Rikke: Users Manual

Olsen, Jens V.; Forskningscenter Risø, Roskilde; Forskningscenter Risø, Roskilde; Forskningscenter Risø, Roskilde; Forskningscenter Risø, Roskilde

Publication date:
1985

Document Version
Publisher's PDF, also known as Version of record

RISØ-M-2480

RIKKE
USERS MANUAL

P. Haastrup, J.V. Olsen, J.R. Taylor, Axel Damborg
and N.K. Vestergaard

Abstract. RIKKE is a computer program for reliability and safety analysis of process plants, electrical systems etc. The program is available in a PDP-11 and a VAX version. The manual gives a description of the use of the program as a tool in the hazard analysis of an actual process plant. Furthermore the manual gives a summary of the principles of building new components as parts of the existing libraries.

February 1985
Risø National Laboratory, DK 4000 Roskilde, Denmark.

ISBN 87-550-1079-2
ISSN 0418-6435

Risø Repro 1985

CONTENTS

Page

1. INTRODUCTION.

RIKKE is a program package intended for support for reliability and safety analysis of process plants, electrical systems, electronic, hydraulic systems etc. The theory underlying plant modelling and failure analysis used in the system is described in Automatic Fault Tree and Consequence Analysis (Taylor and Olsen, 1979).

The system is conceived as a set of small programs running on a small computer (original a PDP-11, but RIKKE is now available in a VAX version) under a command program and making use of a data base describing process plants, electrical circuits etc. The programs permits a relatively inexperienced user to generate fault trees for almost any technical system, provided the necessary component models are available. The command program accepts keyboard commands, and on the basis of these starts other programs. The command input takes the form of a "prompt-response" system. That is, the command program sends a message to the user indicating what command is required next, and the user can then reply. Generally, if in doubt, the user of the program can receive help by pressing the carriage return key on the keyboard. In this case the command program will provide a helping message, most often indicating which range of commands are possible. (See also chapter 6).

The individual programs running under the RIKKE program monitor has a prompt-response input form which is similar to that for the monitor, which means that to the user the system appears as one large interactive program package.

The individual failure analysis programs perform steps such as accepting and storing plant flow sheet, building up a plant function and failure model, generating a fault tree, or printing a fault tree. The programs work by taking some input, in the form af files stored in a disc storage and as commands from the keyboard, and produce outputs in the form of files on disc storage or on a typewriter, line printer, graphic plotter or graphic display.

The programs make use of a data base which describes plant component types, plant flow sheets, plant operating procedure instructions etc. The data base is conceived quite generally, so that it can support a wide range of different plant model types (finite state, equation model, energy and mass flow models, etc.), far beyond the capability of the existing analysis programs. It is hoped that the RIKKE system will provide the basis for a continued development of plant safety and reliability analysis software.

The purpose of this manual is to describe the use of the RIKKE programs, and to describe that part of the structure and working of the programs that is necessary for understanding their use.

It is also the purpose of the manual to provide information about the libraries developed at RISO National Laboratory and the principles for executing models.

It has therefore been the intention to devide this manual into parts, with information on the lowest level given early and with background material in later chapters, in appendices or in references.

The manual has been written with the intention of fulfilling the information needs of the END USER, the PRODUCT TECHNICIAN and the DOMAIN EXPERT. These terms has been defined by Olsen (1984) and the definition can be seen in table 1.1.

Table 1.1 Levels of information.

End User - The Risk Analyst using RIKKE as a tool for his Hazard Analysis on a model of an actual process plant previously fed into the system by a product technician.

Product Technician - A physicist or Engineer with knowledge about the process plant (could be chemical or other type) which is be analyzed by the risk analyst. He uses RIKKE to perform the modelling of the actual plant based on engineering drawings and his personal knowledge together with a library of fault-models for the different types of components (pumps, pipes, valves, tanks etc.) from which the plant is built.

