chung, 1993) described in section 2.4. in order to show that the models created are sufficient to be of use. Equipment Model Builder is not application specific. It could easily be modified to construct models for other applications utilising unit-based SDG or functional equation models.

Equipment Model Builder is implemented in wxCLIPS. WxCLIPS is essentially CLIPS which has been extended to work with an event driven style of programming and a set of graphical user interface functions. CLIPS is an expert system tool developed by NASA.

This chapter presents the design philosophy behind Equipment Model Builder and the basic features are described. Its more advanced features are described in chapters 5 and 6.

4.1. Overview

The architecture of Equipment Model Builder is shown in figure 4.1 (in text). It consists of four main components:

- an output creator;
- a unit model creator;
- a set of module tools;
- a set of verification tools.

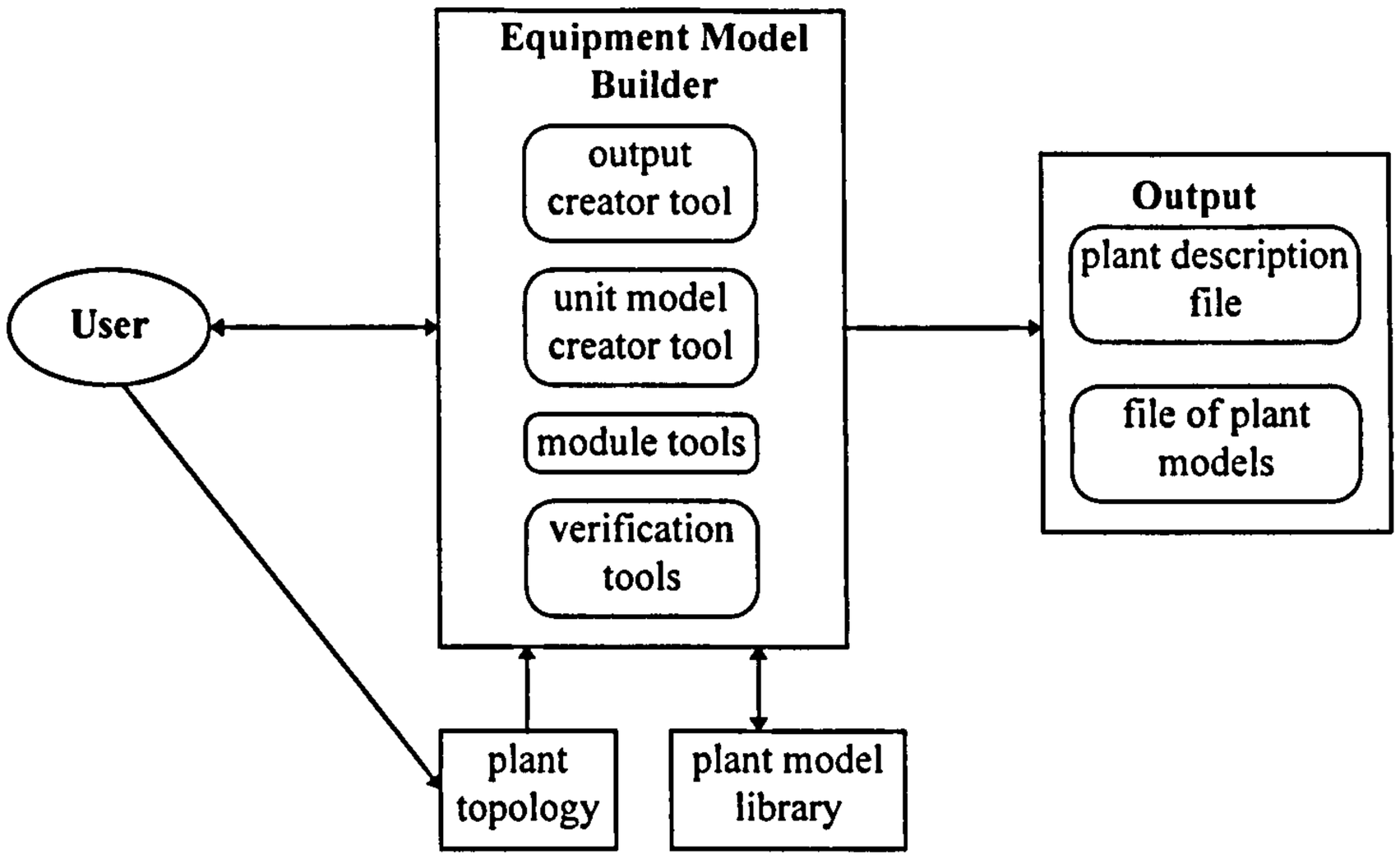


Figure 4.1 Architecture of Equipment Model Builder

The output tool takes as input a description of a plant and a library of plant models and creates a modified plant description file and a file of unit models occurring within the plant. This information forms the input for the expert system as described in section 2.4. The description of the plant topology for Equipment Model Builder is a text file which can be automatically generated by AutoCAD from a drawing or supplied by the user. More details about output creation are given in section 4.2.

The unit model creator tool allows the user to specify new unit models. Each new model specified is added to the model library. This tool will be described in section 4.3.

Within the input plant topology Equipment Model Builder identifies groups of units which could cause ambiguities when combined to form part of the overall plant model due to multiple causal paths. How these types of ambiguities arise is described in chapter 2. Equipment Model Builder applies a module creation approach to remove these ambiguities. To administer this approach Equipment Model Builder possesses a set of module tools. Equipment Model Builder automatically identifies two kinds of unit groupings which could lead to ambiguities within qualitative modelling. Each unit grouping found is checked to see whether a module for it already exists within the model library. If there is no existing module which describes the occurrence identified a new one needs to be specified by the user. Finally, the module is substituted into the plant system, replacing the individual units that make up the module and preventing the ambiguities from occurring. The plant description file created is modified to describe the module. A model of the module is placed into the output file of plant unit models. The user is also

provided with the functionality to define modules from any group of units. Details of the modular approach are given in chapter 5.

Equipment Model Builder possesses a set of verification tools to verify the models created. The verification techniques used by these tools are described in chapter 6.

Upon commencing a session, Equipment Model Builder 'asks' the user to which components access is required. The user may choose from:

- creating a unit model to add to the model library (unit model creator component and verification toolset);
- creating a plant description file and output file of units for QUEEN (output creator component, module toolset and verification toolset);
- creating a user-defined module to add to the model library (part of module toolset and verification toolset).

The design philosophy behind Equipment Model Builder will now be discussed. A need has been identified to develop a method to enable an engineer to construct SDG models. This method should be simple to use and aid the engineer in building correct models. In order to fulfil this requirement ideas from the knowledge acquisition field have been utilised.

Knowledge acquisition tools provide aids to allow the expert to achieve a structured input of information without knowledge of the internal format of the expert systems that these types of tools are acquiring information for. Knowledge acquisition tools are a type of modelling tool although they are not described as such. Knowledge acquisition tools were surveyed in chapter 3 in order to determine their desirable features in relation to qualitative modelling.

Equipment Model Builder utilises the following features:

- a user interface;
- verification testing.

Equipment Model Builder provides a graphical user interface for unit model creation. The interface also forms part of the module and verification toolsets as described in section 5.2.3. and chapter 6. The forms of the interface allow an engineer to enter a fault propagation model relatively easily using a high level representation. Equipment Model Builder converts this into an SDG representation format readable by QUEEN. (See section 2.4.1. for an example of a model written in this format). The user interface provides list browsers, differentiation facilities and ranks the question order. Information may be edited or copied. Details of these facilities are given in section 4.3.

The interface forms provide an explanation facility showing what information a model contains. Menus on the interface identify which information within a model is incomplete (see section 6.1.1.2.). Explanation is also given during the modularisation process. The user is provided with a description of the unit grouping identified as leading to ambiguities.

Equipment Model Builder provides methods to verify the models created (see chapter 6). Viewed as a knowledge acquisition tool Equipment Model Builder falls into the domain oriented category, defining a process plant as its application area.

To ensure no ambiguities due to multiple causal paths arise when equipment unit models are combined to form a plant model Equipment Model Builder utilises a modular approach. This is described in chapter 5.

4.2. Output Creation

This sub-section describes the creation of a modified plant description file and an output file of unit models occurring within the plant by Equipment Model Builder. These two files form the input for QUEEN. To create the modified plant description file Equipment Model Builder takes as input a description of the plant topology. This description is supplied by the user via a text file or a file generated by AutoCAD. An example file of a section of a plant is shown below.

'tail1', 'outlet', '[ ]', 'unspecified'
'pipe1', 'pipe', '[out is [tail1,in]]', 'unspecified'
'valve1', 'valve', '[out is [pipe1,in]]', 'aperture is open'
'pumpJ1', 'pump', '[out is [valve1,in]]', 'unspecified'

Each line describes a unit. The first field consists of an identifier for the unit. The second field is the unit name. The third details the outport connections of the unit. The name of the unit's outport, the identifier of the unit it is connected to and the name of the inport to which its outport is connected are given. The value '[]' indicates that there are no connected units. The example above only shows units with one outport. Units with multiple outports may be connected to several units. For example, the third field of a unit with two outports such as a divider might be '[out1 is [V1,in],out2 is [V2,in]]', were 'V1' and 'V2' are identifiers for valve units. The last field consists of the unit's attributes. Sections 2.4.1. and 2.4.2. provide more detail on unit attributes. The sample described is shown below.

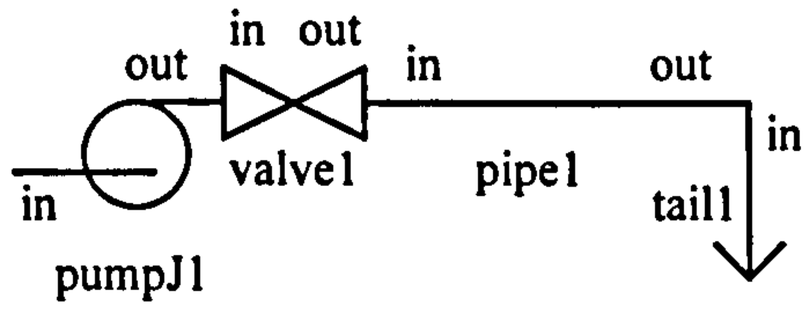


Figure 4.2 A Sample of a Plant Section

To create the plant description file to input to QUEEN Equipment Model Builder re-formats the file for QUEEN. For example, the plant section shown above would be re-written as:

```
instance (tail1 is a outlet,
        [outports info
                [ ]
        ]
).

instance (pipe1 isa pipe,
        [outports info
                [out is [tail1,in]]
        ]
).

instance (valve1 isa valve,
        [outports info
                [out is [pipe1,in]],
                aperture is open
        ]
).

instance (pumpJ1 isa pump,
        [outports info
                [out is [valve1,in]]
        ]
).
```

To build the output file of unit models occurring within the plant the tool creates a list of the types of plant units from the input plant description. The relevant unit models are retrieved from the model library and concatenated to form the output file. The modular approach utilised by Equipment Model Builder may alter the plant description file and unit model file. This is described in section 5.3.

4.3. Model Creation

To simplify model construction a graphical user interface is provided. An overview of this interface is presented and how its features relate to the overall design philosophy is

detailed. The interface must be flexible enough to express the most complicated and unusual of models. Problems encountered in the interface design will be discussed.

The method employed by Equipment Model Builder must take account of the unit model structure. A unit model is composed of structural information (the unit name, parent name and the inports, outports, unit ports and attribute slots) and fault propagation information (the propLinks and conditionLinks slots). A main window, titled 'Unit Description', in the graphical user interface allows structural information to be defined.

The information contained in the conditionLinks forms a specialised subset of the propLinks information. This allows the information in these two slots to be defined together. The propLinks and conditionLinks arcs are used to add SDG propagation arcs to the unit model. There are four types of propLinks and conditionLinks arc:

(i) deviation linked to deviation;

(ii) fault linked to deviation;

(iii) fault linked to consequence;

(iv) deviation linked to consequence.

See section 2.4. for more details of the unit model structure. Four sub-windows are created to allow fault propagation information to be defined, one for each category of arc. The four sub-windows are titled 'Deviation', 'Fault->Deviation', 'Fault->Consequence' and 'Deviation->Consequence'. The windows of the interface have a form layout. This allows model information to be clearly presented and enables the user to differentiate between the different arc types.

On starting a model creation session the main window is shown. This window remains accessible throughout the session. Closing the main window closes the application. The four sub-windows may be opened and closed by the user at will. Each window provides an option to move to any of the other windows. By accessing the main window first the interface ranks the order in which the model is created.

4.3.1. Common Features of the Windows

Each of the windows possesses drop-down menus which allow the user to access the other windows, libraries and verification tools. Three libraries are accessed via the drop-down menus. These libraries contain all the faults, consequences and process conditions for the models. The process conditions consist of attribute values for the conditionLinks slots. The information in the libraries is generic. Each library is displayed using a list browser. This allows the user to differentiate between the information contained within the separate

libraries. Facilities are provided allowing the user to add to or to change or delete members of the libraries.

Physical conditions under which the faults and consequences occur (see section 2.4.1.) are defined by placing the condition after the fault or consequence value and separating the condition from the value by a comma. An example of this for a fault would be 'blocked by frozen fluid,freezing' and an example for a consequence would be 'fire/explosion risk,flammable'. When the value is utilised within a model it is converted to the correct model syntax. For example, for the fault shown the syntax would be '['blocked by frozen fluid',freezing]'. For the consequence the syntax would be '['fire/explosion risk',flammable]'.

To provide the user with quick access a hierarchical structure needs to be implemented within these libraries. Time limitations have prevented this from being carried out. Verification tools accessed via the drop-down menu are described in sections 6.1.1.2. and 6.1.2.3.

All the windows have a 'Save' button and a 'Cancel' button. The 'Save' button has a feature which prevents the user from accidentally over-writing an existing model when the intention may have been to alter the model and save it as a new model. Clicking on the 'Save' button with the mouse stores the model and creates a specialised file for the model. The high level representation of the model information displayed by the forms of the interface is converted into an SDG representation and this information is placed within the specialised file. The specialised files of the model library are utilised by QUEEN. The user is given a choice as to if a specialised file should be created as this might not be desired if the model is not yet complete. The user is also asked as to whether the model saved should be verified. More details of this are given in section 6.2.1.1.

Clicking on the 'Cancel' button replaces the model information displayed with that stored at the most recent save command. Before the model information is altered the user is asked to confirm the cancel command.

The sub-windows are labelled with the name and parent of the model currently being viewed, e.g. 'divider is a pipe'. The 'Deviation' and 'Fault->Deviation' windows provide access to a menu of possible SDG influences. This menu has a list of members of the process condition library adjoining it (see figure 4.3). The 'Fault->Consequence' and 'Deviation->Consequence' windows also provide access to this list. The user may select one or more process conditions from the list of process condition library members to apply to the arcs created. Applying a process condition to an arc means that the arc is placed

with a conditionLinks slot within the model with the condition as its attribute value. The list of process condition library members has a choicebox associated with it. A choicebox is a special kind of menu. The choicebox values allow the user to define the structure of arcs to which conditions are applied as 'include' or 'exclude'. The structure of propLinks arcs is automatically defined as 'info'(see section 2.4.1.).

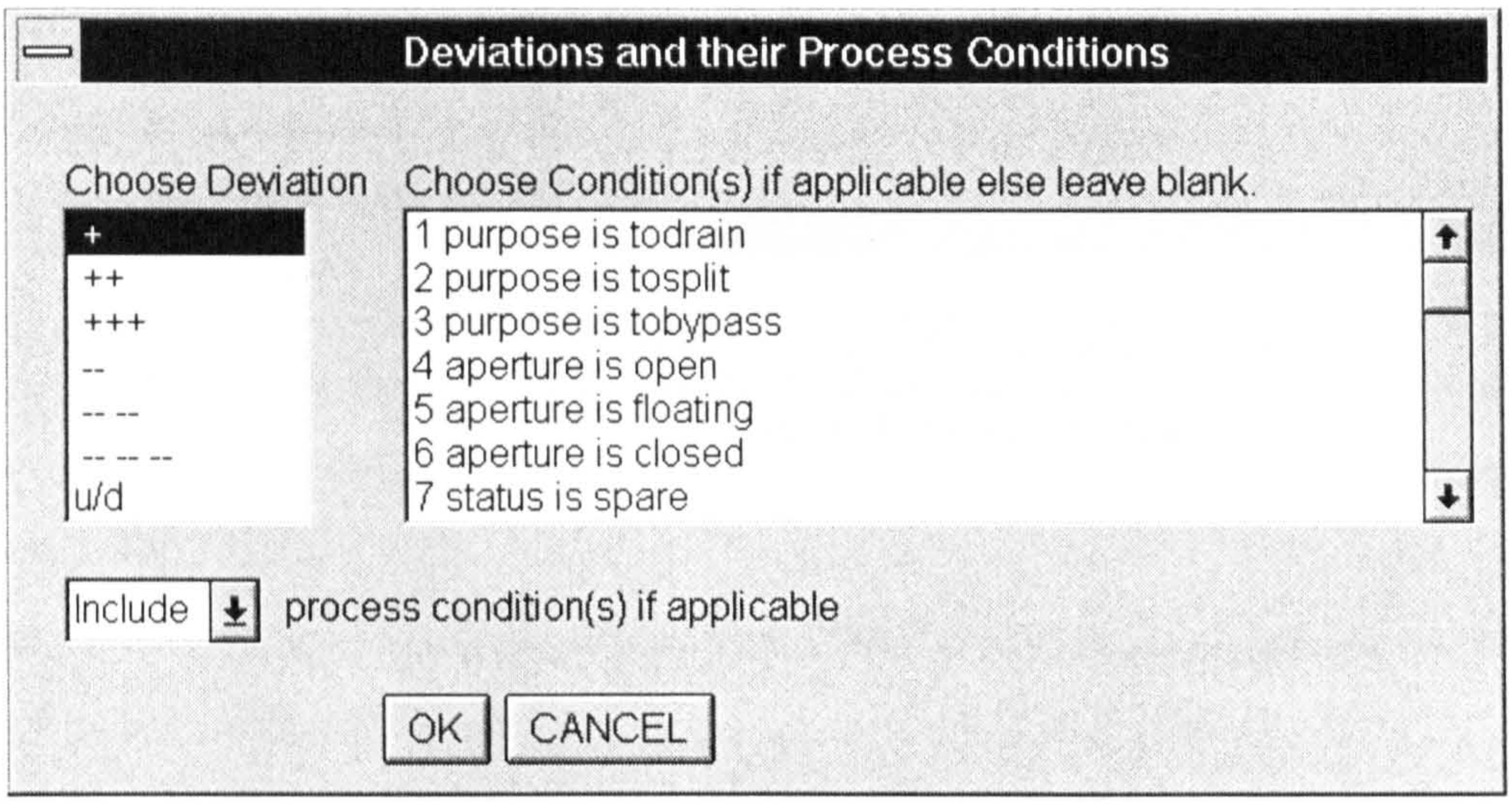


Figure 4.3 Menus of influences and conditions

Each member of the process condition library is numbered. The arcs which have conditions applied to them are labelled with the relevant numbers of the conditions. By referring to the process condition library the user is able to discover which conditions are applicable to the arcs. Space does not allow process condition names to be placed on the sub-windows. Arcs with conditions applied to them are also labelled either with 'in' or 'ex'. 'in' means that the structure of the arcs to which the conditions are applied is defined as 'include'. 'ex' stands for 'exclude'. An example of a condition label for an arc with two conditions applied to it is '[5 or 7 in]'. This states that this arc is applicable when process condition number 5 in the library applies when the model is utilised or if process condition number 7 applies or if both conditions apply. 'in' defines the structure of the arc in both cases. On the 'Deviation' and 'Fault->Deviation' windows the condition label is combined with the SDG influence upon the arc, e.g. '+ [5 or 7 in]'.

4.3.2. The Main Window

The main window provides a set of boxes which allow model structural information to be entered, a 'description' box and a list browser (refer to figure 4.4 in text). The main window also has an 'Exit' button.

By typing in the 'description' box the user may annotate a model. The list browser shows a list of the plant units and modules within the model library. The user may select a

model name in this browser to edit or delete the details of the model. To save effort for the user the graphical interface allows duplication. A user may view a model, edit it and re-save it under a different name. The model names are arranged alphabetically within the list browser, allowing the user to easily locate a model.

When the 'Exit' button is clicked, prior to the application closing, if the model displayed is a new model or if changes have been made to an existing model the user is queried if the model should be saved.

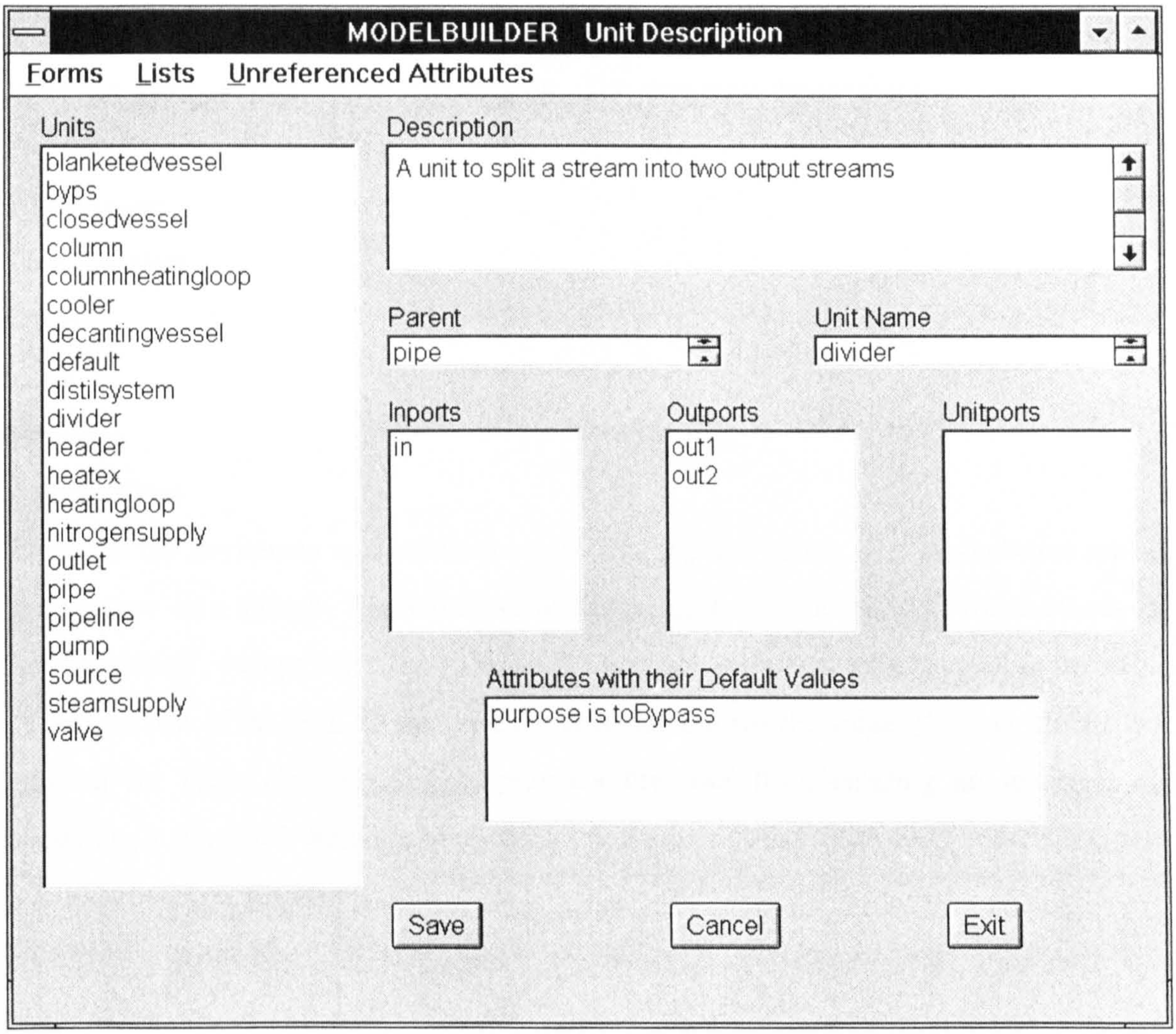


Figure 4.4 Main window

On start-up a model containing default values is shown. Selecting one of the structural information boxes with a mouse allows the user to add or change the value selected or delete the value. The structural information determines the labelling and layout of the sub-forms. This is the reason the user is presented with the main window first. Inputting information into the main window results in changes in the structure of the sub-forms.

4.3.3. The Deviation Window

The 'Deviation' window consists of a grid. The ports of the unit model form the first row and first column of the grid. There are two choiceboxes, one associated with the first row and one associated with the first column of the grid. The choicebox associated with the first row is labelled 'Port To', that associated with the first column is labelled 'Port From'. The choiceboxes each contain a set of process variable deviations. The user is able to select a deviation from the 'Port To' choicebox to act as an influence upon the ports of the first row. A deviation may be selected from the 'Port From' choicebox to act as an influence upon the ports of the first column. The user may choose from:

Pressure

Temperature

Concentration

Level

Flow

No Flow

Reverse Flow.

These are the deviations applicable to continuous process plants. The choiceboxes are set to 'Pressure' as a default. The default value for the grid elements is 'u/d', which stands for 'undetermined', except when the value in the first column of the grid is equal to the value in the first row of the grid. In this case the grid element has the value '/'. User input is not allowed for these elements. This prevents the user from entering an incorrect arc containing a deviation which propagates through a port to influence itself at the same port, e.g. '([in,flow], +, [in,flow])'.

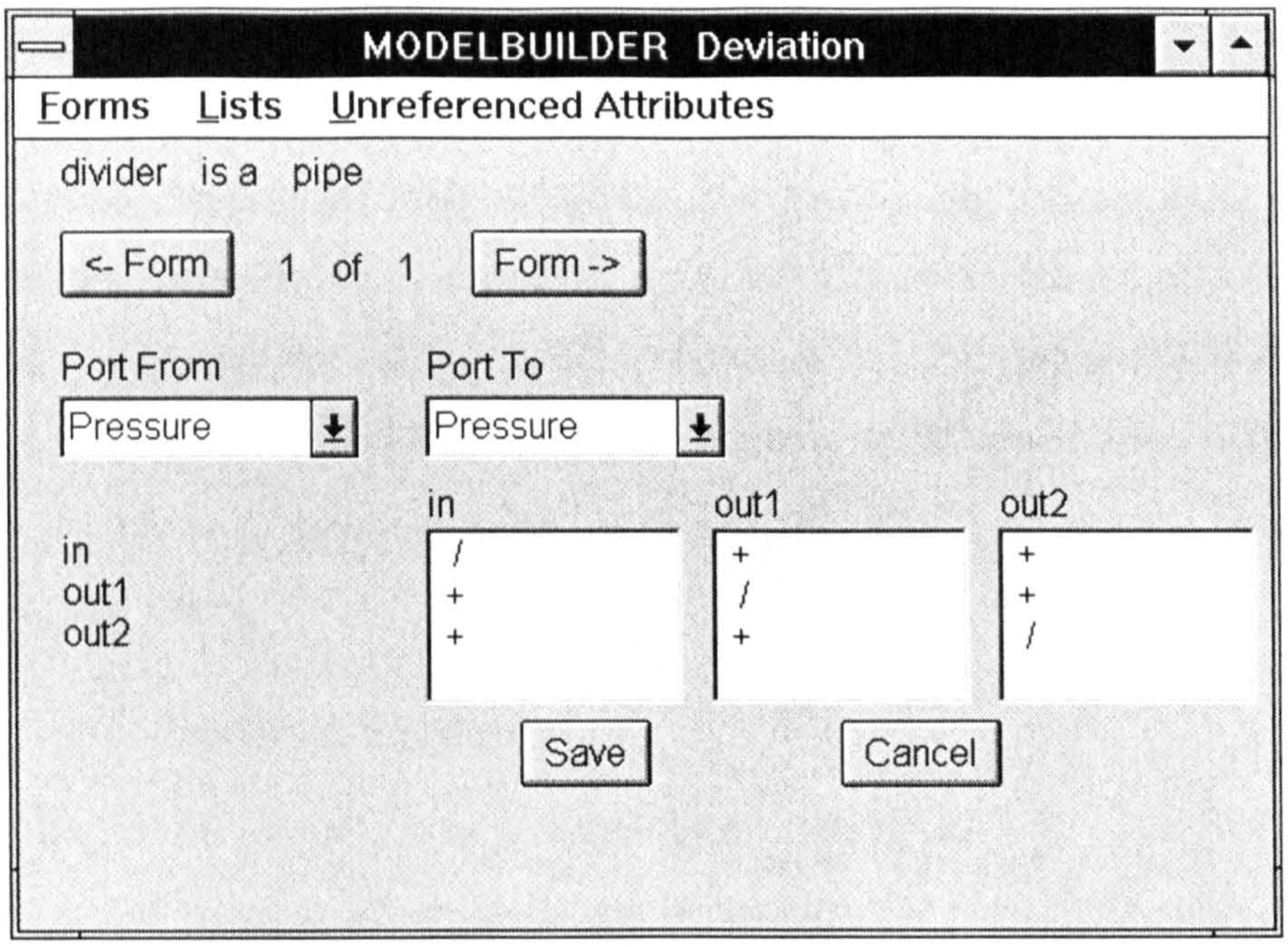


Figure 4.5 Deviation window

The 'Deviation' window has two buttons named '<-Form' and 'Form->'. These allow the user to access further 'Deviation' forms when the number of columns needed by the model is too large to fit into the horizontal space of the window. Clicking on '<-Form' with the mouse shows the previous form, clicking on 'Form->' shows the next form. Error messages are displayed if there is no previous or next form i.e. if the first or last form of the set is being viewed. Between the two buttons a label indicates which one of the 'Deviation' forms is being viewed and how many 'Deviation' forms exist for the model, e.g. '1 of 3'. The vertical space of the 'Deviation' window is limited as this was not thought to present a problem. The problem with the limited horizontal extent of the 'Deviation' window could be resolved instead by using a scrolling window. However the software used to implement Equipment Model Builder does not provide this functionality.

To create arcs on the 'Deviation' form the user first chooses the process variables in which the deviations of the arcs will occur from the choiceboxes. Selecting a value from the 'Port From' choicebox automatically sets the 'Port To' choicebox to the same value. This is to provide ease of use for the user as many process variable deviations propagate through to the same process variable. The user is also able to select another value for the 'Port To' choicebox if appropriate. Once the process variables have been selected the user employs the form's grid to create deviation linked to deviation arcs to which these process variables are applicable.

To make an arc the user links a member of the first column of the grid to a member of the first row by clicking on a default 'u/d' element of the grid with the mouse. This

brings up the menu choice of possible influences for the SDG arc being created. The user chooses a value from this menu. Any conditions which apply to the arc may be selected from the process condition library adjoining the menu. The default grid element is replaced with the value chosen. This allows the effect of a deviation at a port of the model on a deviation at another port of the model to be modelled, forming a deviation linked to deviation arc. To change an existing arc the user selects its influence value within the grid to bring up the menu of possible SDG influences. To delete an arc the value 'u/d' is selected from this menu.

4.3.4. The Fault->Deviation Window

The 'Fault->Deviation' window also presents a grid (see figure 4.6). The first column of the grid consists of a box to which faults can be added. The ports of the unit model form the first row of the grid. A choicebox of the process variables allows the user to select which deviation should influence the ports of the first row. Like the 'Deviation' window, the 'Fault->Deviation' window has two buttons with an identifying label in between. This allows the user to access further 'Fault->Deviation' forms when the number of columns needed by the model is too large to fit into the horizontal space of the window. Initially, before any values are added to the form, the boxes comprising the grid on the 'Fault->Deviation' window are blank.

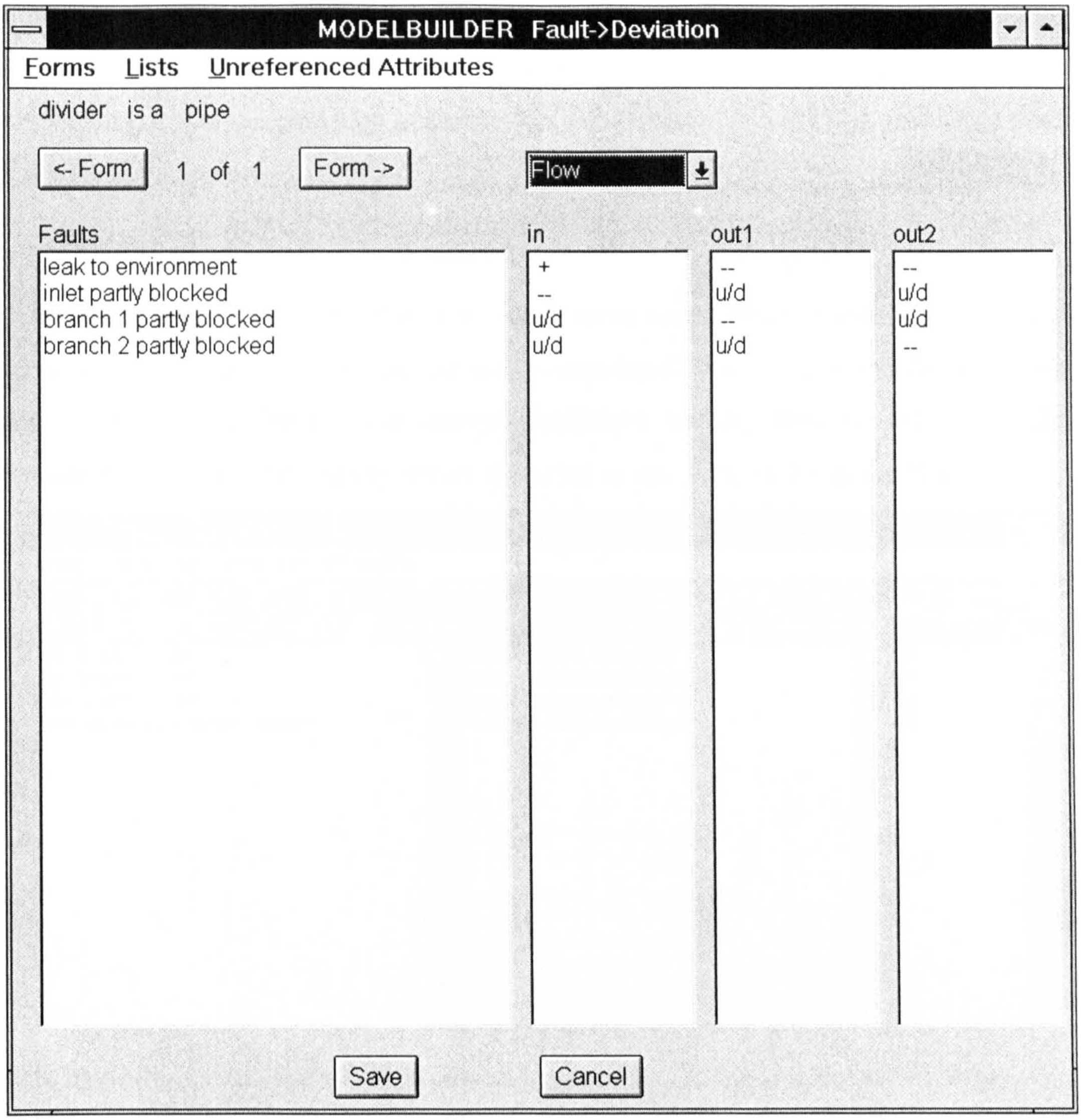


Figure 4.6 Fault->deviation window

In the initial blank grid the user may select the first line of a box on the form to add a default grid line to the form, change the element selected or delete a grid line. If the grid is not blank a grid element may be selected. An error message is displayed if the user attempts to change a grid element or delete a grid line where none exists. A default grid line consists of defining the value 'u/d' for the fault member in the first column and for each of the grid elements.

To create a fault linked to deviation arc the user first selects the process variable deviation which will be caused by the arc from the choicebox on the window. The form's grid is then utilised to create arcs which contain this deviation. Members of the first column of the grid may be defined by selection from a list of fault library members. To make an arc the user links a member of the first column to a member of the first row of the grid by selecting a default 'u/d' element of the grid with the mouse. Selecting an element

brings up the menu of SDG influences and list of process conditions described in sub-section 4.3.1. from which the user may choose values. This allows the effect of a fault propagating to cause a deviation at port within the model to be modelled, forming a fault linked to deviation arc.

4.3.5. The Fault->Consequence Window

The 'Fault->Consequence' window consists of three boxes which enable the faults, and consequences for arcs of this type and the process conditions under which these arcs are applicable to be defined. The process conditions are the attribute values for the conditionLinks slots. Before any values are added to this form the boxes are blank.

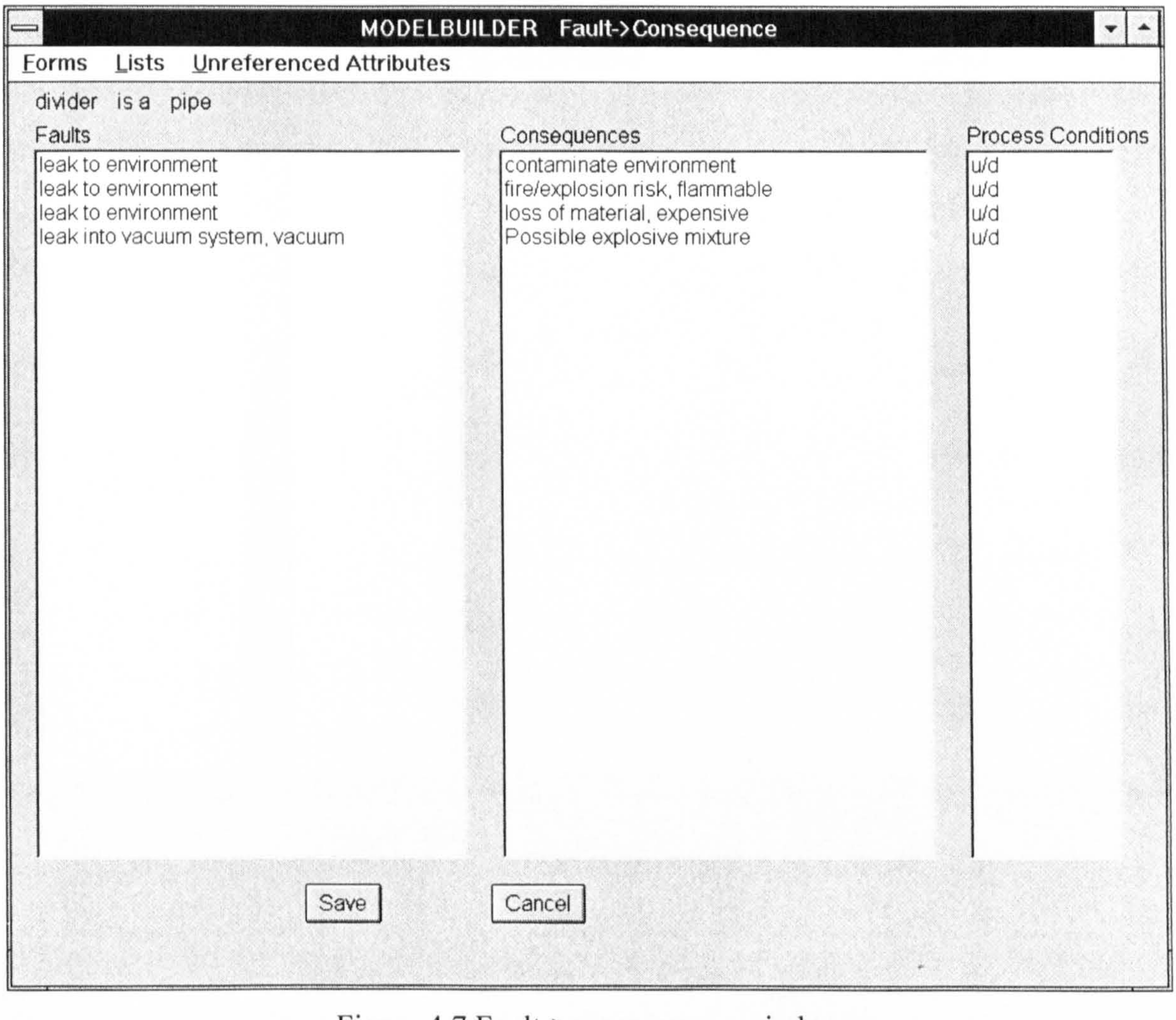


Figure 4.7 Fault->consequence window

A line of default values may be added to the 'Fault->Consequence' form via the same method as described for the 'Fault->Deviation' form. In order to define an arc on this form the user must fill in a fault and a consequence value on a line and values for the process conditions if any are applicable to the arc. The user is able to select these values by accessing lists of members of the fault, consequence and process condition libraries. It

is not necessary to define an influence for this arc type as the value is always '+'. This is automatically substituted in the arc within the model by Equipment Model Builder.

4.3.6. The Deviation->Consequence Window

The 'Deviation->Consequence' window consists of four boxes allowing the deviation labels, ports and consequences for these types of arcs to be defined and in addition the process conditions under which these arcs are applicable (see figure 4.8). Like the 'Fault->Consequence' window the boxes are initially blank.

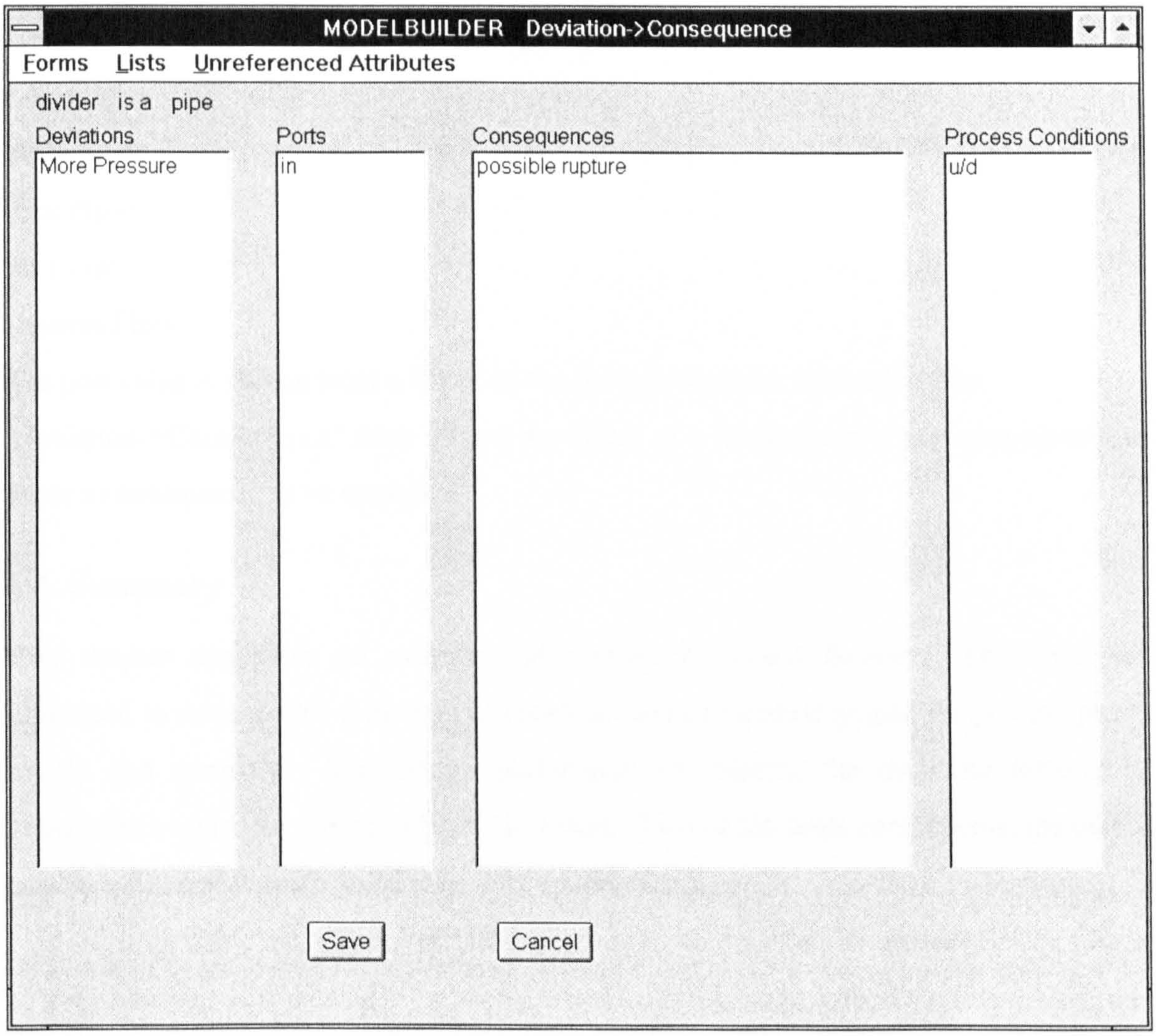


Figure 4.8 Deviation->consequence window

A line of default values may be added to the 'Deviation->Consequence' form, again using the same method as described for the 'Fault->Deviation' form. To define an arc on this form the user must fill in values for the deviation label, port and consequence on a line and values for process conditions if any are applicable to the arc. It is not necessary to define an influence for this arc type as the value is always '+' which is automatically substituted into the model. The consequence and process conditions are

defined in the same way as for the 'Fault->Deviation' form. The deviation label value is chosen from list. The list members are:

More Pressure

Less Pressure

More Temperature

Less Temperature

More Concentration

Less Concentration

More Level

Less Level

More Flow

Less Flow

No Flow

Reverse Flow

The port value is chosen from a list of all the ports defined for the model. The 'Deviation->Consequence' form allows the effect of a deviation at a port propagating to cause a consequence to be modelled.

4.4. Summary

This chapter has given an overview of Equipment Model Builder. This tool was developed to demonstrate a method to construct signed directed graphs for process plants simply and correctly. The design philosophy of utilising the desirable features of knowledge acquisition tools has been discussed. Two of the tools components, the output creator and the unit model creator have been described.

To facilitate model construction a graphical user interface is provided. The format of the interface is based on the model structure. A main window allows structural information about the model to be entered. Sub-windows permit fault propagation information to be input. A unit model has four types of propagation arc:

(i) deviation linked to deviation;

(ii) fault linked to deviation;

(iii) fault linked to consequence;

(iv) deviation linked to consequence.

A separate sub-window exists to allow information to be input for each category of arc.

Lists are provided within the interface from which the user may select values to

input. This saves the user work as the value does not have to be typed in each time it is needed. The list method also ensures that the values are uniform for all the models created by Equipment Model Builder. The interface provides a simple method to allow the user to create models. The interface forms gather all the information from the user that is needed to build a model. This information is automatically transcribed into the correct syntax for the arc of the model when the specialised files are created. This prevents the user from making syntactical errors which are time-consuming to solve.

5. A Modular Approach

The new module creation approach developed provides a method to remove ambiguities due to multiple causal paths. Chapter 2 explains how these ambiguities may arise. This modular approach takes place in three stages: identification, specification and substitution.

Equipment Model Builder takes a file describing a plant as input. Within this process plant the groupings of units together with the connections between them that could lead to ambiguities must be identified. Identical structures consisting of different unit instances may occur within the same plant or different plants. For example, the loop given in figure 5.5 (see section 5.2) might be found within another plant description with different component names. The plant description is searched for a unit grouping which could lead to ambiguities. If an occurrence is found it is checked to see whether a module for it already exists. If there is no existing module which describes the unit group identified a new one needs to be specified.