Domain Expert - is a physicist or engineer with deep knowledge about the individual components according, not only to their behaviour under normal conditions as well as failure modes, but also how they interact when interconnected in more complex structures.
He stores his knowledge in a generic component library from which the Product Technician builds the final model.

It is not the intention of this manual to give information on higher levels of detail than these three, though artificial intelligence experts and system programmers has of cause been involved in development of RIKKE.

Although RIKKE thus contains all elements of an Expert System, and carries out some important expert tasks - it can never replace the expert within its area. Instead it may be seen as an important aid for the Risk Analyst as it carries out some more trivial tasks.

RIKKE may be seen as an intelligent scratch pad.

For the END USER the important information about how to generate and cut a fault tree is found in sections 2.3 and 2.4, and the conversion of results to cutsets is found in section 3.1.

For the PRODUCT TECHNICIAN information about how to make a model of the technical system is found in the sections 2.1 and 2.2.

In practice these two roles are commonly intercorrelated.

For the DOMAIN EXPERT who makes and maintains the libraries, information about the tools provided in the system is found in chapter 4. Further information about the libraries delivered with the system both for DOMAIN EXPERTS and PRODUCT TECHNICIANS is found in chapter 7.

For the DOMAIN EXPERT a discussion of the filosophy of generic modelling and the necessary simplifications is found in chapter 8.

In chapter 5 the commands available is found and a similar list can be found in appendix E: RIKKE COMMANDS AT A GLANCE.

In chapter 6 general information about how to obtain HELP is given.

In the following a short describtion of the RIKKE commands and programs is given.

1.1 The RIKKE commands and programs.

The usual progression of a safety analysis with RIKKE is the following.

(1) A description of a process plant is input to the computer as a flow sheet, circuit drawing, block diagram etc.

(2) The information from the drawing is combined with component information drawn from a library of component models.

(3) Programs are run to carry out different kinds of safety analysis.

(4) Programs are run to simplify the results of the analysis, for example to prune fault trees, generate cutsets etc.

(5) The results are drawn graphically.

Each of these tasks is done with the help of different subprograms in the RIKKE system.

The structure of RIKKE is shown in figure 1.1.

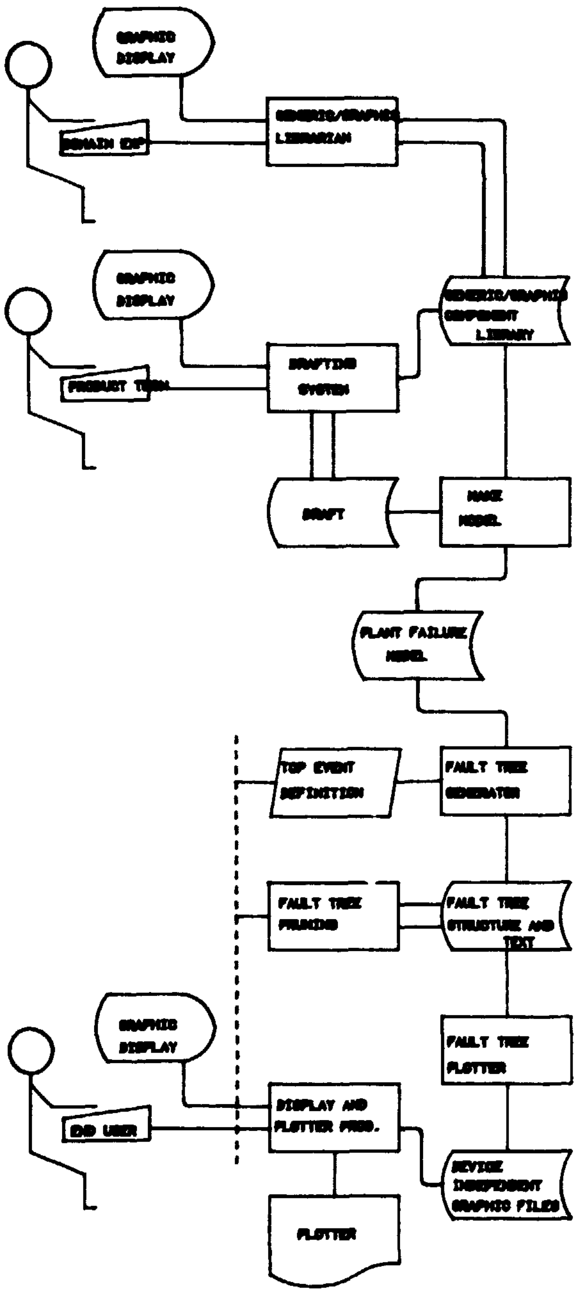


Figure 1.1 Block diagram of RIKKE.

A number of support programs are necessary in order for the system to run. The GENLIB (generic component librarian and editor) program allows new or updated component descriptions to be input to the program library, component descriptions to be extracted from the library, and printouts to be obtained from the component library. There will generally be several component model libraries in a RIKKE system. This is further described in chapter 4.

For input of operating procedures to the system (plant operator procedures or sequential control procedures) the HOPSA program may be included in the system. This allows procedures to be written in a programming language like form, and then to be translated to a 'component' form so that the procedures can be included into plant models in the same way as more conventional plant components.

When starting a 'session' (period of use) of RIKKE the first step is to identify which plant model will be used and which component library. This identification can be made by means of the MODEL command. Alternatively if the MODEL command is not used, any of the programs which need this information will ask (prompt) for it if the information has not been given. The MODEL command is needed when the user wishes to change from one plant model to another during the session. If the user has forgotten which model he is using, he can find out by typing WHAT.

The MODEL and WHAT commands are executed directly by the RIKKE monitor. Most of the other commands cause execution of FORTRAN subprograms. The drawing of the model is further described in chapter 2.

While executing any of the RIKKE subprograms, only the commands appropriate to the subprogram can be issued. Generally a return from a subprogram to the RIKKE monitor is made when the subprogram is completed, when an error occurs, or when the STOP command is given in the subprogram.

The first of the RIKKE subprograms to be described is GRACE, which is activated by the command DRAFT. Its purpose is to allow plant piping diagrams to be entered. This program asks first which model is to be input or modified, whether the model is a new or an old one, and which component model library is to be used. (If the program can discover any of this information for itself, it will not bother to ask for it). Thereafter, the user can construct the diagram by naming and placing components, and linking them together. A detailed description of GRACE is found in the GRACE User Manual (Larsen, 1982).

Once a piping diagram has been prepared, it can be turned into a model of the plant or system using the MAKE command. When this command has been given, no further commands need to be given; and no further information is provided, during execution of the program. A plant model with the current plant name will be built up. (If the MAKE command is issued just after starting, RIKKE will ask for the plant model name). Once MAKE has been completed, a plant failure model exists and fault trees and consequence diagrams can be constructed. (This is described in section 2.2).

The next step in producing a fault tree is to run the actual fault tree construction program using the command FAULT. The program replies by asking which component the TOP event is to occur in, and to identify the TOP event. The fault tree is then constructed in an internal form.

The fault trees produced by the FAULT command have text coded in numeric form. The FTTEXT command transforms the numeric form to text describing fault events. FTTEXT should be used after execution of FAULT, or, if time and disc space are short, after using the CUT command. The CUT command is described in section 2.4.

Once a fault tree has been produced and texted it may be plotted in any of three ways.

The first form of plotting is on a plotter. This requires that a plotting file is first produced, by executing the command FTPLOT. The plotting is then produced on the plotter itself by executing the command PLOT. The result is produced as a series of pages in A4 format, with cross page connections inserted automatically by the FTPLOT program.

The command FTSUPER_PLOT works like FTPLOT, but does not break the fault tree into A4 pages. In stead a larger drawing may be glued together from several pieces following the scissor marks provided.