In the specification stage the units of the plant fragment are amalgamated to form a module. Making a plant fragment into a module removes the causal paths that lead to ambiguities or incorrect behaviour. The effect of deviations and faults upon the units comprising the module are retained. Where multiple causal paths still exist, the shortest path heuristic has the correct influence. The module specified is added to a library of models.

In the final stage, the module is substituted into the plant system, replacing the individual units that make up the module. The plant description is modified to describe the module. A model of the module is placed into an output file which contains the plant's component unit models. The system returns to the identification stage. The plant description is searched until no more unit groupings which could lead to ambiguities are found. Figure 5.1 gives an overview of the modular approach.

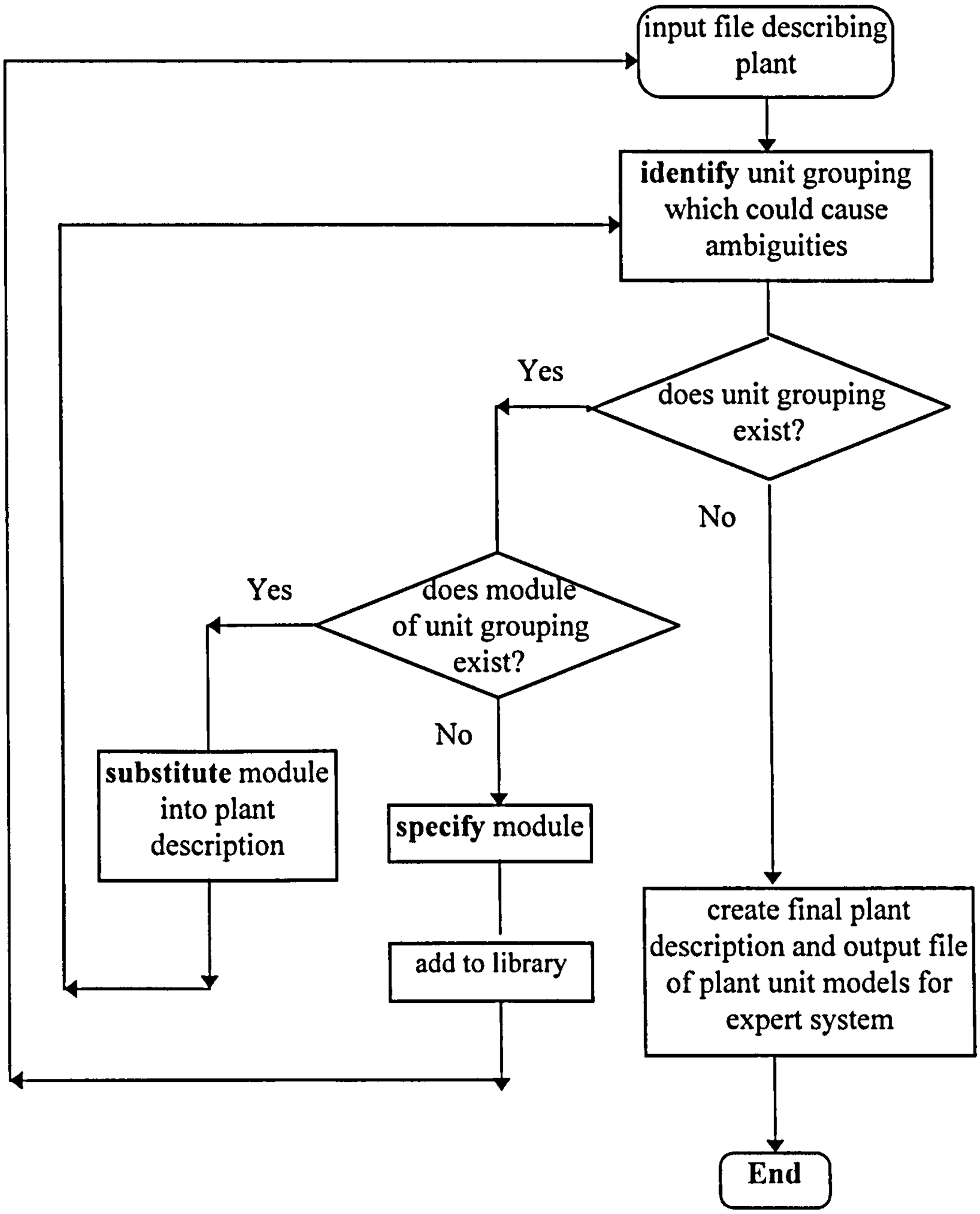


Figure 5.1 The Modular Approach

It can be seen from figure 5.1 that at the specification stage the modular approach is interrupted. This allows the user to verify the module specified using an external expert system (see section 6.2.1.1.). This also allows the user to view similar modules within the model library or to look at the module's constituent units. How Equipment Model Builder executes the three stages of the modular approach will be described. Problems encountered and the solutions employed are discussed.

5.1. Module Automatic Identification

The groupings of units which Equipment Model Builder automatically identifies as leading to ambiguities within qualitative modelling are different possible configurations of recycle loops and divider/header combinations. How some configurations may lead to ambiguities is already described in section 2.2. These groupings of units are automatically identified as they commonly occur within process plants. Other groupings of units which lead to ambiguities may occur. For these groupings of units the user is provided with the functionality to define modules. This is described in section 5.4.

The algorithm used by Equipment Model Builder to search for the unit groupings which could lead to ambiguities is shown in figure 5.2 (in text). Two searches of the plant description are made: one to identify divider/header combinations and one to identify recycle loops. Figure 5.1 shows how these identification procedures relate to the modular approach. The same search algorithm is applied to divider/header combinations and recycle loops but the two types of unit groupings are identified at different positions within the algorithm. Equipment Model Builder first searches for divider/header combinations. It is necessary to detect these combinations first before searching for recycle loops as the substitution of a module of a recycle loop into the plant system would cause any information about possible divider/header combinations located within the recycle loop to be lost. Figure 5.3 provides more detailed version of the search algorithm. The algorithm shown in figure 5.3 is repeated for all the source units.

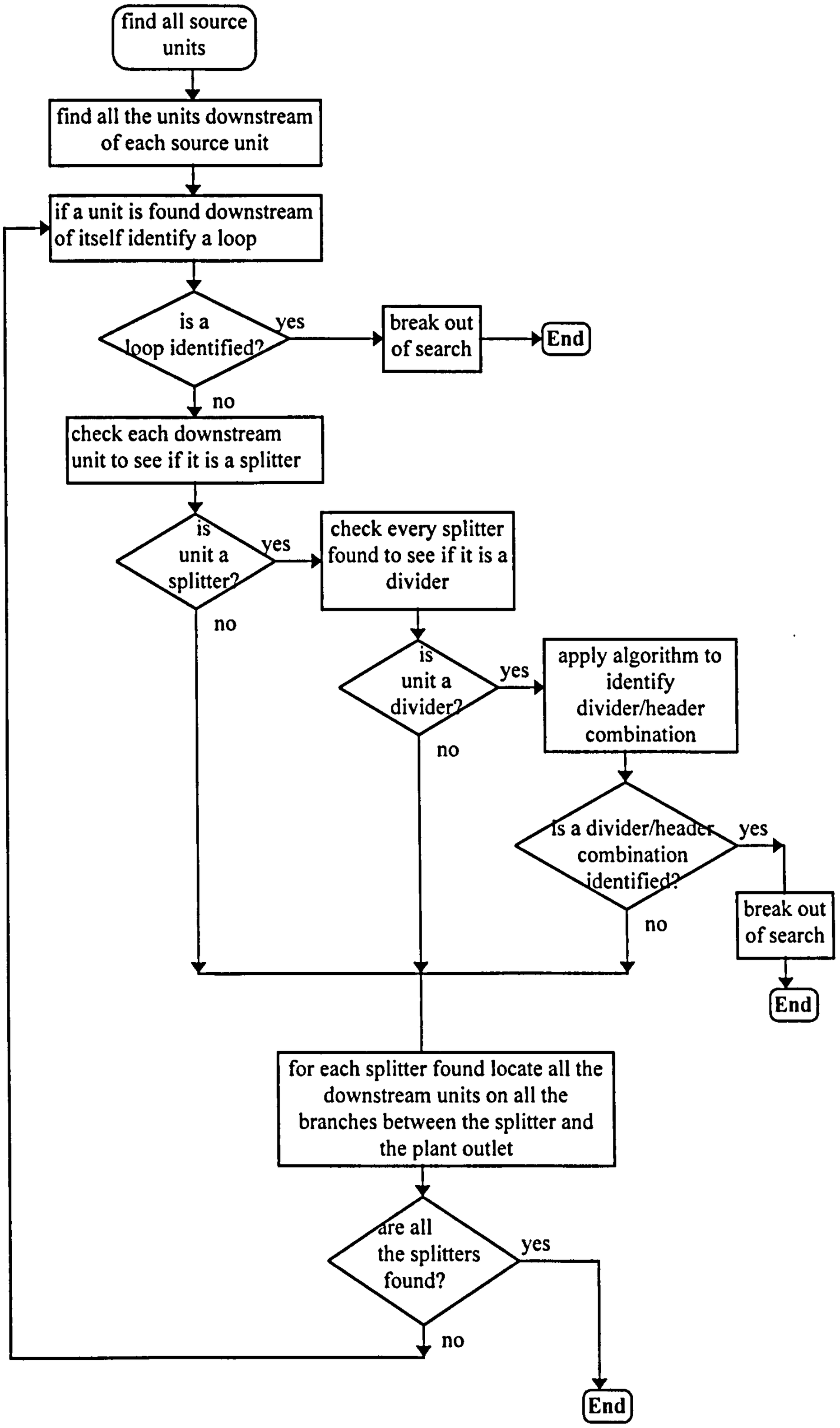


Figure 5.2 The Search Algorithm

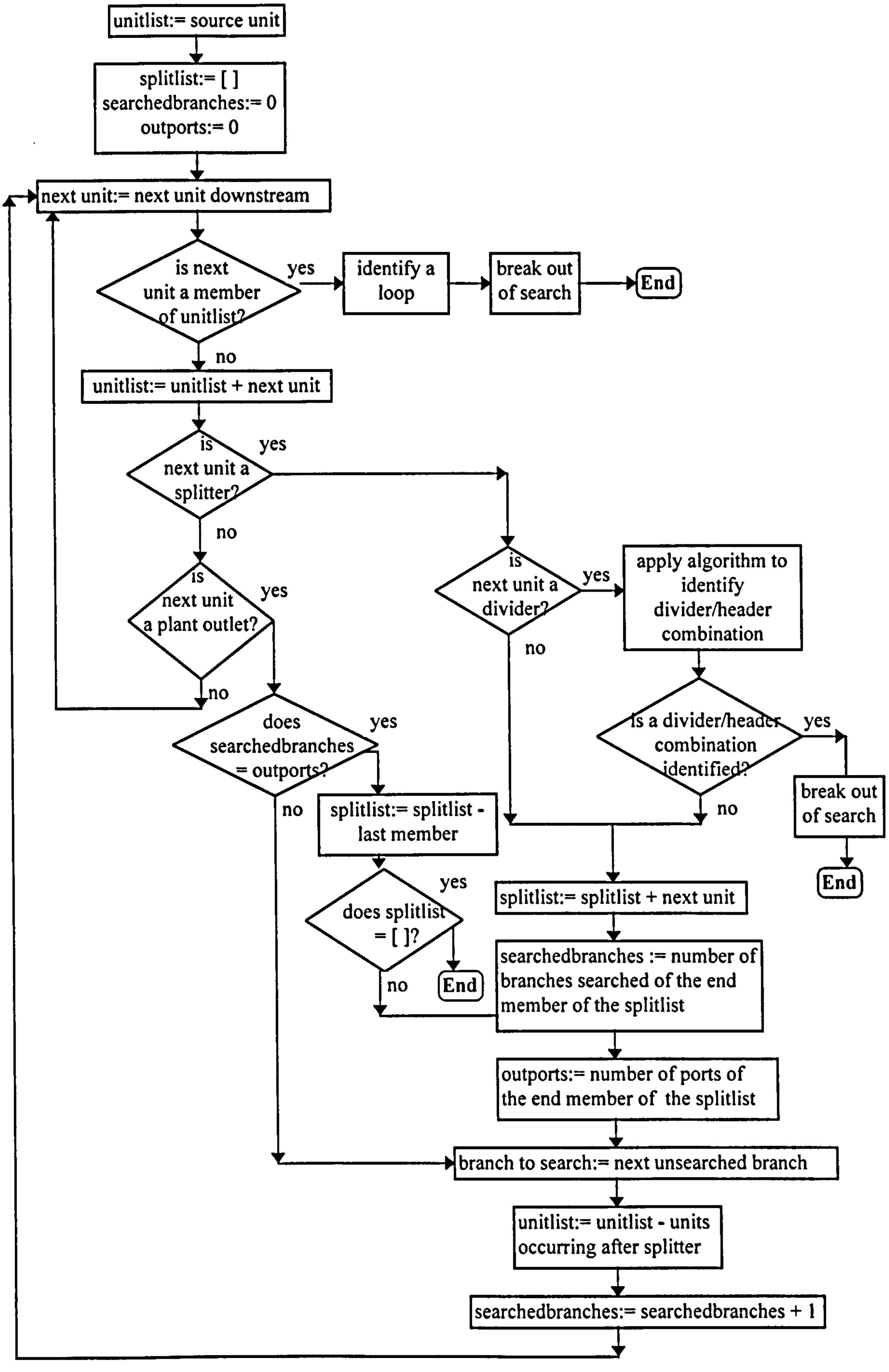


Figure 5.3 Search Algorithm shown in more detail

The search algorithm will be discussed. Problems encountered by the search algorithm and how these are overcome will be described. The methods Equipment Model Builder uses to automatically identify the groupings of units leading to ambiguities will be described briefly. Types of complex plant structures which can occur are discussed. How identification is achieved in the presence of these complex structures is detailed.

5.1.1. The Search Algorithm

Equipment Model Builder takes a file describing the units and the connections between them in a plant as input. The plant description is searched for groupings of units which could lead to ambiguities. The tool conducts a downstream search of the plant. Initially a source unit is searched for. The next unit to be searched for is the unit downstream of the source unit. This unit is found by searching for the unit linked to the outports of the source unit. The unit downstream of the unit linked to the outports of the source unit is searched for. The search continues in this manner until the plant outlet is reached.

It was found to be important to specify a source unit. If a source unit had not been specified it would be necessary to conduct a downstream search from every plant unit. This would be required so that no potential unit groupings which could lead to ambiguities would be missed. Every unit within a plant would have to be treated as a potential plant source. Specifying a source unit considerably reduces the amount of time needed to search a plant description.

In order for all the recycle loops and divider/header combinations within a plant to be located the plant description must be thoroughly searched. To ensure the whole plant is covered, for a plant containing several sources, multiple searches must be carried out initiating from each plant source. The units occurring downstream of every source unit in the plant description must be located. Equipment Model Builder finds and lists all the source units within a plant description. Multiple searches of the plant are made, starting at each source unit.

Whenever a unit grouping is identified which could lead to ambiguities the grouping is checked against the modules of the model library to see if it requires specifying. If it does not the module describing the unit grouping is substituted into the plant description and the search of the plant for more divider/header combinations and recycle loops recommences, starting at the plant sources again.

5.1.1.1. Splitters

The search of the plant is complicated by the presence of certain types of units which shall be called 'splitters'. These units have more than one outport which leads to branching of process flow stream through the plant. This causes branching of the search path. Examples of these types of units are tanks with multiple outports and columns. A column might have three outports: a top reflux outport, a bottom outport and an outport linked to a heating loop. To guarantee complete searching of a complex plant structure containing branches caused by the presence of splitters the following algorithm is used:

For every plant unit found Equipment Model Builder checks if it has multiple outports (i.e. if it is a splitter). If the unit does have multiple outports then the unit's identifier, the number of the unit's outports and the search branch taken is noted. When the plant outlet is reached the search returns to the last splitter located. A search is made down another branch from this splitter to the plant outlet. The search is repeated until all the branches found at the splitter have been searched. The number of branches found is equal to the number of outports at the splitter. The search then returns to the next to last splitter located and the same procedure is carried out, searching all the branches between the splitter and the plant outlet. The plant search continues until the number of branches searched is equal to the number of outports at each splitter multiplied by the number of splitters in the plant.

If a loop in the plant structure is located when a search is being made for divider/header combinations the plant search breaks and restarts at the last splitter found or finishes if there are no remaining splitters noted. This prevents the search from cycling infinitely around the loop. If a loop is found when a search is being made for recycle loops the user is informed that a loop has been detected and the specification and substitution stages commence.

A divider is a type of splitter. Each splitter found is checked to see if it is a divider. If it is then an algorithm which looks for divider/header combinations is utilised. If a divider/header combination is found a module is specified and substituted into the plant description. If no divider/header combination is found the divider is treated as splitter unit and both of its branches are searched using the plant search algorithm detailed above.

5.1.1.2. Heatexchangers

A class of plant units exist with multiple outports which do not divide the process flow stream within the plant so do not cause branching of the plant search. This class of units are heatexchangers. Although a heatexchanger has multiple outports the two process flow streams within the heatexchanger remain separate. The materials of the hot and cold stream do not mix. Only an exchange of energy occurs within a heatexchanger. There is no exchange of mass. This means that when a heatexchanger is located in a plant search it must not be treated as a splitter.

Heatexchangers are treated as a special case. Equipment Model Builder checks each unit found in a plant description to see if it is a heatexchanger. If this is so the tool backtracks to find the name of the inport of the heatexchanger linked to the process flow stream which the tool was searching. The tool must continue to search the plant description using this same stream as no joining or mixing of streams occurs in a heatexchanger. Equipment Model Builder tests the outports of the heatexchanger to find which one is located on the same flow stream as the inport found. The tool continues the plant description search down the branch linked to this outport. For example, assume Equipment Model Builder is searching a stream which forms the heated stream of a heatexchanger. Equipment Model Builder will find the heatexchanger. The tool will backtrack and discover that the inport of the heatexchanger linked to the stream being searched is 'heated in'. Equipment Model Builder will test the two outports of the heatexchanger. These are 'heated out' and 'cooled out'. 'heated out' is identified by name as being present on the same stream as the heated inport. Equipment Model Builder searches the plant description for a unit linked to the heated outport of the heatexchanger to find the plant branch linked to this outport.

5.1.2. Identifying the Unit Groupings Leading to Ambiguities

First, the algorithm used to identify divider/header combinations will be described. This algorithm is shown in figure 5.4 (in text).

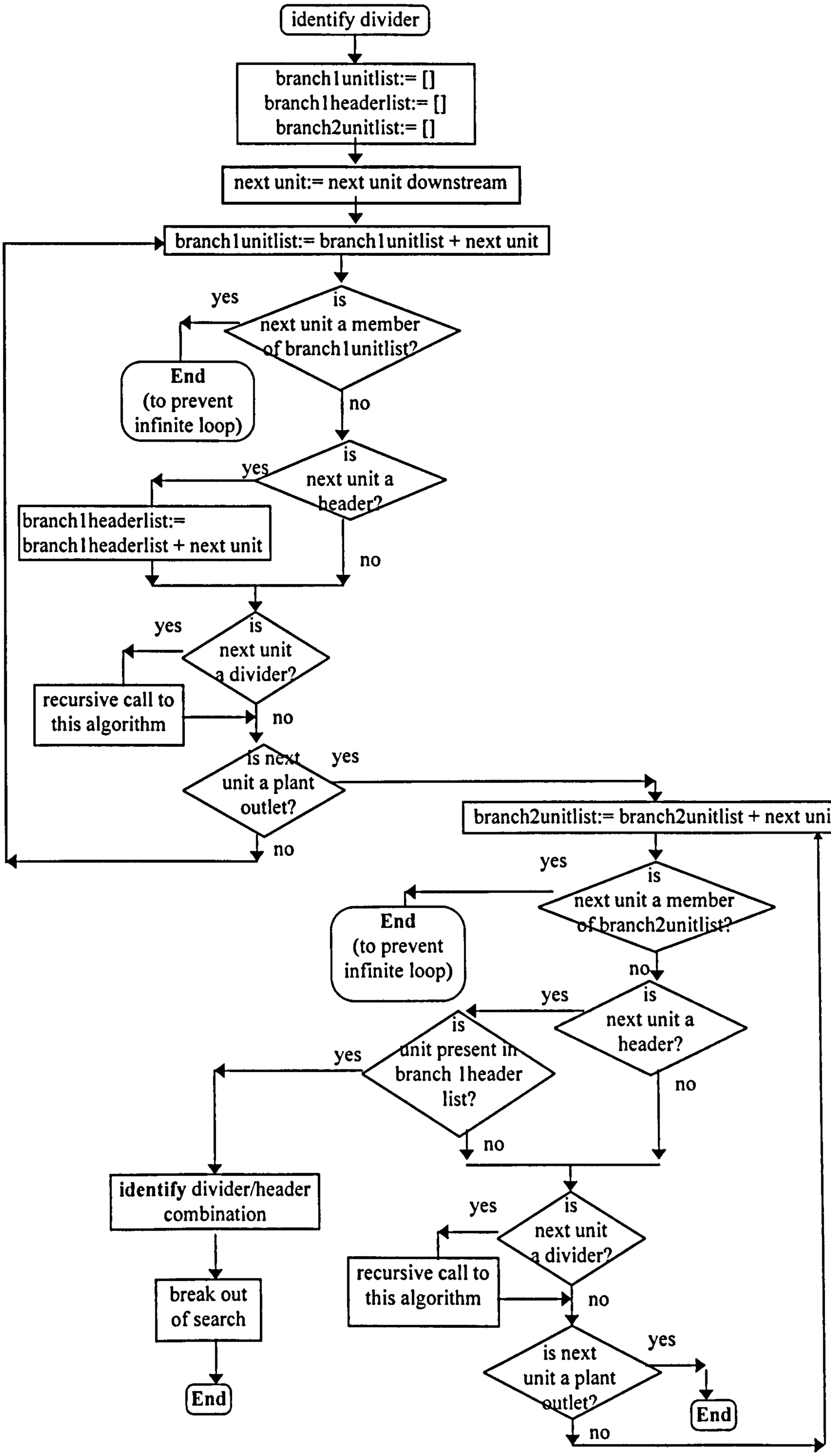


Figure 5.4 Algorithm to Identify A Divider/header Combination

Equipment Model Builder searches the plant description for a divider. If a divider is found units located downstream of branch 1 of the divider are stored. Any headers found among these units are noted. The search continues until the outlet of the plant is reached. Units located downstream of branch 2 of the divider are found and stored. If a header is found in branch 2 that is also in branch 1 then a divider/header combination has been found. The divider/header unit combination consists of the divider, the header and the stored units located between the divider and the header. If a downstream unit is found to be a divider this algorithm is repeated. This allows divider/header combinations located within divider/header combinations to be identified.

Each divider/header combination identified is specified and substituted into the plant description. Each modified plant description created after such a substitution is re-searched for further divider/header combinations. Equipment Model Builder then searches the plant description for recycle loops. The following algorithm is used to identify the simple plant loops:

> Within a plant description if a unit is found downstream of itself (a re-occurring unit) then that unit must be present within a loop. The units of the loop consist of the re-occurring unit and those intervening units which link the two occurrences of this unit.

Each plant loop found is specified and substituted into the plant description. Each modified plant description created after such a substitution is re-searched for further recycle loops. The plant description is searched until no more unit groupings which could lead to ambiguities are found and the plant outlet is reached.

Each grouping of plant units identified which could cause ambiguities is compared with the units of Equipment Model Builder's model library to see if a module for it already exists. If the units, attributes applied to the units and the connecting ports between the units of the plant grouping and a module in the library are the same then the two are identical. The instance names of the constituent units of the two need not be the same as identical structures consisting of different unit instances may occur in different plants or within the same plant. If no existing module is found for the unit grouping then a new one needs to be specified. If an existing unit is found the user is queried as to whether it should be substituted into the plant description.

5.1.3. Complex Plant Structures

This sub-section has discussed how plant groupings which could lead to ambiguities are identified. Problems discovered when searching a plant description for these unit groupings have been explained. How these problems are overcome has been described. The rest of this sub-section will look at the types of complex structures which could possibly occur in plants. How identification of unit groupings which could lead to ambiguities is carried out when these complex structures are encountered in a search by Equipment Model Builder of a plant description is detailed.

Within process plants in addition to simple recycle loops and divider/header combinations there is the possibility for complex structures consisting of multiple loops, multiple divider/header combinations and combinations of loops and divider/header combinations to occur. The following types of configuration will be considered:

(i) divider/header combinations in series;

(ii) a divider/header combination occurring within a divider/header combination;

(iii) loops in series;

(iv) a loop occurring within a loop;

(v) separate loops sharing one or several units;

(vi) a loop occurring within a divider/header combination;

(vii) a divider/header combination occurring within a loop.

5.1.3.1. Divider/header Combinations in Series

Divider/header combinations in series consist of two or more divider/header combinations. The outports of the header of the first combination in the series are linked to the inports of the divider of the second combination in the series by a plant unit or units.

Type (i) configurations are identified by multiple applications of the same method used to identify single occurrences of divider/header combinations. Each combination is identified separately.

5.1.3.2. Nested Divider/header Combinations

Type (ii) configurations consist of a divider/header combination located on a branch within an outer divider/header combination. Equipment Model Builder first identifies the inner divider/header combination. The inner combination is located first as the method to test for a divider/header combination is reapplied to every downstream unit found to be a divider. A module substitution is carried out, replacing the units of the inner unit

grouping. A search of the modified plant description created is made. This time the outer divider/header combination is detected. The module created for the inner divider/header combination is a member of the group of units comprising the outer divider/header combination. Another module substitution is carried out, replacing the units of the outer divider/header combination (including the inner divider/header module). The final result is that a module is created for the entire structure of a divider/header combination lying within another divider/header combination. To avoid loss of information about the unit grouping which comprises the inner module, the inner module must be identified and substituted into the plant description before the outer divider/header combination is detected.

5.1.3.3. Loops in Series

Loops in series consist of two or more simple loops. The outports of the first loop in the series are linked to the inports of the second loop in the series by a plant unit or units. The outports of the second loop may be linked to the inports of a further loop, etc.

Type (iii) loop configurations are identified by multiple applications of the same algorithm used to identify simple loop occurrences. Each loop is identified separately.

5.1.3.4. Nested Loops

Type (iv) configurations consist of a loop whose inports and outports are joined to units which form part of a larger loop. The two loops possess a certain number of units in common. Equipment Model Builder first identifies the inner loop and carries out a module substitution, replacing the units of the inner loop. A search of the modified plant description created is made and the outer loop is identified. The module created for the inner loop is a member of the group of units comprising the outer loop. Another module substitution is carried out, replacing the units of the outer loop (including the inner loop module). The final result is that a module is created for the entire structure of a loop lying within another loop.

No algorithm exists to ensure that the group of unit comprising the 'inner' module of the loop is identified before the 'outer' module is found. This because for a loop within loop configuration the concept of 'inner' and 'outer' loops is relative. The two are interchangeable and which unit grouping is identified as the inner module depends only on the order in which the units were specified in the plant description. However once the unit grouping which comprises the inner module has been identified it is necessary to carry out

a substitution before the outer loop module is detected to avoid loss of information about the units which comprise the inner loop module.

5.1.3.5. Loops Sharing Units

Type (v) configurations consist of two separate loops which share one or more units, i.e. the loops are joined in parallel. A common example of a type (v) configuration is a continuous distillation column system. This consists of a column joined to two separate loops. One loop consists of a heating cycle at the base of the column. The other is a reflux cycle at the top. This types of configuration may be regarded as a specialised kind of type (iv) configuration. A type (v) configuration is type (iv) configuration in which the inner and outer modules share only a small number of units.

The tool identifies an initial loop. This loop contains the unit(s) common to both loops. The module for this initial loop is substituted into the plant description, replacing the units of the initial loop. The modified plant description created is searched and a second loop is identified. This second loop may be regarded as an outer loop of the kind in type (iv) configurations as the initial loop module is a member of the group of units which comprises the second loop. Another module substitution is carried out replacing the units of the second loop. The final result is to create a module for the entire configuration of two separate but joined loops.

Taking the continuous distillation column system as an example, assume the initial loop identified is the reflux cycle. This loop will contain the distillation column along with the other units of the reflux cycle. The module for this loop is substituted into the plant description. A search is made of the modified plant description and another loop, the heating cycle, is identified. This loop may be regarded as an outer loop as amongst the group of units that comprise it, it will contain the recycle module. Module substitution is carried out, replacing the units of the heating cycle (including the recycle loop module). The final effect is to create a module for the continuous distillation column system.

5.1.3.6. Loops Nested in Divider/header Combinations

Type (vi) configurations are formed by a loop whose inports and outports are linked to units which form a branch of a divider/header combination.

For type (vi) configurations Equipment Model Builder identifies the inner loop using the simple loop algorithm. The existence of an outer divider/header combination is not detected. It was not thought likely that these types of configurations would occur in

process plants. Should they occur they may not give rise to ambiguities like the simple divider/header combinations. Not identifying the outer divider/header combination in this type of configuration was not judged to be important.

5.1.3.7. Divider/header Combinations Nested in Loops

Type (vii) configurations consist of a divider/header combination: the inports of the divider of this combination and the outports of the header are linked to units which form part of a loop. An example of this is shown in benzene purification system case study where a divider/header combination to provide a spare pump is located within a reflux loop (see section 7.1.1.).

For type (vii) configurations Equipment Model Builder will first identify the inner divider/header combination. As Equipment Model Builder searches for divider/header combinations before commencing searching for recycle loops this ensures that in this type of configuration the inner divider/header combination is always identified before the outer loop grouping in which it is located. A module substitution is performed, replacing the units of the inner divider/header combination. A search of the modified plant description created is made and the outer loop is identified. The module created for the inner divider/header combination is a member of the group of units comprising the outer loop. Another module substitution is carried out, replacing the units of the outer loop (including the inner divider/header module). The final result is that a module is created for the entire structure of a divider/header combination lying within an outer loop. As with the other configurations containing modules lying within outer modules the inner module must be identified and substituted into the plant description before the outer module is detected to avoid loss of information about the units comprising the grouping of the inner module.

5.2. Module Specification

If there is no existing module within the model library which describes the unit group identified a new one needs to be specified. Modules are specified for the plant fragments identified by amalgamating and modifying existing models of plant units. Modules for unit groupings comprising recycle loops and divider/header combinations are specified in exactly the same way. An interface exists to allow an expert to add extra information to the modules. This interface provides a guide as to what extra arcs are required. The amalgamated units models together with the interface serve as a template to allow the user

to define the module identified. After it has been defined the module is added to a library of component models.

When the units are amalgamated to form a module the causal paths that lead to ambiguities or incorrect behaviour are removed. Internal influences of the module (i.e. influences between or within units found within the plant fragment) are retained. It is important to model the influence of faults if these lead to important consequences within the module or propagate out of the module to cause consequences in upstream or downstream units.

Consider the simple plant loop shown in figure 5.5 below. A partial SDG built up from unit SDGs is shown for the plant loop in figure 5.6. Only arcs for the process variables 'flow' and 'level', two initiating faults and one consequence are shown.

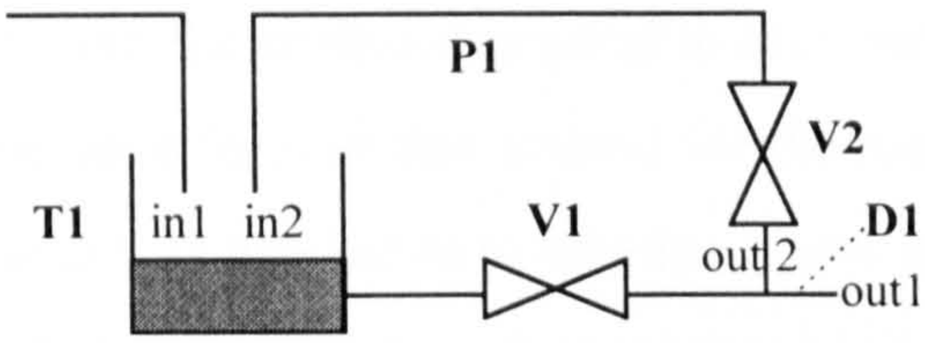


Figure 5.5 A Simple Plant Loop

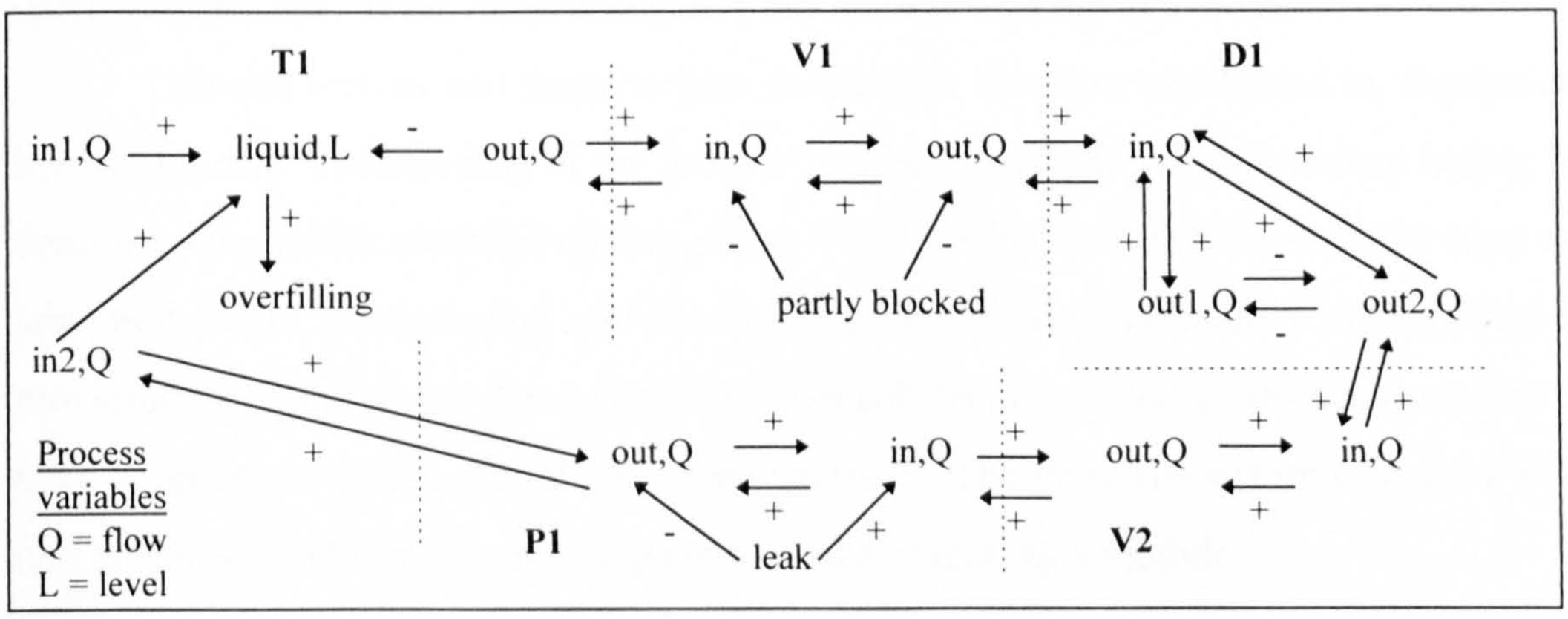


Figure 5.6 Partial Signed Directed Graph for a Simple Plant Loop

Figure 5.7 shows a module of the plant fragment shown in figure 5.5. The module SDG is significantly simpler than the equivalent unit-based SDG in figure 5.6.

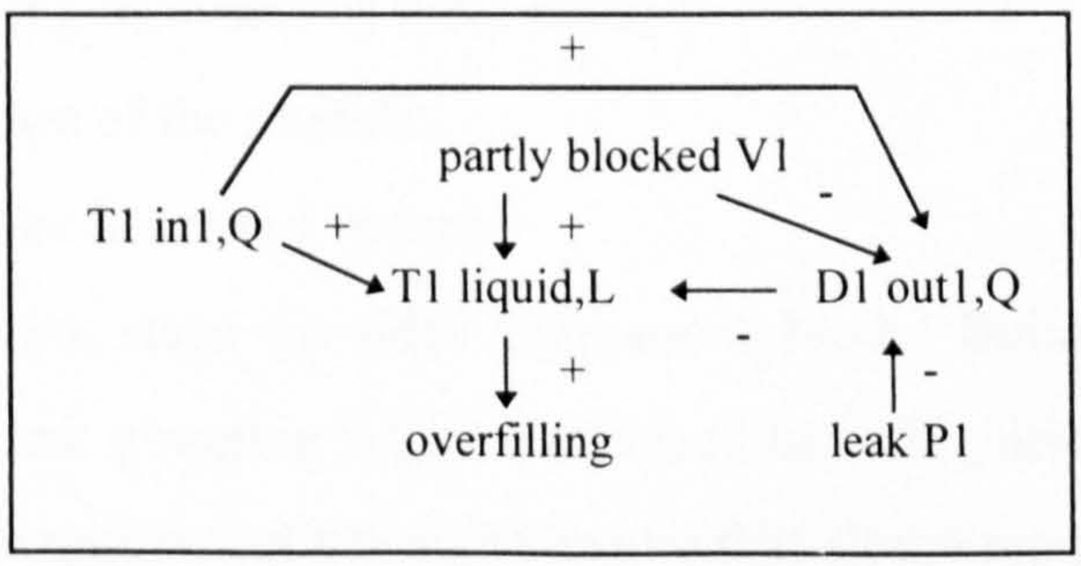


Figure 5.7 A Module of a Simple Plant Loop

When the plant loop is made into a module some of the propagation paths between the units of the module are removed. Propagation paths through the whole module are modelled. In this example the influence of 'flow' through the port 'T1 in1' on 'flow' through 'D1 out 1' is modelled. Important internal influences are retained. The influence of the faults 'partial blockage of V1' and 'leak of P1' are important. The 'partial blockage of V1' may cause the level of T1 to rise sufficiently to lead to 'overfilling' as a consequence. The influence of the 'leak of P1' propagates out of the module through the port 'D1 out1'. This influence may cause consequences in downstream units.

Before specification begins a description of the unit grouping identified as leading to ambiguities is presented to the user. This description consists of the identifiers of the units composing the plant fragment identified, the ports of the units connecting the units within the plant fragment and the attributes applied to each unit within the plant fragment. This description is of the same form as that created for the description slot for the module template (see later). The user is queried as to whether a new module should be created for this unit grouping. If the user agrees Equipment Model Builder commences the specification stage. If the reply is negative the session with the tool is ended.

This sub-section will describe how the module template is provided by Equipment Model Builder. The building of the module will be described. The sub-section begins by describing the initial module building phase. Two further sub-sections consider how the unit models are amalgamated and the interface provided. Difficulties encountered in providing the template are detailed within these sub-sections. Unit groupings can occur in plant descriptions which should not be modularised. The final sub-section discusses these unit groupings and how the user is given a choice of making a module.

5.2.1. Initial Module Building

The initial module building phase consists of:

- checking that all the units which comprise the module identified are modelled;
- creating and labelling instances of these units;
- creating a description of the module.

These procedures will be described in turn.

The identification stage provides Equipment Model Builder with a list of units which comprises the unit grouping which could lead to ambiguities. The tool checks the list of units against the unit model library to ensure that all the models listed are present in the library. If a unit model is found to be missing Equipment Model Builder 'asks' the

user whether a model for the missing unit should be created. If the user confirms this the model building interface is presented. If the reply is negative the session with the tool is ended.

5.2.1.1. Labelling the Units of the Module

Having ensured that all the module's constituent units are present an instance for each unit occurrence is created from its unit model. The tool labels the arcs of each of these individual instances with each instance's identifier taken from the plant description file. For example, assume that the part of a file describing the plant generated by AutoCAD given below has been identified as comprising a divider/header combination.

'D1', 'divider', '[out1 is [V1,in],out2 is [V2,in]]', 'purpose is toBypass'
'V1', 'valve', '[out is [H1,in1]]', 'aperture is open'
'V2', 'valve', '[out is [H1,in2]]', 'aperture is closed'
'H1', 'header', '[out is [tail1,in]]', 'purpose is fromBypass'

This combination is illustrated in figure 5.8 below.

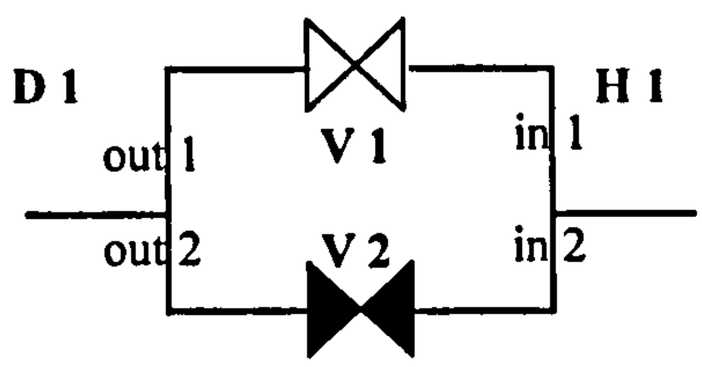


Figure 5.8 A divider/header combination

See section 2.4.2. for an explanation of files describing the plant generated by AutoCAD. The unit instances comprising the module are D1, V1, V2 and H1. Assume that the divider unit model contains the following among its arcs:

([fault,leak to environment],+,[in,flow])
([fault,leak to environment],+,[consequence,contaminate environment])
([deviation,morePressure,in],+,[consequence,possible rupture])

The arcs shown of the divider unit will be labelled as:

([fault,D1 leak to environment],+,[D1_in,flow])
([fault,D1 leak to environment],+,[consequence,contaminate environment])
([deviation,morePressure,D1_in],+,[consequence,possible rupture])

Figure 5.9. (in text) shows part of the template provided by the tool for the divider/header combination in figure 5.8.

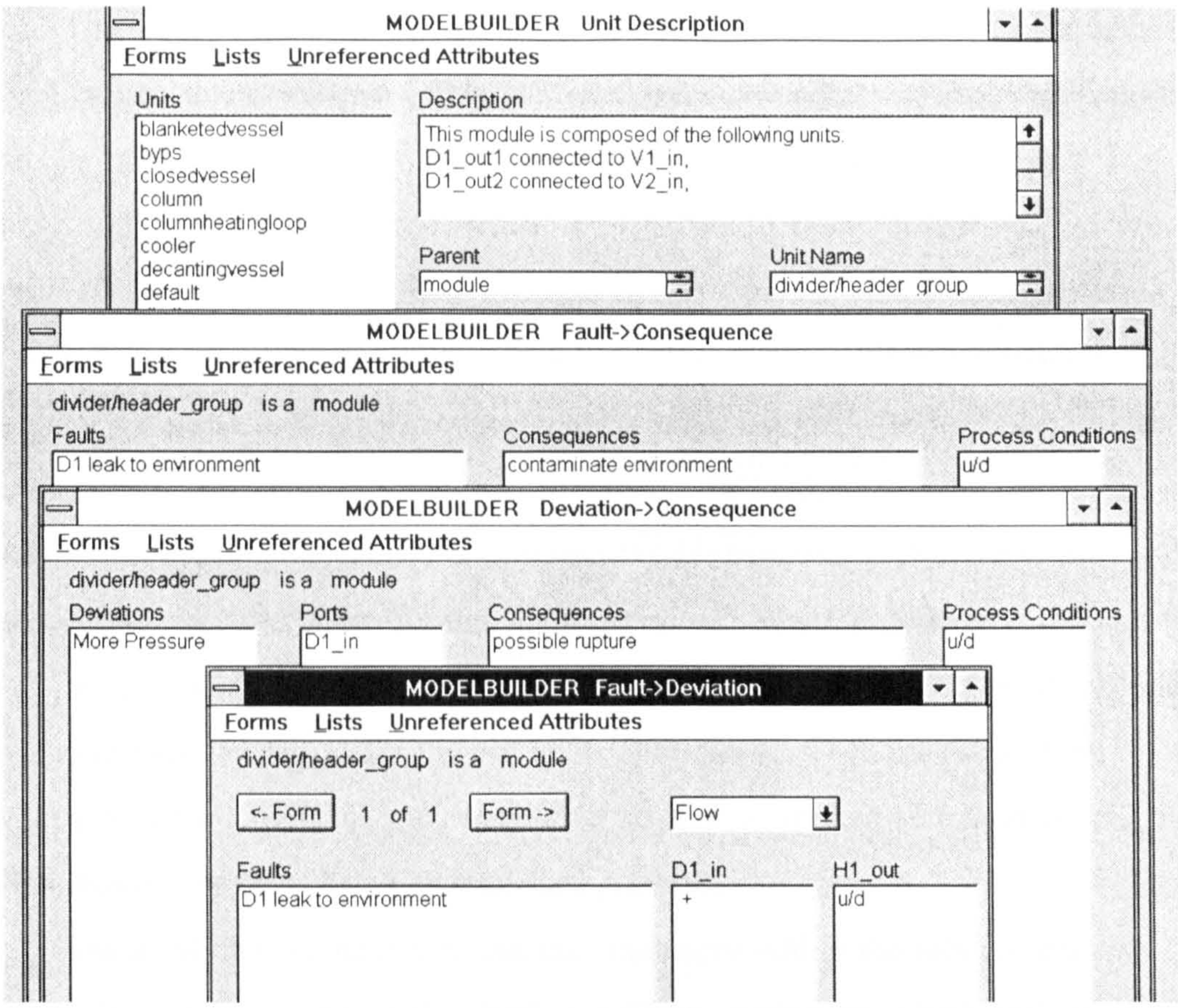


Figure 5.9. Partial Template for a divider/header combination

It can be seen that the faults and the ports of the individual unit arcs are labelled. This is so that the faults and ports of the constituent units of a module may be recognised when the module is placed within a plant description. For example, it might be important to know that the divider of the module shown in figure 5.8 has a 'leak to environment'. If this fault is applied to the whole module it is too broad to be of use. It is important to know which unit within the module is leaking.

The divider of this module has a port 'in'. Other units occurring within the module may have identical port names. For example, the valve units may also have a port 'in'. It is necessary to label the ports to distinguish which units of the module the deviations and faults have an effect upon. For instance, it may be important to know that the fault 'D1 leak to environment' has an effect on the deviation 'flow' at the port 'in' of the divider 'D1' (see arc 1 in the arc series shown above). When more than one unit of the same type occurs within a module this labelling of the ports allows it to be seen which deviations are applicable to the individual unit instances. For example, two instances of a valve unit occur in the unit grouping shown above. The inports of these instances would be labelled 'V1_in' and 'V2_in' so as to distinguish them for the two valve unit instances.

The consequences of the individual unit arcs are not labelled with the identifiers of the individual unit instances. This prevents multiple repeats of the same consequence when the module is placed within a plant description. The next example shows what would result if the consequences were to be labelled. Assume all the units of the module shown in figure . 5.8 contain the arc '([fault,leak to environment],+,[consequence,contaminate environment])'. When the module was placed in a plant description if all the consequences were labelled the following consequences would occur: 'D1 contaminate environment', 'V1 contaminate environment', 'V2 contaminate environment' and 'H1 contaminate environment'. If several units containing identical consequences were present in a group of units comprising a module the user would be overwhelmed with the volume of output. Simply having the consequence 'contaminate environment' for the entire module removes needless repetition. If the user specifically wishes to label a consequence so that it applies to a certain unit within a module this can be done using the interface provided.

The attributes applicable to the unit instances within the module are also labelled, so as to be able to distinguish which condition applies to which unit instance. For example, the following attributes would be created for the unit grouping shown in figure 5.8: 'D1_purpose is toBypass', 'V1_aperture is open', 'V2_aperture is closed' and 'H1_purpose is fromBypass'.

When a module is substituted it is given an instance identifier by Equipment Model Builder (see section 5.3.1.). When an inner module is found to form one of the constituent units of an outer module this inner module will possess an identifier. Configurations of this kind are described in section 5.1.3. In these types of configurations when the arcs of the units comprising the outer module are labelled the arcs of the inner module are not labelled with the inner module's instance identifier. For example, if a divider/header combination occurred within an outer loop module the divider/header module might be allotted the identifier 'divhead1'. An arc '([fault,D1 leak to environment],+,[D1_in,flow])' occurring in the inner divider/header module would not be labelled '([fault,divhead1 D1 leak to environment],+,[divhead1_D1_in,flow])' when the arcs of the individual units of the outer module were labelled but would remain in its original form. This avoids having very complicated identifiers. An additional label is not required as the user is able to determine which unit is being described.

5.2.1.2. Creating a Module Description

A description is created for the module and placed within the description slot of the template defining the module. The description is created in order to inform the user of the module structure. For example, the following description would be created for the module shown in figure 5.8.

> 'This module is composed of the following units:
> D1_out1 connected to V1_in,
> D1_out2 connected to V2_in,
> V1_out connected to H1_in1,
> V2_out connected to H1_in2.
> Conditions of the module are:
> D1_purpose is toBypass,
> V1_aperture is open,
> H1_purpose is fromBypass
> and this completes the module.'

The attributes (conditions) of the individual units of the module are given so that the user is able to distinguish between modules which appear identical in structure but which have different attributes applicable to their constituent units. The descriptions of the units which comprise the module are not included in the module description as these may be found by viewing the individual unit models. The name 'module' is added to the module template's class slot.

5.2.2. Amalgamating Unit Instances

How the unit model instances of the plant fragment identified are amalgamated to form a module template will now be described. As shown in figure 5.7 when the units are amalgamated causal paths existing between or within the units of the module are removed. This is to prevent ambiguities arising. Important internal influences within the module are retained. More detail as to precisely which arcs are kept to retain the important internal influences when a module is created will now be given. In order to determine which arcs of the constituent unit model instances should be retained it is necessary to discover the boundary ports of the module. Finding the ports of the module will be discussed. What 'unlinked module faults' are and how they arise when the unit model instances are amalgamated is described.

5.2.2.1. Arcs Retained

A unit model has four types of arcs:

(i) deviation linked to deviation;

(ii) fault linked to deviation;

(iii) fault linked to consequence;

(iv) deviation linked to consequence.

When the module is created the fault linked to consequence arcs and deviation linked to consequence arcs of the constituent unit model instances are retained. It is important to model the effect of the modules consequences upon a plant. The modules consequences are derived from the arcs of its constituent units.

Fault linked to deviation arcs of the constituent units in which the deviation occurs at a boundary port of the module are retained. This is because the faults within these arc types have the potential to propagate out of the module through its boundary ports to influence units upstream or downstream of the module when the module is present in a plant description. It is necessary to model the effect of these faults.

Fault linked to deviation arcs of the units in which the deviation affects a port which is linked to a consequence are also retained. The port may be linked to a consequence within a deviation linked to consequence arc. This so that the influence of the fault leading to the consequence within the module may be modelled.

Deviation linked to deviation arcs of the units in which the initiating deviation occurs at a boundary port of the module and the caused deviation affects a port which is linked to a consequence are kept too. This is so that potential deviations propagating into the module via its boundary ports to cause consequences may be modelled. These potential deviations may propagate from other plant units when the module is placed within a plant description.

The retention of arcs described enables important internal influences within the module to be described. All the other arcs of the constituent unit model instances are deleted upon making a module to remove multiple causal paths which have the potential to cause ambiguities. The retained arcs of all the unit instances comprising the module are added together and with the interface provided form the module template. An example will be given to illustrate which arcs are retained when the unit model instances of the plant fragment are amalgamated to form a module.

For simplicity, only one model instance of a unit grouping comprising a module will be considered. Assume the module being created is that composed of the plant fragment shown in figure 5.5. The unit to be considered is the tank, instance T1. Assume that the tank model contains the following arcs:

```
arc([in1,flow],+,[liquid,level]),
arc([in2,flow],+,[liquid,level]),
arc([out1,flow],-,[liquid,level]),
```

```
arc([fault,'leak to environment'],+,[in1,flow]),
arc([fault,'leak to environment'],-,[liquid,level]),
arc([fault,'leak to environment'],-,[out1,flow]),
arc([fault,'outlet1 partly blocked'],-,[out1,flow]),

arc([fault,'leak to environment'],+,[consequence,'contaminate environment']),

arc([deviation,[moreLevel,liquid]],+,[consequence,'overfilling']),
arc([deviation,[moreFlow,in1]],+,[consequence,'incomplete separation of water']),
```

When the simple loop module shown in figure 5.5 is constructed instances will be created for each of its constituent units. Among these a tank model will be labelled to form the unit instance T1. The module created will have the inports 'T1_in1' and the outport 'D1_out1'. When arcs of the unit instances constituting the simple loop module are amalgamated the following arcs of instance T1 will be retained within the module template created:

```
arc([T1_in1,flow],+,[T1_liquid,level]),

arc([fault,'T1 leak to environment'],+,[T1_in1,flow]),
arc([fault,'T1 leak to environment'],-,[T1_liquid,level]),

arc([fault,'T1 leak to environment'],+,[consequence,'contaminate environment']),

arc([deviation,[moreLevel,T1_liquid]],+,[consequence,'overfilling']),
arc([deviation,[moreFlow,T1_in1]],+,[consequence,'incomplete separation of water']),
```

Example 5.1 Arcs Present due to Instance T1

The fault linked to consequence arc '([fault,'T1 leak to environment'],1,[consequence,'contaminate environment'])' and the deviation linked to consequence arcs '([deviation,[moreLevel,T1_liquid]],1,[consequence,'overfilling'])' and '([deviation,[moreFlow,T1_in1]],+,[consequence,'incomplete separation of water'])' are retained within the module template. The arc '([fault,'T1 leak to environment'],+,[T1_in1,flow])' is retained as this is a fault linked to deviation arc in which the deviation occurs at a boundary port of the module (inport 'T1_in'). The arc '([fault,'T1 leak to environment'],-,[T1_liquid,level])' is kept as this is a fault linked to deviation arc in which the deviation affects a port which has a consequence. The consequence in this example is 'overfilling'. This consequence is linked to the deviation in the process variable 'level' at the port 'T1_liquid' in the retained deviation linked to consequence arc '([deviation,[moreLevel,T1_liquid]],1,[consequence,'overfilling'])'. The arc '([T1_in1,flow],+,[T1_liquid,level])' is retained as it is a deviation linked to deviation arc in which the initiating deviation occurs at a boundary port of the module and the caused deviation affects a port which is linked to a consequence. The consequence in this

case is also 'overfilling'. The arcs '([T1_in2,flow],+,[T1_liquid,level])', '([T1_out1,flow],+,[T1_liquid,level])', '([fault,'T1 leak to environment'],-,[T1_out1,flow])', and '([fault,'T1 outlet1 partly blocked'],-,[T1_out1,flow])' are deleted to remove multiple causal paths within the module which could result in ambiguities.

5.2.2.2. Identifying Module Ports

Equipment Model Builder identifies the boundary ports of the module from the information contained in the unit model instances and from the information contained in the plant description file about the connections between the units that compose the module. Only the inports and outports of the constituent units model instances have the potential to become inports and outports of the module. The unitports do not. The tool finds the all possible inports and outports of the module from information contained in the unit model instances. Inports and outports of the unit instances which are used to connect the units together to form the module cannot form the inports and outports of the module. The tool finds the information about the connecting ports from the plant description file. The inports and outports of the module are calculated by removing those inports and outports which form inter-unit connections within the module from the list of all the possible in- and outports of the module.

An example will be given to illustrate this. Consider the unit grouping comprising a module shown in figure 5.8. Assume that a divider model has an inport 'in', a valve model has a inport 'in' and header model has two inports 'in1' and 'in2'. Potential inports for the module found in the unit instances are 'D1_in', 'V1_in', 'V2_in', 'H1_in1' and 'H1_in2'. The partial plant description given (in section 5.2.1.1.) shows that 'V1_in', 'V2_in', 'H1_in1' and 'H1_in2' are used to form inter-unit connections within the module. Thus it can be deduced that 'D1_in' forms the inport for the module. Similar reasoning is utilised to derive the outports for the module.

The unitports of the module are formed by the ports of the module's constituent units which have consequences but which do not form the boundary ports of the module. It is necessary to retain these ports as they have consequences. Take the simple unit loop module shown in figure 5.5 as an example. See example 5.1 for the arcs present in the module template due to the unit instance T1. This module would have 'T1_liquid' as a unitport. This is because 'T1_liquid' is linked to a consequence in the arc '([deviation,[moreLevel,T1_liquid]],1,[consequence,'overfilling'])'. The arc

‘([deviation,[moreFlow,T1_in1]],+,[consequence,'incomplete separation of water'])’ does not result in a unitport ‘T1_in1’ as ‘T1_in1’ forms an inport of the module.

5.2.2.3. Unlinked Module Faults

When the unit instances are amalgamated the fault linked to deviation arcs in which the deviation does not occur at a boundary port or in which the deviation does not affect a port which has a consequence are deleted. For example, the arc ‘([fault,'T1 outlet1 partly blocked'],-,[T1_out1,flow])’ is deleted from the instance T1 when it is placed within the simple loop module (see section 5.2.2.1.). These arcs are deleted to remove the multiple causal paths which could lead to ambiguities. However the faults of these arcs must be retained as they may have an important effect within the module. They may also propagate out of the module to cause deviations in other units when the module is placed in a plant description. The final effect of these faults is not known. The effect cannot be determined unambiguously. This is why the arcs of this kind are deleted. The faults from the deleted arcs are stored by Equipment Model Builder as ‘unlinked module faults’.

Faults of deleted arcs are only stored as unlinked module faults if they are not already present within any of the modules arcs. For example the arcs ‘([fault,'T1 leak to environment'],-,[T1_out1,flow])’ and ‘([fault,'T1 outlet1 partly blocked'],-,[T1_out1,flow])’ are deleted from the instance T1 when it is placed within the simple loop module. Only 'T1 outlet1 partly blocked' is stored as an unlinked module fault. The fault fault,'T1 leak to environment' is not stored as it is also present within other arcs within the module. These arcs are ‘([fault,'T1 leak to environment'],+,[T1_in1,flow])’, ‘([fault,'T1 leak to environment'],-,[T1_liquid,level])’ and ‘([fault,'T1 leak to environment'],+,[consequence,'contaminate environment'])’. Unlinked module faults are faults with unknown effects upon the module. The effects of faults of deleted arcs which are also present within the existing arcs of the modules are known. It would be confusing to classify a fault as unlinked if it is causing an effect within the module. A fault of a deleted arc may have been causing a different effect to a duplicate fault present within the existing arcs of the module. This additional effect is unknown. The user interface provided allows more information about the faults to be added to the module.

5.2.3. Module Interface

How the module is initially built up and how its constituent units are combined have been described. The interface provided to allow the user to add further information to the

module will now be considered. The amalgamated arcs of the unit instances comprising the plant fragment and a user interface together make up the template for defining the module. A user interface is required as further user input to the module may be needed to complete the module. Unlinked module faults require consideration. Propagation paths through the module as a whole may need to be added as the causal paths between the constituent units have been removed. These paths were deleted to remove the multiple causal paths which may lead to ambiguities. The user may wish to alter the arcs of the amalgamated unit instances. Some of the arcs may be found to be incorrect or irrelevant to the module.

This interface is superficially similar to the basic interface provided to build unit models. Like the interface for building unit models it consists of a main window and four sub-windows, one for each arc type (see chapter 4). The user is able to add a name to the module. There is a 'Save' button so that the user may add the module to the unit model library. Verification checks are provided (see chapter 6). These are important as new errors may arise when arcs are deleted when the module is formed. The user may add, alter or delete the arcs of the module.

A modification and some additions have been made to the basic interface to form the module building interface. The modification prevents the user from utilising the module interface to create internal propagation paths within the module. These paths are not permitted as they may form the multiple causal paths which cause ambiguities. Additions have been made to manage unlinked module faults. These changes will be described.

5.2.3.1. Preventing Internal Propagation Paths

To prevent internal propagation paths the sub-window for creating deviation linked to deviation arcs is modified. The sub-window provided by the basic interface for building unit models presents a grid. The ports of the unit model form the first row and first column of the grid. The user is able to make a deviation linked to deviation arc by linking a member of the first column to a member of the first row by clicking on an element of the grid with the mouse. This brings up a menu choice of possible influences for the SDG arc being created. The user may replace the default grid element 'u/d' with a member of this menu. This allows the effect of a deviation at a port of the model on a deviation at another port of the model to be modelled, forming a deviation linked to deviation arc. See chapter 4 for a fuller explanation of this.

The deviation linked to deviation sub-window of the module interface presents a modified grid which prevents the user from adding arcs forming internal propagation paths within the module. The first row of the grid remains of the same form on the *module* building interface as on the basic *unit model* building interface. The first row consists of all the ports of the module. The first column of the grid is modified. On the module building interface it consists only of the boundary ports of the module. This interface allows the user to model propagation paths through the whole module, i.e. boundary port deviations linked to boundary port deviations. It also allows the user to model the effect of boundary port deviations on unitports. It is important to model these effects if consequences result from unitport deviations. The interface prevents the user from creating internal influences within the module. This prevents the formation of multiple causal paths.

For example, assume the user was creating propagation paths for the simple loop shown in figure 5.5. The user would be able to add the arc '([T1_in1,flow],+,[D1_out1,flow])' to the module interface. This arc models a boundary port deviation propagating to cause a deviation at another boundary port within the module. The arcs '([T1_in1,flow],+,[T1_liquid,level])', and '([D1_out1,flow],+,[T1_liquid,level])' could be added to the module interface to model boundary port deviations propagating to cause unitports deviations within the module. The user would be unable to create the arc '([T1_liquid,level],+,[D1_out1,flow])' as the unitport 'level' is not present in the first column of the deviation linked to deviation sub-window of the module interface. This arc may not be created as it models an internal influence within the module. This internal influence could form a link in multiple causal paths causing ambiguities. The arcs described here which the user can add to the interface may be seen within the partial SDG for the simple loop module shown in figure 5.7.

5.2.3.2. Managing Unlinked Module Faults

There are two possible ways for the user interface to manage unlinked module faults. One way would be to add the unlinked faults to the fault linked to deviation and fault linked to consequence sub-windows to form partial arcs for the user either to complete or delete. Each fault would be added to a form for each of the process variables (pressure, temperature etc.) on the fault linked to deviation sub-window. This method would result in a large number of arcs for the user to check through. The unlinked module faults may

only be applicable to fault linked to deviation arcs and not to fault linked to consequence arcs as well. It is unlikely that the faults will be applicable to all of the process variables.

The method utilised is to add all of the modules' faults to the fault library. This comprises both unlinked module faults and those faults present within the modules' arcs. The conditions of the module are also added to the end of the process condition library. This allows the user freedom to create further arcs for the module. The user is able to build arcs for the module in the same way in which arcs are constructed for unit models (see section 4.3.). The members of the fault and condition libraries are shown in the list browsers accessed via the drop-down menus in capital letters. The faults and conditions of the module are shown in small letters so as to distinguish them from the library members. When a module creation session is ended the module's faults and conditions are removed from the end of the libraries.

A list of unlinked module faults is provided which may be accessed from a menu. This enables the user to see which of the modules faults are unlinked. This list is updated when the user adds arcs containing faults to or deletes arcs containing faults from the module. The user may delete unlinked faults from this list as some faults of the constituent units may not be relevant to the module created. The deletion of these faults also deletes them from the fault library. If the user attempts to access the list of unlinked module faults when module is not being viewed an error message is displayed.

Equipment Model Builder generates a long list of unlinked faults even for simple plant loops such as that shown in figure 5.5. This means that the user must spend a large amount of time determining the effects of these faults. Fault linked to deviations and faults linked to consequences arcs must be created for these unlinked faults or the faults must be deleted if their effects are not considered to be important.

To prevent the creation of a long list of unlinked faults the following procedure could be used to enhance the exiting method. An inference engine could trace the effects of faults propagating through the units of the plant module. The final effect of the faults within the module would be found and arcs linking the faults to their final effects would be retained within the module. It would only be necessary to list a module fault as unlinked and delete the arcs the fault propagated through if the effects of that fault were found to result in contradictory multiple paths. However the effects of most faults might result in contradictory multiple paths. If this were so the above method would lead to little saving in user time. Further work is required to find out if this is supposition is true.

5.2.4. Allowing a Choice to Modularise

This thesis only considers modularising recycle loops which are relatively small compared to the size of the plant. Creating a module for a large loop encompassing most of the plant would entail the loss of the unit-based approach. Such a module would not be reusable within other plant configurations. Very large loops would group together functionally separate parts of the plant. For example, a user might not wish to place plant units involved with feed within the same module as those in which reaction occurs. How 'large' a recycle loop is will depend on the number of units it comprises compared to the number of units comprising the plant in which it occurs.

No arbitrary value can be assigned to the 'largeness' of a recycle loop as this depends on the relative sizes of the loop and the plant in which it is situated. Therefore the user needs to be given a choice as to whether the modular approach should be applied when a loop is detected in a plant description or whether the original unit models should be retained. One way of providing this choice would be for Equipment Model Builder to query the user each time a loop was detected as to whether it should be modularised.

The actual method used is as follows: the user indicates upon starting a session with Equipment Model Builder that he wishes to generate a plant description (see section 4.1.); the tool provides a menu from which the user chooses the maximum size of recycle loop to modularise; the user is queried as to the name of the file containing the description of the plant and Equipment Model Builder commences generating the plant system. This method has the advantage that the user is not annoyed by queries every time a loop is detected within a plant description. Any recycle loops occurring within the plant description which are larger than the maximum size input by the user are not identified as unit groupings which could lead to ambiguities. If a recycle loop greater than the maximum size specified is located during a plant search the search breaks and restarts at the last splitter found or finishes if there are no remaining splitters noted. This prevents the search from cycling infinitely around the loop.

5.3. Module Substitution

In order for a module to be used it needs to be substituted into the plant description. The individual units which make up the module are replaced with the module. The use of the module within the plant description instead of the individual units which comprise the module removes multiple causal paths which lead to ambiguities within the plant structure.

A unit grouping which could lead to ambiguities is identified within a plant description. A description of the unit grouping identified is presented to the user. This description is of the same form as that created for the description slot for the module template. The unit model library is searched for a module describing the unit grouping. If a module is found the user is 'asked' if this module should be substituted into the plant description. If the user confirms this the substitution stage begins. If the user indicates that no substitution should occur the session with the tool ends. This prevents the user from building a plant model containing unit configurations which could lead to ambiguities.

If no module describing the unit grouping is located within the model library the user is queried as whether a new module should be created for the unit grouping. If the user agrees the specification stage commences. If the user does not wish to create a new module the session with the tool ends.

To amalgamate information to form the input for the expert system QUEEN (Chung, 1993) the tool begins by taking a file describing a plant as input. This file may be a text file or generated by AutoCAD. As described in section 4.2 a modified plant description file is created from this file for use by the expert system. An output file consisting of the models of all the units found within the plant description is also created for use by the expert system. If a unit grouping which could lead to ambiguities is identified within the plant the plant description file is altered to describe the module specified for this unit grouping. A model of the module is placed into the output file. Descriptions of the unit instances comprising the module are removed from the plant description. Unit models of these units are not added to the output file unless they are required to model other instances of the same units located within the plant but outside of the plant fragment comprising the module. This constitutes the substitution stage. How the plant description file and the output file are modified will now be described in more detail.

5.3.1. Modifying the Plant Description

After a module has been identified and specified an instance describing this module is placed within a modified plant description file. An example will be used to illustrate this. Consider that a plant description is being created for the divider/header combination shown in figure 5.8 (section 5.2.1.1.). Assume that the divider/header combination is located between a pipe unit, P1, and a tail unit, tail1. Without module substitution the following instances would be placed in the original plant description file.

```
instance(P1 isa pipe,
   [outports info
      [out is [D1,in]]
   ]
).

instance(D1 isa divider,
   [outports info
      [out1 is [V1,in],out2 is [V2,in]],
      purpose is toBypass
   ]
).

instance(V1 isa valve,
   [outports info
      [out is [H1,in1]],
      aperture is open
   ]
).

instance(V2 isa valve,
   [outports info
      [out is [H1,in2]],
      aperture is closed
   ]
).

instance(H1 isa header,
   [outports info
      [out is [tail1,in]],
      purpose is fromBypass
   ]
).

instance(tail1 isa outlet,
   [outports info [ ]
   ]
).
```

The tool ensures that all unit groupings identified are substituted to prevent ambiguities arising within the plant model. The plant description file is modified as follows:

```
instance(P1 isa pipe,
   [outports info
      [out is [divhead1,D1_in]]
   ]
).

instance(divhead1 isa module_1,
   [outports info
      [H1_out is [tail1,in]],
       D1_purpose istoBypass,
       V1_aperture is open,
       V2_aperture is closed,
       H1_purpose is fromBypass
   ]
).
```

```
instance(tail1 isa outlet,
   [outports info [ ]
   ]
).
```

A module instance describing the divider/header combination is substituted into the plant description. The instances D1, V1, V2 and H1 are removed from the plant description. The identifier of the module instance is composed of the name given to the module by the user amalgamated with a unique numerical identifier assigned by the tool. In this example the user has decided to call the module created to model the plant fragment shown in figure 5.8 'divhead'. The identifier created is 'divhead1'. The connectivities of the module instance are the same as those of its constituent unit instances, except that inter-unit connectivities within the module have been removed. The conditions of the module are composed of the conditions of the units comprising the module. The unit instance, P1, located before the divider/header combination is modified so that instead of linking to the divider, D1, it now links to the module, divhead1.

5.3.2. Modifying the Output file

A unit model for each unit type found within the plant description is placed in an output file created for use by the expert system. In the case of modules a model is created for each instance type and placed within the output file. Each module model created is given the name 'module' combined with a unique numerical identifier, e.g. 'module_3'. The reason a separate model is created for each module instance type is that more than one occurrence of each structure of unit groupings comprising a module may occur within a plant description. For example, consider the following file describing the plant generated by AutoCAD:

'D1', 'divider', '[out1 is [V1,in],out2 is [V2,in]]', 'purpose is toBypass'

'V1', 'valve', '[out is [H1,in1]]', 'aperture is open'

'V2', 'valve', '[out is [H1,in2]]', 'aperture is closed'

'H1', 'header', '[out is [P1,in]]', 'purpose is fromBypass'

'P1', 'pipe', '[out is [D2,in]', 'unspecified'

'D2', 'divider', '[out1 is [V8,in],out2 is [V9,in]]', 'purpose is toBypass'

'V8', 'valve', '[out is [H2,in1]]', 'aperture is open'

'V9', 'valve', '[out is [H2,in2]]', 'aperture is closed'

'H2', 'header', '[out is [tail1,in]]', 'purpose is fromBypass'

It can be seen that this file contains two identical divider/header combinations. The structure of this divider/header combination is shown in figure 5.8.

It is important to recognise the faults and ports of the constituent units of each module located in the plant description. So that these faults and ports may be identified, the arcs of each individual unit instance which forms a constituent unit of a module are labelled with each instances identifier. This means that a model must be automatically created for each module instance within a plant description. This model is built by locating the model describing the module within the tool's model library. The arcs of this model are re-labelled with the unit instance identifiers given in the plant description for the module instance under consideration. Returning to the example described above, two instance module models will be created and placed with the output file. Assume they are named 'module_1' and 'module_2'. Example arcs taken from the module_1 instance are:

([fault,D1 leak to environment],+,[D1_in,flow])
([fault,D1 leak to environment],+,[consequence,contaminate environment])
([deviation,morePressure,D1_in],+,[consequence,possible explosion])

Example arcs from the instance created for module_2 are:

([fault,D2 leak to environment],+,[D1_in,flow])
([fault,D2 leak to environment],+,[consequence,contaminate environment])
([deviation,morePressure,D2_in],+,[consequence,possible explosion])

These examples show the divider unit instance within each module is labelled with its relevant identifier.

Identical occurrences consisting of different unit instances may also occur within separate plant descriptions. For this reason also a model must be created for each module instance. The model is re-labelled with the instance identifiers given in the plant description within which the module is currently being used. This allows reuse of modules.

5.4. User-defined Modules

This section will introduce the concept of user-defined modules. First, why user-defined modules are needed will be explained. In order to demonstrate methods of looking at the concept of user-defined modules a detailed example of where this type of module might be utilised is given. The first sub-section describes the features an ideal method would possess to deal with the concept of user-defined modules. In the following sub-sections three possible ways of looking at the concept are examined and the one implemented is described. These ways are applied to the detailed example. This section ends by

explaining the advantages and disadvantages of these ways. The way chosen to be implemented is detailed. How the features provided by the way chosen compare to those of the ideal way are considered.

User-defined modules allow the user to create modules from a set of plant units in a similar way to which modules are created for recycle loops and divider/header combinations. There are three main reasons why the user might wish to have the functionality to create his own modules.

(1) The user may wish to create a module when the combination of plant units results in an incorrect plant model.

(2) The user may wish to create modules for groups of plant units which commonly occur together in order to save time when creating plant descriptions.

(3) The user may wish to create modules for groups of units which are logically found together.

Combinations of plant units resulting in an incorrect plant model have already been identified as occurring in recycle loops and divider/header combinations. These types of combinations are detected by Equipment Model Builder which defines module templates for them. Other combinations of plant units which lead to an incorrect model may occur. A user-defined module would allow the user to create a correct plant model when these combinations are found in a plant description.

An example of a commonly occurring group of plant units would be a column combined with a reflux loop. Examples of groups of plant units logically found together would be those groups located in the feed and reaction sections of plants.

An example of a group located in the reaction section of a plant will be described below. This provides a basis for discussing methods for looking at the concept of user-defined modules. The group of plant units being used as an example forms part of an MTBE (methyl tertiary butyl ether) process refinery at Jujiang in China (McGreavy et al., 1997). The group consists of a heatexchanger, a reactor and a pressure relief valve. The heated outport (htot) of the heatexchanger is connected to the feed of the reactor. The pressure relief valve is situated on top of the reactor. A diagram of the group of units is given in figure 5.10 below. These units comprise the catalytic reactor of the plant.

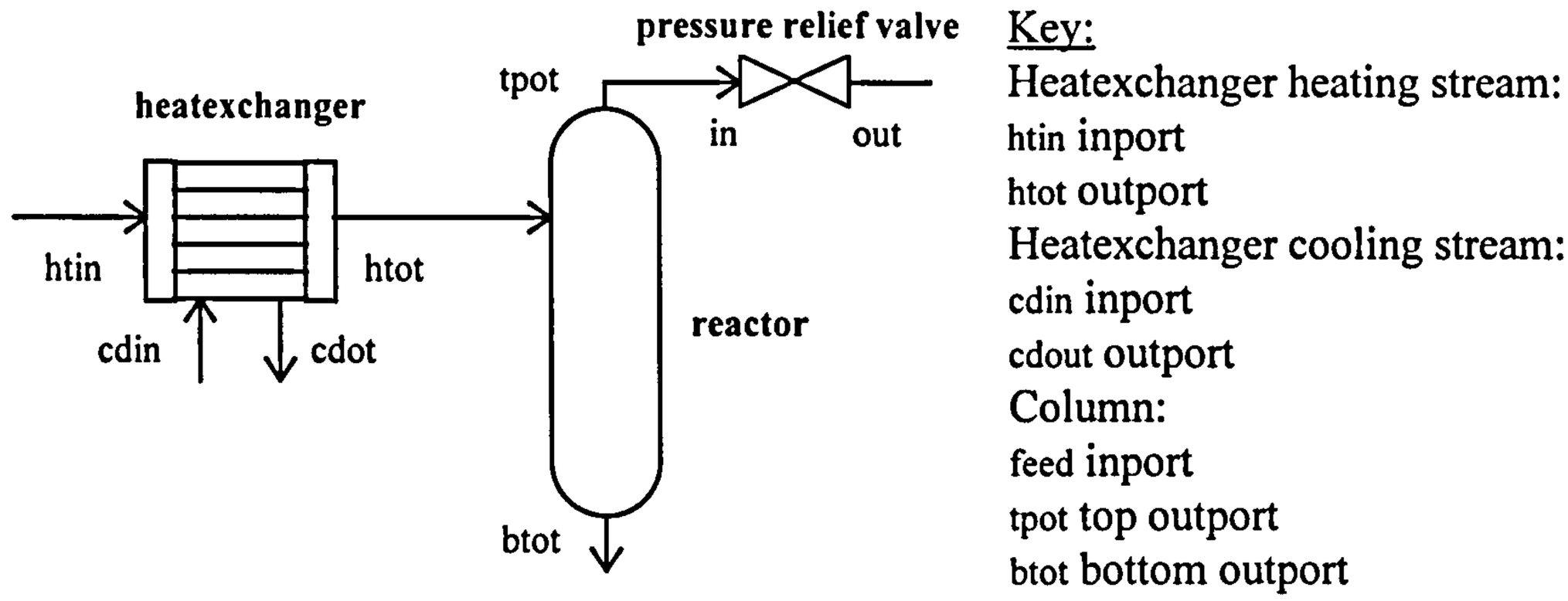


Figure 5.10 Catalytic Reactor of a MTBE Plant

5.4.1. Features of the Ideal Method

The ideal method for dealing with the concept of user-defined modules would cause the minimum of additional work for the user and minimum increase in system complexity. Additional work for the user could arise when specifying the module within a plant description, when defining the module or if a module were unintentionally duplicated.

Ideally, when specifying the user-defined module in a plant description only one icon for the entire module would be required, instead of an icon for each of the units of which the module was composed. This would make drawing and specifying the module much simpler.

To aid the user to define the module the arcs of the units comprising the module should be amalgamated. This would provide the user with a partially defined module and hence reduce the amount of user input required to build the module. The user should be guided as to what extra arcs need to be added or deleted to build the module. This means that a form of module template should be provided. A module template has been defined in section 5.2. as consisting of the amalgamated unit models and a user interface which together guide the user in creating a module. The user should not need to create any additional faults, consequences or conditions for the module. All the variables required to build the arcs of the module should be provided by the module template.

To prevent module duplication the system should possess information on what units the module contains and how these units are connected. The system would be able to use this information to 'check' whether a module being defined is already present in the model library.

Increases in system complexity could occur if additional searching was required to locate plant fragments matching user-defined modules within plant descriptions. Ideally no additional searching should be necessary.

5.4.2. Default Method

To apply this method the user would create an icon for the module within the AutoCAD system. This would be used in the plant diagram created by AutoCAD when the user specifies a plant description for use by Equipment Model Builder. The module would be built using Equipment Model Builder's user interface in the same way in which ordinary units are built, i.e. the module is treated exactly like a single unit. There is no system support to aid the users in defining their own modules at all. The module is specified within a plant description like a unit model.

Using the example given in figure 5.10 the user would create an icon which would represent all the units shown. Equipment Model Builder's user interface would be used to create a 'catalytic reactor' module. Unlike the modules created for recycle loops and divider/header combinations the individual units of modules defined using the default method have no identifiers. The tool's standard user interface used by the default method provides no means of designating identifiers for the individual units of the module.

However it is important to specify which units within the module contain faults. A fault such as 'leak to environment' is too broad to be of use. Knowledge about which unit is leaking is required. This means that the user must create faults containing generic unit identifiers. This is illustrated by examples given below. The examples are of a fault linked to deviation arc and of a fault linked to consequence arc which might be present in the catalytic reactor module when it is defined by the default method.

([fault,heatexchanger partly blocked (heating side)],-,[heatexchanger_cdot,flow])

[fault,reactor leak to environment],+,[consequence,contaminate environment])

The generic unit identifiers are 'heatexchanger' and 'reactor'. The faults containing the generic unit identifiers would be added to Equipment Model Builder's fault library.

This results in a larger and more complex fault library. The library will contain faults applicable to unit models such as 'leak to environment'. It will also contain duplicates of these faults which have been paired with a generic identifier, e.g. 'reactor leak to environment', 'pressure valve leak to environment'.

When a modified plant description is created for QUEEN, for example the MTBE plant, the module built by the default method is specified and used like a unit model (see

section 4.2.). Like a unit model its specialised file is amalgamated with the specialised files of the other units specified within the plant description to form an output file of models occurring within the plant. Unlike the modules created for recycle loops and divider/header combinations the module created by the default method may be directly reused within other plant descriptions with no modifications. The generic identifiers contained within arcs of the module built by the default method are applicable to any plant description in which the module may be located.

5.4.3. Extending the Default Method

The default method could be extended to include more features. Equipment Model Builder would create queries to ask the user to indicate what units the module is to contain. From this information Equipment Model Builder could search the model library for the unit models. The unit models could be used to provide the user with a list of faults present in the units of the module. This gives some help to the user by making clear what possible faults have to be considered for the module. Some of the information required to be present in the module is given. However this method does not allow a template for the module to be defined. None of the advantages provided by having a template would be available to the user.

The default method could be further extended. The tool would create more queries requiring the user to indicate the inports and outports of the module as well as what units the module is to contain. The user would also be asked to give an identifier for each unit of the module. From this information Equipment Model Builder would be able to create a template for defining the module. This template would be created in the same way as the templates created for recycle loops and divider/header combinations. This template provides the method for detecting unlinked module faults, a way of verifying the module. The user is guided in creating a complete module. As in the templates created for recycle loops and divider/header combinations the fault and condition libraries would be temporarily enlarged while the module is being built. Unlike the default method no permanently extended fault library is created.

This method shall be referred to as the 'extended-default' method. The extended-default method has been implemented as part of the Equipment Model Builder tool. Further details of this method will now be discussed. Equipment Model Builder presents the user with a menu listing the units contained in the model library. From this the user chooses a unit to place within the module. Equipment Model Builder provides the user

with a box to fill in an identifier for the unit. The queries are repeated in order to choose another unit and identifier. A module may be composed of a minimum of two unit models. The tool 'asks' the user whether all the models which the user-defined module is comprised of have been chosen. If the reply is negative then the series of menu, box and question is repeated until the user affirms that the set of units the module is comprised of is complete.

A list of possible inports is prepared for the module. Possible inports are the inports of the modules constituent units. Equipment Model Builder labels the inports with the unit identifiers which have been provided by the user. A menu showing this list is presented to the user. From this the user chooses the actual inports of the module. The same method is used to discover the outports of the module from the user. For instance, to create a module for the example given in figure 5.10 the user would choose the following units: 'heatexchanger', 'reactor' and 'pressure relief valve'. Possible identifiers might be 'h1', 'r1' and 'pv1'. The user would choose 'h1_cdin' and 'h1_htin' as the inports of the module and 'h1_cdot', 'r1_btot' and 'pv1_out' as the outports of the module.

When this stage is reached Equipment Model Builder amalgamates the units specified to create a template defining the module. An interface is provided which allows the user to alter the template or to add more information to it. This interface is the same as that provided for recycle loops and divider/header combinations (see section 5.2.3.). It provides a method for detecting unlinked module faults. The faults and ports of the module are labelled by unit identifiers taken from the users specification. For example, if a template were defined for the units shown in figure 5.10 it might contain the following arcs:

```
([fault,h1 partly blocked (heating side)],-,[h1_cdot,flow])
([fault,r1 leak to environment],+,[consequence,contaminate environment])
```

A description is created for the module and placed within the description slot of the template describing the module. As the connectivities of the module are unknown this description cannot be of the same form as that created for recycle loops and divider/header combinations. A simpler description consisting only of the units of the module is created. For example, the following description would be created for the unit shown in figure 5.10:

'This user-defined module is composed of the following units: h1, r1 and pv1.'

When the user module is placed within a plant description faults due to its constituent units are recognised by the identifiers input by the user for the units when specifying the module. Unlike recycle loops and divider/header combinations unit instance names within the module are not replaced by those of each new occurrence of the module. Should two or more instances of the same user-defined module occur within a single plant description it is still be possible to differentiate which user-defined module the faults of the constituent units occur in. This is because although the identifiers of the user-defined modules' constituent models are identical the identifiers of the modules within the plant description are not. For example, assume two instances of a user-defined module are present within a plant description. The identifiers of these instances are 'usermod1' and 'usermod2'. Assume the user-defined module has a valve, identifier 'v1', as one of its constituent units. If this valve were to contain a fault such as 'closed in error' then the following faults would be present in the output file of models created for the plant description.

'usermod1: v1 closed in error'

'usermod2: v1 closed in error'

If a user-defined module forms one of the constituent units of an outer module (i.e. a recycle loop, a divider/header combination or a user-defined module) the module identifier of the constituent user-defined module is retained. This ensures that the faults of the units comprising the new outer module possess unique identifiers. For example, the new module could consist of a valve, identifier v1, and a user-defined module, usermod1. Assume usermod1 has a valve, identifier v1, among its constituent units. The identifier of the new outer module created is newmod1. The following faults would be present in the specialised file describing the new module.

'newmod1: v1 closed in error'

'newmod1: usermod1_v1 closed in error'

It can be seen that retaining the user-defined module identifier allows the faults of a modules constituent units and modules to be uniquely recognisable.

When a recycle loop module forms one of the constituent units of an outer recycle loop module or a divider/header module forms one of the constituent units within a recycle loop module or a divider/header module (see section 5.1.3.) the module identifier of the inner module is removed. The module identifier is not required in this case to maintain unique identifiers for the faults of the new model as unit instance names within the inner

module are replaced by those of each new occurrence of the module. The outer module will not contain units with the same identifiers as the inner module as all unit identifiers will be unique within a plant description.

When recycle loop modules and divider/header modules form constituent units of a user-defined module the module identifier of the inner module (i.e. recycle loop or divider/header) is retained as unit instance names within the inner module are not replaced in this case. This ensures that the faults of the constituent units of the new user-defined module built are unique. For example, assume a user-defined module consists of a valve, identifier v1, and a divider/header module, identifier byps1. Assume byps1 has a valve, identifier, v1 among its constituent units. The identifier of the new outer module created is newmod2. The following faults would be present in the new module's specialised file.

'newmod2: v1 closed in error'

'newmod2: byps1_v1 closed in error'

5.4.4. Automatic-identification Method

This approach would take place in same three stages as the method for modularising recycle loops and divider/header combinations. These are identification, specification and substitution. However unlike recycle loops and divider/header combinations when user-defined modules are created the user is free to specify the unit combination. This means that the structure of a user-defined module is not known, therefore the initial stage must be specification. The automatic-identification method would take place in the following order: specification, identification and substitution.

First, user specifies a module composed of multiple units. This could be done in the same way in which plant descriptions are generated by an AutoCAD file or a text file describing the module. When the user specified the units and their connectivities within a module Equipment Model Builder would be able to store this information within the model library. Whenever a new user-defined module was specified the tool would be able to 'check' the new units and connectivities given against this information. The tool would be able to determine if a module was being specified which was already present as a user-defined module within the model library. Duplication of modules could be prevented.

To build a module defined by the automatic-identification method Equipment Model Builder would amalgamate existing plant unit models as specified by the user. As with modules created for recycle loops and divider/header combinations causal paths existing between the units of the module would be removed. Important internal influences

would be retained. See section 5.2. for more details. The faults and ports of the module would be labelled by unit identifiers taken from the users specification. An interface would exist which would allow the user to add extra information to the module. This would provide a template for defining the module. This template would provide benefits detailed in the extended-default method.

The next stage is identification. From the specification given by the user Equipment Model Builder would be able to identify the groupings of units together with the connections between them that composed the user-defined module within a plant description. Equipment Model Builder would take a plant description as input. In addition to identifying recycle loops and divider/header combinations the tool would also identify modules defined by the automatic-identification method occurring within the plant.

Each time the module was used when a file describing the plant was created the user would have to draw and specify its individual units. When using a text file to create a plant description containing the module this could simply be a matter of copying existing material but when using AutoCAD this could be time-consuming. Additional searching would be required for plant fragments matching user-defined modules within the plant description.

In the final stage, Equipment Model Builder would substitute each occurrence of a user-defined module found into the plant description. The user would be informed that this substitution was occurring. The individual units of the unit group which comprised the user-defined module would be replaced with the module. Unit identifiers within the user-defined module would be replaced with those specified for the new occurrence in the plant description. A specialised file would be created for each new occurrence found. This would be done by updating the existing specialised file for the user-defined model with the identifiers specified for the new occurrence in the plant description. The new specialised files created would be amalgamated with the specialised files of the other units specified within the plant description to form an output file of models occurring within the plant.

How the automatic-identification method may be applied to the example given in figure 5.10 will be described. First the user will specify the module. This could be done by using a description generated by AutoCAD. An example of a description of the group of units shown in figure 5.10 might be:

'heatex1', 'heatexchanger', '[htot is [reactor1,feed],cdot is [tail1,in]]', 'unspecified'

'reactor1', 'reactor', '[btot is [tail2,in],tpot is [presvalve1,in]]', 'unspecified'

'presvalve1', 'pressure relief valve', '[out is [tail3,in]]', 'unspecified'

See section 2.4.2. for an explanation of plant descriptions generated by AutoCAD.

Equipment Model Builder would amalgamate plant unit models for a heatexchanger, reactor and a pressure relief valve to form a template for defining the module. The user would add additional information to this template, for example a name such as 'catalytic reactor'. Example arcs from this template are given below. These arcs show how faults and ports of the module have been created by combining unit identifiers taken from the user's specification with faults and ports of the unit models comprising the module.

([fault,heatex1 partly blocked (heating side)],-,[heatex1_cdot,flow])

([fault,reactor1 leak to environment],+,[consequence,contaminate environment])

Example 5.2 Arcs from Template for Catalytic Reactor Module

From the description given and information contained in the unit models Equipment Model Builder would be able to determine the in- and outports of the module. These are the in- and outports of the constituent units which are not used to link the units of the module. For the catalytic reactor module inports would be 'heatex_htin' and 'heatex_cdin'. Outports would be 'heatex1_cdot', 'reactor1_btot' and 'presvalve1_out'. Arcs containing ports which link the units of the module would be removed. This would remove causal paths existing between units of the module. The first arc shown above is present within the module template as it contains a fault linked to deviation arc in which the deviation occurs at a boundary port of the module. The second arc is present as it is a fault linked to consequence arc.

When a plant description is created, for example the MTBE plant, the identification stage would take place. Using this example, Equipment Model would detect that the cdot port of a heatexchanger is linked to the feed of a reactor. The tool would also detect that the tpot port of this reactor is linked to the inport of a pressure relief valve. From this information Equipment Model Builder would identify that these units constitute a user-defined module present in the model library.

Finally, the individual units of the unit group which comprised the user-defined module would be substituted by Equipment Model Builder with the catalytic reactor module. Unit identifiers within the arcs of the catalytic reactor module would be replaced with those specified for the new occurrence in the plant description. The following

example will show what is meant by this. Assume the unit group which comprises the catalytic reactor was specified in the MTBE plant description as:

'h1', 'heatexchanger', '[htot is [r1,feed],cdot is [tail1,in]]', 'unspecified'

'r1', 'reactor', '[btot is [tail2,in],tpot is [pv1,in]]', 'unspecified'

'pv1', 'pressure relief valve', '[out is [tail3,in]]', 'unspecified'.

The arcs given in example 5.2 would now become:

([fault,h1 partly blocked (heating side)],-,[h1_cdot,flow])

([fault,r1 leak to environment],+,[consequence,contaminate environment])

A specialised file would be created for this new occurrence found.

5.4.5. Method Chosen

The advantages and disadvantages of the three methods detailed above are summarised in tables 5.1 and 5.2 below.

	Method			
Advantages	Ideal	Default	Extended-default	Automatic-identification
Only one icon required for the module within a plant description.	✓	✓	✓	
A template for defining the module exists.	✓		✓	✓
A larger fault library is not required.	✓		✓	✓
User will recourse to this method.		✓		
User specifies the units and how they are connected together to form the module.	✓			✓
No additional system searching needed.	✓	✓	✓	

Table 5.1. Three Ways of Dealing with User-defined Models- Advantages

	Method			
Disadvantages	Ideal	Default	Extended-default	Automatic-identification
No template for defining the module.		✓		
Requires additional work for the user when specifying the module in a plant description.				✓
Larger fault library required.		✓		
System has no information on units and how they are connected to form the module.		✓	✓	
Additional system searching.				✓

Table 5.2. Three Ways of Dealing with User-defined Models- Disadvantages

Any method provided to deal with the concept of user-defined modules needs to be simple and efficient to use otherwise the user will use the default method. As this method is already provided by Equipment Model Builder the user will use it anyway if other methods for creating user-defined modules are found to be too complicated or time-consuming.

The extended-default method was chosen to create user-defined modules and implemented as part of Equipment Model Builder. Table 5.1 shows that this method possesses all but one of the advantages of the ideal method. This method is simple to use. Its disadvantage is that the system has no information on units and unit connectivity of the module. Equipment Model Builder is unable to 'check' whether the module being defined is already in the model library. Duplication of modules cannot be prevented. However ordinary units are not checked to see if a duplicate already exists in the unit library when they are created. Equipment Model Builder only checks that unit names are not duplicated.

This sub-section has described the concept of a user-defined module. Possible methods for modelling user-defined modules have been discussed. From these a method has been derived which is believed to offer most help in defining modules to a user. How this method is applied by Equipment Model Builder was described.

5.5 Summary

This chapter has described a new module creation approach which provides a method to remove ambiguities due to multiple causal paths. This modular approach takes place in three stages: identification, specification and substitution.

The groupings of units which Equipment Model Builder automatically identifies as leading to ambiguities within qualitative modelling are different possible configurations of recycle loops and divider/header combinations. Other groupings of units which lead to ambiguities may occur. For these groupings of units the user is provided with the functionality to define modules.

To identify the unit groupings which could lead to ambiguities Equipment Model Builder makes two searches of the plant description input: one to identify divider/header combinations and one to identify recycle loops. Divider/header combinations are searched for first. It is necessary to detect these combinations first before searching for recycle loops as the substitution of a module of a recycle loop into the plant system would cause

any information about possible divider/header combinations located within the recycle loop to be lost.

The search of the plant is complicated by the presence of certain types of units which have been called 'splitters'. These units have more than one outport which leads to branching of process flow stream through the plant. This causes branching of the search path. Examples of these types of units are tanks with multiple outports and columns. Heatexchangers have multiple outports but do not divide the process flow stream within the plant so do not cause branching of the plant search. Heatexchangers are treated as a special case.

Algorithms used to identify divider/header combinations and recycle loops within plant descriptions have been described. Within process plants in addition to simple recycle loops and divider/header combinations there is the possibility for complex structures consisting of multiple loops, multiple divider/header combinations and combinations of loops and divider/header combinations to occur. How identification of unit groupings which could lead to ambiguities is carried out when these complex structures are encountered has been detailed.

If there is no existing module within the model library which describes the unit group identified a new one needs to be specified. Modules are specified for the plant fragments identified by amalgamating and modifying existing models of plant units. Modules for unit groupings comprising recycle loops and divider/header combinations are specified in exactly the same way. An interface provides the user with a guide as to what extra information needs to be added to the module and enables the user to add this information.

The amalgamated units models together with the interface serve as a template to allow the user to define the module identified. When the units are amalgamated to form a module the causal paths that lead to ambiguities or incorrect behaviour are removed. Internal influences of the module are retained. After the module has been defined it is added to a library of component models.

The initial module building phase consists of:

- checking that all the units which comprise the module identified are modelled;
- creating and labelling instances of these units;
- creating a description of the module.