The PLOT program will also plot plant diagrams, and on issuing the command the program will ask whether a block diagram (answer B) or a fault tree (answer F) is required. However this query will only be made when both fault tree an block diagram plotting files are present.

The second plotting facility is VIEW, which produces a display on the display screen. The format of the display is the same as that produced by PLOT, and requires that the FTPLOT command has been issued prior to execution of VIEW.

The third set of plotting facilities are for use with the lineprinter. The FTSHOW command allows a plot to be produced in abbreviated form on the lineprinter. Examples of this kind of output are shown in figure 1.2. FTSHOW does not require prior execution of FTPLOT.

SYSTEM: DEMO PART: 1

```
               1
               =
               !
               2
               +
              !!
            /-/\-\
            !    !
            3    4
            &    E
           !!
         /-/\-\
         !    !
         5    6
         #    N
         !    !
         7   20
         =    R
         !
         8
         +
        !!!
    /---/!\---\
    !    !    !
    9   10   11
    =    U    E
    !
   12
    #
    !
   13
    =
    !
   14
    +
   !!
 /-/\-\
 !    !
15   16
 U    &
     !!
   /-/\-\
   !    !
  17   18
   E    >
        !
       19
        U
```

Figure 1.2 A fault tree plotted by FTSHOW.

The TEXT command produces a disc file of text for individual events on the fault tree. This text is needed to interpret the output from FTSHOW. The file has the name <model-name>.FTX, for example PLNTMD.FTX. (A list of the extensions used can be seen in appendix A).

As an alternative FTSHOW, when operated from a display screen, may produce its result on a disc file.

Before plotting fault trees, it may be desirable to prune them of unwanted event types. The CUT command allows this pruning to be performed.

After this general introduction, each of the steps in the process of generating fault trees will be described in detail. In the following examples on both the users commands and the programs response are often given. We have adopted the notation of a exclamation mark (!) in the left margin to indicate when a communication to and from the computer is shown. This exclamation mark is of cause not seen on the screen.

2. HOW TO GENERATE A FAULT TREE.

Starting from the monitor in the PDP-11 or VAX system, you call the program (installed at the system) by typing:

```
!
!                                                    RIKKE
!  Welcome to RIKKE
!  What next:
!
```

You are now in the RIKKE monitor, and have a number of commands at your disposal. Here only some of the relevant commands are mentioned. The rest can be found in chapter 5. A list can be obtained by typing carriage return (<CR>) or HELP. In table 2.1 the most important commands are listed. A full list can be found in chapter 6 and in appendix E.

Table 2.1 Some commands in RIKKE.

Possible commands:	Used for:
CHECK	Checking if the library is OK
CONVERT	Convert a fault tree to FAUNET form
CUT	Prune fault tree of unwanted event types
CUTSET	Convert the fault tree to cutset
DRAFT	Activate model drafting
FAULT	Produce a fault tree
FTPLOT	Produce a plotting file / fault tree (A4 sheets)
FTSUPER_PLOT	Produce a plotting file / fault tree on one sheet
FTTEXT	Add readable text to fault tree
HELP	
LIBRARY	
MAKE	Build up a plant model
MODEL	Define or change model name
PLOT	Send plotting file to actual plotter
STOP	Stop execution of RIKKE session
UPDATE	
VIEW	Send plotting file to graphic display screen
WHAT	Ask for current model

The first step in an analysis of a new system is to make a model. This is described in the following section.

2.1 How to make a model.

In order to make a model of your plant you then type:

```
!
!                                                    DRAFT
!
```

You then call the subprogram GRACE, which handles the graphics. The program responds:

```
!
!  GRACE
!  Interactive drafting system
!  Model name:
!
```

You then define the name of the model. This name will identify your model in all parts of the RIKKE system.

Once the plant model name has been identified by using the MODEL command or by answering a prompt query, this model name is fixed, and will be used by most of the programs.

If no model name has been given, programs will ask the name of the plant model to be used.

If the user wishes to change the plant model name, he should use the MODEL command.