When the unit instances are amalgamated the fault linked to deviation arcs in which the deviation does not occur at a boundary port or in which the deviation does not

affect a port which has a consequence are deleted. These arcs are deleted to remove the multiple causal paths which could lead to ambiguities. However the faults of these arcs must be retained as they may have an important effect within the module. They may also propagate out of the module to cause deviations in other units when the module is placed in a plant description. The faults from the deleted arcs are stored by Equipment Model Builder as 'unlinked module faults'. The interface provided allows the user to add more information about these faults to the module.

This thesis only considers modularising recycle loops relatively small compared to the size of the plant. The maximum size of recycle loop to modularise is chosen by the user.

In the substitution stage the individual units which make up the module are replaced with the module. The use of the module within the plant description instead of the individual units which comprise the module removes multiple causal paths which lead to ambiguities within the plant structure.

A unit grouping which could lead to ambiguities is identified within a plant description. The model library is searched for a module describing the unit grouping. If a module is found the user is 'asked' if this module should be substituted into the plant description. If the user confirms this the substitution stage begins. If no module describing the unit grouping is located within the model library the user is queried as whether a new module should be created for the unit grouping. If the user agrees the specification stage commences.

To amalgamate information to form the input for the expert system QUEEN (Chung, 1993) the tool begins by taking a file describing a plant as input. A modified plant description file is created from this file for use by the expert system. An output file consisting of the models of all the units found within the plant description is also created for use by the expert system. If a unit grouping which could lead to ambiguities is identified within the plant the plant description file is altered to describe the module specified for this unit grouping. A model of the module is placed into the output file. Descriptions of the unit instances comprising the module are removed from the plant description. Unit models of these units are not added to the output file unless they are required to model other instances of the same units located within the plant but outside of the plant fragment comprising the module. This comprises the substitution stage.

The penultimate section of this chapter considers the concept of user-defined modules. Why they are needed is explained. Three possible methods of looking at the concept were examined:

1. the default method;
2. the extended-default method;
3. the automatic identification method.

The default method would use Equipment Model Builder's user interface to build user-defined modules in the same way in which ordinary units are built, i.e. the module would be treated exactly like a single unit.

The extended-default method queries the user as what units the module is to contain and the identity of the module outports. The user is be asked to give an identifier for each unit of the module. From this information Equipment Model Builder is able to create a template for defining the module. The user is guided in creating a complete module.

The automatic identification method would take place in same three stages as the method for modularising recycle loops and divider/header combinations. However the automatic-identification method cannot take place in the same order as the other modular approaches as the structure of a user-defined module is unknown. The automatic-identification method would take place in the following order: specification, identification and substitution.

The advantages and disadvantages of the three methods were discussed. The method offering the user most help in defining modules was found to be the extended-default method. This was the method chosen and implemented to aid the user in creating user-defined modules.

6. Verification

Verification has been defined in chapter 3. Verification methods were split into three categories: those that check for completeness, correctness and conciseness. In this chapter the verification methods used in Equipment Model Builder are discussed within the context of these categories. The development of models to evaluate the verification methods is described.

Some of the methods shown below are purely verification methods whereas others may also be thought of as part of the user interface, duplication checking or explanation facilities (i.e. part of the knowledge acquisition facilities) provided by Equipment Model Builder.

What an 'ideal' model consists of will be described for each category. Verification techniques used by Equipment Model Builder to achieve this 'ideal' model will be discussed. Some techniques may fall into more than one category. Any shortcomings with the unit model after it has been verified will be detailed.

6.1. Completeness

A complete model has no missing information, i.e. all the information that ought to be in the model is contained in the model structure. It will be able to function in all possible situations that arise in the application. As what information the model should contain is unknown it is impossible to ensure that a model is complete. The set of all the possible ways that the model might be required to behave is unknown. Although there are ways of identifying some aspects of missing information, completeness cannot be guaranteed.

This section explains the different ways in which incomplete information can occur in the unit models created by Equipment Model Builder. Techniques for identifying this incomplete information are described. Other types of missing information are then considered.

Two types of incomplete information will be discussed. The first type is incomplete arcs. Each arc within a unit model should contain an initial node, an influence and an influenced node. Equipment Model Builder will not allow a user to save a model until all the arcs within the model are complete. The user is informed which forms contain incomplete arcs. This makes up part of Equipment Model Builder's user interface and will not be discussed further. The second type of incomplete information described is unreferenced attributes.

6.1.1. Unreferenced Attributes

The term 'unreferenced attributes' has been taken from the body of work describing verification of rule-based systems (see section 3.4.2.2.). There are two types of incompleteness check for unreferenced attributes relevant to this thesis: the dead-end conclusion check and the unreachable condition check.

An SDG may be compared to a rule. Consider the arc shown below.

$$X \xrightarrow{+/-} Y$$

The node 'X' may be compared to the conditions or left hand side of a rule. The node 'Y' may be compared to the conclusions or right hand side of a rule. The completeness checks for rule-based systems are also applicable to systems constructing SDG models. The following description explains how unreferenced attributes may occur in the unit models created by Equipment Model Builder.

As described previously (section 2.4.1.) the unit model has four types of arcs. These are:

(i) deviation linked to deviation;

(ii) fault linked to deviation;

(iii) fault linked to consequence;

(iv) deviation linked to consequence.

The initiating node 'X' of the SDG shown could be a process variable deviation or a fault. The influenced node 'Y' could be a deviation or consequence. A consequence may occur directly as a result of a fault (arc type iii). e.g.

([fault, leak to environment], +,[consequence, [loss of material, expensive]]).

It may also occur as a result of a fault propagating through a deviation or series of deviations. For example, in a tank model the following fault propagation might occur, leading to the consequence shown:

([tank,fault,external fire], +,[tank,vapour,temperature])

([tank,vapour,temperature], +, [tank,liquid,temperature])

([tank,deviation,[moreTemperature,liquid]], +,[tank,consequence,crystallisation]).

The nodes in the path are of the types ii, i and iv respectively. The fault (failure mode) initiating a propagation path need not be present in the same unit as the resulting

consequence. The fault may occur in an upstream unit and cause a deviation which propagates into another unit via its inports, or in a downstream unit and cause a deviation which propagates into another unit via its outports, to result in a consequence. For example, a pipe model linked upstream of the tank model could cause the following consequence in the tank model:

([pipe,fault,partly blocked], -,[pipe,out,flow])

([pipe,out,flow], +,[tank,in1,flow])

([tank,in1,flow], +,[tank,liquid,level])

([tank,deviation,[lessLevel,liquid]], +,[tank,consequence,vessel emptying])

Thus in order for a fault to have the potential to cause a consequence it must fall into one of the following categories:

1. The fault must be directly linked to the consequence.
2. The fault must cause a deviation which propagates within the unit model to cause a consequence.
3. The fault must cause a deviation which propagates out of the unit model via its inports or outports resulting in a consequence in another unit.

Any fault that does not fall into one of more of these categories is an unreferenced attribute as it has no overall effect. An unreferenced fault (failure mode) causes a deviation with no effect.

Deviations with no effect may be present in fault linked to deviation arcs as described above. They may also be present in deviation linked to deviation arcs in which the deviation does not propagate to a boundary port. Deviations propagating from boundary ports (i.e. in and out ports) are assumed to have an effect as they may propagate out of the unit model resulting in a consequence in another unit. For example, the deviation 'out,flow' in the arc '([pipe,in,flow], +,[pipe,out,flow])' could propagate out of the pipe unit to cause a potential effect in a downstream unit if the pipe unit was present in a plant description.

In order for a consequence to occur it must either be:

1. directly linked to a fault;
2. result from a fault causing a deviation to propagate within a unit model;
3. be linked to the inports or outports of the unit model. This is so that deviations may propagate in to the unit to cause the consequence. An example of this last case for a tank model would be:

([tank,deviation,[moreFlow,out]], +,[tank,consequence,vessel emptying])

Any consequences not fulfilling these criteria are unreferenced attributes as they will never occur. An unreferenced consequence is linked to a deviation without a cause.

Deviations without causes may be present in deviation linked to consequence arcs (described above). They may also be present in deviation linked to deviation arcs in which the deviation does not initiate at a boundary port. Deviations initiating at boundary ports are assumed to have a cause. The cause may propagate into the unit via its boundary ports from elsewhere in the plant. For example, the deviation 'in1,flow' in the arc '([tank,in1,flow], +, [tank,liquid,level])' may be caused by a fault propagating into the tank unit from an upstream unit when the tank unit is present in a plant description.

Another type of unreferenced attribute arises when modules are created. Unlinked faults occur when fault linked to deviation arcs are deleted within the module. This has been described in section 5.2.2.3. The final type of unreferenced attributes to be described are undetected faults.

6.1.1.1. Undetected Faults

The unit models created using the Equipment Model Builder interface are intended for use in systems undertaking hazard identification (e.g. Parmar and Lees, 1987; Chung, 1993;Vaidhyanathan and Ventkatasubramanian, 1995; 1996). Systems of this type examine deviations associated with the units. Causes are identified that give rise to these process deviations. Propagation paths are followed from the causes to identify consequences.

Only faults linked to process deviations are detected and the consequences of those faults identified. Within systems undertaking hazard identification consequences linked to undetected faults will not be located. In order to detect consequences in the fault linked to consequence arcs of a unit model the faults of those arcs must also be present in the fault linked to deviation arcs of the model.

6.1.1.2. Identification Techniques

For a given model, Equipment Model Builder maintains lists of deviations with no effect, deviations without causes and undetected faults. Each new arc added or deleted is checked to see if it causes these lists to require updating.

To allow a deviation with no effect to cause a potential effect the new arc must allow the deviation to propagate to an inport, outport or consequence. The new arc must be

applicable under the same model conditions as the process variable deviation with no effect. This means it must share the same conditions as the deviation with no effect or have no specific conditions, making it applicable to the whole model. Instead of allowing the deviation with no effect to propagate to an effect a new arc might simply alter the deviation by lengthening the propagation path. For example, a new arc '([vapour,temperature], +, [liquid,temperature])' might simply change a deviation with no effect from 'temperature vapour' to 'temperature liquid'.

In order to allow a cause to propagate to a deviation without a cause the new arc must contain the correct influence and process variable. For example, the arc '([liquid,level], - -, [in,flow])' cannot bring about the deviation 'moreFlow, in'. Although a propagation path is provided to the process variable 'flow' the influence '- -' cannot cause an increase in the variable flow. The new arc must also be applicable under the same model conditions as the deviation without a cause.

In order to allow a fault within a fault to consequence arc to be linked to a deviation the fault linked to consequence arc must share the same model conditions as the fault linked to deviation arc.

Equipment Model Builder allows the user to access the lists of deviations with no effect, deviations without causes, and undetected faults from a drop-down menu. This means that the lists need not be visible until the user wishes to verify the model. The user will not be annoyed by the presence of extra information until it is required. The lists also contain the conditions under which the deviations with no effect, deviations without causes and undetected faults are applicable. When the model is saved the user is informed if any deviations with no effect, deviations without causes or undetected faults remain.

Unlinked module faults are also listed. Again, this list may be accessed from a drop-down menu. This is described more fully in section 5.2.3.2.

6.1.2. Missing Information

Three kinds of missing information will be considered:

1. missing units within the unit library;
2. missing faults and consequences;
3. missing propagation paths.

6.1.2.1. Missing Units

Equipment Model Builder's unit library needs to be complete in order for it to create modules and complete output files of models of a plant's units. When a module or output file is created Equipment Model Builder checks that all the component units are present within its model library. If a model for a unit is found to be missing the user is queried as to whether he wishes to create the missing model.

6.1.2.2. Missing Faults and Consequences

Omissions by the user will result in missing faults and consequences within the unit model. These are difficult to detect simply because it is not known what is missing. A missing fault could result in an unreferenced consequence and which would lead to a deviation without a cause. However, a unit model with no consequences or deviations without causes does not necessarily have no missing faults. The consequences or deviations may be caused by more than one fault. If any of the faults is missing the consequences and deviations will still have a cause. No deviation without a cause will be detected.

Missing consequences can result in unreferenced faults and hence deviations without effects in a unit model. However, a fault may cause multiple consequences within a unit model or propagate out of the unit model to cause consequences in other units within a plant description. If one of the multiple consequences is missing this fault will still be able to cause a consequence. A deviation without an effect will not be detected.

In order to aid the user to discover missing faults Equipment Model Builder could present the user with a different view of the model. This could be done by showing the user a list of model faults (failure modes) and querying the user if this list were complete. Missing faults could be added to the list. The user could then be asked whether the fault lead to a consequence or a deviation. When this was established the fault could be placed on the relevant forms within Equipment Model Builder. A similar list could also be provided for model consequences. Missing consequences could be added to the list. The user could be asked if the consequence were linked to a deviation or a fault. The consequence could also be placed on the relevant forms.

6.1.2.3. Missing Propagation Paths

In order for unit models to function correctly within a plant model deviations in process variables will need to be able to propagate through them. This means that process variable deviations need to be able to propagate from a unit's inports to its outports and from its outports to its inports. Exceptions to this are models for the source and outlet of the plant. Some unit models may not propagate all deviations. For example, an open tank will not propagate an increase in flow from its outport to its inport. However most units will propagate most deviations.

Equipment Model Builder identifies process variables with no propagation path through a unit model and provides the user with a list, accessible via a drop-down menu. These variables will be called unlinked boundary deviations. This does not mean that the model is not complete if there are process variables with no propagation path. For example, in the case of the open tank there is no propagation path for flow from the tank's outports to its inports. It only means that the model *might* be not be complete. The list is intended to act as a memory aid for the user. The process variable 'level' is not tested for propagation as this tends only to be linked to the internal ports of models. This list of process variables with no propagation paths will help to ensure that the unit models behaviour is plausible.

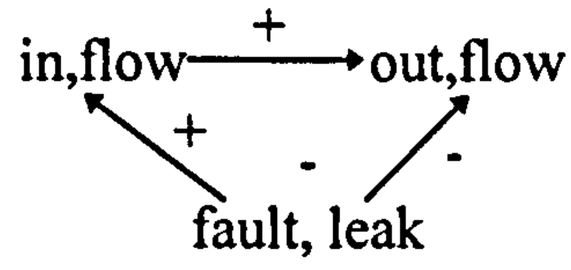
6.2. Correctness

A correct model is an accurate representation. A correct model has no conflicting information or illegitimate attributes. An illegitimate attribute is one which does not occur within the set of attribute values allowed for the model. A correct model contains no wrong information. A correct model will function correctly. However, there may be misconceptions about how a model should function. The models role or the type of systems in which it will be used may not be understood.

This sub-section will first look at how conflicting information occurs within a model and a technique will be described as to how its effects might be detected. Techniques for preventing illegitimate attributes will be discussed. Techniques for identifying wrong information are looked at.

6.2.1. Conflicting Information

Conflict may occur within an SDG model when there is more than one possible propagation path between the nodes within the model. The following SDG will be used to illustrate this:



From this SDG two propagation paths may be traced between the fault 'leak' and 'out,flow'. The two paths are:

$$\text{fault,leak} \xrightarrow{-} \text{out,flow}$$

$$\text{fault,leak} \xrightarrow{+} \text{in,flow} \xrightarrow{+} \text{out,flow}$$

For the first path the effect of a leak is an decrease of 'out,flow'. For the second path the effect is a increase of 'out,flow'.

The qualitative analysis results in two contradictory paths with the first path having the correct influence. In order to deal with ambiguities a heuristic that is commonly used is that when there is more than one acyclic path through the SDG the shortest path is used. This heuristic is used as it is applicable to many cases.

6.2.1.1. Detecting the Effects of Conflicting Information

A technique has been devised which allows the user to check that the shortest path within the unit model leads to correct model behaviour. To avoid duplication of work the technique for verifying the shortest path uses the QUEEN (Chung, 1993) system. Equipment Model Builder 'asks' the user when a model is saved as to whether the shortest paths should be verified. If the user wishes to verify the shortest paths Equipment Model Builder builds a plant description and specialised file for the model to enable QUEEN to test the model. Equipment Model Builder prepares a file of queries to send to QUEEN. QUEEN generates the shortest paths and their effects from these queries. The user verifies that these shortest paths are correct. To verify the model certain query types are needed to cover all the kinds of propagation path which may occur between the node types within the SDG model.

Within the unit model deviations may propagate to cause effects along four different types of path. The deviation may propagate:

(1) from a boundary port to a boundary port;

(2) from a fault to cause a consequence;

(3) from a fault to a boundary port;

(4) from a boundary port to cause a consequence.

Equipment Model Builder prepares an exhaustive list of queries to test for shortest paths between all the two node combinations which might occur within the model. The tool tests all possible paths. Not all of the paths tested for may exist. To provide a base for describing these queries a simple model is given below (in text). It is intended for illustration only.

```
frame(pipe isa unit,
        [  inports info  [ in ],
           outports info  [ out ],
           propLinks info [

           %propagation

           arc([in,pressure],+,[out,pressure]),
           arc([out,pressure],+,[in,pressure]),
           arc([in,temperature],+,[out,temperature]),
           arc([in,flow],+,[out,flow]),
           arc([out,flow],+,[in,flow]),


          %faults
          arc([fault,'partly blocked'],-,[out,flow]),
          arc([fault,['leak into vacuum system', vacuum]],+,[in,pressure]),

           %consequences resulting from faults
           arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
           arc([fault,'leak to environment'],+,[consequence, 'loss of material']),

           %consequences resulting from deviations
           arc([deviation,[morePressure,in]],+,[consequence,'possible rupture'])
         ]
      ]
).
```

To verify type (1) paths a complete list of process variable deviations is prepared for every boundary port. Queries are prepared to test for shortest paths between each of the process variable deviations at a boundary port and each of the process variable deviations at the other boundary ports within the model. Examples of some of the paths queried for the pipe model shown are 'in,pressure' propagating to 'out,pressure', 'in,pressure' propagating to 'out,temperature' and 'out,pressure' propagating to 'in,pressure'. Queries are also prepared to test for shortest paths for deviations propagating from a boundary port back to the same port but a different process variable, e.g. 'in,flow' propagating to 'in,temperaure'. Queries are not set up for deviations

propagating from a boundary port back to the same variable at the same port, e.g. 'in,flow' propagating to 'in flow'. To provide more detail an example set of paths queried for a tank model is shown in appendix A.2. The resulting output from the QUEEN system for these queries is shown in appendix A.3. The unit model these queries were generated for is shown in appendix A.1.

To verify type (2) shortest paths Equipment Model Builder compiles a list of faults from the fault linked to deviation arcs within the model. Equipment Model Builder also creates a list of consequences contained in the deviation linked to consequence arcs within the model. Consequences contained in the fault linked to consequence arcs are not added to the list as the shortest paths propagating to these consequences are known. Queries are prepared to test for the shortest paths between each of the faults in the fault list and each of the consequences in the consequence list. The paths queried for the pipe model would be 'partly blocked' propagating to 'possible rupture' and 'leak into vacuum system' propagating to 'possible rupture'.

For type (3) paths queries are written to test for shortest paths from each of the faults in the fault list to each of process variable deviations at each of the boundary ports. Examples of some of the paths queried for the pipe model are 'partly blocked' propagating to 'in,pressure', 'leak into vacuum system' propagating to 'in,pressure' and 'leak into vacuum system' propagating to 'out,temperature'.

To test type (4) paths queries are prepared to test for shortest paths between each of the process variable deviations at each of the boundary ports and each of the consequences in the consequence list. Examples of paths queried for the pipe model would be 'in,pressure' propagating to 'possible rupture', 'in,temperature' propagating to 'possible rupture' and 'out,pressure' propagating to 'possible rupture'.

As all possible shortest paths within the model are tested for, the user is able to see where paths do not exist as well as where they do. The omission of an arc may mean that a path does not exist where the user might expect to find one. QUEEN returns the shortest path and (where relevant) the effect of this path. The effect is not relevant for type (4) paths. The value of the deviations propagating into the unit's boundary ports in these paths is not known. Therefore it is not known whether these deviations will have an effect (i.e. cause a consequence) or not. QUEEN's output enables the user to check that both the shortest path and the effect are correct.

The technique for verifying the shortest path is intended to be the final verification procedure performed upon the model. Errors which could result in missing or incorrect

shortest paths such as deviations with no effect and deviations without causes are detected prior to this test to reduce the number of problems found. The model at this stage should be complete and correct to the best of the user's knowledge. The user tests the unit model using the technique for verifying the shortest path. If any errors are found within the model the user will correct them and test the model again. A cycle of testing and correcting the model is carried out until the user is satisfied with it.

6.2.1.2. A Limitation with the Detection Technique

A limitation with this technique for verifying that the shortest path within the unit model leads to correct model behaviour is that it is exhaustive. All possible paths within the model are tested for. This results in a large numbers of queries generated and a large amount of information for checking. Appendix A.3. gives an indication of the size of the output for a tank model for queries testing the existence of shortest paths between boundary ports. Possible ways of simplifying the results would be:

- to list positive output (where paths exist) in a separate file to negative output;
- to allow the user a choice of which results to view;
- to limit the pairing of variables;
- to write the output more concisely.

The user may not wish to view all of the results generated. The user may only wish to look at paths which contain more than two arcs or those containing a certain process variable deviation. Looking at paths containing two or more arcs would detect errors caused by the unforeseen interaction of arcs, which the user may feel to be the most likely source of error. For example, the arc from the pipe model '([fault,'leak into vacuum system',vacuum], +, [pipe,in,pressure])' is correct, as is the arc '([pipe,in,pressure], +, [pipe,out,pressure])'. However the these arcs lead to the path:

$$\text{leak into vacuum system} \xrightarrow{+} \text{in,pressure} \xrightarrow{+} \text{out,pressure}$$

The effect of this path is an increase in 'out,pressure'. This effect may not be intended to occur within the model.

The user may feel it is only necessary to view paths containing a certain process variable. For example, if a heatexchanger were being modelled the user might decide that only the shortest paths containing the process variable deviation temperature were of interest. The other paths might be assumed to be correct. If a blanketed vessel model

were being created by extending a closed vessel model the user might wish to limit the results to those shortest paths containing pressure. These paths would be present as a result of new arcs added when the closed vessel model was extended. Other shortest paths within the blanketed vessel model would already have been verified as the arcs containing these paths would have been copied from the closed vessel model. The closed vessel model would have been verified when it was created. The user may also wish to condense the results viewed in other ways. Providing a choice would allow the user to restrict the output to those results of interest.

Shortest paths queried between pairs of variables could be limited to possible paths. For example, paths propagating from the process variable 'level' need not be tested. A deviation will not propagate from the process variable 'noFlow' to the variable 'flow'.

To write the output more concisely instead of returning:

There is no path between [tank1,in1,pressure] and [tank1,in1,temperature]

There is no path between [tank1,in1,pressure] and [tank1,in1,concentration]

There is no path between [tank1,in1,pressure] and [tank1,in1,level]

There is no path between [tank1,in1,pressure] and [tank1,in1,flow]

There is no path between [tank1,in1,pressure] and [tank1,in1,noFlow]

There is no path between [tank1,in1,pressure] and [tank1,in1,reverseFlow]

the output could return:

There is no path between [tank1,in1,pressure] and [tank1,in1; temperature,concentration, level, flow, noFlow and reverseFlow].

6.2.2. Preventing Illegitimate Attributes

The port names allowed for the model are defined by the user. Any ports found within the model that are not members of the set of names defined by the user are illegitimate attributes. This may occur when the user deletes or changes a port name. Equipment Model Builder deletes any arcs within the model containing the port which the user has deleted or changed.

Equipment Model Builder creates unit models for the QUEEN (Chung, 1993) expert system. The QUEENs syntax places some constraints upon the values allowed within the model. The parent and unit names of the model may not contain any spaces. In order for QUEEN to function correctly the parent name of the model must be different to the unit name. Equipment Model Builders user interface prevents the user from inputting values into the model which violate these constraints. The user is requested to enter another value if an incorrect value is entered.

Equipment Model Builders user interface also prevents the user from entering an arc containing a deviation which propagates through a port to influence itself at the same port, e.g. ([in,flow], +, [in,flow]).

The following set of values are allowable as model influences: +,-, ++, --,+++ and - - -. When the user indicates he wishes to input an influence Equipment Model Builder presents with a list of influences to chose from. This ensures that no values outside of this list can be entered. For the process variables 'noFlow' and 'revFlow' only the influence values '++' and '- -' are valid. When influences for these process variables are being entered Equipment Model Builder will present the user with a list of only these two values. For fault linked to deviation arcs only the influence values '+' and '-' are valid. When influences are being entered on the 'Fault->Deviation' form Equipment Model Builder lists only these two values.

A file describing the plant is created for Equipment Model Builder using AutoCAD. The file output from AutoCAD should be checked to ensure that it is in a correct format to be read into Equipment Model Builder. The syntax of the file should be correct, e.g. all the brackets and commas within the file should be correctly matched. This will ensure that the modified plant description files that Equipment Model Builder creates to be input into QUEEN are correct. All the units referred to within the file should either be linked to a plant outlet or be linked to other units from which a link to a plant outlet can be traced. This will prevent search algorithms within Equipment Model Builder from looping infinitely. The unitport names referred to within the AutoCAD file should be identical to those within the models in Equipment Model Builders unit library in order for QUEEN to function correctly.

6.2.3. Identifying Wrong Information

Unreferenced attributes might be present because the arcs containing them are wrong. There may be no propagation path to or from the unreferenced attribute because it should not occur. As the deviations with no effect, deviations without causes and undetected faults may be wrong Equipment Model Builder does not force the user to link them. When a model is saved the user is simply informed if any exist.

As what comprises an accurate model is not known it is impossible to say which of the pieces of information it contains might be wrong. The user might make errors when inputting information. For example, the arc '([fault,leak], +, [in,flow])' might be input as '([fault,blockage], +, [in,flow])'. This mistake cannot be detected by checking the internal

structure of the model as no unreferenced attributes or conflicting information may result from it. The only technique by which this kind of error might be found is to test the model within a simple system to see if it behaves in the way expected. A malfunctioning model will indicate an error is present. The onus is on the user to trace which error is causing the malfunction. This technique should be performed as a final test after Equipment Model Builder has carried out all its verification procedures on the internal structure of the model. Otherwise extra work would be given to the user in tracing errors which could have been more easily detected by tests on the internal model structure.

6.3. Conciseness

A concise model has no redundant, duplicated or superfluous information. No model can be assumed to be totally concise. As the set of all the possible functions that the model might be required to perform is unknown, whether any of the information within the model is superfluous to requirements is also unknown. It is not known what these requirements might be. A model which is not concise is confusing to the user. The user would find the model hard to understand. Duplication within the model would allow errors to be easily made.

This sub-section will describe how Equipment Model Builder tests for redundant information and prevents the addition of duplicated information to the model. Unreferenced attributes (deviations with no effect and deviations without causes) might be present because the information they contain is redundant.

Equipment Model Builder's interface contains checks to prevent the addition of duplicated information to the model. The user is prevented from adding duplicated port names to the model. This ensures that all port names are unique. When the user defines a module the user interface checks that the identifiers of the modules constituent units are not duplicated.

The prevention of duplicate ports within a unit model and duplicate identifiers within a module model allows a user interface check to be made preventing the addition of duplicate arcs to a model.

Equipment Model Builder tests for duplication within the lists of fault, conditions and consequences. In spite of this test duplication within these lists and hence within the model created could still occur. The reason for this is the wide variety of ways of conveying the same meaning within the English language. For example, the fault 'upstream fire' could also be listed as 'fire upstream', 'conflagration upstream' or

'upstream conflagration'. The development of a natural language processor would be required to prevent duplication of this kind. This is outside the scope of this thesis.

6.4. Verification Testing

To test the effectiveness of the major verification techniques described above a series of test models has been developed. In order to ensure thorough checking testing was done on both unit and module models. The techniques tested are: verifying propagation paths; detection of deviations without causes; detection of deviations with no effect and verifying the shortest path. To avoid creating complicated models separate model series were developed to test each of the verification methods. The models were constructed to test if the verification techniques would correctly detect the following types of propagation paths:

- propagation of deviations between different process variables;
- propagation when conditional arcs are present;
- propagation under all the influence (sign) types.

To complete verification testing propagation paths were removed from the models to show that Equipment Model Builder could detect missing propagation paths. For example, a simplified version of the model built to test the verification of propagation paths is shown:

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   proplinks info [

   %propagation
   arc([in1,flow], +,[out1,flow])
   ]]
).
```

This model shows that a propagation path is detected between 'in1 flow' and 'out1 flow'. Removing the arc '([in1,flow], +,[out1,flow])' results in 'flow in1' being placed in the unlinked boundary deviation list. This shows that Equipment Model Builder has detected that there is no propagation path for the variable flow from the port in1 through the model. Examples of the test models developed are listed in appendix B.

6.5. Conclusions

Verification is necessary to detect modelling errors which may give rise to wrong results when the models are utilised. A series of verification techniques for signed directed graph models has been described. However, some modelling errors may still remain as the

expert may be unaware what information is missing or incorrect. As a final verification technique the models should be tested in a simple system to show that they are behaving in an expected manner and are satisfactory for their intended purpose.

In this thesis verification methods have been split into three categories: those that check for completeness, correctness and conciseness. The verification methods used in Equipment Model Builder have been discussed within the context of these categories.

Main causes of incompleteness were found to result from:

- unreferenced attributes;
- missing propagation paths.

Correctness is checked for by looking for conflicting information, preventing the entry of illegitimate attributes into the model and identifying wrong information. Conflict may arise within a SDG model as a result of multiple propagation paths between two nodes within the model.

Conciseness is checked for by testing for redundant information and preventing the addition of duplicated information to the module.

The development of models to evaluate the verification methods has been described.

7. Case Studies

This chapter presents two case studies in order to assess whether:

1. the unit models created by using Equipment Model Builder are sufficient to be of use;
2. the tool's user interface is easy to use;
3. the verification techniques described in this thesis are effective;
4. the modular approach developed by this thesis removes ambiguous inference.

The application QUEEN (Chung, 1993) was chosen to demonstrate the use of the models as it is an expert system utilising SDG unit models to which access was available. QUEEN's HAZOP emulation system was employed to show the models in use. HAZOP emulation was used in preference to fault tree analysis as a complete HAZOP assessment allows the entire plant model to be thoroughly tested. Fault tree analysis only considers those parts of the plant model affected by certain faults.

For each test case two sets of input data were prepared. For the first set of data a plant description was created using AutoCAD. Models of the plant units were built using Equipment Model Builder's user interface and added to the tool's library. The plant description was supplied to the tool. A modified plant description file and a file of unit models occurring within the plant was created by the tool to form the input for QUEEN. The second set of data was prepared in the same way but for this set the modular approach was applied. The models and modules were verified after they were created. HAZOP emulation was applied to both sets of input. Modified plant descriptions, files of models occurring in the plants and examples from the HAZOP results for both sets of input data are given in appendix C for the first test case. Appendix D contains this data for the second test case.

This chapter begins by describing the two test cases. Further sections consider how Equipment Model Builder performed when creating models for the case studies. Equipment Model Builder's user interface, verification techniques and modular approach are evaluated. Limitations with the tool are described. Improvements are suggested which could be implemented if more time were available.

7.1 Plant Descriptions

The test cases described are public domain examples of plant systems. The first case consists of the purification section of a plant producing benzene (Wells and Seagrave, 1976). The second case is an olefin dimerisation plant which has been used in a well-

known example of an application of the HAZOP methodology (Lawley, 1974). The plants used as test cases were chosen to contain recycle loops and divider/header combinations in order to assess the modular approach.

7.1.1. Benzene Purification System

This purification system forms part of a plant producing benzene by the catalytic dehydroalkylation of toluene (Wells and Seagrave, 1976). The test case was restricted to the purification section of this plant in order to provide a relatively small example. The plant description used is given in figure 7.1. The distillation column (T101) separates the toluene and benzene components in its feed. The benzene is produced as the top product and the toluene as the bottom product. The top product is condensed by cooling water in exchanger E104 and then collected in reflux drum D103. The benzene is pumped by pump P101a from the reflux drum and is divided into a reflux stream and a product stream. The product stream is cooled by cooling water in exchanger E105 before going to storage.

Pump P101b is spare. All the valves in the system have open apertures. Figure 7.1 differs from the Wells and Seagrave plant in that there is no kick-back line from the P101 pumps and control facilities are omitted. The control facilities are omitted as Equipment Model Builder is currently unable to create unit models that have control structures.

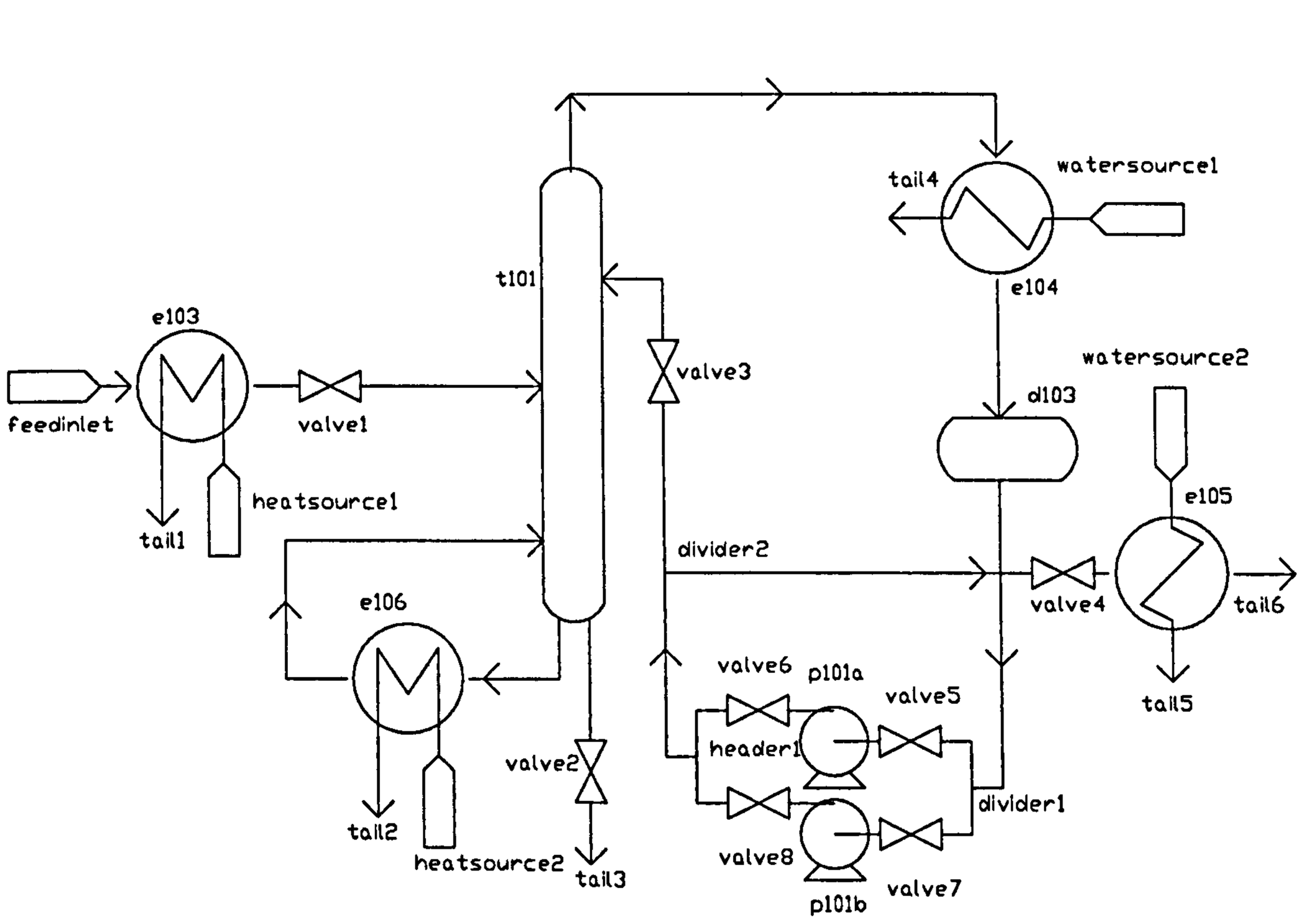


Figure 7.1. Benzene Purification System

7.1.2. Olefin Dimerisation Plant

The plant description of this test case is given in figure 7.2 and is based on figure 1 from Lawley (1974). Pumps J1a and J2a are working, pumps J1b and J2b are spare. Valves 5, 6, 9, 13 and 14 have closed apertures. All other valves in the plant have open apertures. Notable differences between this plant description and Lawley's example are the omission of kickback lines from the J2 pumps, the pressure relief valve on the heat exchanger and of control facilities for the plant.

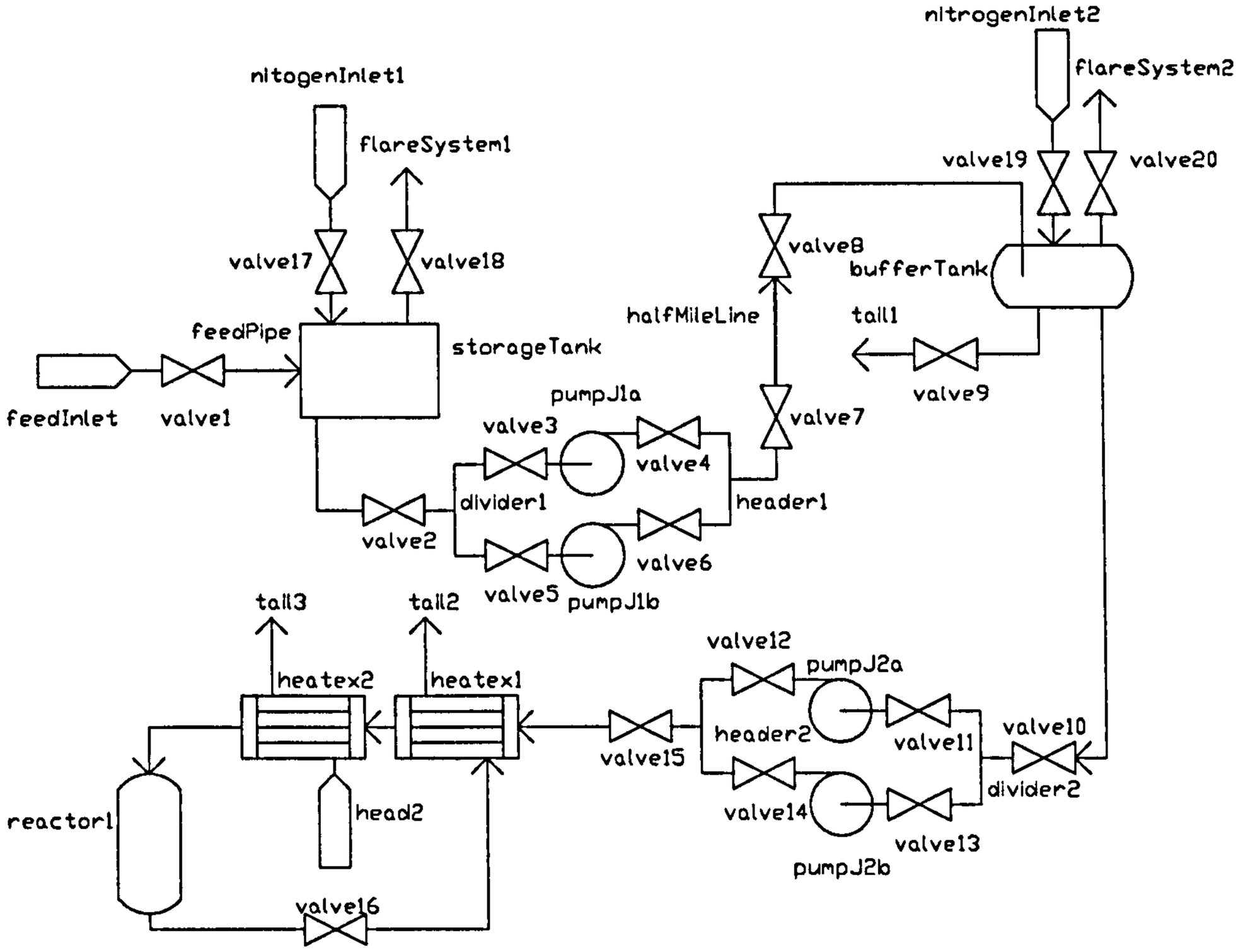


Figure 7.2 Olefin Dimerisation Plant

7.2. Evaluation of the User Interface

To thoroughly evaluate the user-friendliness of the interface a full set of user trials would be required. However, no major difficulties were encountered with the interface when creating models for the case studies. The interface achieved its objective of being simple to use. Although it is a difficult thing for the designer herself to judge, the interface was believed to be reasonably intuitive.

Compared to the original method of creating models by using a text editor it was found to be easier to use Equipment Model Builder. It was considerably easier to create arcs comprising the models by clicking on a grid element with the mouse or by choosing

nodes from a menu than by typing in the whole arc as in the previous method. It was possible to create some models by modifying similar models, saving time and work. For example, when creating models for the olefin dimerisation plant described in section 7.1.2., the closedvessel model (instantiated as 'reactor1') was modified by adding further arcs and ports to describe a nitrogen blanket. This modified model formed the decantingvessel model (instantiated as 'storagetank').

A limitation with the interface was discovered. This was the limited vertical extent of the 'Fault->Consequence' and 'Deviation->Consequence' forms. (See section 4.3. for a description of the forms of the interface). This limitation became apparent when modules were created as these are large models. The limitation did not restrict the creation of modules for the case studies presented in this chapter. However if large, complex models were created the limited vertical extent of the interface forms would cause a problem which would also extend to the 'Deviation' and 'Fault->Deviation' forms. The software used to implement Equipment Model Builder does not provide the functionality to have vertical scroll. However, a method of overcoming this problem needs to be found.

Two improvements which could be made to the user interface will be considered. The first improvement would be to provide the user with a menu detailing which of the forms of the 'Deviation' and 'Fault->Deviation' windows contain arcs. Selecting a process variable from a choicebox on one of these windows displays a new form showing the arcs possessing that process variable. The 'Deviation' window has two choiceboxes containing process variable deviations. This gives 49 forms for this window. The 'Fault->Deviation' window has 7 forms. The user does not wish to search all these forms to check what arcs have been created. The majority of the forms will not contain any arcs. A menu of forms with contents would indicate which forms to view.

The second improvement would allow a partially built model to be saved. At present, to prevent a model with incomplete information being created, the user is prevented from saving partial models. However to create a large model takes quite a length of time. To enable the user to break from the task the facility to retain the information in a partially defined model needs to be provided. To avoid possible sources of error partial models would need to be clearly labelled as such and stored separately to the members of the model library.

7.3. Evaluation of the Verification Techniques

Verification of the models was found to be useful. A user would find that the verification techniques offered by Equipment Model Builder provided sufficient reason in themselves to choose to use the tool. The techniques found to be of greatest benefit were those used to check for deviations with no effect, deviations without causes (see section 6.1.1.) and the technique for verifying the shortest path (see section 6.2.1.1.). The presence of deviations with no effect or deviations without causes provided a clear indication that the model created was faulty. It was found most useful to check for the presence of deviations with no effect and deviations without causes when the model was considered to be complete to check that it was not faulty.

Verifying the shortest path presented a different view of the model, thus allowing mistakes to be detected. This technique detected the following types of errors:

- simple mistakes;
- unexpected paths;
- missing arcs.

Simple mistakes caused paths to be present where they were not expected. Examples of simple mistakes located in the models created for the case studies are:

'arc([in1,flow],+,[in2,revFlow])', should be 'arc([out,flow],+,[in2,revFlow])',

'arc([in,level],+,[out,flow])', should be 'arc([in,flow],+,[out,flow])'.

These are entry errors caused by the user clicking on the wrong grid element of the interface with the mouse or selecting the wrong variable from a choice box.

Unexpected pathways occur when arcs the user has entered into the model interact in a way which the user did not expect. For example when creating a divider model for the case studies, assume the user has entered the following arcs;

'arc([out2,flow],+,[in,flow])'

'arc([in,flow],+,[out1,flow])'

The divider model is instantiated as 'divider1' and 'divider2' in both the benzene purification system and olefin dimerisation plant. The arcs shown are both individually correct but lead to the path:

$$\text{out2,flow} \xrightarrow{+} \text{in,flow} \xrightarrow{+} \text{out1,flow}$$

The effect of this path would be for an increase in 'out2, flow' to increase 'out1,flow'. This may be not intended to occur within the model.

Missing arcs may be detected when expected propagation paths in the model are not found or when unexpected propagation paths are found. For example, if 'arc([out2,flow],-,[in,flow])' were missing from the divider model the unexpected propagation path shown above might result.

A limitation with the technique for verifying the shortest path is the amount of output that it generates. This limitation and possible ways of overcoming are discussed in section 6.2.1.2.

If time permitted the verification techniques for deviations with no effect and deviations without causes could be improved by providing the functionality to show the user which arcs they are derived from. This would allow the user to clearly see how these unreferenced attributes have arisen and provide a guide as to what steps are needed to eliminate them.

7.4. Evaluation of the Modular Approach

This section will discuss the modular approach and look at its advantages and limitations. Modules created for the case studies are detailed. For each test case, HAZOP emulation results output when the modular approach is applied are compared to those of the standard approach (appendices C and D contain the HAZOP results). The aim of the modular approach to remove ambiguities occurring due to multiple causal paths when unit models are combined was found to succeed. Details of ambiguities eliminated or replaced in the case studies by the modular approach are given in section 7.4.2.

The HAZOP analyses of the data sets to which the modular approach was applied had faster run-times than those of the standard data sets. The benzene purification system had a run-time of 43.10 seconds on a SPARC station LX for HAZOP analysis of the data to which the modular approach was applied. The run-time for the standard data set was 275.25 seconds. The olefin dimerisation plant had a run-time of 186.68 seconds for HAZOP analysis of the modular data standard data set and 1522.37 seconds for the standard data set. The reason for this was that the data sets to which the modular approach was applied contained fewer models than the standard data sets. This was because several models had been combined to form modules. This meant that the propagation paths were shorter when the modular approach was applied to the plant systems and there were fewer paths to explore, resulting in faster analysis times.

The results of the HAZOP analyses to which the modular approach was applied were of a similar length to the results generated by the standard approach. The report length for the benzene purification system was 4778 words for modular approach and 5220 words for the standard approach. The length of the result report for the olefin dimerisation plant was 8390 words when the modular approach was applied and 7674 words for the standard approach. The two approaches produced results of similar complexity.

If appendices C and D are viewed it will be seen that there are further differences between the HAZOP analysis results of the modular data sets and the standard data sets of the case studies other than the errors described in the following sub-section 7.4.2. The differences arise due to the result filtering mechanism employed by QUEEN. This mechanism reduces the amount of output.

7.4.1. Module Creation

This sub-section will describe the modules created for each of the case studies in turn. It will be seen that one of the modules created for the first case study is re-used in the second case study where it is used twice. Two limitations found with the creation of modules will be discussed.

When the modular approach is applied to the plant description of the benzene purification system Equipment Model Builder first identifies a divider/header combination consisting of units: divider1, valve5, pump101a, valve6, valve7, pump101b, valve8 and header1 (see figure 7.1). This is because the tool first searches for divider/header combinations before searching for loops. A module for this unit grouping is specified and substituted into the plant system. Next, Equipment Model Builder identifies a recycle loop consisting of the distillation column T101 and the heatexchanger E106. A module for this is specified and substituted into the plant description. Finally, the tool identifies an outer recycle loop consisting of units: cooler E104, reflux drum D103, divider2, valve3 and the divider/header combination and recycle loop modules already identified. An outer module for this loop is specified. Another module substitution is carried out, replacing the units of the outer recycle loop (including the two inner modules).

When the modular approach is applied to olefin dimerisation plant Equipment Model Builder first identifies a divider/header combination consisting of units: divider1, valve3, pumpJ1a, valve4, valve5, pumpJ1b, valve6 and header1 (see figure 7.2). This unit grouping is identical to the divider/header combinations found in the benzene purification plant. A module already exists within the tools library for this divider/header

combination. The module is re-labelled with the identifiers of the unit grouping located in the olefin dimerisation plant and substituted into the plant description. Next, Equipment Model Builder identifies a divider/header combination consisting of units: divider2, valve11, pumpJ2a, valve12, valve13, pumpJ2b, valve14 and header2. This unit grouping is the same as the first unit grouping detected. The existing module is re-labelled with the identifiers of the second unit grouping and substituted. Finally, Equipment Model Builder identifies a loop consisting of units heatex1, heatex2, reactor1 and valve16. A module for this loop is specified and substituted into the plant system.

The first limitation when a module is created is that Equipment Model Builder generates a long list of unlinked module faults. These occur when arcs are deleted to remove potential ambiguities when unit models are combined to form a module (see section 5.2.2.3). The user must spend a large amount of time determining the effects of these faults. Fault linked to deviations and faults linked to consequences arcs must be created for these unlinked faults or the faults must be deleted if their effects are not considered to be important.

The second limitation is that some arcs that should be deleted upon modularisation are retained. Upon modularising fault linked to deviation arcs of the units in which the deviation affects a port which is linked to a consequence are retained (see section 5.2.2.1.). Deviation linked to deviation arcs of the units in which the initiating deviation occurs at a boundary port of the module and the caused deviation affects a port which is linked to a consequence are kept too. The arcs retained need to be restricted to those arcs in which the deviation affects a port which is linked to a consequence for that same deviation.

Consider the following example of the type of error currently arising. Assume a module contains the arc '([deviation,[moreLevel,unit1_liquid]],+,[consequence,'overfilling'])'. This would allow an arc '([fault,'unit1 leak to environment'],-,[unit1_liquid,level])' to be retained as this is a fault linked to deviation arc in which the deviation affects a port which has a consequence. However an arc '([fault,'external fire'],-,[unit1_liquid,temp])' would also be wrongly kept within the module.

7.4.2. Comparison of Results

For each test case, the results of the HAZOP analysis for the set of input to which the modular approach was applied are compared to those for the standard set of input data. Errors which occur in the results of the standard set of input data due to ambiguous

inference are tabulated. Only the causes of deviations are shown as no erroneous consequences due to ambiguous inference were found. How these errors are eliminated or correctly replaced when the modular approach is applied is explained.

No errors due to ambiguous inference were found when analysing the recycle loops of the benzene purification plant. Errors were only found to result from the divider/header combination. For this reason a second case study was carried out on a larger plant, the olefin dimerisation plant. This was in order to demonstrate that the modular approach does remove ambiguities caused by recycle loops. Limitations discovered when the modular approach was applied to the benzene purification system are detailed.

7.4.2.1. Benzene Purification System Results

Error no.	Deviation	Cause
1	D103 less flow out1	divider1: branch2 partly blocked
2	D103 more level liquid	divider1: branch2 partly blocked
3	P101b less flow out	valve8: thermal expansion of contents
4	P101b more flow out	valve8: leak to environment P101b: spare unit turned on
5	P101b more pressure out	valve8: thermal expansion of contents P101b: spare unit turned on
6	valve8 no flow out	header1: branch2 completely blocked
7	valve8 less flow out	header1: branch2 partly blocked
8	valve8 more flow out	valve8: open or passing

Table 7.1 Errors in the HAZOP Analysis of the Standard Result Set for the Benzene Purification Plant.

Error numbers 3 to 8 shown in table 7.1 occur with the divider/header combination of the benzene purification system. This unit grouping consists of plant units: divider1, valve5, pump101a, valve6, valve7, pump101b, valve8 and header1 (see figure 7.1). Errors 1 to 8 may not take place as the plant line to which they refer has no flow. When the modular approach is applied to the plant system these errors are eliminated.

Two limitations discovered with the modular approach while undertaking this case study will now be discussed. The first limitation concerns the sequence in which Equipment Model Builder searches a plant. This sequence depends upon the order in which the outports of any splitter units within the plant are written in the plant description (see section 5.1.1.1.). The search sequence affects the order in which loops are detected. For simple loop structures this is not important. However for nested loops and loops

sharing units, such as the plant fragment located in the benzene purification system, the order in which loops are detected is of relevance.

How this applies to the benzene purification system will be detailed. The distillation column T101 is a splitter unit. If its outports were written in a different order, instead of detecting the loops described above, Equipment Model Builder would first identify a loop consisting of T101, cooler E104, the divider/header module, divider2 and valve3. The user would be asked to specify a module for this which would be substituted into the plant description. Equipment Model Builder would then identify an outer loop consisting of the loop already identified and heatexchanger E106. The user would be asked to specify a module for this and another module substitution would be carried out.

Assume a plant fragment identical to the one found in the benzene purification system was located in another plant. This plant fragment would only be recognised as being comprised of existing modules in the tool's model library if the plant description of the other plant was written in the same order as that of the benzene purification plant. This shows that modules specified for nested loops and loops sharing units are of limited re-use.

The second limitation is that Equipment Model Builder requires a user to specify the inner modules of plant fragments composed of complex structures. Complex structures are loops nested inside loops, divider/header nested inside loops, loops sharing units and combinations of these. Specifying the inner modules is necessary to avoid loss of information about the unit grouping which comprises the inner module. When the outer module is specified causal paths existing between or within the units of the module are removed. Arcs created for the inner module may have to be re-specified when the outer module is created. This leads to duplication of work for the user.

7.4.2.2. Olefin Dimerisation Plant Results

Error no.	Deviation	Cause
1	storagetank less flow out1	divider1: branch2 partly blocked
2	storagetank more level liquid	divider1: branch2 partly blocked
3	divider1 no flow out2	divider1: branch 2 completely blocked
4	valve5 more flow out	pumpJ1b: leak to environment
5	pumpJ1b less flow out	valve6: thermal expansion of contents
6	pumpJ1b more flow out	valve6: leak to environment pumpJ1b: spare unit turned on
7	pumpJ1b more pressure out	valve6: thermal expansion of contents pumpJ1b: spare unit turned on
8	valve6 no flow out	header1: branch 2 completely blocked
9	valve6 less flow out	header1: branch2 partly blocked
10	valve6 more flow out	valve6: open or passing
11	buffertank less flow out1	divider2: branch2 partly blocked
12	buffertank more level liquid	divider2: branch2 partly blocked
13	divider2 no flow out2	divider2: branch2 completely blocked
14	valve13 more flow out	pumpJ2b: leak to environment
15	pumpJ2b less flow out	valve14: thermal expansion of contents
16	pumpJ2b more flow out	valve14: leak to environment pumpJ2b: spare unit turned on
17	pumpJ2b more pressure out	valve14: thermal expansion of contents pumpJ2b: spare unit turned on
18	valve14 no flow out	header2: branch2 completely blocked
19	valve14 less flow out	header2: branch2 partly blocked
20	valve14 more flow out	valve14: open or passing
21	reactor1 less level liquid	heatex1: holed heatexchanger

Table 7.2 Errors in the HAZOP Analysis of the Standard Result Set for the Olefin Dimerisation Plant.

In table 7.2 errors number 3 to 10 form an identical set to errors 13 to 20. This is because these two error sets occur within identical divider/header combinations located in the olefin dimerisation plant. Error numbers 3 to 10 occur with the divider/header combination consisting of plant units: divider1, valve3, pumpJ1a, valve4, valve5, pumpJ1b, valve6 and header1. Errors 13 to 20 occur within the combination consisting of units: divider2, valve11, pumpJ2a, valve12, valve13, pumpJ2b, valve14 and header2. Error numbers 1 to 20 may not take place as the plant lines to which they refer have no

flow. When the modular approach is applied to the plant system these errors are eliminated. Although these divider/header combinations are identical to the one located in the benzene purification system the sets of errors occurring in these unit groupings differ to the error set resulting from the divider/header combination located in the benzene purification plant. This is because of the result filtering mechanism employed by QUEEN.

Error number 21 arises as a result of ambiguous inference occurring within the unit grouping comprising a loop. The loop consists of units heatex1, heatex2, reactor1 and valve16. This loop is a heating recycle loop not a true recycle loop, hence it might be supposed that ambiguous inference would not take place. However, when the fault 'heatex1: holed heatexchanger' shown in error 21 occurs, this allows the loop to operate as a mass recycle loop. This means that ambiguous inference may take place and an error results. 'Heatex1: holed heatexchanger' causes the flow out of the heating loop to be decreased. This cannot lead to the deviation 'reactor1 less level liquid' shown in table 7.2 as the decrease in flow out increases the overall volume of liquid within the heating loop. The HAZOP analysis of the data set to which the modular approach was applied replaces this erroneous result and shows the fault 'heatex1: holed heatexchanger' causing the deviation 'reactor more level liquid'.

7.5. Conclusions

These case studies show that the models created by using Equipment Model Builder are sufficient for a purpose. In order to show this the models were used by the expert system QUEEN to perform HAZOP emulation. However, the models could be utilised by any application which requires SDG models. The case studies also show that the models may be re-used, both within the same plant and within different plant systems. The tool's user interface was found to be simple and easy to use. Despite some limitations the verification techniques were found to be useful.

The modular approach developed by this thesis removed ambiguities which occurred due to multiple causal paths when unit models were combined. Applying the modular approach eliminated errors due to this ambiguous inference in the HAZOP analysis results of the case studies. Using the modular approach produced faster run-times for the HAZOP emulation and did not create larger or more complicated outputs of results.

Modules specified for nested loops and loops sharing units were found to be of limited re-use. The user had to duplicate work when specifying modules for complex structures. For these reasons automatically identifying these types of structures may not

be desirable. To remove ambiguities it is still necessary to apply the modular approach to these structures, but it may be better for the user to define modules. The user-defined modules would be completely re-usable. There would be no need to specify inner modules to avoid loss of information about the unit groupings comprising the inner modules as information about the structure's units is supplied by the user.

8. Conclusions and Future Work

The research described in this thesis develops a method for creating and testing equipment models for process plants. A computer-aided modelling tool, Equipment Model Builder, has been constructed to demonstrate this method. The models created describe how the different plant equipment behaves in qualitative terms. The models use the SDG representation and may be utilised by applications requiring a library of unit models of this type. The models are used to model fault propagation in process plants. The SDG representation has been utilised by computer aids for fault tree synthesis, HAZOP emulation and diagnosis.

This chapter will describe the contributions made by this thesis. Limitations of the work will be considered and possibilities for future work discussed.

8.1. Contributions

A method has been developed to construct unit models simply and correctly. Three main contributions have been made by this thesis. These are:

1. Creating a front end interface enabling an engineer to enter modelling information directly;
2. Developing a novel modular approach which provides a methodology to remove ambiguities caused by multiple causal paths when unit models are combined to form a system;
3. Developing verification techniques for the models created.

8.1.1. The User Interface

The models were previously built by engineers writing the models in an application specific format. An engineer unfamiliar with the application would find this difficult. An alternative is to use a knowledge engineer to gather information from the expert and convert it into the application format. This is expensive and both methods would take up a lot of the expert's time. A front end interface has been designed and built which enables an expert to enter information directly without needing to understand details of the model format or the application. This interface incorporates ideas from the knowledge acquisition field in order to produce a tool that is simple to use.

Knowledge acquisition tools provide aids to allow the expert to achieve a structured input of information without knowledge of the internal format of the expert

systems that these types of tools are acquiring information for. Knowledge acquisition tools were surveyed in order to determine their desirable features in relation to qualitative modelling. The desirable features identified are:

(i) a differentiation facility;

(ii) verification testing;

(iii) an explanation facility;

(iv) the ability to rank information;

(v) a user interface.

Equipment Model Builder possesses:

- a user interface;
- verification testing.

Differentiation facilities, an explanation facilities and the ability to rank information are implemented as part of the user interface.

The features need to take account of the structure of the unit models created. A unit model is composed of structural information (the unit name, parent name and the inports, outports, unit ports and attribute slots) and fault propagation information (the propLinks and conditionLinks slots). The information in the propLinks and conditionLinks slots is defined together. There are four types of propLinks and conditionLinks arc:

(i) deviation linked to deviation;

(ii) fault linked to deviation;

(iii) fault linked to consequence;

(iv) deviation linked to consequence.

The interface has four sub-windows which allow fault propagation information to be defined. There is one sub-window for each category of arc. This enables the user to differentiate between the different arc types.

The user interface provides drop-down menus which give access to libraries. For example, there are libraries containing all the faults, consequences and process conditions for the models. Each library is displayed using a list browser. This allows the user to differentiate between the information contained within the separate libraries.

The interface forms provide an explanation facility showing what information a model contains. Explanation is also given during the modularisation process. The user is provided with a description of the unit grouping identified as leading to ambiguities.

On starting a model creation session a 'main' window is shown. The main window provides a set of boxes which allows model structural information to be entered. By accessing the main window first the interface ranks the order in which the model is created.

The user interface provides list browsers. Information may be edited or copied. The interface allows well-defined, uniform models to be built. The models created were found to be sufficient for their purpose. The models may also be re-used. With minor modifications these models may also be used by a range of applications.

8.1.2. The Modular Approach

Unit-based qualitative modelling can lead to incorrect or ambiguous inference. The type of ambiguities considered by this thesis are those due to multiple causal paths. Problems can occur when one path has a contradictory effect compared to another. The result of the addition of the effects of these paths cannot be determined unambiguously. The methodology developed here identifies some situations were these types of ambiguities may arise. The situations identified occur when groups of unit models are found to comprise divider/header combinations or recycle loops. A new modular approach has been developed to overcome the problem caused by these ambiguities. The advantage of this modular approach is that the user is able to see clearly what influences are described by the modules.

This modular approach takes place in three stages: identification, specification and substitution. Equipment Model Builder takes a file describing a plant as input. Within this process plant the groupings of units together with the connections between them that could lead to ambiguities are identified. Each unit grouping found is checked to see whether a module for it already exists within the model library. If there is no existing module which describes the occurrence identified a new one needs to be specified by the user. In the specification stage the units of the plant fragment are amalgamated to form a module. Making a plant fragment into a module removes the causal paths that lead to ambiguities or incorrect behaviour. Finally, the module is substituted into the plant system, replacing the individual units that make up the module. The plant description created is modified to describe the module. A model of the module is placed into an output file which contains the plant's component unit models. In addition to those automatically identified, other groupings of units which lead to ambiguities may occur. For these groupings of units the user is provided with the functionality to define modules.

8.1.3. Verification Techniques

All forms of modelling errors occur due to omissions and mistakes. Verification is necessary to detect modelling errors which may give rise to wrong results when the models are utilised. The method developed presents a series of verification techniques for signed directed graph models. Verification is defined in this thesis as ensuring that the internal structure of each model is, as far as possible, complete, correct and consistent. Verification should also ensure that the model's behaviour is plausible.

Verification consists of a number of techniques. The choice of techniques used will depend on the system being verified. The verification techniques must ensure as far as possible that the model is complete, correct and consistent. In order for a model to be consistent it must be correct and concise.

- A complete model is free from missing information. It contains sufficient information to be able to function in all possible situations that arise in the application. All the information that ought to be in the model is contained in the model structure.
- A correct model is an accurate representation and contains no wrong information. It has no conflicting information or illegitimate attributes. An illegitimate attribute is one which does not occur within the set of attribute values allowed for the model. A correct model will function correctly.
- A concise model has no redundant or duplicated information. It has no information which is superfluous or unnecessary. A model which is not concise may lead to additional processing and may be ambiguous.

Some of the techniques described are purely verification techniques whereas others may be thought of as part of the tool's user interface, duplication checking or explanation facilities. A series of models has been developed to test the verification techniques used.

Incompleteness can occur due to a number of reasons. Main causes of incompleteness result from:

- unreferenced attributes;
- missing propagation paths.

Unreferenced attributes occur within SDG arcs when an initial node 'X' of the arc can never be effected or an influenced node of the arc 'Y' can never cause an effect. The unreferenced attribute 'X' is a process variable deviation without a cause. Causes may occur within the unit model or the unit model may have the potential for causes to propagate into it from other units within the plant. 'Y' is a process variable deviation with no effect. Deviations may cause effects within the unit model or the unit model may

allow a deviation to propagate out of it to cause an effect within another plant unit. For a given model, Equipment Model Builder maintains lists of deviations with no effect and deviations without causes.

In order for unit models to function correctly within a plant model deviations in process variables will need to be able to propagate through them. This means that process variable deviations need to be able to propagate from a units inports to its outports and from its outports to its inports. Exceptions to this are models for the source and outlet of the plant. Some unit models may not propagate all deviations. For example, an open tank will not propagate an increase in flow from its outport to its inport. However most units will propagate most deviations.

Equipment Model Builder identifies process variables with no propagation path through a unit model and provides the user with a list. This does not mean that the model is not complete if there are process variables with no propagation path. For example, in the case of the open tank there is no propagation path for flow from the tanks outports to its inports. It only means that the model *might* be not be complete. The list is intended to act as a memory aid for a user.

Correctness is checked for by identifying wrong information, preventing the entry of illegitimate attributes into the model and looking for conflicting information within the model. Deviations with no effect and deviations without causes may also occur because they contain wrong information. The layout of Equipment Model Builders front end interface prevents the user from entering illegitimate attributes.

Conflict may occur within the component model when there is more than one possible path through the SDG. Problems occur when one path has a contradictory effect compared to another. In order to deal with ambiguities a heuristic that is commonly used is that when there is more than one acyclic path through the SDG the shortest path is used. A method has been devised which allows the user to check that the shortest path within the component model leads to correct model behaviour. To avoid duplication of work the technique for verifying the shortest path uses the QUEEN (Chung, 1993) system. A file of queries to test the effects of the shortest paths within the model is prepared. This file and the component model are given to QUEEN. The user checks the output from QUEEN to ensure that the model functions correctly.

Equipment Model Builder checks for conciseness by testing for redundant information and preventing the addition of duplicated information to the model. Deviations with no effect and deviations without causes might be present because the

information they contain is redundant. Equipment Model Builders user interface prevents the addition of duplicate arcs to the model.

8.2. Limitations

Equipment Model Builder has been shown to be versatile with the ability to cope with many types of units and plant configurations. However, some limitations with the method have been identified. Certain limitations exist because insufficient time was available to remove them. How future work could remove these limitations is discussed in sub-section 8.3.1. Three remaining limitations are described in this section. The way to solve these limitations is unclear. These are all limitations of the modular approach.

The first limitation is that when a module is created Equipment Model Builder generates a long list of unlinked module faults. These occur when arcs are deleted to remove potential ambiguities when unit models are combined to form a module (see section 5.2.2.3). The user must spend a large amount of time determining the effects of these faults. Fault linked to deviations and faults linked to consequences arcs must be created for these unlinked faults or the faults must be deleted if their effects are not considered to be important. An enhancement to shorten the list of unlinked faults was suggested in section 5.2.3.2. This consisted of utilising the expert system QUEEN to determine the effects of the faults propagating through the units of the plant module. It would only be necessary to list a module fault as unlinked and delete the arcs the fault propagated through if the effects of that fault were found to result in contradictory multiple paths. However the effects of most faults might result in contradictory multiple paths. It is not known whether this method would be effective.

The second limitation concerns the modularisation of recycle loops. This thesis only considers modularising recycle loops relatively small compared to the size of the plant. Creating a module for a large loop encompassing most of the plant would entail the loss of the unit-based approach. There is no way of assessing the 'largeness' of a recycle loop. This depends on the relative sizes of the loop and the plant in which it is situated. To overcome this problem the user is required to state the maximum size of recycle loop to modularise.

Complex plant structures represent the third limitation. When specifying modules for complex structures containing modules within modules the user has to duplicate work. Arcs created for the inner module may have to be re-specified when the outer module is

created. Modules specified for nested loops and loops sharing units were found to be of limited re-use.

8.3. Recommendations for Future Work

The first set of recommendations discussed in this sub-section looks at ways of removing limitations identified with Equipment Model Builder and some ways in which the tool may be enhanced. Time constraints have prevented the removal of the limitations. The second set considers how the method defined in this thesis may be further developed.

8.3.1. Removal of Limitations

These recommendations fall into four categories: improved interface; a correction to the method for creating models; improving the output from the verification techniques and improved software compatibility.

Improvements to the interface are:

- To implement a hierarchical structure within the tool's fault, consequence and condition libraries in order to provide the user with quick access.
- To provide lists of the faults and consequences within a model and to query the user if these lists are complete. This would serve as a further check to ensure model completeness (see section 6.1.2.2.).
- To provide a vertical scroll to enable large models to be built.
- To provide a menu of 'Deviation' and 'Fault->Deviation' forms with contents indicating which forms to view.
- To allow the information in a partly defined model to be retained.

A correction to the method for specifying modules is required to prevent some arcs that should be deleted upon modularisation being retained. Upon modularising fault linked to deviation arcs of the units in which the deviation affects a port which is linked to a consequence are retained (see section 5.2.2.1.). Deviation linked to deviation arcs of the units in which the initiating deviation occurs at a boundary port of the module and the caused deviation affects a port which is linked to a consequence are kept too. The arcs retained need to be restricted to those arcs in which the deviation affects a port which is linked to a consequence for that same deviation.

Improving the output from the verification techniques would consist of:

- Reducing the amount of output generated by the technique for verifying the shortest path (see section 6.2.1.2.).

- The verification techniques for deviations with no effect and deviations without causes could be improved by providing the functionality to show the user which arcs they are derived from.

Improvements to the compatibility are:

- The file output from AutoCAD should be checked to ensure that it is in a correct format to be read into Equipment Model Builder (see section 6.2.2.). The length of unit names within the file should be restricted to be compatible to the size of the tool's interface.

8.3.2. Future Developments

Four ways in which the method may be further developed are considered. These are:

- To construct unit models using the functional equation representation;
- To allow control structures to be modelled;
- To ensure that the user interface is as user-friendly as possible.
- To assess where the modular approach could be of most use.

Currently models are constructed using the SDG representation. As shown in section 2.1.2. these two representations are equivalent. The method could be extended to allow the user to construct unit models using either representation. This would increase the number of applications which could employ the models created by using Equipment Model Builder.

The same high level representation of the fault propagation model input into the tool's interface could be used to create two models, one using the SDG representation and one using the functional equation representation. This would mean that a single high level representation could converted to be input into applications utilising the SDG representation and applications utilising the functional equation representation. This would increase the number of applications which could use a single high level representation after it was converted to the appropriate format. The number of times each high level representation could be used during the design and commissioning of a plant would be increased. This would save time and work for the expert.

Allowing control structures to be modelled would enlarge the number of plant units which could be modelled using Equipment Model Builder and allow existing models to be given more structure. This development would be necessary to allow the tool to pass beyond a prototype stage. All functioning plant systems will contain some form of control

structure. In order to be of use the method which Equipment Model Builder is based on will need to consider control.

Control loops may lead to the presence of ambiguous inference. When considering fault propagation this may lead to further interpretations being found in addition to the true fault origin or the exclusion of the true fault origin. The work of Chang and Yu (1990) is an initial attempt to resolve this problem. Further work could look at whether it would be appropriate to extend the modular approach developed in this thesis to control loops. This would consider whether the modular approach could remove ambiguities within control loops.

This thesis has surveyed knowledge acquisition tools in order to determine their desirable features in relation to qualitative modelling. Part of this survey considered the features that a user interface should possess. Future developments could look at how to increase the user-friendliness of Equipment Model Builder's interface. Guidelines are emerging which direct the developer when designing and implementing a user interface (Hunt, 1990). How these guidelines relate to the tool could be examined. Applying these guidelines to qualitative modelling would be novel. Increasing the user-friendliness of the tool would increase its acceptance and use among users.

The modular approach developed in this thesis has been shown to be effective in eliminating the ambiguities caused by multiple causal paths. Currently recycle loops and divider/header combinations are automatically identified as unit groupings which lead to ambiguities. Work is required to determine more types of plant unit grouping which lead to ambiguities. This would enable these groupings to be automatically identified and increase the usefulness of the modular approach.

The modular approach requires extra user input in order to create the modules. Modules for complex plant structures were found to be of limited re-use. More case studies are required to assess where the modular approach could be most effective. This would be for plant fragments in which ambiguities are definitely found to occur, as opposed to where they might occur, and for commonly occurring plant fragments. This would justify the extra work involved by the modular approach.

References

Aldersey, M.L. et al. (1991) Knowledge Elicitation and Representation for Diagnostic Tasks in the Process Industries:Forms of Representation and Translation Between these Forms. In TransIChemE,Vol69, PartB,November, pp187-194.

Allen, D.J. and Rao, M.S. (1980) New Algorithms for the Synthesis and Analysis of Fault Trees. In Ind. Eng. Chem. vol 19, pp79-85.

Allen, D.J. (1984) Digraphs and Fault Trees. In Ind. Eng. Chem. Fundam. vol 23, pp175-180.

Andow, P.K. and Lees, F.P. (1975) Process Computer Alarm Analysis: Outline Of A Method Based On List Processing. In Trans. Instn. Chem. Engrs. vol 55, pp195-207.

Andert, E.P. (1992) Integrated Knowledge-based System-Design and Validation for solving problems in uncertain environments. In International Journal of Man-Machine Studies vol 36, no.2, pp357-373.

Andrews, J. and Morgan, J.M. (1986) Application of the Digraph Method of Fault Tree Construction to a Process Plant. In Reliability Engineering vol 14, pp85-106.

Andrews, J. and Brennan, G. (1990) Application of the Digraph Method of Fault Tree Construction to a Complex Control Configuration. In Reliability Engineering and System Safety, vol 28, pp357-384.

Boose, J.H. (1986) Expertise Transfer for Expert System Design.

Boose, J.H. (1989a) A survey of knowledge acquisition techniques and tools. Knowledge Acquisition vol 1 (1), pp3-37.

Boose, J.H. (1989b) Knowledge Acquisition for Knowledge-Based Systems: Notes on the State of the Art. Machine Learning Journal vol 4, pp377-394.

Boose, J.H. (1990) Knowledge Acquisition Tools, Methods, and Mediating Representations. In Motoda, R. et al. (eds). Proceedings of the First Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop: JKAW.

Boose, J.H. and Bradshaw, J.M. (1988) Expertise transfer and complex problems: using AQUINAS as a knowledge-acquisition workbench for knowledge-based systems. In Boose, J.H. and Gaines, B.R. (eds). Knowledge Acquisition for Expert Systems, Knowledge-based Systems vol 2, pp39-64.

Bratko, I. (1990) Prolog Programming for Artificial Intelligence.

Catino, C.A. et al. (1991) Automatic Generation of Qualitative Models of Process Units. In Computers chem. Engng. vol 15, pp583-599.

Catino, C.A. and Lyle, L.H. (1995) Model-Based Approach to Automated Hazard Identification of Chemical Plants. In Process Systems Engineering vol 41, pp97-109.

Chang, C. and Hwang, H. (1992) New Developments of the Digraph-Based Techniques for Fault-Tree Synthesis. In Ind. Eng. Chem. Res. vol 31, pp1490-1502.

Chang, C. and Yu, C. (1990) On-Line Fault Diagnosis Using the Signed Directed Graph. In Ind. Eng. Chem. Res. vol 29, pp290-1299.

Chung, P.W.H. (1993) Qualitative Analysis of Process Plant Behaviour. In Chung, P.W. H., Lovegrove, G. and Ali, M. (eds) The Proceedings of the Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems June 1993, pp277-283.

Chemical Industries Association Ltd. (1977) A Guide to Hazard and Operability Studies. Chemical Industry Safety and Health Council of the Chemical Industries Association.

Cooke, N.M. and McDonald, J.E. (1988) The application of psychological scaling techniques to knowledge elicitation for knowledge-based systems. In Boose, J.H. and

Gaines, B.R. (eds). Knowledge Acquisition for Expert Systems, Knowledge-based Systems vol 2, pp65-82.

Diederich, J et al. (1988) KRITON: a knowledge-acquisition tool for expert systems. In Boose, J.H. and Gaines, B.R. (eds). Knowledge Acquisition for Expert Systems, Knowledge-based Systems vol 2, pp83-94.

Eriksson, H. (1992a) A Survey of Knowledge Acquisition Techniques and Tools and their Relationship to Software Engineering. Journal of Systems and Software vol 19 (1), pp97-107.

Eriksson, H. (1992b) Domain-Oriented Knowledge-Acquisition Tool for Protein Purification Planning. Journal of Chemical Information and Computer Sciences vol 32 (1), pp90-95.

Eriksson, H. (1993) Specification and Generation of Custom-Tailored Knowledge-Acquisition Tools. Artificial Intelligence vol 172, pp510-515.

Eriksson, H. (1994) Models for knowledge acquisition tool design. Knowledge Acquisition vol 6 (1), pp47-74.

Eriksson, H. and Larses, P (1992) ALF-A: A Knowledge Acquisition Tool for Troubleshooting of Laboratory Equipment. Journal of Chemical Information and Computer Sciences vol 32 (2), pp139-144.

Eriksson, H. and Musen, M. (1993) Metatools for Knowledge Acquisition. IEEE Software vol 10 (3), pp23-29.

Eshelman, L. (1988) MOLE: A Knowledge Acquisition Tool for Cover-and-Differentiate Systems. In Marcus, S. (ed). Automating Knowledge Acquisition for Expert Systems, pp37-80.

Falkenhainer, B. and Forbus, K.D. (1990) Setting up Large-Scale Qualitative Models. In Weld, D.S. and Kleer, J. de (eds). Readings in Qualitative Reasoning About Physical Systems, pp553-558.

Fanti, M. et al. (1993) Resolving Ambiguity in Qualitative Models (Applied to Fault Propagation) through High Level Constraints. In IChemE Research Event, pp630-632, Rugby, UK.

Gupta, U.M. (1993) Validation and Verification of Knowledge-Based Systems: A Survey. In Journal of Applied Intelligence, vol 3, pp. 343-363.

Hoppe, T. and Meseguer, P. (1993) VVT Terminology: A Proposal. In IEEE Expert June, pp48-55.

Hunt, J. (1990) Human-computer interfaces for knowledge-based systems. In Price, C.J. (ed). Knowledge Engineering Toolkits, pp39-60.

Iri, M. et al. (1981) An Algorithm For Diagnosis Of System Failures In The Chemical Process. In Computers and Chemical Engineering, vol 3, pp489-493.

Jefferson, M. et al. (1995) Automated Hazard Identification By Emulation Of Hazard And Operability Studies. In Proceedings of the Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems June6-8 1995, pp. 765-770.

Kahn, G. (1988) MORE: From Observing Knowledge Engineers to Automating Knowledge Acquisition. In Marcus, S. (ed). Automating Knowledge Acquisition for Expert Systems, pp7-35.

Kahn, G. et al. (1988) A mixed-initiative workbench for knowledge acquisition. In Boose, J.H. and Gaines, B.R. (eds). Knowledge Acquisition for Expert Systems, Knowledge-based Systems vol 2, pp161-173

Kawaguchi, A. et al. (1991) Interview-Based Knowledge Acquisition Using Dynamic Analysis. IEEE Expert, Oct. pp47-60.

Kitto, C.M. and Boose, J.H. (1988) Heuristic for expertise transfer: an implementation of a dialog manager. In Boose, J.H. and Gaines, B.R. (eds). Knowledge Acquisition for Expert Systems, Knowledge-based Systems vol 2, pp175-194

Klinker, G. et al. (1988) KNACK---Report-driven knowledge acquisition. In Boose, J.H. and Gaines, B.R. (eds). Knowledge Acquisition for Expert Systems, Knowledge-based Systems vol 2, pp195-209.

Kohda, T. and Henley, E.J. (1988) On Digraphs, Fault Trees, and Cut Sets. In Reliability Engineering and System Safety vol 20, pp35-61.

Kokawa, M. et al.. (1983) Fault Location Using Digraph And Inverse Direction Search with Application. In Automatica vol 9, no.6, pp729-735.

Kumamoto, H. and Henley, E.J. (1986) Automated Fault Tree Synthesis by Disturbance Analysis. In Ind. Eng. Chem. Fundam. vol 25, pp233-239.

Kuo, D.H. et al.. (1997) A Prototype for Integrating Automatic Fault Tree/Event Tree/HAZOP Analysis. In Computers and Chemical Engineering vol 21, ppS923-S928.

Kramer, M.A. and Palowitch, B.L. (1987) A Rule-Based Approach to Fault Diagnosis Using the Signed Directed Graph. In AIChE Journal vol 33, no. 7, pp1067-1078.

Lapp, S.A. and Powers, G.J. (1977) Computer-aided Synthesis of Fault-trees. In IEEE Transactions on Reliability, April, pp2-13.

Larkin, F.D. et al. (1997) Computer-Aided Hazard Identification: Methodology and System Architecture. In IChemE Symposium Series Hazards XIII Process Safety - The Future vol 141, pp337-348.

Lawley, H.G. (1974) Operability Studies and Hazard Analysis. Chemical Engineering Progress vol 70, no.4, pp45-56.

Leone, H. (1996) A Knowledge-Based System For HAZOP Studies. The Knowledge Representation Structure. Computers and Chemical Engineering vol 20, ppS369-S374.

Major, N. (1991) CATO: an automated card sort tool. In Proc. Fifth European Knowledge Acquisition for Knowledge-Based Systems Workshop, EKAW '91, Crieff, Scotland.

Marcus, S. (1988) SALT: A Knowledge-Acquisition Tool for Propose-and-Revise Systems. In Marcus, S. (ed). Automating Knowledge Acquisition for Expert Systems, pp81-123.

Martin-Solis, G.A. et al. (1982) Fault Tree Synthesis For Design and Real Time Applications. In Trans IChemE vol 60, pp14-25.

McCoy, S.A. and Rushton, A.G. (1997) A Case Study in Qualitative Reasoning About Process Plant Hazards. In IChemE Jubilee Research Event vol 2, pp665-668, Rugby, UK.

McGreavy, C. et al., (1997) Use of a Dynamic Simulator to Improve Start-up for a MTBE Reactor and a Reactive Distillation Process. In AIChE Spring Meeting, Houston, USA., March.

Mohindra, S. and Clark, P.A. (1993) A Distributed Fault Diagnosis Method Based On Digraph Models: Steady-State Analysis. In Computers and Chemical Engineering vol 17, no.2, pp193-209.

McDermott, J. (1988) Preliminary Steps Toward a Taxonomy of Problem-Solving Methods. In Marcus, S. (ed). Automating Knowledge Acquisition for Expert Systems, pp225-256.

Mukherjee, R. et al. (1997) Classifying and Detecting Anomalies in Hybrid Knowledge-based Systems. In Decision Support Systems vol 21, pp231-251.

Musen, M. A. et al. (1988) Use of a domain model to drive an interactive knowledge editing tool. In Boose, J.H. and Gaines, B.R. (eds). Knowledge Acquisition for Expert Systems, Knowledge-based Systems vol 2, pp257-273.

Nam, D.S. et al.. (1996) Automatic Construction of Extended Symptom-Fault Associations From the Signed Digraph. In Computers and Chemical Engineering vol 20, ppS605-S610.

Neale, I. (1988) First generation expert systems: A review of knowledge acquisition methods. The Knowledge Engineering Review, vol 3 (2), pp105-136.

O'Keefe, R.M. and O'Leary, D.E. (1993) Expert System Verification and Validation: a survey and tutorial. In Artificial Intelligence Review vol 7, pp3-42.

Oyeleye, O.O. and Kramer, M.A. (1988) Qualitative Simulation of Chemical Process Systems: Steady-State Analysis. In AICHE Journal vol 34, pp1441-1453.

Parmar, J.C. and Lees, F.P. (1987) The Propagation of Faults in Process Plants:Hazard Identification. In Reliability Engineering vol 17, pp277-302.

Price, C.J. et al. (1997) Combining functional and structural reasoning for safety analysis of electrical designs. In The Knowledge Engineering Review vol 12, part 3, pp271-287.

Price, C.J. (1998) Function-directed electrical design analysis. In Artificial Intelligence in Engineering vol 12, pp445-456.

Qian, D. Q. (1990) An Improved Method For Fault Location Of Chemical Plants. In Computers and Chemical Engineering vol 14, no. 1, pp41-48.

Renard, F. et al. (1993) Knowledge Verification in Expert Systems Combining Declarative and Procedural Representations. In Computers and Chemical Engineering vol 17 (11), pp1067-1090.

Rose, P. and Kramer, M.A. (1991) Qualitative Analysis of Causal Feedback, *AIAI-91*.

Schut, C. and Bredeweg, B. (1993) Automatic Enhancement of Model Parsimony. In Proceedings of the Seventh International Workshop on Qualitative Reasoning pp.194-203 Seattle,WA. May

Schut, C. and Bredeweg, B. (1994) Supporting Qualitative Model Specification. In Proceedings of the Second International Conference on Intelligent Systems Engineering, pp37-42.

Schut, C. and Bredeweg, B. (1995) Supporting Qualitative Model Construction: · Eliminating incorrectly predicted derivatives. In Proceedings of the Nineth International Workshop on Qualitative Reasoning pp.163-172 Amsterdam,May.

Shaeiwitz, J.A. et al. (1977) Fault Tree Analysis of Sequential Systems. In Ind. Eng. Chem., Process Des. Dev. vol 16, no. 4, pp529-549.

Shafaghi, A. et al. (1984a) Fault Tree Synthesis Based on Control Loop Structure. In Chem Eng Res Des vol 62, March, pp101-110.

Shafaghi, A. et al. (1984b) An illustrative Example of Fault Tree Synthesis Based on Control Loop Structure. In Reliability Engineering vol 8, pp. 193-233.

Shaw, M.L.G. and Gaines, B.R. (1988) KITTEN: Knowledge initiation and transfer tools for experts and novices. In Boose, J.H. and Gaines, B.R. (eds). Knowledge Acquisition for Expert Systems, Knowledge-based Systems vol 2, pp309-338.

Shiozaki, J. et al. (1985) An Improved Algorithm For Diagnosis Of System Failures In The Chemical Process. In Computers and Chemical Engineering vol 9, no. 3, pp285-293.

Srinivasan, R. et al. (1997) Integrating Knowledge-based and Mathematical Programming Approaches for Process Safety Verification. In Computers and Chemical Engineering vol 21, ppS905-S910.

Srinivasan, R. et al. (1998) Safety Verification Using a Hybrid Knowledge-Based Mathematical Programming Framework. In AIChE Journal vol 44, no. 2, pp361-371.

Tarifa, E.E. and Scenna, N.J. (1997) Fault Diagnosis, direct graphs, and fuzzy logic. In Computers and Chemical Engineering, vol 21, ppS649-S654.

Tepandi, J. (1997) Quality Assurance of Knowledge-based Systems. In Engineering Applications of Artificial Intelligence vol 10 (3), pp231-242.

Ulerich, N.H. and Powers, G.J. (1988) On-Line Hazard Aversion and Fault Diagnosis in Chemical Processes:The Diagraph + Fault-Tree Method. In IEEE Transactions on Reliability vol 37, pp71-177.

Umeda, T. et al. (1980) A Graphical Approach To Cause and Effect Analysis Of Chemical Processing Systems. In Chemical Engineering Science vol 35, pp2379-2388.

Vaidhyanathan, R. and Ventkatasubramanian, V. (1995) Digraph-Based Models for Automated HAZOP Analysis. In Reliability Engineering and System Safety vol 50, pp33-49.

Vaidhyanathan, R. and Ventkatasubramanian, V. (1996) A Semi-quantitative reasoning methodology for filtering and ranking HAZOP results in HAZOPExpert. In Reliability Engineering and System Safety vol 53, pp185-203.

Van Heijst, G. et al. (1992) Using Generalised Directive Models in Knowledge Acquisition. Lecture Notes in Artificial Intelligence vol 559, pp112-132.

Wilcox, N.A. and Himmelblau, D.M. (1994) The Possible Cause And Effect Graphs (PCEG) Model For Fault Diagnosis-I. Methodology. In Computers Chem, Engng. vol 18, pp103-116.

Wells, G.L. and Seagrave, C.J. (1976) Flowsheeting for Safety, a guide on safety measures to consider during the design of chemical plant, IChemE.

Appendix A

Verification Example

A.1. Tank Model Verified

```
/*-----------------------------------------------*/
/*
vin and vout are input and output for nitrogen
/*
/*-----------------------------------------------*/

frame(tank isa closedvessel,
  [ inports info  [ in1, vin ],
    outports info  [ out1, vout ],
    unitports info  [ liquid, vapour ],
    propLinks info [

    %propagation
    arc([in1,temperature],+,[liquid,temperature]),
    arc([liquid,temperature],+,[out1,temperature]),
    arc([in1,flow],+,[liquid,level]),
    arc([out1,flow],-,[liquid,level]),
    arc([vin,flow],+,[vapour,pressure]),
    arc([vout,flow],-,[vapour,pressure]),
    arc([vout,flow],+,[in1,flow]),
    arc([vin,flow],-,[in1,flow]),
    arc([in1,noFlow],- -,[liquid,level]),
    arc([out1,noFlow],++,[liquid,level]),
    arc([vout,noFlow],++,[vapour,pressure]),
    arc([in1,noFlow],++,[out1,noFlow]),
    arc([vin,noFlow],++,[in1,flow]),
    arc([vout,noFlow],- -,[in1,flow]),
    arc([out1,reverseFlow],++,[liquid,level]),
    arc([in1,pressure],+,[vapour,pressure]),
    arc([vin,pressure],+,[vapour,pressure]),
    arc([vout,pressure],+,[vapour,pressure]),
    arc([in1,pressure],+,[liquid,level]),
    arc([out1,pressure],+,[liquid,level]),
    arc([out1,pressure],+,[vapour,pressure]),
    arc([in1,concentration],+,[liquid,concentration]),
    arc([liquid,concentration],+,[out1,concentration]),
    arc([vin,concentration],+,[vapour,concentration]),
    arc([vapour,concentration],+,[vout,concentration]),

    %faults
    arc([fault,'outlet1 partly blocked'],-,[out1,flow]),
    arc([fault,'outlet1 completely blocked'],+,[out1,noFlow]),
    arc([fault,'leak to environment'],-,[liquid,level]),
    arc([fault,'external fire'],+,[liquid,temperature]),
    arc([fault,'cold weather'],-,[liquid,temperature]),
    arc([fault,'inlet1 siphon breaker blockage'],+,[in1,reverseFlow]),
    arc([fault,['leak into vacuum system', vacuum]],+,[in1,pressure]),
    arc([fault,'leak to environment'],-,[out1,flow]),
    arc([fault,'leak to environment'],+,[in1,flow]),
    arc([fault,'outlet1 partly blocked'],+,[vapour,pressure]),
```

```
arc([fault,'outlet1 completely blocked'],+,[vapour,pressure]),

%consequences resulting from faults
arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'external fire'],+,[consequence,'structural weakening']),
arc([fault,['leak into vacuum system', vacuum]],+,[consequence,'possible explosive    mixture']),

%consequences resulting from deviations
arc([deviation,[moreLevel,liquid]],+,[consequence,'overfilling']),
arc([deviation,[moreConcentration,liquid]],+,[consequence,'rubber lining corrosion in hot caustic']),
arc([deviation,[moreConcentration,liquid]],+,[consequence,'emulsification of contents ']),
arc([deviation,[moreTemperature,liquid]],+,[consequence,'increased evaporation']),
arc([deviation,[moreTemperature,liquid]],+,[consequence,'flammable or toxic vapour release']),
arc([deviation,[moreTemperature,liquid]],+,[consequence,'crystallisation']),
arc([deviation,[moreTemperature,liquid]],+,[consequence,'viscosity increase']),
arc([deviation,[lessTemperature,liquid]],+,[consequence,'viscosity decrease']),
arc([deviation,[lessLevel,liquid]],+,[consequence,'vessel emptying']),
arc([deviation,[morePressure,vapour]],+,[consequence,'possible rupture']),
arc([deviation,[noFlow,vin]],+,[consequence,['loss of blanket and explosion risk', flammable]]),
arc([deviation,[lessFlow,vin]],+,[consequence,'vacuum collapse']),
arc([deviation,[moreFlow,vout]],+,[consequence,['loss of material', expensive]]),
arc([deviation,[moreFlow,in1]],+,[consequence,'incomplete separation of water'])
]
]
).
```

A.2. Verification Queries to Test for the Existence of Shortest Paths Between Boundary Ports For Tank Model

Examples from a set of queries prepared for the QUEEN expert system (Chung, 1993) are shown below.

Key: P = Path, E = Effect

```
% Testing all possibilities for shortest paths from boundary ports to boundary ports
:- spe([tank1,in1,pressure],[tank1,in1,temperature],P,E).
:- spe([tank1,in1,pressure],[tank1,in1,concentration],P,E).
:- spe([tank1,in1,pressure],[tank1,in1,level],P,E).
:- spe([tank1,in1,pressure],[tank1,in1,flow],P,E).
:- spe([tank1,in1,pressure],[tank1,in1,noFlow],P,E).
:- spe([tank1,in1,pressure],[tank1,in1,reverseFlow],P,E).
:- spe([tank1,in1,pressure],[tank1,vin,pressure],P,E).
:- spe([tank1,in1,pressure],[tank1,vin,temperature],P,E).
:- spe([tank1,in1,pressure],[tank1,vin,concentration],P,E).
:- spe([tank1,in1,pressure],[tank1,vin,level],P,E).
:- spe([tank1,in1,pressure],[tank1,vin,flow],P,E).
:- spe([tank1,in1,pressure],[tank1,vin,noFlow],P,E).
:- spe([tank1,in1,pressure],[tank1,vin,reverseFlow],P,E).
:- spe([tank1,in1,pressure],[tank1,out1,pressure],P,E).
:- spe([tank1,in1,pressure],[tank1,out1,temperature],P,E).
:- spe([tank1,in1,pressure],[tank1,out1,concentration],P,E).
:- spe([tank1,in1,pressure],[tank1,out1,level],P,E).
:- spe([tank1,in1,pressure],[tank1,out1,flow],P,E).
:- spe([tank1,in1,pressure],[tank1,out1,noFlow],P,E).
:- spe([tank1,in1,pressure],[tank1,out1,reverseFlow],P,E).
:- spe([tank1,in1,pressure],[tank1,vout,pressure],P,E).
:- spe([tank1,in1,pressure],[tank1,vout,temperature],P,E).
```

:- spe([tank1,in1,pressure],[tank1,vout,concentration],P,E).
:- spe([tank1,in1,pressure],[tank1,vout,level],P,E).
:- spe([tank1,in1,pressure],[tank1,vout,flow],P,E).
:- spe([tank1,in1,pressure],[tank1,vout,noFlow],P,E).
:- spe([tank1,in1,pressure],[tank1,vout,reverseFlow],P,E).
:- spe([tank1,in1,temperature],[tank1,in1,pressure],P,E).
:- spe([tank1,in1,temperature],[tank1,in1,concentration],P,E).
:- spe([tank1,in1,temperature],[tank1,in1,level],P,E)...etc.
:- spe([tank1,in1,concentration],[tank1,in1,pressure],P,E)...etc.
:- spe([tank1,in1,level],[tank1,in1,pressure],P,E)...etc.
:- spe([tank1,in1,flow],[tank1,in1,pressure],P,E)...etc.
:- spe([tank1,in1,noFlow],[tank1,in1,pressure],P,E)...etc.
:- spe([tank1,in1,reverseFlow],[tank1,in1,pressure],P,E)...etc.
:- spe([tank1,vin,pressure],[tank1,in1,pressure],P,E)...etc.

:- spe([tank1,out1,pressure],[tank1,in1,pressure],P,E)...etc.

:- spe([tank1,vout,pressure],[tank1,in1,pressure],P,E)...etc.
:- seen.

A.3. Queen Output for Verification Testing for the Existence of Shortest Paths Between Boundary Ports

Examples of output from the QUEEN expert system are given below.

There is no path between [tank1,in1,pressure] and [tank1,in1,temperature]...etc.

The shortest path between [tank1,in1,temperature] and [tank1,out1,temperature] is:
[[tank1,in1,temperature],[tank1,liqd,temperature],[tank1,out1,temperature]]
The effect of propagating an initial increase is: 1
The effect of propagating an initial decrease is: -1

There is no path between [tank1,in1,temperature] and [tank1,out1,concentration]...etc.

The shortest path between [tank1,in1,concentration] and [tank1,out1,concentration] is:
[[tank1,in1,concentration],[tank1,liqd,concentration],[tank1,out1,concentration]]
The effect of propagating an initial increase is: 1
The effect of propagating an initial decrease is: -1

There is no path between [tank1,in1,concentration] and [tank1,out1,level]...etc.

The shortest path between [tank1,in1,noFlow] and [tank1,out1,noFlow] is:
[[tank1,in1,noFlow],[tank1,out1,noFlow]]
The effect of propagating an initial increase is: 1
The effect of propagating an initial decrease is: 0

There is no path between [tank1,in1,noFlow] and [tank1,out1,reverseFlow]...etc.

The shortest path between [tank1,vin,concentration] and [tank1,vout,concentration] is:
[[tank1,vin,concentration],[tank1,vapr,concentration],[tank1,vout,concentration]]
The effect of propagating an initial increase is: 1
The effect of propagating an initial decrease is: -1

There is no path between [tank1,vin,concentration] and [tank1,vout,level]...etc.

The shortest path between [tank1,vin,flow] and [tank1,in1,flow] is:
[[tank1,vin,flow],[tank1,in1,flow]]

The effect of propagating an initial increase is: -1
The effect of propagating an initial decrease is: 1

There is no path between [tank1,vin,flow] and [tank1,in1,noFlow]...etc.

The shortest path between [tank1,vin,noFlow] and [tank1,in1,flow] is:
[[tank1,vin,noFlow],[tank1,in1,flow]]
The effect of propagating an initial increase is: 1
The effect of propagating an initial decrease is: 0

There is no path between [tank1,vin,noFlow] and [tank1,in1,noFlow]...etc.

The shortest path between [tank1,vout,flow] and [tank1,in1,flow] is:
[[tank1,vout,flow],[tank1,in1,flow]]
The effect of propagating an initial increase is: 1
The effect of propagating an initial decrease is: -1

There is no path between [tank1,vout,flow] and [tank1,in1,noFlow]...etc.

The shortest path between [tank1,vout,noFlow] and [tank1,in1,flow] is:
[[tank1,vout,noFlow],[tank1,in1,flow]]
The effect of propagating an initial increase is: -1
The effect of propagating an initial decrease is: 0

There is no path between [tank1,vout,noFlow] and [tank1,in1,noFlow]...etc.

Appendix B

Verification Testing Examples

These models were constructed to test if the propagation paths they contain would be correctly detected by Equipment Model Builder's verification techniques. These models are correct base models. They test that no spurious errors are detected by the verification techniques. To complete verification testing arcs are removed from these models to introduce errors into the models. Removing arcs leads to missing propagation paths. Errors are introduced in order to test that they are detected by the verification techniques. For example, removing the arc '([in1,flow], +, [liquid,level])' from model B.2.1. would result in 'level liquid' being placed in the list of deviations without causes as no deviation will now be able to propagate to the unit port liquid.

B.1. Test Models for Verifying Propagation Paths

B.1.1. Model Testing Propagation of Deviations Between Different Process Variables

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports info [liquid],
   proplinks info [

   %propagation
   arc([in1,flow], +, [liquid,level]),
   arc([liquid,level], +, [out1,flow])
]]
).
```

B.1.2. Models Testing Propagation when Conditional Arcs are Present

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports info [vapour],
   proplinks info [ ],
   conditionLinks info [
     [status is spare,
       [propLinks include [

   %propagation
     arc([out1,pressure],1,[vapour,pressure]),
     arc([vapour,pressure],1,[in1,pressure])
]]]]
]
).

frame(test1 isa unit,
  [inports info [in1],
```

```
   outports info [out1],
   unitports info [liquid],
   proplinks info [

   % propagation
   arc ([out1,temperature],1,[liquid,temperature])
 ],
   conditionLinks info [
     [status is spare,
       [propLinks include [
   %propagation
     arc([liquid,temperature],1,[in1,temperature])
]]]]
]
).
```

B.1.3. Model Testing Propagation Under All the Influence (sign) types

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports info [liquid],
   proplinks info [

   %propagation
   arc([out1,flow], +, [liquid,level]),
   arc([liquid,level],- -, [in1,flow])
]]
).
```

Arcs containing examples of all the sign types (i.e. +,-, ++,- -,+++,- - -) were added to the model to test that Equipment Model Builder will detect a propagation path for all sign types.