As an example we have chosen a system (see figure 2.1) which consist of two separators, one at high pressure, the other at low pressure. The system is a let down system, as in an ammonia plant.

Gas containing liquid enters separator 1, and gas without liquid leaves at the top. The liquid with disolved gas passes on to separator 2 in which the dissolved gas is released at a lower pressure. The pressure in separator 1 is usually around 300 bar and in separator 2 around 25 bar.

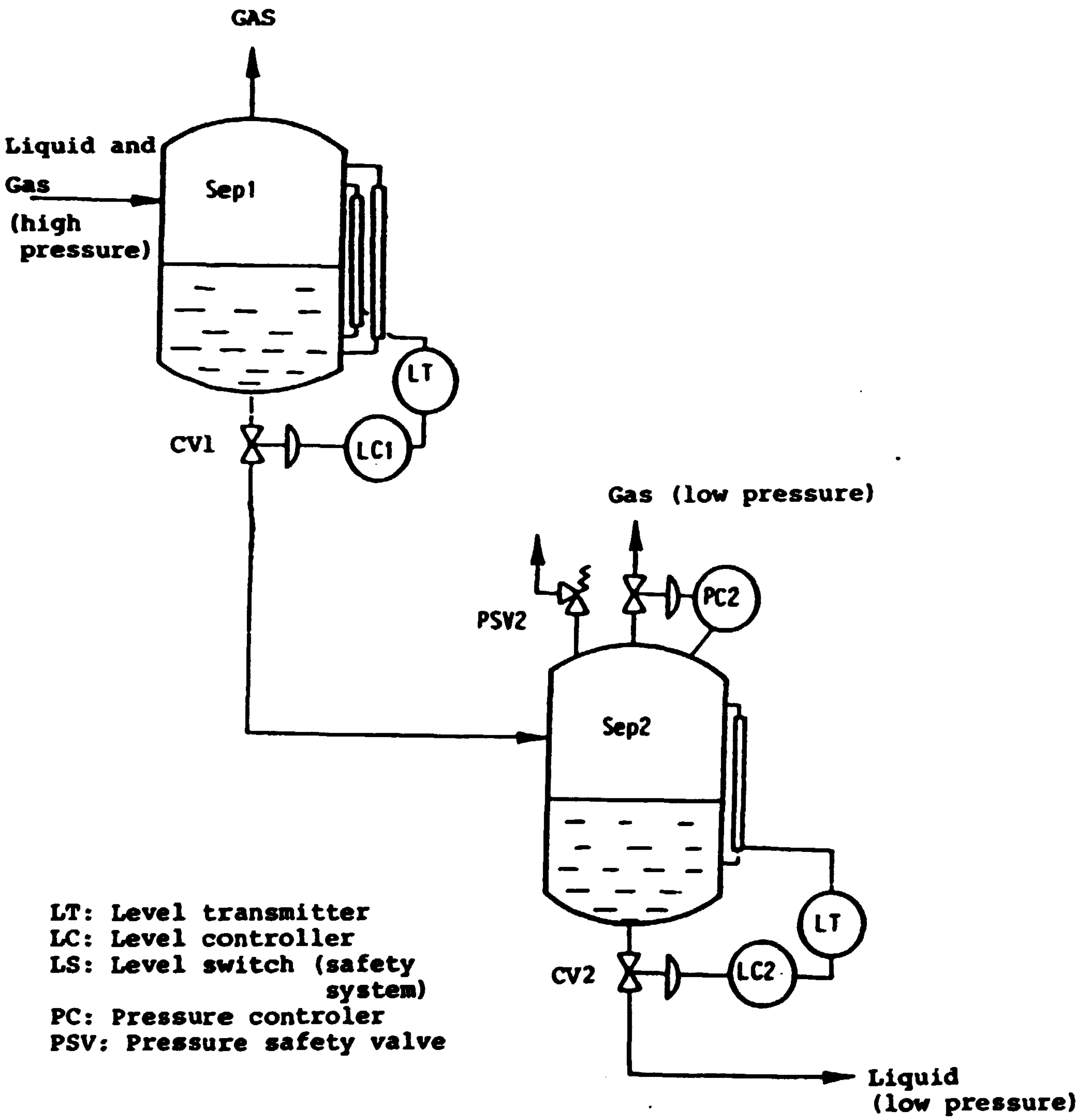


Figure 2.1 Piping and instrumentation diagram of a let down system.