B.2. Test Models for the Detection of Deviations without Causes

As described in section 6.1.1. deviations without causes may be present in deviation linked to consequence arcs and deviation linked to deviation arcs. Two series of test models were developed, one for each of the arc types. To avoid repetition only the models developed to test propagation of deviations between different process variables are given. Models to test propagation when conditional arcs are present and propagation under all the influence (sign) types were also created.

B.2.1. Models Testing Detection in Deviation Linked to Consequence Arcs

When a deviation linked to consequence arc is present in an arc series propagation paths may exist from boundary ports to consequences as shown in the model below.

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports info [liquid],
   proplinks info [

   %propagation
```

```
   arc([in1,flow], +, [liquid,level]),
   %consequences resulting from deviations
   arc([deviation,[moreLevel,liquid]], +,[consequence,overflowing])
]]
).
```

Propagation paths may also exist from faults to consequences as the model below demonstrates.

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports info [liquid, vapour],
   proplinks info [

   %propagation
   arc([vapour,temperature], -, [liquid,level]),
   %faults
   arc([fault,external fire], +, [vapour,temperature]),
   %consequences resulting from deviations
   arc([deviation,[lessLevel,liquid]], +,[consequence,vessel emptying])
]]
).
```

B.2.2. Models Testing Detection in Deviation Linked to Deviation Arcs

Deviations propagating to deviation linked to deviation arcs located in propagation paths may initiate at a boundary port within a deviation linked to deviation arc.

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports info [liquid,vapour],
   proplinks info [

   %propagation
   arc([in1,flow], +, [liquid,level]),
   arc([liquid,level], +, [vapour,pressure])
]]
).
```

The deviations may also initiate with a fault within a fault linked to deviation arc.

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports info [liquid, vapour],
   proplinks info [

   %propagation
   arc([vapour,temperature], +, [liquid,level]),
   %faults
   arc([fault,cold weather], +, [vapour temperature])
]]
).
```

B.3. Test Models for the Detection of Deviations with no Effect

Deviations with no effect may be present in fault linked to deviation arcs and deviation linked to deviation arcs as discussed in section 6.1.1. Again two test model series were developed, one for each of the arc types. Only the models developed to test propagation of deviations between different process variables are given. Models to test propagation when conditional arcs are present and propagation under all the influence (sign) types were also created.

B.3.1. Models Testing Detection in Fault Linked to Deviations Arcs

Deviations initiating in fault to deviation arcs located within propagation paths may propagate into deviation linked to deviation arcs.

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports info [liquid, vapour],
   proplinks info [

   %propagation
   arc([liquid,temperature], -, [vapour,concentration]),
   arc([vapour,concentration], +, [out1,concentration]),
   %faults
   arc([fault,external fire], +, [liquid,temperature])
]]
).
```

Deviations initiating in fault to deviation arcs may also propagate into deviation linked to consequence arcs in propagation paths.

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports info [liquid, vapour],
   proplinks info [

   %propagation
   arc([vapour,temperature], +, [liquid,level]),
   %faults
   arc([fault,external fire], +, [vapour,temperature]),
   %consequences resulting from deviations
   arc([deviation,[moreLevel,liquid]], +,[consequence,overflowing])
]]
).
```

B.3.2. Models Testing Detection in Deviation Linked to Deviation Arcs

Deviations may propagate from deviation linked to deviation arcs to other deviation linked to deviation arcs located in propagation paths.

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
```

```
   unitports info [liquid,vapour],
   proplinks info [

   %propagation
  arc([in1,concentration], +,[vapour,concentration]),
  arc([vapour,concentration], +, [liquid,temperature]),
  arc([liquid,temperature], +, [out1,temperature])
]]
).
```

Deviations may also propagate from deviation linked to deviation to deviation linked to consequence arcs located on propagation paths.

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports info [liquid,vapour],
   proplinks info [

   %propagation
   arc([in1,flow], +,[liquid,flow]),
   arc([liquid,flow], +, [vapour,pressure]),
   %consequences resulting from deviations
   arc([deviation,[morePressure,vapour]], +,[consequence,risk of explosion])
]]
).
```

B.4. Test Models for Verifying the Shortest Path

A series of four test models was developed to test the verification of different types of shortest path. See section 6.2.1.1.for an explanation of path types. These models were created to test:

shortest paths occurring between boundary ports;

shortest paths occurring in arc series between faults and consequences;

shortest paths occurring in arc series between faults and boundary ports;

shortest paths occurring in arc series between boundary ports and consequences.

The following types of propagation path were tested:

- propagation of deviations between different process variables;
- propagation under all the influence (sign) types.

To avoid repetition only the models developed to test propagation of deviations between different process variables are given.

B.4.1. Test Model for Paths Between Boundary Ports

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports info [liquid,vapour],
   proplinks info [

   %propagation
```

```
   arc([in1,temperature], +,[vapour,temperature]),
   arc([vapour,temperature], -,[liquid,concentration]),
   arc([liquid,concentration], +,[out1,concentration])
]]
).
```

B.4.2. Test Model for Paths Between Faults and Consequences

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports [liquid,vapour],
   proplinks info [
   %propagation
   arc([liquid,temperature], +,[vapour,pressure]),
   %faults
   arc([fault,reactor coking], +[liquid,temperature]),
   % consequences resulting from deviations
   arc([deviation,[morePressure,vapour]], +,[consequence,possible rupture])
]]
).
```

B.4.3. Test Model for Paths Between Faults and Boundary Ports

```
frame(test1 isa unit,
  [inports info [in1,in2],
   outports info [out1],
   unitports [liquid],
   proplinks info [
   %propagation
   arc([vapour,temp], -,[liquid,concentration]),
   arc([liquid,concentration], +,[out1,concentration]),
   %faults
   arc([fault,external fire], +,[vapour,temp])
]]
).
```

B.4.4. Test Model for Paths between Boundary Ports and Consequences

```
frame(test1 isa unit,
  [inports info [in1],
   outports info [out1],
   unitports [liquid],
   proplinks info [
   %propagation
   arc([in1,flow], +,[liquid,level]),
   % consequences resulting from deviations
   arc([deviation,[moreLevel,liquid]], +,[consequence,overfilling])
]]
).
```

B.5. Test Module Models

The same techniques that are used to verify unit models are also applicable to module models. In order to ensure thorough checking some test modules were constructed. This sub-section describes the structure of the test modules built. The same types of

propagation paths used to test the verification techniques for unit models were also tested within module models. These propagation paths are:

- propagation of deviations between different process variables;
- propagation when conditional arcs are present;
- propagation under all the influence (sign) types.

A unit may have one of three position-types within a module. The unit may be the initial, the intermediate or the terminal unit of the module. This holds true for all the module types (i.e. loop modules, divider-header combinations and user-defined modules). Therefore it is only necessary to test the verification techniques for one module type as the results will also apply to the other module types. The module type chosen to be tested was the user-defined module as this allowed very simple modules to built.

The test modules built contained two types of units which shall be called complex units and simple units. Complex units possess all four arc types (see section 2.4.1.). For simplicity only one complex unit was placed within a module. The simple unit contains only deviation linked to deviation arcs. Deviation linked to deviation arcs are needed so that arcs modelling boundary port deviations propagating to boundary port deviations can be placed into the unit. This is necessary to prevent unlinked boundary port deviations. Before being placed into a module both unit types were verified.

Three modules were built to test the verification techniques, one for each of the positions a unit may occupy in a module. The modules were formed by amalgamating instances of the simple and complex units in the order shown below.

Test Module 1. simple unit → complex unit → simple unit

Test Module 2. complex unit → simple unit

Test Module 3. simple unit → complex unit

Figure B.1 Test Module Structures

All the boundary ports of the initial and final units were chosen to be the new boundary ports of the modules.

Appendix C

Benzene Purification System

This appendix contains input data for the application QUEEN (Chung, 1993) prepared for a benzene purification system. Two sets of input data have been prepared: a standard set and a set to which the modular approach has been applied. Each set of input data consists of a modified plant description and a file of models occurring within the plant. Example HAZOP results created by QUEEN are given for both sets of data.

C.1. Modified Plant Description (Standard Approach)

See section 7.1. for a plant diagram.

```
instance(feedinlet isa source,
   [outports info
      [out is [e103,htin]]
   ]
).

instance(e103 isa heatex,
   [outports info
      [cdot is [tail1,in],htot is [valve1,in]]
   ]
).

instance(tail1 isa outlet,
   [outports info
      []
   ]
).

instance(heatsource1 isa source,
   [outports info
      [out is [e103,cdin]]
   ]
).

instance(t101 isa column,
   [outports info
      [hto is [e106,htin],topo is [e104,stin],boto is
[valve2,in]]
   ]
).

instance(valve1 isa valve,
   [outports info
      [out is [t101,feed]],
      aperture is open
   ]
).

instance(e106 isa heatex,
   [outports info
      [cdot is [tail2,in],htot is [t101,htr]]
   ]
).

instance(tail2 isa outlet,
   [outports info
      []
   ]
).

instance(heatsource2 isa source,
   [outports info
      [out is [e106,cdin]]
   ]
).

instance(valve2 isa valve,
   [outports info
      [out is [tail3,in]],
      aperture is open
   ]
).

instance(tail3 isa outlet,
   [outports info
      []
   ]
).

instance(e104 isa cooler,
   [outports info
      [wot is [tail4,in],stot is [d103,in1]]
   ]
).

instance(tail4 isa outlet,
   [outports info
      []
   ]
).
```

```
instance(watersource1 isa source,
   [outports info
      [out is [e104,win]]
   ]
).

instance(d103 isa closedvessel,
   [outports info
      [out1 is [divider1,in]]
   ]
).

instance(p101a isa pump,
   [outports info
      [out is [valve6,in]]
   ]
).

instance(divider2 isa divider,
   [outports info
      [out1 is [valve3,in],out2 is [valve4,in]],
      purpose is tosplit
   ]
).

instance(valve3 isa valve,
   [outports info
      [out is [t101,topr]],
      aperture is open
   ]
).

instance(valve4 isa valve,
   [outports info
      [out is [e105,stin]],
      aperture is open
   ]
).

instance(e105 isa cooler,
   [outports info
      [wot is [tail5,in],stot is [tail6,in]]
   ]
).

instance(tail5 isa outlet,
   [outports info
      []
   ]
).

instance(tail6 isa outlet,
   [outports info
      []
   ]
).

instance(watersource2 isa source,
   [outports info
      [out is [e105,win]]
   ]
).

instance(p101b isa pump,
   [outports info
      [out is [valve8,in]],
      status is spare
   ]
).

instance(valve5 isa valve,
   [outports info
      [out is [p101a,in]],
      aperture is open
   ]
).

instance(valve7 isa valve,
   [outports info
      [out is [p101b,in]],
      aperture is closed
   ]
).

instance(valve6 isa valve,
   [outports info
      [out is [header1,in1]],
      aperture is open
   ]
).

instance(valve8 isa valve,
   [outports info
      [out is [header1,in2]],
      aperture is closed
   ]
).

instance(header1 isa header,
   [outports info
      [out is [divider2,in]],
      purpose is frombypass
   ]
).

instance(divider1 isa divider,
   [outports info
      [out1 is [valve5,in],out2 is [valve7,in]],
      purpose is tobypass
   ]
).
```

C.2. Plant Models (Standard Approach)

```
/*-------------------------------------------------*/
/*
Dummy source of plant
/*
/*-------------------------------------------------*/

frame(source isa unit,
[  outports info  [ out ],
   propLinks info [

   %faults
   arc([fault,'inlet completely blocked'],+,[out,noFlow]),
   arc([fault,'inlet partly blocked'],-,[out,flow]),
   arc([fault,'inlet completely blocked'],-,[out,pressure]),
   arc([fault,'cold weather'],-,[out,temperature]),
   arc([fault,'leak to environment'],-,[out,flow]),

   %consequences resulting from faults
   arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
   arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
   arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]])
   ]
 ]
).
/*-------------------------------------------------*/
/*
Basic heatexchanger model.
This model is a heater.  Cdin is actually a steam source.
heated stream: htin = in port, htot = out port
heating  stream: cdin = in port, cdot = out port
/*
/*-------------------------------------------------*/

frame(heatex isa unit,
   [  inports info  [ htin, cdin ],
      outports info  [ htot, cdot ],
      propLinks info [

      %propagation
      arc([htot,flow],+,[htin,flow]),
      arc([htin,flow],+,[htot,flow]),
      arc([cdot,flow],+,[cdin,flow]),
      arc([cdin,flow],+,[cdot,flow]),
      arc([cdin,flow],+,[htot,temperature]),
      arc([cdin,flow],+,[cdot,temperature]),
      arc([htin,flow],-,[htot,temperature]),
      arc([htin,flow],-,[cdot,temperature]),
      arc([htin,temperature],+,[htot,temperature]),
      arc([cdin,temperature],+,[cdot,temperature]),
      arc([cdin,temperature],+,[htot,temperature]),
      arc([htin,temperature],+,[cdot,temperature]),
      arc([htin,pressure],+,[htot,pressure]),
      arc([cdin,pressure],+,[cdot,pressure]),
      arc([htot,pressure],+,[htin,pressure]),
```

```
arc([cdot,pressure],+,[cdin,pressure]),
arc([htot,reverseFlow],++,[htin,reverseFlow]),
arc([cdot,reverseFlow],++,[cdin,reverseFlow]),
arc([htot,noFlow],++,[htin,noFlow]),
arc([htin,noFlow],++,[htot,noFlow]),
arc([cdot,noFlow],++,[cdin,noFlow]),
arc([cdin,noFlow],++,[cdot,noFlow]),
arc([cdin,noFlow],- -,[htot,temperature]),
arc([htin,noFlow],++,[cdot,temperature]),
arc([htin,concentration],+,[htot,concentration]),
arc([cdin,concentration],+,[cdot,concentration]),

%faults
arc([fault,'partly blocked (heating side)'],-,[cdot,flow]),
arc([fault,'partly blocked (heated side)'],-,[htot,flow]),
arc([fault,'fouling'],-,[htot,temperature]),
arc([fault,'fouling'],+,[cdot,temperature]),
arc([fault,'blockage (heating side)'],+,[cdot,noFlow]),
arc([fault,'blockage (heated side)'],+,[htot,noFlow]),
arc([fault,'blockage (heating side)'],+,[cdin,pressure]),
arc([fault,'blockage (heating side)'],-,[cdot,pressure]),
arc([fault,'blockage (heated side)'],+,[htin,pressure]),
arc([fault,'blockage (heated side)'],-,[htot,pressure]),
arc([fault,'blockage (heating side)'],-,[htot,temperature]),
arc([fault,'blockage (heated side)'],+,[cdot,temperature]),
arc([fault,'holed heatexchanger'],-,[cdot,flow]),
arc([fault,'holed heatexchanger'],+,[cdin,flow]),
arc([fault,'holed heatexchanger'],-,[cdot,temperature]),
arc([fault,'holed heatexchanger'],+,[htot,temperature]),

%consequences resulting from faults
arc([fault,'holed heatexchanger'],+,[consequence,'contamination of product']),

%consequences resulting from deviations
arc([deviation,[moreTemperature,htot]],+,[consequence,'boiling in heatexchanger']),
arc([deviation,[lessTemperature,cdot]],+,[consequence,'freezing in heatexchanger']),
arc([deviation,[moreTemperature,htot]],+,[consequence,'polymerisation']),
arc([deviation,[morePressure,cdin]],+,[consequence,'possible rupture']),
arc([deviation,[morePressure,htin]],+,[consequence,'possible rupture']),
arc([deviation,[moreTemperature,htot]],+,[consequence,'mechanical failure due to overtemperature'])
]
]
).
/*-------------------------------------------------*/
/*

/*
/*-------------------------------------------------*/

frame(outlet isa unit,
  [ inports info  [ in ],
      propLinks info [

    %faults
    arc([fault,'partly blocked'],-,[in,flow]),
    arc([fault,'blocked'],+,[in,noFlow]),
    arc([fault,'blocked'],+,[in,pressure]),
    arc([fault,'leak to environment'],+,[in,flow]),
    arc([fault,['leak into vacuum system', vacuum]],+,[in,pressure]),

    %consequences resulting from faults
```

```
    arc([fault,['leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
    arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
    arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
    arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]])
    ]
   ]
).
/*-------------------------------------------------*/
/*
Distillation column
A column which connects to a heating loop
feed = feed in port,
heated stream: htr = heated return in port, hto = heated out port
topr = top return in port, tpot = top out port
btot = bottom out port,
/*
/*-------------------------------------------------*/

frame(column isa unit,
  [  inports info  [ feed, topr, htr ],
     outports info  [ topo, hto, boto ],
     unitports info  [ self ],
     propLinks info [

     %propagation
     arc([feed,flow],+,[self,level]),
     arc([topr,flow],+,[self,level]),
     arc([htr,flow],+,[self,level]),
     arc([topo,flow],-,[self,level]),
     arc([hto,flow],-,[self,level]),
     arc([boto,flow],-,[self,level]),
     arc([feed,temperature],+,[self,temperature]),
     arc([topr,temperature],+,[self,temperature]),
     arc([htr,temperature],+,[self,temperature]),
     arc([self,temperature],+,[boto,temperature]),
     arc([self,temperature],+,[hto,temperature]),
     arc([feed,noFlow],- -,[self,level]),
     arc([topr,noFlow],- -,[self,level]),
     arc([htr,noFlow],- -,[self,level]),
     arc([topo,noFlow],++,[self,level]),
     arc([hto,noFlow],++,[self,level]),
     arc([boto,noFlow],++,[self,level]),
     arc([feed,concentration],+,[self,concentration]),
     arc([self,concentration],+,[topo,concentration]),
     arc([self,concentration],+,[boto,concentration]),
     arc([topr,concentration],+,[self,concentration]),
     arc([self,concentration],+,[hto,concentration]),
     arc([htr,concentration],+,[self,concentration]),
     arc([boto,reverseFlow],++,[self,level]),
     arc([topo,reverseFlow],++,[self,level]),
     arc([hto,reverseFlow],++,[self,level]),

     %faults
     arc([fault,'packing support collapses'],+,[boto,noFlow]),
     arc([fault,'external fire'],+,[self,temperature]),

     %consequences resulting from deviations
     arc([deviation,[moreLevel,self]],+,[consequence,'column floods'])
     ]
    ]
).
```

```
/*-----------------------------------------------*/
/*
The most general unit for a valve.
/*
/*-----------------------------------------------*/

frame(valve isa unit,
  [  inports info  [ in ],
     outports info  [ out ],
     aperture is unknown,
     propLinks info [

     %faults
     arc([fault,'leak to environment'],+,[in,flow]),
     arc([fault,['leak into vacuum system', vacuum]],+,[in,pressure]),
     arc([fault,'thermal expansion of contents'],+,[in,pressure]),

     %consequences resulting from faults
     arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
     arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]]),
     arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
     arc([fault,'thermal expansion of contents'],+,[consequence,'flange leak'])
     ],

     conditionLinks info [
      [aperture is open,
        [propLinks include [

     %propagation
     arc([out,flow],+,[in,flow]),
     arc([in,flow],+,[out,flow]),
     arc([in,temperature],+,[out,temperature]),
     arc([in,pressure],+,[out,pressure]),
     arc([out,pressure],+,[in,pressure]),
     arc([out,reverseFlow],++,[in,reverseFlow]),
     arc([in,noFlow],++,[out,noFlow]),
     arc([out,noFlow],++,[in,noFlow]),
     arc([in,concentration],+,[out,concentration]),

     %faults
     arc([fault,'partly blocked'],-,[out,flow]),
     arc([fault,'completely blocked'],+,[out,noFlow]),
     arc([fault,['blocked by frozen fluid', freezing]],+,[out,noFlow]),
     arc([fault,'leak to environment'],-,[out,flow]),
     arc([fault,'closed in error'],+,[in,pressure]),
     arc([fault,'closed in error'],-,[out,pressure]),
     arc([fault,'thermal expansion of contents'],+,[out,pressure])
      ]]],

      [aperture is floating,
        [propLinks include [

     %propagation
     arc([out,flow],+,[in,flow]),
     arc([in,flow],+,[out,flow]),
     arc([in,temperature],+,[out,temperature]),
     arc([in,pressure],+,[out,pressure]),
     arc([out,pressure],+,[in,pressure]),
     arc([out,reverseFlow],++,[in,reverseFlow]),
     arc([in,noFlow],++,[out,noFlow]),
     arc([out,noFlow],++,[in,noFlow]),
```

```
arc([in,concentration],+,[out,concentration]),

%faults
arc([fault,'partly blocked'],-,[out,flow]),
arc([fault,'completely blocked'],+,[out,noFlow]),
arc([fault,['blocked by frozen fluid', freezing]],+,[out,noFlow]),
arc([fault,'leak to environment'],-,[out,flow]),
arc([fault,'fails at closed'],+,[out,noFlow]),
arc([fault,'fails at low aperture '],-,[out,flow]),
arc([fault,'fails at high aperture'],+,[out,flow]),
arc([fault,'passes when no flow is desired'],+,[out,flow]),
arc([fault,'closed in error'],+,[in,pressure]),
arc([fault,'closed in error'],-,[out,pressure]),
arc([fault,'thermal expansion of contents'],+,[out,pressure])
 ]]],

 [aperture is closed,
   [propLinks include [

 %faults
 arc([fault,'open or passing'],+,[out,flow]),
 arc([fault,'leak to environment'],+,[out,reverseFlow])
 ]]]]
 ]
).
/*-----------------------------------------------*/
/*
A heatexchanger used for cooling, win is cooling water source.
stin is stream source. Stream is being cooled.
cooled stream: stin =stream  in port, stot =stream out port
cooling stream: win = water in port, wot = water out port
/*
/*-----------------------------------------------*/

frame(cooler isa unit,
  [  inports info  [ win, stin ],
     outports info  [ wot, stot ],
     propLinks info [

     %propagation
     arc([stot,flow],+,[stin,flow]),
     arc([stin,flow],+,[stot,flow]),
     arc([wot,flow],+,[win,flow]),
     arc([win,flow],+,[wot,flow]),
     arc([win,flow],-,[stot,temperature]),
     arc([win,flow],-,[wot,temperature]),
     arc([stin,flow],+,[stot,temperature]),
     arc([stin,flow],+,[wot,temperature]),
     arc([stin,temperature],+,[stot,temperature]),
     arc([win,temperature],+,[wot,temperature]),
     arc([win,temperature],+,[stot,temperature]),
     arc([stin,temperature],+,[wot,temperature]),
     arc([stin,pressure],+,[stot,pressure]),
     arc([win,pressure],+,[wot,pressure]),
     arc([stot,pressure],+,[stin,pressure]),
     arc([wot,pressure],+,[win,pressure]),
     arc([wot,reverseFlow],++,[win,reverseFlow]),
     arc([stot,reverseFlow],++,[stin,reverseFlow]),
     arc([stot,noFlow],++,[stin,noFlow]),
     arc([stin,noFlow],++,[stot,noFlow]),
     arc([wot,noFlow],++,[win,noFlow]),
```

```
arc([win,noFlow],++,[wot,noFlow]),
arc([win,noFlow],++,[stot,temperature]),
arc([stin,noFlow],- -,[wot,temperature]),
arc([win,concentration],+,[wot,concentration]),
arc([stin,concentration],+,[stot,concentration]),

%faults
arc([fault,'partly blocked (cooling side)'],-,[wot,flow]),
arc([fault,'partly blocked (cooled side)'],-,[stot,flow]),
arc([fault,'fouling'],+,[stot,temperature]),
arc([fault,'fouling'],-,[wot,temperature]),
arc([fault,'blockage (cooling side)'],+,[wot,noFlow]),
arc([fault,'blockage (cooled side)'],+,[stot,noFlow]),
arc([fault,'blockage (cooling side)'],+,[win,pressure]),
arc([fault,'blockage (cooling side)'],-,[wot,pressure]),
arc([fault,'blockage (cooled side)'],+,[stin,pressure]),
arc([fault,'blockage (cooled side)'],-,[stot,pressure]),
arc([fault,'blockage (cooling side)'],+,[stot,temperature]),
arc([fault,'blockage (cooled side)'],-,[wot,temperature]),
arc([fault,'holed heatexchanger'],-,[wot,flow]),
arc([fault,'holed heatexchanger'],+,[win,flow]),
arc([fault,'holed heatexchanger'],+,[wot,temperature]),
arc([fault,'holed heatexchanger'],-,[stot,temperature]),

%consequences resulting from faults
arc([fault,'holed heatexchanger'],+,[consequence,'contamination of product']),

%consequences resulting from deviations
arc([deviation,[lessTemperature,wot]],+,[consequence,'freezing in heatexchanger']),
arc([deviation,[moreTemperature,stot]],+,[consequence,'boiling in heatexchanger']),
arc([deviation,[morePressure,win]],+,[consequence,'possible rupture']),
    arc([deviation,[morePressure,stin]],+,[consequence,'possible rupture'])
    ]
   ]
).
/*-----------------------------------------------*/
/*
Reactor for lawley plant
With one inlet and one outlet
liqd = liquid, vapr = vapour
/*
/*-----------------------------------------------*/

frame(closedvessel isa vessel,
  [  inports info  [ in1 ],
     outports info  [ out1 ],
     unitports info  [ liqd, vapr ],
     propLinks info [

%propagation
arc([in1,temperature],+,[liqd,temperature]),
arc([liqd,temperature],+,[out1,temperature]),
arc([in1,flow],+,[liqd,level]),
arc([out1,flow],-,[liqd,level]),
arc([in1,noFlow],- -,[liqd,level]),
arc([out1,noFlow],++,[liqd,level]),
arc([in1,noFlow],++,[out1,noFlow]),
arc([out1,reverseFlow],++,[liqd,level]),
arc([in1,pressure],+,[vapr,pressure]),
arc([in1,pressure],+,[liqd,level]),
arc([out1,pressure],+,[liqd,level]),
```

```
arc([out1,pressure],+,[vapr,pressure]),
arc([in1,concentration],+,[liqd,concentration]),
arc([liqd,concentration],+,[out1,concentration]),
arc([liqd,concentration],+,[in1,concentration]),

%faults
arc([fault,'outlet1 partly blocked'],-,[out1,flow]),
arc([fault,'outlet1 completely blocked'],+,[out1,noFlow]),
arc([fault,'leak to environment'],-,[liqd,level]),
arc([fault,'external fire'],+,[liqd,temperature]),
arc([fault,'cold weather'],-,[liqd,temperature]),
arc([fault,'inlet1 siphon breaker blockage'],+,[in1,reverseFlow]),
arc([fault,['leak into vacuum system', vacuum]],+,[in1,pressure]),
arc([fault,'leak to environment'],-,[out1,flow]),
arc([fault,'leak to environment'],+,[in1,flow]),

%consequences resulting from faults
arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'external fire'],+,[consequence,'structural weakening']),
arc([fault,['leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'external fire'],+,[consequence,'possible rupture']),

%consequences resulting from deviations
arc([deviation,[moreLevel,liqd]],+,[consequence,'overfilling']),
arc([deviation,[moreConcentration,liqd]],+,[consequence,'rubber lining corrosion in hot caustic']),
arc([deviation,[moreConcentration,liqd]],+,[consequence,'emulsification of contents ']),
arc([deviation,[moreTemperature,liqd]],+,[consequence,'increased evaporation']),
arc([deviation,[moreTemperature,liqd]],+,[consequence,'flammable or toxic vapour release']),
arc([deviation,[moreTemperature,liqd]],+,[consequence,'crystallisation']),
    arc([deviation,[moreTemperature,liqd]],+,[consequence,'viscosity increase']),
    arc([deviation,[lessTemperature,liqd]],+,[consequence,'viscosity decrease']),
    arc([deviation,[lessLevel,liqd]],+,[consequence,'vessel emptying']),
    arc([deviation,[morePressure,vapr]],+,[consequence,'possible rupture']),
    arc([deviation,[moreFlow,in1]],+,[consequence,'incomplete separation of water'])
    ]
   ]
).
/*-----------------------------------------------*/
/*
Status = spare or running
/*
/*-----------------------------------------------*/

frame(pump isa pressureraiser,
  [ inports info  [ in ],
    outports info  [ out ],
status is unknown,
propLinks info [

%propagation
arc([in,pressure],+,[out,flow]),
arc([in,pressure],+,[out,pressure]),
arc([out,pressure],-,[out,flow]),
arc([in,temperature],+,[out,temperature]),
arc([out,noFlow],++,[in,noFlow]),
arc([in,noFlow],++,[out,noFlow]),
arc([in,concentration],+,[out,concentration]),

%faults
```

```
arc([fault,'partly blocked'],-,[out,flow]),
arc([fault,'completely blocked'],+,[out,noFlow]),
arc([fault,'leak to environment'],+,[in,flow]),
arc([fault,'leak to environment'],-,[out,flow]),
arc([fault,'air lock'],-,[out,flow]),
arc([fault,'air lock'],-,[out,pressure]),
arc([fault,'power supply fails'],+,[out,noFlow]),
arc([fault,'power supply fails'],-,[out,pressure]),
arc([fault,'impellor failure'],-,[out,flow]),
arc([fault,'impellor failure'],-,[out,pressure]),
arc([fault,['leak into vacuum system', vacuum]],+,[in,pressure]),
arc([fault,'pump fails'],+,[in,reverseFlow]),

%consequences resulting from faults
arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,['leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),

%consequences resulting from deviations
arc([deviation,[morePressure,out]],+,[consequence,'possible rupture']),
arc([deviation,[moreTemperature,out]],+,[consequence,'overheating']),
arc([deviation,[moreTemperature,out]],+,[consequence,'gland failure']),
arc([deviation,[reverseFlow,out]],+,[consequence,'pump driven backwards']),
arc([deviation,[noFlow,in]],+,[consequence,'overheating']),
arc([deviation,[noFlow,in]],+,[consequence,'gland failure'])
],

conditionLinks info [
 [status is spare,
   [propLinks include [
     %faults
     arc([fault,'spare unit turned on'],+,[out,flow]),
     arc([fault,'spare unit turned on'],+,[out,pressure]),
     arc([fault,'leak to environment'],+,[out,reverseFlow])
      ]]]]
    ]
).
/*-----------------------------------------------*/
/*
A unit to split a stream into two output streams
/*
/*-----------------------------------------------*/

frame(divider isa pipe,
  [  inports info  [ in ],
     outports info  [ out1, out2 ],
purpose is toBypass,
propLinks info [

%propagation
arc([out1,flow],+,[in,flow]),
arc([out2,flow],+,[in,flow]),
arc([in,flow],+,[out1,flow]),
arc([in,flow],+,[out2,flow]),
arc([out1,flow],-,[out2,flow]),
arc([out2,flow],-,[out1,flow]),
arc([in,temperature],+,[out1,temperature]),
arc([in,temperature],+,[out2,temperature]),
arc([in,pressure],+,[out1,pressure]),
```

```
arc([in,pressure],+,[out2,pressure]),
arc([out1,pressure],+,[in,pressure]),
arc([out2,pressure],+,[in,pressure]),
arc([out2,pressure],+,[out1,pressure]),
arc([out1,pressure],+,[out2,pressure]),
arc([in,noFlow],++,[out1,noFlow]),
arc([in,noFlow],++,[out2,noFlow]),
arc([in,concentration],+,[out1,concentration]),
arc([in,concentration],+,[out2,concentration]),

%faults
arc([fault,'leak to environment'],+,[in,flow]),
arc([fault,'leak to environment'],-,[out1,flow]),
arc([fault,'leak to environment'],-,[out2,flow]),
arc([fault,['blocked by frozen fluid', freezing]],+,[in,noFlow]),
arc([fault,'inlet completely blocked'],+,[in,noFlow]),
arc([fault,'branch 1 completely blocked'],+,[out1,noFlow]),
arc([fault,'branch 2 completely blocked'],+,[out2,noFlow]),
arc([fault,'branch 1 completely blocked'],+,[out2,flow]),
arc([fault,'branch 2 completely blocked'],+,[out1,flow]),
arc([fault,'inlet partly blocked'],-,[in,flow]),
arc([fault,'branch 1 partly blocked'],-,[out1,flow]),
arc([fault,'branch 2 partly blocked'],-,[out2,flow]),
arc([fault,['leak into vacuum system', vacuum]],+,[in,pressure]),

%consequences resulting from faults
arc([fault,'leak to environment'],+,[consequence,'contaminate environment']),
arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,['leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),

    %consequences resulting from deviations
    arc([deviation,[morePressure,in]],+,[consequence,'possible rupture'])
    ],

    conditionLinks info [
     [purpose is tosplit,
       [propLinks include [

    %propagation
    arc([out1,noFlow],-,[in,flow]),
    arc([out2,noFlow],-,[in,flow])
     ]]]]
   ]
).
/*-----------------------------------------------*/
/*
To combine two unit streams
/*
/*-----------------------------------------------*/

frame(header isa pipe,
  [  inports info  [ in1, in2 ],
     outports info  [ out ],
     purpose is fromBypass,
     propLinks info [

     %propagation
     arc([out,flow],+,[in1,flow]),
     arc([out,flow],+,[in2,flow]),
     arc([in1,temperature],+,[out,temperature]),
```

```
arc([in2,temperature],+,[out,temperature]),
arc([out,pressure],+,[in1,pressure]),
arc([out,pressure],+,[in2,pressure]),
arc([out,reverseFlow],++,[in1,reverseFlow]),
arc([out,reverseFlow],++,[in2,reverseFlow]),
arc([out,noFlow],++,[in1,noFlow]),
arc([out,noFlow],++,[in2,noFlow]),
arc([in1,concentration],+,[out,concentration]),
arc([in2,concentration],+,[out,concentration]),

%faults
arc([fault,'branch 1 partly blocked'],-,[in1,flow]),
arc([fault,'branch 2 partly blocked'],-,[in2,flow]),
arc([fault,'outlet partly blocked'],-,[out,flow]),
arc([fault,'branch 1 completely blocked'],+,[in1,noFlow]),
arc([fault,'branch 2 completely blocked'],+,[in2,noFlow]),
arc([fault,'outlet completely blocked'],+,[out,noFlow]),
arc([fault,['blocked by frozen fluid', freezing]],+,[out,noFlow]),
arc([fault,['leak into vacuum system', vacuum]],+,[in1,pressure]),
arc([fault,['leak into vacuum system', vacuum]],+,[in2,pressure]),
arc([fault,'leak to environment'],+,[in1,flow]),
arc([fault,'leak to environment'],+,[in2,flow]),
arc([fault,'leak to environment'],-,[out,flow]),

%consequences resulting from faults
arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,['leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture'])
]
]
).
```

C.3. HAZOP Results (Standard Approach)

Examples from the HAZOP emulation results created by QUEEN are shown. For brevity the full set of results is not given. The examples include the errors in the HAZOP analysis identified in section 7.4.2.1.

feedinlet no flow out	valve1: completely blocked, feedinlet: inlet completely blocked, e103: blockage (heated side)		
feedinlet less flow out	valve1: partly blocked, feedinlet: inlet partly blocked, e103: partly blocked (heated side)		
	feedinlet: leak to environment	contaminate environment, fire/explosion risk, loss of material	

feedinlet more flow out	valve1: leak to environment	contaminate environment, loss of material, fire/explosion risk
feedinlet less temperature out	feedinlet: cold weather	
feedinlet more pressure out	valve1: thermal expansion of contents	flange leak
	valve1: closed in error	
e103 no flow cdin	tail1: blocked, heatsource1: inlet completely blocked, e103: blockage (heating side)	
e103 less flow cdin	tail1: partly blocked, heatsource1: inlet partly blocked, e103: partly blocked (heating side)	
	heatsource1: leak to environment	contaminate environment, fire/explosion risk, loss of material

- etc.-

d103 less flow out1	valve5: partly blocked, divider1: inlet partly blocked, branch 2 partly blocked, branch 1 partly blocked, d103: outlet1 partly blocked	

- etc.-

d103 more level liqd	valve7: thermal expansion of contents	overfilling, flange leak
	valve5: partly blocked, closed in error, divider1: inlet partly blocked, inlet completely blocked, branch 2 partly blocked, branch 1 partly blocked, d103: outlet1 partly blocked, outlet1 completely blocked	overfilling

- etc.-

p101b less flow out	valve8: thermal expansion of contents	flange leak
p101b more flow out	valve8: leak to environment	contaminate environment, loss of material, fire/explosion risk
	p101b: spare unit turned on	
p101b more pressure out	valve8: thermal expansion of contents	possible rupture, flange leak
	p101b: spare unit turned on	possible rupture
valve7 no flow in	divider1: branch 2 completely blocked	
valve8 no flow out	header1: branch 2 completely blocked	
valve8 less flow out	header1: branch 2 partly blocked	
valve8 more flow out	valve8: open or passing	

C.4. Modified Plant Description (Modular Approach)

See section 7.1. for a plant diagram.

```
instance(feedinlet isa source,
   [outports info
      [out is [e103,htin]]
   ]
).

instance(e103 isa heatex,
   [outports info
      [cdot is [tail1,in],htot is [valve1,in]]
   ]
).

instance(tail1 isa outlet,
   [outports info
      []
   ]
).

instance(heatsource1 isa source,
   [outports info
      [out is [e103,cdin]]
   ]
).

instance(valve1 isa valve,
   [outports info
      [out is [distilsystem3_,t101_feed]],
      aperture is open
   ]
).

instance(tail2 isa outlet,
   [outports info
      []
   ]
).

instance(heatsource2 isa source,
   [outports info
      [out is [distilsystem3_,e106_cdin]]
   ]
).

instance(valve2 isa valve,
   [outports info
      [out is [tail3,in]],
      aperture is open
   ]
```

```
).

instance(tail3 isa outlet,
   [outports info
      []
   ]
).

instance(tail4 isa outlet,
   [outports info
      []
   ]
).

instance(watersource1 isa source,
   [outports info
      [out is [distilsystem3_,e104_win]]
   ]
).

instance(valve4 isa valve,
   [outports info
      [out is [e105,stin]],
      aperture is open
   ]
).

instance(e105 isa cooler,
   [outports info
      [wot is [tail5,in],stot is [tail6,in]]
   ]
).

instance(tail5 isa outlet,
   [outports info
      []
   ]
).

instance(tail6 isa outlet,
   [outports info
      []
   ]
).

instance(watersource2 isa source,
   [outports info
      [out is [e105,win]]
   ]
).

instance(distilsystem3_ isa module_3,
   [outports info
         [t101_boto is [valve2,in],
         e106_cdot is [tail2,in],
         e104_wot is [tail4,in],
         divider2_out2 is [valve4,in]],
      divider1_purpose is tobypass,
      valve5_aperture is open,
      valve6_aperture is open,
      valve7_aperture is closed,
      p101b_status is spare,
      valve8_aperture is closed,
      header1_purpose is frombypass,
      divider2_purpose is tosplit,
      valve3_aperture is open
   ]
).
```

C.5. Plant Models (Modular Approach)