We call the system LDDRUM:

```
!
!  Model name:                                    LDDRUM
!  Old, New or continue:                          NEW
!
```

The program needs to know from which library the components are to be chosen. With the RIKKE package two libraries are delivered: HAZLB2, with about 25 components, and FTLIB3 (the original safety library) with about 60 components. Here we have chosen to use FTLIB3. A full list of the components in the libraries can be found in chapter 7.

```
!
!  Library:                                       FTLIB3
!  Loading library
!  (blank screen)
!  What now:
!
```

We are now in the graphic editoring system, and can draw, include components from the library and link them together. If the carriage return is pressed, the possible commands are shown.Some of the most important are shown in table 2.2.

Table 2.2 Some commands in GRACE.

Command	Used for
All	Draft all components in the library.
Alter	Modify the parameters of an existing component in the draft.
Component	Include a new component in the draft.
Draw	Drawing lines, arcs and circles.
Duplicate	Duplicate a drawing.
Erase	Erase a drawing or component.
Find	Find a specified component in the draft and redraw it in a bigger window.
Grid	Draw a grid on the screen.
In	Define a new window with 1/4 of the current area.
Link	Link two ports.
Library	Change the library.
Move	Move a component.
Out	Define a new window with 4 times the current area.
Quit	Quit the whole draft.
Redraw	Redraw the current draft.
Relink	Delete and reenter connection between two components.
Save	Save the current draft data base.
Setup	Define the drawing facilities.
Shift	Move the whole draft.
Stop	Terminate drafting.
Text	Put a text on the draft.
Unlink	Delete a specified link between two ports.
Window	Define a part of the draft to be shown on the whole screen.

A further description of the graphic editors is found in GRACE User Manual (Larsen, 1982). In order to set the drawing facilities as desired, the command SETUP is used (default values in parenthesis):

```
!
!                                                          SETUP
!
!  Advanced drafting?  (No)                                <CR>
!  Names in output:
!  Components?  (No)                                       YES
!  Occupied ports?(No)                                     YES
!  Free ports?(No)                                         YES
!  Text in output:
!  Component text?(Yes)                                    NO
!  Free text(Yes)                                          <CR>
!  Text new components?(No)                                <CR>
!  Grid(x,y):(100,100)                                     <CR>
!  Step(20)                                                <CR>
!  Individual scaling?(No)                                 YES
!  Smooth links(Yes)                                       <CR>
!  Dotted links(No)                                        <CR>
!  What now:
!
```

The setup is now as desired for the first component to be included. Any other setup can of cause be used. If one wish to terminate the setup list on the way, this is done by typing an X.

The setup chosen will be active until the drafting is finished. The next time the Draft command is used, a new setup is required.

We then wish to add a component to the draft:

```
!
!  What now:                                               COMPONENT
!  Type:                                                   SEPARA
!  Form:                                                   1
.  Component name:                                         SEP1
!
```

The program responds with an activation of the position system. Point out the position and type the number from 0 to 7 or a space according to which rotation is desired. The orientation is as shown in figure 2.2.