Models 'source', 'heatex', 'outlet', 'valve' and 'cooler' are duplicates of the same models given for the standard approach (section C.2) and have been removed to avoid repetition.

```
/*-------------------------------------------------*/
/*
The module is composed of the following units:
columnheatingloop2_ t101_topo connected to e104 stin,
e104 stot connected to d103 in1,
d103 out1 connected to byps1_ divider1_in,
byps1_ header1_out connected to divider2 in,
divider2 out1 connected to valve3 in,
valve3 out connected to columnheatingloop2_ t101_topr.
Conditions of module are:
      divider1_purpose is tobypass,
      valve5_aperture is open,
      valve6_aperture is open,
      valve7_aperture is closed,
      p101b_status is spare,
      valve8_aperture is closed,
      header1_purpose is frombypass,
      divider2_purpose is tosplit,
      valve3_aperture is open
and this completes the module.
```

```
/*
/*-----------------------------------------------*/

frame(module_3 isa module,
  [ inports info [ t101_feed, e106_cdin, e104_win ],
    outports info [ t101_boto, e106_cdot, e104_wot, divider2_out2 ],
    unitports info [ t101_self, e106_htin, e106_htot, e104_stin, e104_stot, d103_liqd, d103_vapr,
          divider1_in, p101a_in, p101a_out, divider2_in ],
    divider1_purpose is toBypass,
    valve5_aperture is unknown,
    p101a_status is unknown,
    valve6_aperture is unknown,
    valve7_aperture is unknown,
    p101b_status is unknown,
    valve8_aperture is unknown,
    header1_purpose is fromBypass,
    divider2_purpose is toBypass,
    valve3_aperture is unknown,
    propLinks info [

    %propagation
    arc([t101_feed,flow],+,[t101_self,level]),
    arc([t101_boto,noFlow],++,[t101_self,level]),
    arc([t101_boto,reverseFlow],++,[t101_self,level]),
    arc([e106_cdot,flow],+,[e106_cdin,flow]),
    arc([e106_cdin,flow],+,[e106_cdot,flow]),
    arc([e106_cdin,flow],+,[e106_cdot,temperature]),
    arc([e106_cdin,flow],+,[t101_boto,temperature]),
    arc([e106_cdin,flow],+,[e106_htot,temperature]),
    arc([e106_cdin,temperature],+,[e106_cdot,temperature]),
    arc([t101_feed,temperature],+,[t101_boto,temperature]),
    arc([e106_cdin,temperature],+,[e106_htot,temperature]),
    arc([e106_cdin,temperature],+,[t101_boto,temperature]),
    arc([e106_cdin,pressure],+,[e106_cdot,pressure]),
    arc([e106_cdot,pressure],+,[e106_cdin,pressure]),
    arc([e106_cdot,reverseFlow],++,[e106_cdin,reverseFlow]),
    arc([e106_cdot,noFlow],++,[e106_cdin,noFlow]),
    arc([e106_cdin,noFlow],++,[e106_cdot,noFlow]),
    arc([e106_cdin,noFlow],- -,[t101_boto,temperature]),
    arc([e106_cdin,concentration],+,[e106_cdot,concentration]),
    arc([t101_feed,concentration],+,[t101_boto,concentration]),
    arc([e104_wot,flow],+,[e104_win,flow]),
    arc([e104_win,flow],+,[e104_wot,flow]),
    arc([e104_win,flow],-,[e104_stot,temperature]),
    arc([e104_win,flow],-,[e104_wot,temperature]),
    arc([e104_win,temperature],+,[e104_wot,temperature]),
    arc([e104_win,temperature],+,[e104_stot,temperature]),
    arc([e104_win,pressure],+,[e104_wot,pressure]),
    arc([e104_wot,pressure],+,[e104_win,pressure]),
    arc([e104_wot,reverseFlow],++,[e104_win,reverseFlow]),
    arc([e104_wot,noFlow],++,[e104_win,noFlow]),
    arc([e104_win,noFlow],++,[e104_wot,noFlow]),
    arc([e104_win,noFlow],++,[e104_stot,temperature]),
    arc([e104_win,concentration],+,[e104_wot,concentration]),
    arc([e104_win,temperature],+,[p101a_out,temperature]),
    arc([divider2_out2,pressure],+,[divider2_in,pressure]),

    %faults
    arc([fault,'t101 packing support collapses'],+,[t101_boto,noFlow]),
    arc([fault,'t101 external fire'],+,[e106_htot,temperature]),
```

```
arc([fault,'t101 external fire'],+,[t101_boto,temperature]),
arc([fault,'e106 partly blocked (heating side)'],-,[e106_cdot,flow]),
arc([fault,'e106 fouling'],+,[e106_cdot,temperature]),
arc([fault,'e106 fouling'],-,[t101_boto,temperature]),
arc([fault,'e106 blockage (heating side)'],+,[e106_cdot,noFlow]),
arc([fault,'e106 blockage (heating side)'],+,[e106_cdin,pressure]),
arc([fault,'e106 blockage (heating side)'],-,[e106_cdot,pressure]),
arc([fault,'e106 blockage (heated side)'],+,[e106_htin,pressure]),
arc([fault,'e106 blockage (heating side)'],-,[t101_boto,temperature]),
arc([fault,'e106 blockage (heated side)'],+,[e106_cdot,temperature]),
arc([fault,'e106 blockage (heated side)'],-,[t101_boto,temperature]),
arc([fault,'e106 holed heatexchanger'],-,[e106_cdot,flow]),
arc([fault,'e106 holed heatexchanger'],+,[e106_cdin,flow]),
arc([fault,'e106 holed heatexchanger'],-,[e106_cdot,temperature]),
arc([fault,'e106 holed heatexchanger'],+,[e106_htot,temperature]),
arc([fault,'e106 holed heatexchanger'],+,[t101_boto,temperature]),
arc([fault,'e106 partly blocked (heated side)'],-,[t101_boto,temperature]),
arc([fault,'e104 partly blocked (cooling side)'],-,[e104_wot,flow]),
arc([fault,'e104 partly blocked (cooled side)'],-,[divider2_out2,flow]),
arc([fault,'e104 fouling'],+,[e104_stot,temperature]),
arc([fault,'e104 fouling'],-,[e104_wot,temperature]),
arc([fault,'e104 fouling'],+,[divider2_out2,temperature]),
arc([fault,'e104 blockage (cooling side)'],+,[e104_wot,noFlow]),
arc([fault,'e104 blockage (cooled side)'],+,[divider2_out2,noFlow]),
arc([fault,'e104 blockage (cooling side)'],+,[e104_win,pressure]),
arc([fault,'e104 blockage (cooling side)'],-,[e104_wot,pressure]),
arc([fault,'e104 blockage (cooled side)'],+,[e104_stin,pressure]),
arc([fault,'e104 blockage (cooling side)'],+,[e104_stot,temperature]),
arc([fault,'e104 blockage (cooling side)'],+,[divider2_out2,temperature]),
arc([fault,'e104 blockage (cooled side)'],-,[e104_wot,temperature]),
arc([fault,'e104 holed heatexchanger'],-,[e104_wot,flow]),
arc([fault,'e104 holed heatexchanger'],+,[e104_win,flow]),
arc([fault,'e104 holed heatexchanger'],+,[e104_wot,temperature]),
arc([fault,'e104 holed heatexchanger'],-,[divider2_out2,temperature]),
arc([fault,'d103 leak to environment'],-,[d103_liqd,level]),
arc([fault,'d103 external fire'],+,[d103_liqd,temperature]),
arc([fault,'d103 external fire'],+,[divider2_out2,temperature]),
arc([fault,'d103 cold weather'],-,[d103_liqd,temperature]),
arc([fault,'d103 cold weather'],-,[divider2_out2,temperature]),
arc([fault,'d103 leak to environment'],-,[divider2_out2,flow]),
arc([fault,'divider1 leak to environment'],-,[divider2_out2,flow]),
arc([fault,['divider1 blocked by frozen fluid', freezing]],+,[p101a_in,noFlow]),
arc([fault,['divider1 blocked by frozen fluid', freezing]],+,[divider2_out2,noFlow]),
arc([fault,'divider1 inlet completely blocked'],+,[p101a_in,noFlow]),
arc([fault,'divider1 inlet completely blocked'],+,[divider2_out2,noFlow]),
arc([fault,'divider1 inlet partly blocked'],-,[divider2_out2,flow]),
arc([fault,'p101a partly blocked'],-,[divider2_out2,flow]),
arc([fault,'p101a completely blocked'],+,[divider2_out2,noFlow]),
arc([fault,'p101a leak to environment'],-,[divider2_out2,flow]),
arc([fault,'p101a air lock'],-,[divider2_out2,flow]),
arc([fault,'p101a air lock'],+,[divider1_in,pressure]),
arc([fault,'p101a power supply fails'],+,[divider2_out2,noFlow]),
arc([fault,'p101a power supply fails'],+,[divider1_in,pressure]),
arc([fault,'p101a impellor failure'],-,[divider2_out2,flow]),
arc([fault,'p101a impellor failure'],+,[divider1_in,pressure]),
arc([fault,['p101a leak into vacuum system', vacuum]],+,[divider1_in,pressure]),
arc([fault,'header1 outlet partly blocked'],-,[divider2_out2,flow]),
arc([fault,'header1 outlet completely blocked'],+,[p101a_in,noFlow]),
arc([fault,'header1 outlet completely blocked'],+,[divider2_out2,noFlow]),
arc([fault,['header1 blocked by frozen fluid', freezing]],+,[p101a_in,noFlow]),
arc([fault,['header1 blocked by frozen fluid', freezing]],+,[divider2_out2,noFlow]),
```

arc([fault,'divider1 branch 1 completely blocked'],+,[p101a_in,noFlow]),
arc([fault,'divider1 branch 1 completely blocked'],+,[divider2_out2,noFlow]),
arc([fault,'divider1 branch 1 partly blocked'],-,[divider2_out2,flow]),
arc([fault,'valve5 partly blocked'],-,[divider2_out2,flow]),
arc([fault,'valve6 partly blocked'],-,[divider2_out2,flow]),
arc([fault,'valve5 completely blocked'],+,[p101a_in,noFlow]),
arc([fault,'valve6 completely blocked'],+,[divider2_out2,noFlow]),
arc([fault,'valve6 completely blocked'],+,[p101a_in,noFlow]),
arc([fault,'valve6 completely blocked'],+,[divider2_out2,noFlow]),
arc([fault,['valve5 blocked by frozen fluid', freezing]],+,[p101a_in,noFlow]),
arc([fault,['valve5 blocked by frozen fluid', freezing]],+,[divider2_out2,noFlow]),
arc([fault,['valve6 blocked by frozen fluid', freezing]],+,[p101a_in,noFlow]),
arc([fault,['valve6 blocked by frozen fluid', freezing]],+,[divider2_out2,noFlow]),
arc([fault,['valve5 leak into vacuum system', vacuum]],+,[divider1_in,pressure]),
arc([fault,['valve6 leak into vacuum system', vacuum]],+,[divider1_in,pressure]),
arc([fault,'valve5 closed in error'],+,[divider1_in,pressure]),
arc([fault,'valve5 closed in error'],+,[p101a_in,noFlow]),
arc([fault,'valve5 closed in error'],+,[divider2_out2,noFlow]),
arc([fault,'valve6 closed in error'],+,[p101a_out,pressure]),
arc([fault,'valve6 closed in error'],+,[divider1_in,pressure]),
arc([fault,'valve6 closed in error'],+,[p101a_in,noFlow]),
arc([fault,'valve6 closed in error'],+,[divider2_out2,noFlow]),
arc([fault,['valve5 leak into vacuum system', vacuum]],+,[divider1_in,pressure]),
arc([fault,'header1 branch 1 partly blocked'],-,[divider2_out2,flow]),
arc([fault,'header1 branch 1 completely blocked'],+,[p101a_in,noFlow]),
arc([fault,'header1 branch 1 completely blocked'],+,[divider2_out2,noFlow]),
arc([fault,'header1 outlet completely blocked'],+,[p101a_out,pressure]),
arc([fault,['header1 blocked by frozen fluid', freezing]],+,[p101a_out,pressure]),
arc([fault,'header1 branch 1 completely blocked'],+,[p101a_out,pressure]),
arc([fault,'valve6 completely blocked'],+,[p101a_out,pressure]),
arc([fault,['valve6 blocked by frozen fluid', freezing]],+,[p101a_out,pressure]),
arc([fault,'valve5 leak to environment'],-,[divider2_out2,flow]),
arc([fault,'valve5 leak to environment'],-,[divider2_out2,flow]),
arc([fault,'valve6 leak to environment'],-,[divider2_out2,flow]),
arc([fault,'valve5 thermal expansion'],+,[divider1_in,pressure]),
arc([fault,'valve6 thermal expansion'],+,[divider1_in,pressure]),
arc([fault,'header1 leak to environment'],-,[divider2_out2,flow]),
arc([fault,['header1 leak into vacuum system', vacuum]],+,[divider1_in,pressure]),
arc([fault,'valve5 thermal expansion'],+,[divider1_in,pressure]),
arc([fault,'divider2 leak to environment'],-,[divider2_out2,flow]),
arc([fault,['divider2 blocked by frozen fluid', freezing]],+,[divider2_out2,noFlow]),
arc([fault,'divider2 inlet completely blocked'],+,[divider2_out2,noFlow]),
arc([fault,'divider2 branch 2 completely blocked'],+,[divider2_out2,noFlow]),
arc([fault,'divider2 inlet partly blocked'],-,[divider2_out2,flow]),
arc([fault,'divider2 branch 2 partly blocked'],-,[divider2_out2,flow]),
arc([fault,['divider2 leak into vacuum system', vacuum]],+,[divider2_in,pressure]),
arc([fault,'e104 partly blocked (cooled side)'],-,[d103_liqd,level]),
arc([fault,'e104 blockage (cooled side)'],-,[d103_liqd,level]),
arc([fault,'divider1 leak to environment'],-,[d103_liqd,level]),
arc([fault,'p101a leak to environment'],-,[d103_liqd,level]),
arc([fault,'header1 leak to environment'],-,[d103_liqd,level]),
arc([fault,'valve5 leak to environment'],-,[d103_liqd,level]),
arc([fault,'valve6 leak to environment'],-,[d103_liqd,level]),
arc([fault,'valve5 leak to environment'],-,[d103_liqd,level]),
arc([fault,'p101a pump fails'],+,[d103_liqd,level]),
arc([fault,'divider1 inlet partly blocked'],+,[d103_liqd,level]),
arc([fault,'valve5 partly blocked'],+,[d103_liqd,level]),
arc([fault,['valve5 blocked by frozen fluid', freezing]],+,[d103_liqd,level]),
arc([fault,'valve5 completely blocked'],+,[d103_liqd,level]),
arc([fault,'valve5 closed in error'],+,[d103_liqd,level]),
arc([fault,'header1 branch 1 completely blocked'],+,[d103_liqd,level]),

```
arc([fault,'d103 outlet1 partly blocked'],-,[divider2_out2,flow]),
arc([fault,'d103 outlet1 completely blocked'],+,[divider2_out2,noFlow]),
arc([fault,'d103 outlet1 partly blocked'],+,[d103_liqd,level]),
arc([fault,'d103 outlet1 completely blocked'],+,[d103_liqd,level]),
arc([fault,'valve5 open or passing'],-,[d103_liqd,level]),
arc([fault,'valve3 thermal expansion of contents'],+,[divider2_in,pressure]),

%consequences resulting from faults
arc([fault,'e106 holed heatexchanger'],+,[consequence,'contamination of product']),
arc([fault,'e104 holed heatexchanger'],+,[consequence,'contamination of product']),
arc([fault,'d103 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'d103 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'d103 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'d103 external fire'],+,[consequence,'structural weakening']),
arc([fault,['d103 leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'d103 external fire'],+,[consequence,'possible rupture']),
arc([fault,'divider1 leak to environment'],+,[consequence,'contaminate environment']),
arc([fault,'divider1 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'divider1 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'valve5 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'valve5 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'valve5 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'valve5 thermal expansion'],+,[consequence,'flange leak']),
arc([fault,'p101a leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'p101a leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'p101a leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,['p101a leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'valve6 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'valve6 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'valve6 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'valve6 thermal expansion'],+,[consequence,'flange leak']),
arc([fault,'valve5 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'valve5 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'valve5 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'valve5 thermal expansion'],+,[consequence,'flange leak']),
arc([fault,'header1 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'header1 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'header1 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,['header1 leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'valve7 open or passing'],+,[consequence,'spare pump possible rupture']),
arc([fault,'divider2 leak to environment'],+,[consequence,'contaminate environment']),
arc([fault,'divider2 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'divider2 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,['divider2 leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'valve3 thermal expansion of contents'],+,[consequence,'flange leak']),

%consequences resulting from deviations
arc([deviation,[moreLevel,t101_self]],+,[consequence,'column floods']),
arc([deviation,[moreTemperature,e106_htot]],+,[consequence,'boiling in heatexchanger']),
arc([deviation,[lessTemperature,e106_cdot]],+,[consequence,'freezing in heatexchanger']),
arc([deviation,[moreTemperature,e106_htot]],+,[consequence,'polymerisation']),
arc([deviation,[morePressure,e106_cdin]],+,[consequence,'possible rupture']),
arc([deviation,[morePressure,e106_htin]],+,[consequence,'possible rupture']),
arc([deviation,[moreTemperature,e106_htot]],+,[consequence,'mechanical failure due to
    overtemperature']),
arc([deviation,[lessTemperature,e104_wot]],+,[consequence,'freezing in heatexchanger']),
arc([deviation,[moreTemperature,e104_stot]],+,[consequence,'boiling in heatexchanger']),
arc([deviation,[morePressure,e104_win]],+,[consequence,'possible rupture']),
arc([deviation,[morePressure,e104_stin]],+,[consequence,'possible rupture']),
arc([deviation,[moreLevel,d103_liqd]],+,[consequence,'overfilling']),
arc([deviation,[moreTemperature,d103_liqd]],+,[consequence,'increased evaporation']),
```

```
arc([deviation,[moreTemperature,d103_liqd]],+,[consequence,'flammable or toxic vapour release']),
arc([deviation,[moreTemperature,d103_liqd]],+,[consequence,'crystallisation']),
arc([deviation,[moreTemperature,d103_liqd]],+,[consequence,'viscosity increase']),
arc([deviation,[lessTemperature,d103_liqd]],+,[consequence,'viscosity decrease']),
arc([deviation,[lessLevel,d103_liqd]],+,[consequence,'vessel emptying']),
arc([deviation,[morePressure,divider1_in]],+,[consequence,'possible rupture']),
arc([deviation,[morePressure,p101a_out]],+,[consequence,'possible rupture']),
arc([deviation,[moreTemperature,p101a_out]],+,[consequence,'overheating']),
arc([deviation,[moreTemperature,p101a_out]],+,[consequence,'gland failure']),
arc([deviation,[noFlow,p101a_in]],+,[consequence,'overheating']),
arc([deviation,[noFlow,p101a_in]],+,[consequence,'gland failure']),
arc([deviation,[morePressure,divider2_in]],+,[consequence,'possible rupture'])
]
]
).
```

C.6. HAZOP Results (Modular Approach)

Examples from the HAZOP emulation results created by QUEEN are shown. For brevity the full set of results is not given.

feedinlet no flow out	valve1: completely blocked, feedinlet: inlet completely blocked, e103: blockage (heated side)		
feedinlet less flow out	valve1: partly blocked, feedinlet: inlet partly blocked, e103: partly blocked (heated side)		
	feedinlet: leak to environment	contaminate environment, fire/explosion risk, loss of material	
feedinlet more flow out	valve1: leak to environment	contaminate environment, loss of material, fire/explosion risk	
feedinlet less temperature out	feedinlet: cold weather		
feedinlet more pressure out	valve1: thermal expansion of contents	flange leak	
	valve1: closed in error		
e103 no flow cdin	tail1: blocked, heatsource1: inlet completely blocked, e103: blockage (heating side)		

e103 less flow cdin	tail1: partly blocked, heatsource1: inlet partly blocked, e103: partly blocked (heating side)		
	heatsource1: leak to environment	contaminate environment, fire/explosion risk, loss of material	

- etc. -

e105 more temperature stot	watersource2: leak to environment, tail6: leak to environment	boiling in heatexchanger, contaminate environment, fire/explosion risk, loss of material	
	watersource2: inlet partly blocked, inlet completely blocked, tail5: partly blocked, blocked, e105: partly blocked (cooling side), fouling, blockage (cooling side), distilsystem3_: e104 fouling, e104 blockage (cooling side)	boiling in heatexchanger	
	distilsystem3_: d103 external fire	boiling in heatexchanger, structural weakening, possible rupture	
tail5 less temperature in	e105: fouling		
distilsystem3_ more pressure e104_win	tail4: blocked, distilsystem3_: e104 blockage (cooling side)	possible rupture	
distilsystem3_ more pressure e106_cdin	tail2: blocked, distilsystem3_: e106 blockage (heating side)	possible rupture	
distilsystem3_ less temperature e106_cdot	tail2: partly blocked, heatsource2: inlet partly blocked, cold weather, distilsystem3_: e106 partly blocked	freezing in heatexchanger	

	(heating side)		
	heatsource2: leak to environment	freezing in heatexchanger, contaminate environment, fire/explosion risk, loss of material	
	distilsystem3_: e106 holed heatexchanger	freezing in heatexchanger, contamination of product	
distilsystem3_ less temperature e104_wot	watersource1: cold weather, distilsystem3_: e104 fouling	freezing in heatexchanger	
	tail4: leak to environment	freezing in heatexchanger, contaminate environment, fire/explosion risk, loss of material	
distilsystem3_ more level t101_self	valve2: completely blocked, tail3: blocked, distilsystem3_: t101 packing support collapses	column floods	
distilsystem3_ more pressure e106_htin	distilsystem3_: e106 blockage (heated side)	possible rupture	
distilsystem3_ more temperature e106_htot	tail2: leak to environment	boiling in heatexchanger, polymerisation, mechanical failure due to overtemperature, contaminate environment, fire/explosion risk, loss of material	
	distilsystem3_: t101 external fire	boiling in heatexchanger, polymerisation, mechanical failure due to overtemperature	
	distilsystem3_: e106 holed heatexchanger	boiling in heatexchanger, polymerisation,	

		mechanical failure due to overtemperature, contamination of product
distilsystem3_ more pressure e104_stin	distilsystem3_: e104 blockage (cooled side)	possible rupture
distilsystem3_ more temperature e104_stot	watersource1: leak to environment	boiling in heatexchanger, contaminate environment, fire/explosion risk, loss of material
	watersource1: inlet partly blocked, inlet completely blocked, tail4: partly blocked, blocked, distilsystem3_: e104 partly blocked (cooling side)	boiling in heatexchanger
distilsystem3_ more temperature d103_liqd	distilsystem3_: d103 external fire	increased evaporation, flammable or toxic vapour release, crystallisation, viscosity increase, structural weakening, possible rupture
distilsystem3_ less temperature d103_liqd	distilsystem3_: d103 cold weather	viscosity decrease
distilsystem3_ more level d103_liqd	distilsystem3_: valve5 partly blocked, valve5 completely blocked, valve5 closed in error, p101a pump fails, header1 branch 1 completely blocked, divider1 inlet partly blocked, d103 outlet1 partly blocked, d103 outlet1 completely blocked	overfilling
distilsystem3_ less level d103_liqd	distilsystem3_: valve6 leak to environment, header1 leak to environment	vessel emptying, contaminate environment, loss of material, fire/explosion risk
	distilsystem3_:	vessel emptying,

	valve7 open or passing	spare pump possible rupture	
	distilsystem3_: valve5 leak to environment	vessel emptying, contaminate environment, loss of material, fire/explosion risk, contaminate environment, loss of material, fire/explosion risk	
	distilsystem3_: p101a leak to environment, divider1 leak to environment, d103 leak to environment	vessel emptying, contaminate environment, fire/explosion risk, loss of material	
	distilsystem3_: e104 partly blocked (cooled side), e104 blockage (cooled side)	vessel emptying	
distilsystem3_ more pressure divider1_in	distilsystem3_: valve6 thermal expansion	possible rupture, flange leak	
	distilsystem3_: valve6 closed in error, valve5 closed in error, p101a power supply fails, p101a impellor failure, p101a air lock	possible rupture	
	distilsystem3_: valve7 thermal expansion	possible rupture, flange leak, flange leak	
distilsystem3_ no flow p101a_in	distilsystem3_: valve6 completely blocked, valve6 closed in error, valve5 completely blocked, valve5 closed in error, header1 outlet completely blocked, header1 branch 1 completely blocked, divider1 inlet completely blocked, divider1 branch 1 completely blocked	overheating, gland failure	
distilsystem3_ more pressure p101a_out	distilsystem3_: valve6 completely blocked, header1	possible rupture	

	outlet completely blocked, header1 branch 1 completely blocked		
distilsystem3_ more pressure divider2_in	valve4: closed in error	possible rupture	
	distilsystem3_: valve3 thermal expansion of contents	possible rupture, flange leak	

Appendix D

Olefin Dimerisation Plant

This appendix contains input data for the application QUEEN (Chung, 1993) prepared for an olefin dimerisation plant. Two sets of input data have been prepared: a standard set and a set to which the modular approach has been applied. Each set of input data consists of a modified plant description and a file of models occurring within the plant. Example HAZOP results created by QUEEN are given for both sets of data.

D.1. Modified Plant Description (Standard Approach)

See section 7.1. for a plant diagram.

```
instance(feedinlet isa source,
   [outports info
      [out is [valve1,in]]
   ]
).

instance(valve1 isa valve,
   [outports info
      [out is [feedpipe,in]],
      aperture is open
   ]
).

instance(feedpipe isa pipe,
   [outports info
      [out is [storagetank,in1]]
   ]
).

instance(storagetank isa blanketedvessel,
   [outports info
      [out1 is [valve2,in],vout is [valve18,in]]
   ]
).

instance(valve2 isa valve,
   [outports info
      [out is [divider1,in]],
      aperture is open
   ]
).

instance(divider1 isa divider,
   [outports info
      [out1 is [valve3,in],out2 is [valve5,in]],
      purpose is tobypass
   ]
).

instance(valve3 isa valve,
   [outports info
      [out is [pumpj1a,in]],
      aperture is open
   ]
).

instance(valve5 isa valve,
   [outports info
      [out is [pumpj1b,in]],
      aperture is closed
   ]
).

instance(pumpj1a isa pump,
   [outports info
      [out is [valve4,in]]
   ]
).

instance(pumpj1b isa pump,
   [outports info
      [out is [valve6,in]],
      status is spare
   ]
).

instance(valve4 isa valve,
   [outports info
      [out is [header1,in1]],
      aperture is open
   ]
).

instance(valve6 isa valve,
   [outports info
      [out is [header1,in2]],
      aperture is closed
   ]
).
```

```
instance(header1 isa header,
   [outports info
      [out is [valve7,in]],
      purpose is frombypass
   ]
).

instance(valve7 isa valve,
   [outports info
      [out is [halfmileline,in]],
      aperture is open
   ]
).

instance(halfmileline isa pipeline,
   [outports info
      [out is [valve8,in]]
   ]
).

instance(valve8 isa valve,
   [outports info
      [out is [buffertank,in1]],
      aperture is open
   ]
).

instance(buffertank isa decantingvessel,
   [outports info
         [out1 is [valve10,in],out2 is [valve9,in],
         vout is [valve20,in]]
   ]
).

instance(valve9 isa valve,
   [outports info
      [out is [tail1,in]],
      aperture is closed
   ]
).

instance(tail1 isa outlet,
   [outports info
      []
   ]
).

instance(valve10 isa valve,
   [outports info
      [out is [divider2,in]],
      aperture is open
   ]
).

instance(divider2 isa divider,
   [outports info
      [out1 is [valve11,in],out2 is [valve13,in]],
      purpose is tobypass
   ]
).

instance(valve11 isa valve,
   [outports info
      [out is [pumpj2a,in]],
      aperture is open
   ]
).

instance(valve13 isa valve,
   [outports info
      [out is [pumpj2b,in]],
      aperture is closed
   ]
).

instance(pumpj2a isa pump,
   [outports info
      [out is [valve12,in]]
   ]
).

instance(pumpj2b isa pump,
   [outports info
      [out is [valve14,in]],
      status is spare
   ]
).

instance(valve12 isa valve,
   [outports info
      [out is [header2,in1]],
      aperture is open
   ]
).

instance(valve14 isa valve,
   [outports info
      [out is [header2,in2]],
      aperture is closed
   ]
).

instance(header2 isa header,
   [outports info
      [out is [valve15,in]],
      purpose is frombypass
   ]
).

instance(valve15 isa valve,
   [outports info
      [out is [heatex1,htin]],
      aperture is open
   ]
).

instance(heatex1 isa heatex,
   [outports info
      [cdot is [tail2,in],htot is [heatex2,htin]]
   ]
).
```

```
instance(tail2 isa outlet,
   [outports info
      []
   ]
).

instance(heatex2 isa heatex,
   [outports info
      [cdot is [tail3,in],htot is [reactor1,in1]]
   ]
).

instance(tail3 isa outlet,
   [outports info
      []
   ]
).

instance(reactor1 isa closedvessel,
   [outports info
      [out1 is [valve16,in]]
   ]
).

instance(valve16 isa valve,
   [outports info
      [out is [heatex1,cdin]],
      aperture is open
   ]
).

instance(head2 isa steamsupply,
   [outports info
      [out is [heatex2,cdin]]
   ]
).

instance(nitogeninlet1 isa nitrogensupply,
   [outports info
      [out is [valve17,in]]
   ]
).

instance(valve17 isa valve,
   [outports info
      [out is [storagetank,vin]],
      aperture is open
   ]
).

instance(valve18 isa valve,
   [outports info
      [out is [flaresystem1,in]],
      aperture is open
   ]
).

instance(flaresystem1 isa outlet,
   [outports info
      []
   ]
).

instance(nitrogeninlet2 isa nitrogensupply,
   [outports info
      [out is [valve19,in]]
   ]
).

instance(valve19 isa valve,
   [outports info
      [out is [buffertank,vin]],
      aperture is open
   ]
).

instance(valve20 isa valve,
   [outports info
      [out is [flaresystem2,in]],
      aperture is open
   ]
).

instance(flaresystem2 isa outlet,
   [outports info
      []
   ]
).
```

D.2. Plant Models (Standard Approach)

Models 'source', 'valve', 'divider', 'pump', 'header', 'closedvessel', 'outlet' and 'heatex' are duplicates of the same models given for the standard approach for the benzene purification system (section C.2) and have been removed to avoid repetition.

```
/*-------------------------------------------*/
/*
Short length of  pipe
/*
/*-------------------------------------------*/

frame(pipe isa unit,
  [  inports info  [ in ],
```

```
outports info  [ out ],
construction is unknown,
lagged is unknown,
propLinks info [

%propagation
arc([out,pressure],+,[in,pressure]),
arc([in,pressure],+,[out,pressure]),
arc([in,temperature],+,[out,temperature]),
arc([out,flow],+,[in,flow]),
arc([in,flow],+,[out,flow]),
arc([out,reverseFlow],++,[in,reverseFlow]),
arc([out,noFlow],++,[in,noFlow]),
arc([in,noFlow],++,[out,noFlow]),
arc([in,concentration],+,[out,concentration]),

%faults
arc([fault,'partly blocked'],-,[out,flow]),
arc([fault,'completely blocked'],+,[out,noFlow]),
arc([fault,['blocked by frozen fluid', freezing]],+,[out,noFlow]),
arc([fault,'leak to environment'],-,[out,flow]),
arc([fault,'leak to environment'],+,[in,flow]),
arc([fault,'pipe fracture'],+,[in,flow]),
arc([fault,'pipe fracture'],-,[out,flow]),
arc([fault,['leak into vacuum system', vacuum]],+,[in,pressure]),

%consequences resulting from faults
arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'pipe fracture'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
    arc([fault,['leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),

    %consequences resulting from deviations
    arc([deviation,[morePressure,in]],+,[consequence,'possible rupture'])
    ]
   ]
).
/*-----------------------------------------------*/
/*
Blanketed vessel for lawley plant

vin and vout are input and output for nitrogen
liqd = liquid, vapr = vapr
/*
/*-----------------------------------------------*/

frame(blanketedvessel isa closedvessel,
[  inports info  [ in1, vin ],
   outports info  [ out1, vout ],
   unitports info  [ liqd, vapr ],
   propLinks info [

   %propagation
   arc([in1,temperature],+,[liqd,temperature]),
   arc([liqd,temperature],+,[out1,temperature]),
   arc([in1,flow],+,[liqd,level]),
   arc([out1,flow],-,[liqd,level]),
   arc([vin,flow],+,[vapr,pressure]),
   arc([vout,flow],-,[vapr,pressure]),
   arc([vout,flow],+,[in1,flow]),
```

```
arc([vin,flow],-,[in1,flow]),
arc([in1,noFlow],- -,[liqd,level]),
arc([out1,noFlow],++,[liqd,level]),
arc([vout,noFlow],++,[vapr,pressure]),
arc([in1,noFlow],++,[out1,noFlow]),
arc([vin,noFlow],++,[in1,flow]),
arc([vout,noFlow],- -,[in1,flow]),
arc([out1,reverseFlow],++,[liqd,level]),
arc([in1,pressure],+,[vapr,pressure]),
arc([vin,pressure],+,[vapr,pressure]),
arc([vout,pressure],+,[vapr,pressure]),
arc([in1,pressure],+,[liqd,level]),
arc([out1,pressure],+,[liqd,level]),
arc([out1,pressure],+,[vapr,pressure]),
arc([in1,concentration],+,[liqd,concentration]),
arc([liqd,concentration],+,[out1,concentration]),
arc([vin,concentration],+,[vapr,concentration]),
arc([vapr,concentration],+,[vout,concentration]),

%faults
arc([fault,'outlet1 partly blocked'],-,[out1,flow]),
arc([fault,'outlet1 completely blocked'],+,[out1,noFlow]),
arc([fault,'leak to environment'],-,[liqd,level]),
arc([fault,'external fire'],+,[liqd,temperature]),
arc([fault,'cold weather'],-,[liqd,temperature]),
arc([fault,'inlet1 siphon breaker blockage'],+,[in1,reverseFlow]),
arc([fault,['leak into vacuum system', vacuum]],+,[in1,pressure]),
arc([fault,'leak to environment'],-,[out1,flow]),
arc([fault,'leak to environment'],+,[in1,flow]),
arc([fault,'outlet1 partly blocked'],+,[vapr,pressure]),
arc([fault,'outlet1 completely blocked'],+,[vapr,pressure]),

%consequences resulting from faults
arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'external fire'],+,[consequence,'structural weakening']),
arc([fault,['leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),

%consequences resulting from deviations
arc([deviation,[moreLevel,liqd]],+,[consequence,'overfilling']),
arc([deviation,[moreConcentration,liqd]],+,[consequence,'rubber lining corrosion in hot caustic']),
arc([deviation,[moreConcentration,liqd]],+,[consequence,'emulsification of contents ']),
arc([deviation,[moreTemperature,liqd]],+,[consequence,'increased evaporation']),
arc([deviation,[moreTemperature,liqd]],+,[consequence,'flammable or toxic vapour release']),
arc([deviation,[moreTemperature,liqd]],+,[consequence,'crystallisation']),
arc([deviation,[moreTemperature,liqd]],+,[consequence,'viscosity increase']),
arc([deviation,[lessTemperature,liqd]],+,[consequence,'viscosity decrease']),
    arc([deviation,[lessLevel,liqd]],+,[consequence,'vessel emptying']),
    arc([deviation,[morePressure,vapr]],+,[consequence,'possible rupture']),
    arc([deviation,[noFlow,vin]],+,[consequence,['loss of blanket and explosion risk', flammable]]),
    arc([deviation,[lessFlow,vin]],+,[consequence,'vacuum collapse']),
    arc([deviation,[moreFlow,vout]],+,[consequence,['loss of material', expensive]]),
    arc([deviation,[moreFlow,in1]],+,[consequence,'incomplete separation of water'])
    ]
   ]
).
/*-----------------------------------------------*/
/*
```

The pipeline model is to represent the fact that pipelines can be subject to hydraulic hammer because of their length.

```
/*
/*-----------------------------------------------*/

frame(pipeline isa pipe,
  [  inports info  [ in ],
     outports info  [ out ],
     construction is unknown,
     lagged is unknown,
     propLinks info [

     %propagation
     arc([out,pressure],+,[in,pressure]),
     arc([in,pressure],+,[out,pressure]),
     arc([in,temperature],+,[out,temperature]),
     arc([out,flow],+,[in,flow]),
     arc([in,flow],+,[out,flow]),
     arc([out,reverseFlow],++,[in,reverseFlow]),
     arc([out,noFlow],++,[in,noFlow]),
     arc([in,noFlow],++,[out,noFlow]),
     arc([in,concentration],+,[out,concentration]),

     %faults
     arc([fault,'partly blocked'],-,[out,flow]),
     arc([fault,'completely blocked'],+,[out,noFlow]),
     arc([fault,['blocked by frozen fluid', freezing]],+,[out,noFlow]),
     arc([fault,'leak to environment'],-,[out,flow]),
     arc([fault,'leak to environment'],+,[in,flow]),
     arc([fault,'pipe fracture'],+,[in,flow]),
     arc([fault,'pipe fracture'],-,[out,flow]),
     arc([fault,['leak into vacuum system', vacuum]],+,[in,pressure]),
     arc([fault,'liquid hammer'],+,[out,pressure]),
     arc([fault,'external heat'],+,[out,temperature]),
     arc([fault,'external cold'],-,[out,temperature]),
     arc([fault,'thermal expansion'],+,[in,pressure]),
     arc([fault,'thermal expansion'],+,[out,pressure]),

     %consequences resulting from faults
     arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
     arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]]),
     arc([fault,'pipe fracture'],+,[consequence,['contaminate environment', toxic]]),
     arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
     arc([fault,['leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
     arc([fault,'liquid hammer'],+,[consequence,'pipeline supports broken']),
     arc([fault,'external cold'],+,[consequence,'freeze up']),
     arc([fault,'thermal expansion'],+,[consequence,'possible rupture']),

     %consequences resulting from deviations
     arc([deviation,[morePressure,in]],+,[consequence,'possible rupture'])
     ]
  ]
).
/*-----------------------------------------------*/
/*
decanting vessel for lawley plant
vin and vout are input and output for nitrogen
out1 is main outlet
liqd = liquid, vapr = vapour
/*
/*-----------------------------------------------*/

frame(decantingvessel isa closedvessel,
```

```
[  inports info  [ in1, vin ],
   outports info  [ out1, vout, out2 ],
   unitports info  [ liqd, vapr ],
   propLinks info [

   %propagation
   arc([in1,temperature],+,[liqd,temperature]),
   arc([liqd,temperature],+,[out1,temperature]),
   arc([liqd,temperature],+,[out2,temperature]),
   arc([in1,flow],+,[liqd,level]),
   arc([out1,flow],-,[liqd,level]),
   arc([out2,flow],-,[liqd,level]),
   arc([vin,flow],+,[vapr,pressure]),
   arc([vout,flow],-,[vapr,pressure]),
   arc([vin,flow],-,[in1,flow]),
   arc([vout,flow],+,[in1,flow]),
   arc([in1,noFlow],- -,[liqd,level]),
   arc([out1,noFlow],++,[liqd,level]),
   arc([in1,noFlow],++,[out1,noFlow]),
   arc([out2,noFlow],++,[liqd,level]),
   arc([vout,noFlow],++,[vapr,pressure]),
   arc([in1,noFlow],++,[out2,noFlow]),
   arc([vin,noFlow],++,[in1,flow]),
   arc([vout,noFlow],- -,[in1,flow]),
   arc([out1,reverseFlow],++,[liqd,level]),
   arc([out2,reverseFlow],++,[liqd,level]),
   arc([in1,pressure],+,[vapr,pressure]),
   arc([vin,pressure],+,[vapr,pressure]),
   arc([vout,pressure],+,[vapr,pressure]),
   arc([in1,pressure],+,[liqd,level]),
   arc([out1,pressure],+,[liqd,level]),
   arc([out2,pressure],+,[liqd,level]),
   arc([out1,pressure],+,[vapr,pressure]),
   arc([out2,pressure],+,[vapr,pressure]),
   arc([in1,concentration],+,[liqd,concentration]),
   arc([liqd,concentration],+,[out1,concentration]),
   arc([vin,concentration],+,[vapr,concentration]),
   arc([vapr,concentration],+,[vout,concentration]),
   arc([liqd,concentration],+,[out2,concentration]),

   %faults
   arc([fault,'outlet1 partly blocked'],-,[out1,flow]),
   arc([fault,'outlet1 completely blocked'],+,[out1,noFlow]),
   arc([fault,'leak to environment'],-,[liqd,level]),
   arc([fault,'external fire'],+,[liqd,temperature]),
   arc([fault,'cold weather'],-,[liqd,temperature]),
   arc([fault,'inlet1 siphon breaker blockage'],+,[in1,reverseFlow]),
   arc([fault,['leak into vacuum system', vacuum]],+,[in1,pressure]),
   arc([fault,'outlet2 completely blocked'],+,[out2,noFlow]),
   arc([fault,'outlet2 partly blocked'],-,[out2,flow]),
   arc([fault,'leak to environment'],-,[out1,flow]),
   arc([fault,'leak to environment'],-,[out2,flow]),
   arc([fault,'leak to environment'],+,[in1,flow]),

   %consequences resulting from faults
   arc([fault,'leak to environment'],+,[consequence,['contaminate environment', toxic]]),
   arc([fault,'leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
   arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]]),
   arc([fault,'external fire'],+,[consequence,'structural weakening']),
   arc([fault,['leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
   arc([fault,'external fire'],+,[consequence,'possible rupture']),
```

```
%consequences resulting from deviations
arc([deviation,[moreLevel,liqd]],+,[consequence,'overfilling']),
arc([deviation,[moreConcentration,liqd]],+,[consequence,'rubber lining corrosion in hot caustic']),
arc([deviation,[moreConcentration,liqd]],+,[consequence,'emulsification of contents ']),
arc([deviation,[moreTemperature,liqd]],+,[consequence,'increased evaporation']),
arc([deviation,[moreTemperature,liqd]],+,[consequence,'flammable or toxic vapour release']),
arc([deviation,[moreTemperature,liqd]],+,[consequence,'crystallisation']),
arc([deviation,[moreTemperature,liqd]],+,[consequence,'viscosity increase']),
arc([deviation,[lessTemperature,liqd]],+,[consequence,'viscosity decrease']),
arc([deviation,[lessLevel,liqd]],+,[consequence,'vessel emptying']),
arc([deviation,[morePressure,vapr]],+,[consequence,'possible rupture']),
arc([deviation,[noFlow,vin]],+,[consequence,['loss of blanket and explosion risk', flammable]]),
arc([deviation,[lessFlow,vin]],+,[consequence,'vacuum collapse']),
arc([deviation,[moreFlow,vout]],+,[consequence,['loss of material', expensive]]),
    arc([deviation,[moreFlow,in1]],+,[consequence,'incomplete separation of water'])
    ]
   ]
).
/*-----------------------------------------------*/
/*
Dummy source of plant
/*
/*-----------------------------------------------*/

frame(nitrogensupply isa unit,
  [ outports info  [ out ],
    propLinks info [

    %faults
    arc([fault,'inlet completely blocked'],+,[out,noFlow]),
    arc([fault,'inlet partly blocked'],-,[out,flow]),
    arc([fault,'inlet completely blocked'],-,[out,pressure]),
    arc([fault,'leak to environment'],-,[out,flow]),

    %consequences resulting from faults
    arc([fault,'leak to environment'],+,[consequence,['loss of material', expensive]])
    ]
   ]
).
```

D.3. HAZOP Results (Standard Approach)

Examples from the HAZOP emulation results created by QUEEN are shown. For brevity the full set of results is not given. The examples include the errors in the HAZOP analysis identified in section 7.4.2.2.

feedinlet no flow out	valve1: completely blocked, feedpipe: completely blocked, feedinlet: inlet completely blocked		
feedinlet less flow out	valve18: partly blocked, completely blocked, flaresystem1: partly blocked, blocked, feedpipe: partly		

	blocked, feedinlet: inlet partly blocked		
	feedinlet: leak to environment	contaminate environment, fire/explosion risk, loss of material	
feedinlet more flow out	valve18: leak to environment, feedpipe: leak to environment	contaminate environment, loss of material, fire/explosion risk	
	valve17: partly blocked, completely blocked, nitogeninlet1: inlet partly blocked, inlet completely blocked		
	storagetank: leak to environment, flaresystem1: leak to environment	contaminate environment, fire/explosion risk, loss of material	
	nitogeninlet1: leak to environment	loss of material	
	feedpipe: pipe fracture	contaminate environment	
feedinlet less temperature out	feedinlet: cold weather		
feedinlet more pressure out	valve1: thermal expansion of contents	flange leak	
	valve1: closed in error		
feedinlet reverse flow out	storagetank: inlet1 siphon breaker blockage		
feedpipe more pressure in	valve1: thermal expansion of contents	possible rupture, flange leak	
storagetank no flow vin	valve17: completely blocked, nitogeninlet1: inlet completely blocked	loss of blanket and explosion risk	

- etc.-

storagetank less flow out1	valve3: partly blocked, storagetank: outlet1 partly blocked, divider1: inlet partly blocked,	

	branch 2 partly blocked, branch 1 partly blocked		
	- etc.-		
storagetank more level liqd	valve5: thermal expansion of contents	overfilling, flange leak	
	valve3: partly blocked, closed in error, valve2: completely blocked, storagetank: outlet1 partly blocked, outlet1 completely blocked, nitogeninlet1: inlet partly blocked, inlet completely blocked, divider1: inlet partly blocked, inlet completely blocked, branch 2 partly blocked, branch 1 partly blocked	overfilling	
	valve18: leak to environment	overfilling, contaminate environment, loss of material, fire/explosion risk	
	nitogeninlet1: leak to environment	overfilling, loss of material	
	flaresystem1: leak to environment	overfilling, contaminate environment, fire/explosion risk, loss of material	
	- etc.-		
divider1 no flow out2	divider1: branch 2 completely blocked		
valve5 more flow out	valve5: open or passing		
	pumpj1b: leak to environment	contaminate environment, fire/explosion risk, loss of material	
	- etc.-		
pumpj1b less flow out	valve6: thermal	flange leak	

	expansion of contents		
pumpj1b more flow out	valve6: leak to environment	contaminate environment, loss of material, fire/explosion risk	
	pumpj1b: spare unit turned on		
pumpj1b more pressure out	valve6: thermal expansion of contents	possible rupture, flange leak	
	pumpj1b: spare unit turned on	possible rupture	
valve6 no flow out	header1: branch 2 completely blocked		
valve6 less flow out	header1: branch 2 partly blocked		
valve6 more flow out	valve6: open or passing		

- etc.-

buffertank less flow out1	valve11: partly blocked, divider2: inlet partly blocked, branch 2 partly blocked, branch 1 partly blocked, buffertank: outlet1 partly blocked		

- etc.-

buffertank more level liqd	valve9: thermal expansion of contents	overfilling, flange leak	
	valve20: leak to environment	overfilling, contaminate environment, loss of material, fire/explosion risk	
	valve19: partly blocked, completely blocked, valve11: closed in error, nitrogeninlet2: inlet partly blocked, inlet completely blocked, divider2: inlet partly blocked, inlet completely blocked, branch 2 partly blocked, branch 1 partly blocked, buffertank: outlet2	overfilling	

	partly blocked, outlet2 completely blocked, outlet1 partly blocked, outlet1 completely blocked		
	nitrogeninlet2: leak to environment	overfilling, loss of material	
	halfmileline: thermal expansion	overfilling, possible rupture	
	halfmileline: liquid hammer	overfilling, pipeline supports broken	
	flaresystem2: leak to environment	overfilling, contaminate environment, fire/explosion risk, loss of material	

- etc.-

divider2 no flow out	divider2: branch 2 completely blocked		
valve13 more flow out	valve13: open or passing		
	pumpj2b: leak to environment	contaminate environment, fire/explosion risk, loss of material	

- etc.-

pumpj2b less flow out	valve14: thermal expansion of contents	flange leak	
pumpj2b more flow out	valve14: leak to environment	contaminate environment, loss of material, fire/explosion risk	
	pumpj2b: spare unit turned on		
pumpj2b more pressure out	valve14: thermal expansion of contents	possible rupture, flange leak	
	pumpj2b: spare unit turned on	possible rupture	
valve14 no flow out	header2: branch 2 completely blocked		
valve14 less flow out	header2: branch 2		

	partly blocked		
valve14 more flow out	valve14: open or passing		
	- etc.-		
reactor1 more level liqd	valve16: thermal expansion of contents	overfilling, flange leak	
	valve16: partly blocked, completely blocked, closed in error, tail2: partly blocked, blocked, reactor1: outlet1 partly blocked, outlet1 completely blocked, heatex1: partly blocked (heating side), blockage (heating side)	overfilling	
	-etc. -		
reactor1 less level liqd	valve16: leak to environment, header2: leak to environment	vessel emptying, contaminate environment, loss of material, fire/explosion risk	
	valve15: partly blocked, completely blocked, closed in error, heatex2: partly blocked (heated side), blockage (heated side), header2: outlet partly blocked, outlet completely blocked	vessel emptying	
	tail2: leak to environment, reactor1: leak to environment	vessel emptying, contaminate environment, fire/explosion risk, loss of material	
	heatex1: holed heatexchanger	vessel emptying, contamination of product	
reactor1 more pressure vapr	valve16: closed in error	possible rupture	

D.4. Modified Plant Description (Modular Approach)

See section 7.1. for a plant diagram.

```
instance(feedinlet isa source,
   [outports info
      [out is [valve1,in]]
   ]
).

instance(valve1 isa valve,
   [outports info
      [out is [feedpipe,in]],
      aperture is open
   ]
).

instance(feedpipe isa pipe,
   [outports info
      [out is [storagetank,in1]]
   ]
).

instance(storagetank isa blanketedvessel,
   [outports info
      [out1 is [valve2,in],vout is [valve18,in]]
   ]
).

instance(valve2 isa valve,
   [outports info
      [out is [byps1_,divider1_in]],
      aperture is open
   ]
).

instance(valve7 isa valve,
   [outports info
      [out is [halfmileline,in]],
      aperture is open
   ]
).

instance(halfmileline isa pipeline,
   [outports info
      [out is [valve8,in]]
   ]
).

instance(valve8 isa valve,
   [outports info
      [out is [buffertank,in1]],
      aperture is open
   ]
).

instance(buffertank isa decantingvessel,
   [outports info
         [out1 is [valve10,in],out2 is [valve9,in],
         vout is [valve20,in]]
   ]
).

instance(valve9 isa valve,
   [outports info
      [out is [tail1,in]],
      aperture is closed
   ]
).

instance(tail1 isa outlet,
   [outports info
      []
   ]
).

instance(valve10 isa valve,
   [outports info
      [out is [byps2_,divider2_in]],
      aperture is open
   ]
).

instance(valve15 isa valve,
   [outports info
      [out is [heatingloop3_,heatex1_htin]],
      aperture is open
   ]
).

instance(tail2 isa outlet,
   [outports info
      []
   ]
).

instance(tail3 isa outlet,
   [outports info
      []
   ]
).

instance(head2 isa steamsupply,
   [outports info
      [out is [heatingloop3_,heatex2_cdin]]
   ]
).

instance(nitogeninlet1 isa nitrogensupply,
   [outports info
      [out is [valve17,in]]
   ]
).

instance(valve17 isa valve,
   [outports info
      [out is [storagetank,vin]],
      aperture is open
```

```
    ]
).

instance(valve18 isa valve,
    [outports info
        [out is [flaresystem1,in]],
        aperture is open
    ]
).

instance(flaresystem1 isa outlet,
    [outports info
        []
    ]
).

instance(nitrogeninlet2 isa nitrogensupply,
    [outports info
        [out is [valve19,in]]
    ]
).

instance(valve19 isa valve,
    [outports info
        [out is [buffertank,vin]],
        aperture is open
    ]
).

instance(valve20 isa valve,
    [outports info
        [out is [flaresystem2,in]],
        aperture is open
    ]
).

instance(flaresystem2 isa outlet,
    [outports info
        []
    ]
).

instance(byps1_ isa module_1,
    [outports info
        [header1_out is [valve7,in]],
        divider1_purpose is tobypass,
        valve3_aperture is open,
        valve4_aperture is open,
        valve5_aperture is closed,
        pumpj1b_status is spare,
        valve6_aperture is closed,
        header1_purpose is frombypass
    ]
).

instance(byps2_ isa module_2,
    [outports info
        [header2_out is [valve15,in]],
        divider2_purpose is tobypass,
        valve11_aperture is open,
        valve12_aperture is open,
        valve13_aperture is closed,
        pumpj2b_status is spare,
        valve14_aperture is closed,
        header2_purpose is frombypass
    ]
).

instance(heatingloop3_ isa module_3,
    [outports info
        [heatex1_cdot is [tail2,in],
        heatex2_cdot is [tail3,in]],
        valve16_aperture is open
    ]
).
```

D.5. Plant Models (Modular Approach)

Models 'pipe', 'blanketedvessel', 'pipeline', 'decantingvessel', 'steamsupply' and 'nitrogensupply' are duplicates of models given for the standard approach (section D.2). Models 'source', 'valve' and 'outlet' are duplicates of the same models given for the standard approach for the benzene purification system (section C.2). These models have been removed to avoid repetition.

```
/*-----------------------------------------------*/
/*
The module is composed of the following units:
heatex1 htot connected to heatex2 htin,
heatex2 htot connected to reactor1 in1,
reactor1 out1 connected to valve16 in,
valve16 out connected to heatex1 cdin.
Conditions of module are:
      valve16_aperture is open
and this completes the module.
```

```
/*
/*-------------------------------------------------*/

frame(module_3 isa module,
  [ inports info  [ heatex1_htin, heatex2_cdin ],
    outports info  [ heatex1_cdot, heatex2_cdot ],
    unitports info  [ heatex1_cdin, heatex1_htot, heatex2_htin, heatex2_htot, reactor1_in1, reactor1_liqd,
                      reactor1_vapr ],
    valve16_aperture is unknown,
    propLinks info [

    %propagation
    arc([heatex1_htin,flow],+,[heatex1_htot,temperature]),
    arc([heatex1_cdot,pressure],+,[heatex1_cdin,pressure]),
    arc([heatex2_cdot,flow],+,[heatex2_cdin,flow]),
    arc([heatex2_cdin,flow],+,[heatex2_cdot,temperature]),
    arc([heatex1_htin,flow],+,[reactor1_in1,flow]),
    arc([heatex1_htin,flow],+,[reactor1_liqd,level]),
    arc([heatex2_cdin,flow],+,[reactor1_liqd,temperature]),
    arc([heatex1_htin,flow],+,[heatex2_htot,temperature]),
    arc([heatex1_cdot,flow],-,[reactor1_liqd,level]),
    arc([heatex2_cdin,flow],+,[heatex1_cdot,temperature]),
    arc([heatex2_cdin,flow],+,[heatex2_cdot,flow]),
    arc([heatex2_cdin,flow],+,[heatex2_htot,temperature]),
    arc([heatex2_cdin,temperature],+,[heatex2_cdot,temperature]),
    arc([heatex1_htin,temperature],+,[heatex2_htot,temperature]),
    arc([heatex1_htin,temperature],+,[heatex1_htot,temperature]),
    arc([heatex1_htin,temperature],+,[heatex1_cdot,temperature]),
    arc([heatex1_htin,temperature],+,[heatex2_cdot,temperature]),
    arc([heatex2_cdin,temperature],+,[heatex1_cdot,temperature]),
    arc([heatex2_cdin,temperature],+,[heatex2_htot,temperature]),
    arc([heatex2_cdin,pressure],+,[heatex2_cdot,pressure]),
    arc([heatex1_htin,pressure],+,[reactor1_vapr,pressure]),
    arc([heatex2_cdot,pressure],+,[heatex2_cdin,pressure]),
    arc([heatex2_cdot,reverseFlow],++,[heatex2_cdin,reverseFlow]),
    arc([heatex2_cdot,noFlow],++,[heatex2_cdin,noFlow]),
    arc([heatex2_cdin,noFlow],++,[heatex2_cdot,noFlow]),
    arc([heatex1_htin,noFlow],- -,[reactor1_liqd,level]),
    arc([heatex2_cdin,noFlow],- -,[reactor1_liqd,temperature]),
    arc([heatex1_htin,noFlow],++,[heatex1_cdot,noFlow]),
    arc([heatex1_cdot,noFlow],++,[reactor1_liqd,level]),
    arc([heatex2_cdin,noFlow],- -,[heatex1_cdot,temperature]),
    arc([heatex2_cdin,concentration],+,[heatex2_cdot,concentration]),
    arc([heatex1_htin,concentration],+,[reactor1_liqd,concentration]),
    arc([heatex1_htin,concentration],+,[heatex1_cdot,concentration]),

    %faults
    arc([fault,'heatex2 holed heatexchanger'],+,[heatex2_cdin,flow]),
    arc([fault,'heatex2 holed heatexchanger'],-,[heatex2_cdot,flow]),
    arc([fault,'heatex1 holed heatexchanger'],-,[heatex1_cdot,flow]),
    arc([fault,'heatex1 partly blocked (heating side)'],-,[heatex1_cdot,flow]),
    arc([fault,'heatex1 fouling'],+,[heatex1_cdot,temperature]),
    arc([fault,'heatex1 fouling'],-,[reactor1_liqd,temperature]),
    arc([fault,'heatex1 blockage (heating side)'],+,[heatex1_cdot,noFlow]),
    arc([fault,'heatex1 blockage (heated side)'],+,[heatex1_cdot,noFlow]),
    arc([fault,'heatex1 blockage (heated side)'],+,[heatex1_htin,noFlow]),
    arc([fault,'heatex1 blockage (heating side)'],+,[heatex1_cdin,pressure]),
    arc([fault,'heatex1 blockage (heating side)'],-,[heatex1_cdot,pressure]),
    arc([fault,'heatex1 blockage (heated side)'],+,[heatex1_htin,pressure]),
```

```
arc([fault,'heatex1 blockage (heated side)'],-,[reactor1_vapr,pressure]),
arc([fault,'heatex1 blockage (heating side)'],-,[reactor1_liqd,temperature]),
arc([fault,'heatex2 partly blocked (heating side)'],-,[heatex2_cdot,flow]),
arc([fault,'heatex2 fouling'],+,[heatex2_cdot,temperature]),
arc([fault,'heatex2 fouling'],-,[heatex1_cdot,temperature]),
arc([fault,'heatex2 fouling'],-,[reactor1_liqd,temperature]),
arc([fault,'heatex2 blockage (heating side)'],+,[heatex2_cdot,noFlow]),
arc([fault,'heatex2 blockage (heated side)'],+,[heatex1_cdot,noFlow]),
arc([fault,'heatex2 blockage (heated side)'],+,[heatex1_htin,noFlow]),
arc([fault,'heatex2 blockage (heating side)'],+,[heatex2_cdin,pressure]),
arc([fault,'heatex2 blockage (heating side)'],-,[heatex2_cdot,pressure]),
arc([fault,'heatex2 blockage (heated side)'],+,[heatex2_htin,pressure]),
arc([fault,'heatex2 blockage (heated side)'],+,[heatex1_htin,pressure]),
arc([fault,'heatex2 blockage (heated side)'],-,[reactor1_vapr,pressure]),
arc([fault,'heatex2 blockage (heating side)'],-,[heatex1_cdot,temperature]),
arc([fault,'heatex2 blockage (heating side)'],-,[reactor1_liqd,temperature]),
arc([fault,'heatex2 blockage (heated side)'],+,[heatex2_cdot,temperature]),
arc([fault,'reactor1 leak to environment'],-,[reactor1_liqd,level]),
arc([fault,'reactor1 external fire'],+,[reactor1_liqd,temperature]),
arc([fault,'reactor1 external fire'],+,[heatex1_cdot,temperature]),
arc([fault,'reactor1 cold weather'],-,[reactor1_liqd,temperature]),
arc([fault,'reactor1 cold weather'],-,[heatex1_cdot,temperature]),
arc([fault,'reactor1 inlet1 siphon breaker blockage'],+,[heatex1_htin,reverseFlow]),
arc([fault,'reactor1 leak to environment'],-,[heatex1_htin,pressure]),
arc([fault,['reactor1 leak into vacuum system', vacuum]],+,[heatex1_htin,pressure]),
arc([fault,'reactor1 outlet1 partly blocked'],-,[heatex1_cdot,flow]),
arc([fault,'reactor1 outlet1 completely blocked'],+,[heatex1_cdot,noFlow]),
arc([fault,'reactor1 outlet1 completely blocked'],+,[heatex1_htin,noFlow]),
arc([fault,'valve16 partly blocked'],-,[heatex1_cdot,flow]),
arc([fault,'valve16 partly blocked'],+,[reactor1_liqd,level]),
arc([fault,'valve16 completely blocked'],+,[reactor1_liqd,level]),
arc([fault,'reactor1 outlet1 partly blocked'],+,[reactor1_liqd,level]),
arc([fault,'reactor1 outlet1 completely blocked'],+,[reactor1_liqd,level]),
arc([fault,'valve16 completely blocked'],+,[heatex1_cdot,noFlow]),
arc([fault,'valve16 completely blocked'],+,[heatex1_htin,noFlow]),
arc([fault,['valve16 blocked by frozen fluid', freezing]],+,[heatex1_cdot,noFlow]),
arc([fault,['valve16 blocked by frozen fluid', freezing]],+,[heatex1_htin,noFlow]),
arc([fault,'valve16 closed in error'],+,[heatex1_cdot,noFlow]),
arc([fault,'valve16 closed in error'],+,[heatex1_htin,noFlow]),
arc([fault,['valve16 blocked by frozen fluid', freezing]],+,[reactor1_liqd,level]),
arc([fault,['valve16 leak into vacuum system', vacuum]],+,[reactor1_liqd,level]),
arc([fault,'valve16 leak to environment'],-,[heatex1_cdot,flow]),
arc([fault,'valve16 leak to environment'],-,[reactor1_liqd,level]),
arc([fault,'reactor1 leak to environment'],+,[heatex1_htin,flow]),
arc([fault,'reactor1 leak to environment'],-,[heatex1_cdot,flow]),
arc([fault,'heatex2 partly blocked (heated side)'],-,[heatex1_htin,flow]),
arc([fault,'heatex2 partly blocked (heated side)'],-,[reactor1_liqd,level]),
arc([fault,'heatex1 partly blocked (heated side)'],-,[reactor1_liqd,level]),
arc([fault,'heatex1 blockage (heating side)'],+,[reactor1_liqd,level]),
arc([fault,'heatex1 blockage (heated side)'],-,[reactor1_liqd,level]),
arc([fault,'heatex1 partly blocked (heated side)'],-,[heatex1_htin,flow]),
arc([fault,'valve16 closed in error'],+,[reactor1_liqd,level]),
arc([fault,'valve16 thermal expansion of contents'],+,[heatex1_cdot,pressure]),
arc([fault,'valve16 thermal expansion of contents'],+,[heatex1_cdin,pressure]),
arc([fault,'heatex2 holed heatexchanger'],-,[heatex2_cdot,temperature]),

%consequences resulting from faults
arc([fault,'heatex2 holed heatexchanger'],+,[consequence,'contamination of product']),
arc([fault,'heatex1 holed heatexchanger'],+,[consequence,'contamination of product']),
arc([fault,'reactor1 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'reactor1 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
```

```
arc([fault,'reactor1 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'reactor1 external fire'],+,[consequence,'structural weakening']),
arc([fault,['reactor1 leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'reactor1 external fire'],+,[consequence,'possible rupture']),
arc([fault,'valve16 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'valve16 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'valve16 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'valve16 thermal expansion of contents'],+,[consequence,'flange leak']),

%consequences resulting from deviations
arc([deviation,[moreTemperature,heatex1_htot]],+,[consequence,'boiling in heatexchanger']),
arc([deviation,[lessTemperature,heatex1_cdot]],+,[consequence,'freezing in heatexchanger']),
arc([deviation,[moreTemperature,heatex1_htot]],+,[consequence,'polymerisation']),
arc([deviation,[morePressure,heatex1_cdin]],+,[consequence,'possible rupture']),
arc([deviation,[morePressure,heatex1_htin]],+,[consequence,'possible rupture']),
arc([deviation,[moreTemperature,heatex1_htot]],+,[consequence,'mechanical failure due to
    overtemperature']),
arc([deviation,[moreTemperature,heatex2_htot]],+,[consequence,'boiling in heatexchanger']),
arc([deviation,[lessTemperature,heatex2_cdot]],+,[consequence,'freezing in heatexchanger']),
arc([deviation,[moreTemperature,heatex2_htot]],+,[consequence,'polymerisation']),
arc([deviation,[morePressure,heatex2_cdin]],+,[consequence,'possible rupture']),
arc([deviation,[morePressure,heatex2_htin]],+,[consequence,'possible rupture']),
arc([deviation,[moreTemperature,heatex2_htot]],+,[consequence,'mechanical failure due to
    overtemperature']),
arc([deviation,[moreLevel,reactor1_liqd]],+,[consequence,'overfilling']),
arc([deviation,[moreConcentration,reactor1_liqd]],+,[consequence,'rubber lining corrosion in hot
    caustic']),
arc([deviation,[moreConcentration,reactor1_liqd]],+,[consequence,'emulsification of contents ']),
arc([deviation,[moreTemperature,reactor1_liqd]],+,[consequence,'increased evaporation']),
arc([deviation,[moreTemperature,reactor1_liqd]],+,[consequence,'flammable or toxic vapour release']),
arc([deviation,[moreTemperature,reactor1_liqd]],+,[consequence,'crystallisation']),
    arc([deviation,[moreTemperature,reactor1_liqd]],+,[consequence,'viscosity increase']),
    arc([deviation,[lessTemperature,reactor1_liqd]],+,[consequence,'viscosity decrease']),
    arc([deviation,[lessLevel,reactor1_liqd]],+,[consequence,'vessel emptying']),
    arc([deviation,[morePressure,reactor1_vapr]],+,[consequence,'possible rupture']),
    arc([deviation,[moreFlow,reactor1_in1]],+,[consequence,'incomplete separation of water'])
    ]
   ]
).
/*-------------------------------------------------*/
/*
The module is composed of the following units:
divider2 out1 connected to valve11 in,
divider2 out2 connected to valve13 in,
valve11 out connected to pumpj2a in,
pumpj2a out connected to valve12 in,
valve12 out connected to header2 in1,
valve13 out connected to pumpj2b in,
pumpj2b out connected to valve14 in,
valve14 out connected to header2 in2.
Conditions of module are:
    divider2_purpose is tobypass,
    valve11_aperture is open,
    valve12_aperture is open,
    valve13_aperture is closed,
    pumpj2b_status is spare,
    valve14_aperture is closed,
    header2_purpose is frombypass
and this completes the module.

/*
```

```
/*-----------------------------------------------*/

frame(module_2 isa module,
  [  inports info  [ divider2_in ],
     outports info  [ header2_out ],
     unitports info  [ pumpj2a_in, pumpj2a_out, pumpj2b_in, pumpj2b_out ],
     divider2_purpose is toBypass,
     valve11_aperture is unknown,
     pumpj2a_status is unknown,
     valve12_aperture is unknown,
     valve13_aperture is unknown,
     pumpj2b_status is unknown,
     valve14_aperture is unknown,
     header2_purpose is fromBypass,
     propLinks info [

     %propagation
     arc([divider2_in,noFlow],++,[header2_out,noFlow]),
     arc([header2_out,noFlow],++,[divider2_in,noFlow]),
     arc([divider2_in,noFlow],++,[pumpj2a_in,noFlow]),
     arc([divider2_in,temperature],+,[pumpj2a_out,temperature]),
     arc([divider2_in,temperature],+,[header2_out,temperature]),
     arc([divider2_in,pressure],+,[pumpj2a_out,pressure]),
     arc([divider2_in,pressure],+,[header2_out,pressure]),
     arc([header2_out,reverseFlow],++,[pumpj2a_out,reverseFlow]),
     arc([divider2_in,concentration],+,[header2_out,concentration]),

     %faults
     arc([fault,'divider2 leak to environment'],+,[divider2_in,flow]),
     arc([fault,'divider2 leak to environment'],-,[header2_out,flow]),
     arc([fault,['divider2 blocked by frozen fluid', freezing]],+,[divider2_in,noFlow]),
     arc([fault,['divider2 blocked by frozen fluid', freezing]],+,[header2_out,noFlow]),
     arc([fault,['divider2 blocked by frozen fluid', freezing]],+,[pumpj2a_in,noFlow]),
     arc([fault,'divider2 inlet completely blocked'],+,[divider2_in,noFlow]),
     arc([fault,'divider2 inlet completely blocked'],+,[header2_out,noFlow]),
     arc([fault,'divider2 inlet completely blocked'],+,[pumpj2a_in,noFlow]),
     arc([fault,'divider2 inlet partly blocked'],-,[divider2_in,flow]),
     arc([fault,'divider2 inlet partly blocked'],-,[header2_out,flow]),
     arc([fault,'pumpj2a partly blocked'],-,[divider2_in,flow]),
     arc([fault,'pumpj2a partly blocked'],-,[header2_out,flow]),
     arc([fault,'pumpj2a completely blocked'],+,[header2_out,noFlow]),
     arc([fault,'pumpj2a completely blocked'],+,[divider2_in,noFlow]),
     arc([fault,'pumpj2a leak to environment'],+,[divider2_in,flow]),
     arc([fault,'pumpj2a leak to environment'],-,[header2_out,flow]),
     arc([fault,'pumpj2a air lock'],-,[divider2_in,flow]),
     arc([fault,'pumpj2a air lock'],-,[header2_out,flow]),
     arc([fault,'pumpj2a air lock'],-,[header2_out,pressure]),
     arc([fault,'pumpj2a air lock'],+,[divider2_in,pressure]),
     arc([fault,'pumpj2a power supply fails'],+,[header2_out,noFlow]),
     arc([fault,'pumpj2a power supply fails'],+,[divider2_in,noFlow]),
     arc([fault,'pumpj2a power supply fails'],-,[header2_out,pressure]),
     arc([fault,'pumpj2a power supply fails'],+,[divider2_in,pressure]),
     arc([fault,'pumpj2a impellor failure'],-,[divider2_in,flow]),
     arc([fault,'pumpj2a impellor failure'],-,[header2_out,flow]),
     arc([fault,'pumpj2a impellor failure'],-,[header2_out,pressure]),
     arc([fault,'pumpj2a impellor failure'],+,[divider2_in,pressure]),
     arc([fault,['pumpj2a leak into vacuum system', vacuum]],+,[divider2_in,pressure]),
     arc([fault,'pumpj2a pump fails'],+,[divider2_in,reverseFlow]),
     arc([fault,'pumpj2a pump fails'],+,[header2_out,reverseFlow]),
     arc([fault,'header2 outlet partly blocked'],-,[header2_out,flow]),
     arc([fault,'header2 outlet partly blocked'],-,[divider2_in,flow]),
```

arc([fault,'header2 outlet completely blocked'],+,[header2_out,noFlow]),
arc([fault,'header2 outlet completely blocked'],+,[divider2_in,noFlow]),
arc([fault,'header2 outlet completely blocked'],+,[pumpj2a_in,noFlow]),
arc([fault,['header2 blocked by frozen fluid', freezing]],+,[header2_out,noFlow]),
arc([fault,['header2 blocked by frozen fluid', freezing]],+,[divider2_in,noFlow]),
arc([fault,['header2 blocked by frozen fluid', freezing]],+,[pumpj2a_in,noFlow]),
arc([fault,'divider2 branch 1 completely blocked'],+,[divider2_in,noFlow]),
arc([fault,'divider2 branch 1 completely blocked'],+,[header2_out,noFlow]),
arc([fault,'divider2 branch 1 completely blocked'],+,[pumpj2a_in,noFlow]),
arc([fault,'divider2 branch 1 partly blocked'],-,[divider2_in,flow]),
arc([fault,'divider2 branch 1 partly blocked'],-,[header2_out,flow]),
arc([fault,'valve11 partly blocked'],-,[divider2_in,flow]),
arc([fault,'valve11 partly blocked'],-,[header2_out,flow]),
arc([fault,'valve12 partly blocked'],-,[header2_out,flow]),
arc([fault,'valve12 partly blocked'],-,[divider2_in,flow]),
arc([fault,'valve11 completely blocked'],+,[divider2_in,noFlow]),
arc([fault,'valve11 completely blocked'],+,[header2_out,noFlow]),
arc([fault,'valve11 completely blocked'],+,[pumpj2a_in,noFlow]),
arc([fault,'valve12 completely blocked'],+,[divider2_in,noFlow]),
arc([fault,'valve12 completely blocked'],+,[header2_out,noFlow]),
arc([fault,'valve12 completely blocked'],+,[pumpj2a_in,noFlow]),
arc([fault,['valve11 blocked by frozen fluid', freezing]],+,[divider2_in,noFlow]),
arc([fault,['valve11 blocked by frozen fluid', freezing]],+,[header2_out,noFlow]),
arc([fault,['valve11 blocked by frozen fluid', freezing]],+,[pumpj2a_in,noFlow]),
arc([fault,['valve12 blocked by frozen fluid', freezing]],+,[divider2_in,noFlow]),
arc([fault,['valve12 blocked by frozen fluid', freezing]],+,[header2_out,noFlow]),
arc([fault,['valve12 blocked by frozen fluid', freezing]],+,[pumpj2a_in,noFlow]),
arc([fault,['valve11 leak into vacuum system', vacuum]],+,[divider2_in,pressure]),
arc([fault,['valve12 leak into vacuum system', vacuum]],+,[divider2_in,pressure]),
arc([fault,'valve11 closed in error'],+,[divider2_in,pressure]),
arc([fault,'valve11 closed in error'],-,[pumpj2a_out,pressure]),
arc([fault,'valve11 closed in error'],-,[header2_out,pressure]),
arc([fault,'valve11 closed in error'],+,[pumpj2a_in,noFlow]),
arc([fault,'valve11 closed in error'],+,[header2_out,noFlow]),
arc([fault,'valve11 closed in error'],+,[divider2_in,noFlow]),
arc([fault,'valve12 closed in error'],+,[pumpj2a_out,pressure]),
arc([fault,'valve12 closed in error'],+,[divider2_in,pressure]),
arc([fault,'valve12 closed in error'],-,[header2_out,pressure]),
arc([fault,'valve12 closed in error'],+,[header2_out,noFlow]),
arc([fault,'valve12 closed in error'],+,[divider2_in,noFlow]),
arc([fault,'valve12 closed in error'],+,[pumpj2a_in,noFlow]),
arc([fault,['valve13 leak into vacuum system', vacuum]],+,[divider2_in,pressure]),
arc([fault,'header2 branch 1 partly blocked'],-,[header2_out,flow]),
arc([fault,'header2 branch 1 partly blocked'],-,[divider2_in,flow]),
arc([fault,'header2 branch 1 completely blocked'],+,[header2_out,noFlow]),
arc([fault,'header2 branch 1 completely blocked'],+,[divider2_in,noFlow]),
arc([fault,'header2 branch 1 completely blocked'],+,[pumpj2a_in,noFlow]),
arc([fault,'header2 outlet completely blocked'],+,[pumpj2a_out,pressure]),
arc([fault,['header2 blocked by frozen fluid', freezing]],+,[pumpj2a_out,pressure]),
arc([fault,'header2 branch 1 completely blocked'],+,[pumpj2a_out,pressure]),
arc([fault,'valve12 completely blocked'],+,[pumpj2a_out,pressure]),
arc([fault,['valve12 blocked by frozen fluid', freezing]],+,[pumpj2a_out,pressure]),
arc([fault,'valve13 open or passing'],+,[divider2_in,flow]),
arc([fault,'valve13 leak to environment'],+,[divider2_in,flow]),
arc([fault,'valve13 leak to environment'],-,[header2_out,flow]),
arc([fault,'valve11 leak to environment'],+,[divider2_in,flow]),
arc([fault,'valve11 leak to environment'],-,[header2_out,flow]),
arc([fault,'valve12 leak to environment'],+,[divider2_in,flow]),
arc([fault,'valve12 leak to environment'],-,[header2_out,flow]),
arc([fault,'valve11 thermal expansion'],+,[divider2_in,pressure]),
arc([fault,'valve12 thermal expansion'],+,[divider2_in,pressure]),

```
arc([fault,'header2 leak to environment'],-,[header2_out,flow]),
arc([fault,'header2 leak to environment'],+,[divider2_in,flow]),
arc([fault,['divider2 leak into vacuum system', vacuum]],+,[divider2_in,pressure]),
arc([fault,['header2 leak into vacuum system', vacuum]],+,[divider2_in,pressure]),
arc([fault,'valve13 thermal expansion'],+,[divider2_in,pressure]),

%consequences resulting from faults
arc([fault,'divider2 leak to environment'],+,[consequence,'contaminate environment']),
arc([fault,'divider2 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'divider2 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,['divider2 leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'valve11 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'valve11 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'valve11 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'valve11 thermal expansion'],+,[consequence,'flange leak']),
arc([fault,'pumpj2a leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'pumpj2a leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'pumpj2a leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,['pumpj2a leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'valve12 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'valve12 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'valve12 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'valve12 thermal expansion'],+,[consequence,'flange leak']),
arc([fault,'valve13 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'valve13 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'valve13 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'valve13 thermal expansion'],+,[consequence,'flange leak']),
arc([fault,'header2 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'header2 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'header2 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,['header2 leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'valve13 open or passing'],+,[consequence,'spare pump possible rupture']),

%consequences resulting from deviations
arc([deviation,[morePressure,divider2_in]],+,[consequence,'possible rupture']),
arc([deviation,[morePressure,pumpj2a_out]],+,[consequence,'possible rupture']),
arc([deviation,[moreTemperature,pumpj2a_out]],+,[consequence,'overheating']),
arc([deviation,[moreTemperature,pumpj2a_out]],+,[consequence,'gland failure']),
arc([deviation,[reverseFlow,pumpj2a_out]],+,[consequence,'pump driven backwards']),
arc([deviation,[noFlow,pumpj2a_in]],+,[consequence,'overheating']),
arc([deviation,[noFlow,pumpj2a_in]],+,[consequence,'gland failure'])
]
]
).
/*-----------------------------------------------*/
/*
The module is composed of the following units:
divider1 out1 connected to valve3 in,
divider1 out2 connected to valve5 in,
valve3 out connected to pumpj1a in,
pumpj1a out connected to valve4 in,
valve4 out connected to header1 in1,
valve5 out connected to pumpj1b in,
pumpj1b out connected to valve6 in,
valve6 out connected to header1 in2.
Conditions of module are:
    divider1_purpose is tobypass,
    valve3_aperture is open,
    valve4_aperture is open,
    valve5_aperture is closed,
    pumpj1b_status is spare,
```

valve6_aperture is closed,
header1_purpose is frombypass
and this completes the module.

```
/*
/*-------------------------------------------------*/

frame(module_1 isa module,
  [ inports info  [ divider1_in ],
    outports info  [ header1_out ],
    unitports info  [ pumpj1a_in, pumpj1a_out, pumpj1b_in, pumpj1b_out ],
    divider1_purpose is toBypass,
    valve3_aperture is unknown,
    pumpj1a_status is unknown,
    valve4_aperture is unknown,
    valve5_aperture is unknown,
    pumpj1b_status is unknown,
    valve6_aperture is unknown,
    header1_purpose is fromBypass,
    propLinks info [

    %propagation
    arc([divider1_in,noFlow],++,[header1_out,noFlow]),
    arc([header1_out,noFlow],++,[divider1_in,noFlow]),
    arc([divider1_in,noFlow],++,[pumpj1a_in,noFlow]),
    arc([divider1_in,temperature],+,[pumpj1a_out,temperature]),
    arc([divider1_in,temperature],+,[header1_out,temperature]),
    arc([divider1_in,pressure],+,[pumpj1a_out,pressure]),
    arc([divider1_in,pressure],+,[header1_out,pressure]),
    arc([header1_out,reverseFlow],++,[pumpj1a_out,reverseFlow]),
    arc([divider1_in,concentration],+,[header1_out,concentration]),

    %faults
    arc([fault,'divider1 leak to environment'],+,[divider1_in,flow]),
    arc([fault,'divider1 leak to environment'],-,[header1_out,flow]),
    arc([fault,['divider1 blocked by frozen fluid', freezing]],+,[divider1_in,noFlow]),
    arc([fault,['divider1 blocked by frozen fluid', freezing]],+,[header1_out,noFlow]),
    arc([fault,['divider1 blocked by frozen fluid', freezing]],+,[pumpj1a_in,noFlow]),
    arc([fault,'divider1 inlet completely blocked'],+,[divider1_in,noFlow]),
    arc([fault,'divider1 inlet completely blocked'],+,[header1_out,noFlow]),
    arc([fault,'divider1 inlet completely blocked'],+,[pumpj1a_in,noFlow]),
    arc([fault,'divider1 inlet partly blocked'],-,[divider1_in,flow]),
    arc([fault,'divider1 inlet partly blocked'],-,[header1_out,flow]),
    arc([fault,'pumpj1a partly blocked'],-,[divider1_in,flow]),
    arc([fault,'pumpj1a partly blocked'],-,[header1_out,flow]),
    arc([fault,'pumpj1a completely blocked'],+,[header1_out,noFlow]),
    arc([fault,'pumpj1a completely blocked'],+,[divider1_in,noFlow]),
    arc([fault,'pumpj1a leak to environment'],+,[divider1_in,flow]),
    arc([fault,'pumpj1a leak to environment'],-,[header1_out,flow]),
    arc([fault,'pumpj1a air lock'],-,[divider1_in,flow]),
    arc([fault,'pumpj1a air lock'],-,[header1_out,flow]),
    arc([fault,'pumpj1a air lock'],-,[header1_out,pressure]),
    arc([fault,'pumpj1a air lock'],+,[divider1_in,pressure]),
    arc([fault,'pumpj1a power supply fails'],+,[header1_out,noFlow]),
    arc([fault,'pumpj1a power supply fails'],+,[divider1_in,noFlow]),
    arc([fault,'pumpj1a power supply fails'],-,[header1_out,pressure]),
    arc([fault,'pumpj1a power supply fails'],+,[divider1_in,pressure]),
    arc([fault,'pumpj1a impellor failure'],-,[divider1_in,flow]),
    arc([fault,'pumpj1a impellor failure'],-,[header1_out,flow]),
    arc([fault,'pumpj1a impellor failure'],-,[header1_out,pressure]),
    arc([fault,'pumpj1a impellor failure'],+,[divider1_in,pressure]),
```

arc([fault,['pumpj1a leak into vacuum system', vacuum]],+,[divider1_in,pressure]),
arc([fault,'pumpj1a pump fails'],+,[divider1_in,reverseFlow]),
arc([fault,'pumpj1a pump fails'],+,[header1_out,reverseFlow]),
arc([fault,'header1 outlet partly blocked'],-,[header1_out,flow]),
arc([fault,'header1 outlet partly blocked'],-,[divider1_in,flow]),
arc([fault,'header1 outlet completely blocked'],+,[header1_out,noFlow]),
arc([fault,'header1 outlet completely blocked'],+,[divider1_in,noFlow]),
arc([fault,'header1 outlet completely blocked'],+,[pumpj1a_in,noFlow]),
arc([fault,['header1 blocked by frozen fluid', freezing]],+,[header1_out,noFlow]),
arc([fault,['header1 blocked by frozen fluid', freezing]],+,[divider1_in,noFlow]),
arc([fault,['header1 blocked by frozen fluid', freezing]],+,[pumpj1a_in,noFlow]),
arc([fault,'divider1 branch 1 completely blocked'],+,[divider1_in,noFlow]),
arc([fault,'divider1 branch 1 completely blocked'],+,[header1_out,noFlow]),
arc([fault,'divider1 branch 1 completely blocked'],+,[pumpj1a_in,noFlow]),
arc([fault,'divider1 branch 1 partly blocked'],-,[divider1_in,flow]),
arc([fault,'divider1 branch 1 partly blocked'],-,[header1_out,flow]),
arc([fault,'valve3 partly blocked'],-,[divider1_in,flow]),
arc([fault,'valve3 partly blocked'],-,[header1_out,flow]),
arc([fault,'valve4 partly blocked'],-,[header1_out,flow]),
arc([fault,'valve4 partly blocked'],-,[divider1_in,flow]),
arc([fault,'valve3 completely blocked'],+,[divider1_in,noFlow]),
arc([fault,'valve3 completely blocked'],+,[header1_out,noFlow]),
arc([fault,'valve3 completely blocked'],+,[pumpj1a_in,noFlow]),
arc([fault,'valve4 completely blocked'],+,[divider1_in,noFlow]),
arc([fault,'valve4 completely blocked'],+,[header1_out,noFlow]),
arc([fault,'valve4 completely blocked'],+,[pumpj1a_in,noFlow]),
arc([fault,['valve3 blocked by frozen fluid', freezing]],+,[divider1_in,noFlow]),
arc([fault,['valve3 blocked by frozen fluid', freezing]],+,[header1_out,noFlow]),
arc([fault,['valve3 blocked by frozen fluid', freezing]],+,[pumpj1a_in,noFlow]),
arc([fault,['valve4 blocked by frozen fluid', freezing]],+,[divider1_in,noFlow]),
arc([fault,['valve4 blocked by frozen fluid', freezing]],+,[header1_out,noFlow]),
arc([fault,['valve4 blocked by frozen fluid', freezing]],+,[pumpj1a_in,noFlow]),
arc([fault,['valve3 leak into vacuum system', vacuum]],+,[divider1_in,pressure]),
arc([fault,['valve4 leak into vacuum system', vacuum]],+,[divider1_in,pressure]),
arc([fault,'valve3 closed in error'],+,[divider1_in,pressure]),
arc([fault,'valve3 closed in error'],-,[pumpj1a_out,pressure]),
arc([fault,'valve3 closed in error'],-,[header1_out,pressure]),
arc([fault,'valve3 closed in error'],+,[pumpj1a_in,noFlow]),
arc([fault,'valve3 closed in error'],+,[header1_out,noFlow]),
arc([fault,'valve3 closed in error'],+,[divider1_in,noFlow]),
arc([fault,'valve4 closed in error'],+,[pumpj1a_out,pressure]),
arc([fault,'valve4 closed in error'],+,[divider1_in,pressure]),
arc([fault,'valve4 closed in error'],-,[header1_out,pressure]),
arc([fault,'valve4 closed in error'],+,[header1_out,noFlow]),
arc([fault,'valve4 closed in error'],+,[divider1_in,noFlow]),
arc([fault,'valve4 closed in error'],+,[pumpj1a_in,noFlow]),
arc([fault,['valve5 leak into vacuum system', vacuum]],+,[divider1_in,pressure]),
arc([fault,'header1 branch 1 partly blocked'],-,[header1_out,flow]),
arc([fault,'header1 branch 1 partly blocked'],-,[divider1_in,flow]),
arc([fault,'header1 branch 1 completely blocked'],+,[header1_out,noFlow]),
arc([fault,'header1 branch 1 completely blocked'],+,[divider1_in,noFlow]),
arc([fault,'header1 branch 1 completely blocked'],+,[pumpj1a_in,noFlow]),
arc([fault,'header1 outlet completely blocked'],+,[pumpj1a_out,pressure]),
arc([fault,['header1 blocked by frozen fluid', freezing]],+,[pumpj1a_out,pressure]),
arc([fault,'header1 branch 1 completely blocked'],+,[pumpj1a_out,pressure]),
arc([fault,'valve4 completely blocked'],+,[pumpj1a_out,pressure]),
arc([fault,['valve4 blocked by frozen fluid', freezing]],+,[pumpj1a_out,pressure]),
arc([fault,'valve5 open or passing'],+,[divider1_in,flow]),
arc([fault,'valve5 leak to environment'],+,[divider1_in,flow]),
arc([fault,'valve5 leak to environment'],-,[header1_out,flow]),
arc([fault,'valve3 leak to environment'],+,[divider1_in,flow]),

```
arc([fault,'valve3 leak to environment'],-,[header1_out,flow]),
arc([fault,'valve4 leak to environment'],+,[divider1_in,flow]),
arc([fault,'valve4 leak to environment'],-,[header1_out,flow]),
arc([fault,'valve3 thermal expansion'],+,[divider1_in,pressure]),
arc([fault,'valve4 thermal expansion'],+,[divider1_in,pressure]),
arc([fault,'header1 leak to environment'],-,[header1_out,flow]),
arc([fault,'header1 leak to environment'],+,[divider1_in,flow]),
arc([fault,['divider1 leak into vacuum system', vacuum]],+,[divider1_in,pressure]),
arc([fault,['header1 leak into vacuum system', vacuum]],+,[divider1_in,pressure]),
arc([fault,'valve5 thermal expansion'],+,[divider1_in,pressure]),

%consequences resulting from faults
arc([fault,'divider1 leak to environment'],+,[consequence,'contaminate environment']),
arc([fault,'divider1 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'divider1 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,['divider1 leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'valve3 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'valve3 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'valve3 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'valve3 thermal expansion'],+,[consequence,'flange leak']),
arc([fault,'pumpj1a leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'pumpj1a leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'pumpj1a leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,['pumpj1a leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'valve4 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'valve4 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'valve4 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'valve4 thermal expansion'],+,[consequence,'flange leak']),
arc([fault,'valve5 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'valve5 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'valve5 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,'valve5 thermal expansion'],+,[consequence,'flange leak']),
arc([fault,'header1 leak to environment'],+,[consequence,['contaminate environment', toxic]]),
arc([fault,'header1 leak to environment'],+,[consequence,['loss of material', expensive]]),
arc([fault,'header1 leak to environment'],+,[consequence,['fire/explosion risk', flammable]]),
arc([fault,['header1 leak into vacuum system', vacuum]],+,[consequence,'Possible explosive mixture']),
arc([fault,'valve5 open or passing'],+,[consequence,'spare pump possible rupture']),

%consequences resulting from deviations
arc([deviation,[morePressure,divider1_in]],+,[consequence,'possible rupture']),
arc([deviation,[morePressure,pumpj1a_out]],+,[consequence,'possible rupture']),
arc([deviation,[moreTemperature,pumpj1a_out]],+,[consequence,'overheating']),
arc([deviation,[moreTemperature,pumpj1a_out]],+,[consequence,'gland failure']),
arc([deviation,[reverseFlow,pumpj1a_out]],+,[consequence,'pump driven backwards']),
arc([deviation,[noFlow,pumpj1a_in]],+,[consequence,'overheating']),
arc([deviation,[noFlow,pumpj1a_in]],+,[consequence,'gland failure'])
]
]
).
```

D.6. HAZOP Results (Modular Approach)

Examples from the HAZOP emulation results created by QUEEN are shown. For brevity the full set of results is not given.

feedinlet no flow out	valve1: completely blocked, feedpipe: completely blocked, feedinlet: inlet		

	completely blocked		
feedinlet less flow out	valve18: partly blocked, completely blocked, flaresystem1: partly blocked, blocked, feedpipe: partly blocked, feedinlet: inlet partly blocked		
	feedinlet: leak to environment	contaminate environment, fire/explosion risk, loss of material	

- etc.-

storagetank less flow out1	valve2: partly blocked, storagetank: outlet1 partly blocked, byps1_: valve4 partly blocked, valve3 partly blocked, pumpj1a partly blocked, pumpj1a impellor failure, pumpj1a air lock, header1 outlet partly blocked, header1 branch 1 partly blocked, divider1 inlet partly blocked, divider1 branch 1 partly blocked		

- etc.-

storagetank more level liqd	valve8: completely blocked, valve2: partly blocked, closed in error, storagetank: outlet1 partly blocked, outlet1 completely blocked, nitogeninlet1: inlet partly blocked, inlet completely blocked, halfmileline: completely blocked, byps1_: valve4 partly blocked, valve4 completely blocked, valve4 closed in error, valve3 partly blocked, valve3 completely blocked, valve3 closed in	overfilling	

		error, pumpj1a pump fails, pumpj1a power supply fails, pumpj1a partly blocked, pumpj1a impellor failure, pumpj1a completely blocked, pumpj1a air lock, header1 outlet partly blocked, header1 outlet completely blocked, header1 branch 1 partly blocked, header1 branch 1 completely blocked, divider1 inlet partly blocked, divider1 inlet completely blocked, divider1 branch 1 partly blocked, divider1 branch 1 completely blocked	
		valve2: thermal expansion of contents, byps1_: valve5 thermal expansion, valve4 thermal expansion, valve3 thermal expansion	overfilling, flange leak
		valve18: leak to environment	overfilling, contaminate environment, loss of material, fire/explosion risk
		nitogeninlet1: leak to environment	overfilling, loss of material
		flaresystem1: leak to environment	overfilling, contaminate environment, fire/explosion risk, loss of material
	- etc. -		
buffertank less flow out1		valve10: partly blocked, byps2_: valve12 partly blocked, valve11 partly blocked, pumpj2a partly blocked, pumpj2a impellor failure, pumpj2a air lock, header2 outlet partly	

	blocked, header2 branch 1 partly blocked, divider2 inlet partly blocked, divider2 branch 1 partly blocked, buffertank: outlet1 partly blocked		

- etc.-

buffertank more level liqd	valve9: thermal expansion of contents, byps2_: valve13 thermal expansion, valve12 thermal expansion, valve11 thermal expansion	overfilling, flange leak	
	valve20: leak to environment	overfilling, contaminate environment, loss of material, fire/explosion risk	
	valve19: partly blocked, completely blocked, valve10: closed in error, nitrogeninlet2: inlet partly blocked, inlet completely blocked, heatingloop3_: valve16 completely blocked, valve16 closed in error, reactor1 outlet1 completely blocked, heatex2 blockage (heated side), heatex1 blockage (heated side), byps2_: valve12 partly blocked, valve12 completely blocked, valve12 closed in error, valve11 partly blocked, valve11 completely blocked, valve11 closed in error, pumpj2a pump fails, pumpj2a power supply fails, pumpj2a partly blocked, pumpj2a impellor failure, pumpj2a completely blocked, pumpj2a air lock, header2 outlet partly	overfilling	

	blocked, header2 outlet completely blocked, header2 branch 1 partly blocked, header2 branch 1 completely blocked, divider2 inlet partly blocked, divider2 inlet completely blocked, divider2 branch 1 partly blocked, divider2 branch 1 completely blocked, buffertank: outlet2 partly blocked, outlet2 completely blocked, outlet1 partly blocked, outlet1 completely blocked		
	nitrogeninlet2: leak to environment	overfilling, loss of material	
	halfmileline: thermal expansion	overfilling, possible rupture	
	halfmileline: liquid hammer	overfilling, pipeline supports broken	
	flaresystem2: leak to environment	overfilling, contaminate environment, fire/explosion risk, loss of material	
	- etc.-		
byps1_ no flow pumpj1a_in	valve8: completely blocked, storagetank: outlet1 completely blocked, halfmileline: completely blocked, feedpipe: completely blocked, feedinlet: inlet completely blocked, byps1_: valve4 completely blocked, valve4 closed in error, valve3 completely blocked, valve3 closed in error, pumpj1a power supply fails, pumpj1a completely blocked, header1 outlet	overheating, gland failure	

	completely blocked, header1 branch 1 completely blocked, divider1 inlet completely blocked, divider1 branch 1 completely blocked		
byps1_ more temperature pumpj1a_out	storagetank: external fire	overheating, gland failure, structural weakening	
byps1_ more pressure pumpj1a_out	byps1_: valve4 completely blocked, header1 outlet completely blocked, header1 branch 1 completely blocked	possible rupture	
byps2_ no flow pumpj2a_in	valve15: completely blocked, heatingloop3_: valve16 completely blocked, valve16 closed in error, reactor1 outlet1 completely blocked, heatex2 blockage (heated side), heatex1 blockage (heated side), byps2_: valve12 completely blocked, valve12 closed in error, valve11 completely blocked, valve11 closed in error, pumpj2a power supply fails, pumpj2a completely blocked, header2 outlet completely blocked, header2 branch 1 completely blocked, divider2 inlet completely blocked, divider2 branch 1 completely blocked, buffertank: outlet1 completely blocked	overheating, gland failure	
byps2_ more temperature pumpj2a_out	halfmileline: external heat	overheating, gland failure	
	buffertank: external fire	overheating, gland failure, structural weakening, possible rupture	
byps2_ more pressure	byps2_: valve12	possible rupture	

pumpj2a_out	completely blocked, header2 outlet completely blocked, header2 branch 1 completely blocked	

- etc.-

heatingloop3_ more level reactor1_liqd	tail2: partly blocked, blocked heatingloop3_: valve16 partly blocked, reactor1 outlet1 partly blocked, heatex1 partly blocked (heating side), heatex1 blockage (heating side)	overfilling
	heatingloop3_: heatex1 holed heatexchanger	overfilling, contamination of product
heatingloop3_ less level reactor1_liqd	valve15: partly blocked, completely blocked, heatingloop3_: heatex2 partly blocked (heated side), heatex2 blockage (heated side), heatex1 partly blocked (heated side), heatex1 blockage (heated side), byps2_: valve12 partly blocked, valve12 completely blocked, valve12 closed in error, valve11 partly blocked, valve11 completely blocked, valve11 closed in error, pumpj2a power supply fails, pumpj2a partly blocked, pumpj2a impellor failure, pumpj2a completely blocked, pumpj2a air lock, header2 outlet partly blocked, header2 outlet completely blocked, header2 branch 1 partly blocked, header2 branch 1 completely blocked, divider2 inlet partly blocked,	vessel emptying

	divider2 inlet completely blocked, divider2 branch 1 partly blocked, divider2 branch 1 completely blocked, buffertank: outlet1 completely blocked		
	valve15: leak to environment, heatingloop3_: valve16 leak to environment	vessel emptying, contaminate environment, loss of material, fire/explosion risk	
	tail2: leak to environment, heatingloop3_: reactor1 leak to environment	vessel emptying, contaminate environment, fire/explosion risk, loss of material	

Appendix E

Published Papers Resulting From Work Carried Out For This Thesis

Palmer, C. and Chung, P.W.H. (1997) Constructing Qualitative Models. In IChemE Jubilee Research Event, vol2, pp725-728, IChemE, Rugby, UK.

Palmer, C. and Chung, P.W.H. (1998) Eliminating Ambiguities in Qualitative Models. In Computers and Chemical Engineering, (Suppl) vol22, ppS843-S846.

Palmer, C. and Chung, P.W.H. (1999) Verifying Signed Directed Graph Models for Process Plants. To be published in Computers and Chemical Engineering, (Suppl).