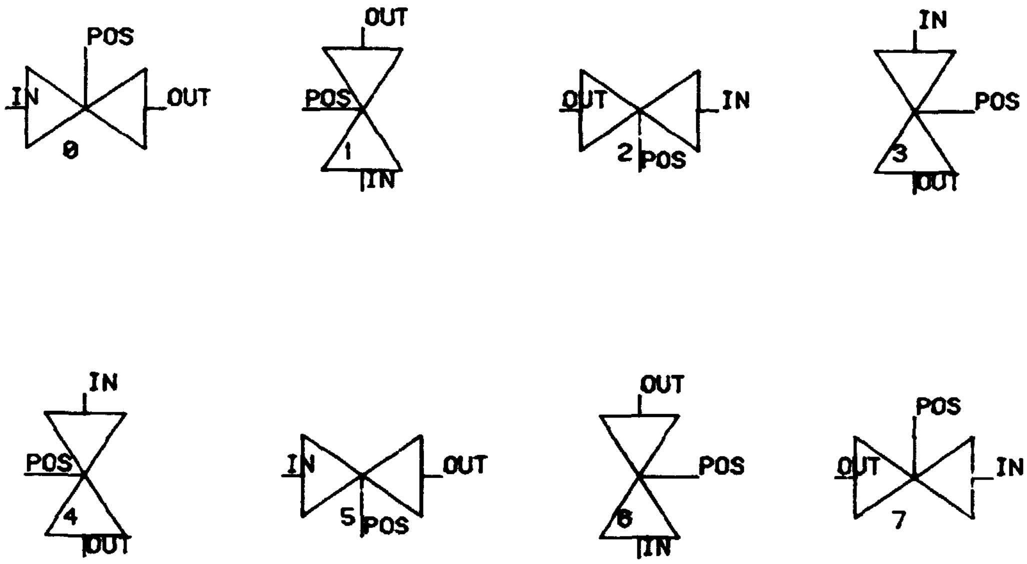


Figure 2.2 Orientation of a component.

```
!
!                                                    <SP>
!  Scale                                             2
!
```

The component is now seen on the screen. A good advise: Use an individual upscaling of 2 or 3 on the main components and the standard scaling (1) on for instance valves, sensors and controllers. The option "scaling" is chosen in the setup mode. The order is confirmed by

```
!
!                                                    E
!
!  What now:                                         COMPONENT
!  Type:                                             REGVLV
!  Form:                                             1
!  Component name:                                   RV
!
```

Point out position and type:

```
!
!                                                    3
!  Scale:                                            1
!
```

The component is shown, confirm by

```
!
!                                                E
!
```

We now wish to link the two components. We choose to link using the cursor to point position (on a VT 105 or 240 screen) or the sighting (on a 4014 screen). First we use the command LINK, then we point out components and confirm:

```
!
!  What now:                                     LINK
!
!
!                                                Y
!
```

Here you may get the responce "Too far away" which means that the cursor or sighting is pointing to a point too far away from a component or port to identify the component or port.

Then we point out the port and confirm by:

```
!
!                                                Y
!
```

We have now defined the beginning of the link and we then point out the second component, confirm and point out the port on the second component and confirm by:

```
!
!                                                Y
!
!
!                                                Y
!
```

A number of different link types are available (see table 2.3). A list on the screen can be obtained by typing a question mark here.

Table 2.3 Link types

```
-----------------------------------------------------------
     F  (For full line)
     U  (For up)
     D  (For down)
     L  (For left)
     R  (For right)
     C  (For connect)
     A  (For arc, using a <SP>
          to define the middle point in a curve)
     B  (For begin)
     E  (For end)
     M  (For moveable)
-----------------------------------------------------------
```

We responded:

```
!
!                                                  F
!  What now:                                       STOP
!  You have no hardcopy file
!  Want one before exit?(Yes)                      YES
!  Want a peekhole?(No)                            NO
!
!  Writing hardcopy file
!
```

The hardcopy file is the file, where the graphic information is stored. A further description is found in Larsen (1984). The peekhole command is used if you want to draw only part of the system, defined with a window.

```
!
!  Current draft not saved
!
!  SAVE before exit?(Yes)                          YES
!  Keep draft database(No)                         YES
!  Picture name was:  LDDRUM
!
!  What next:
!
```

We are now back in the RIKKE monitor.

```
!
!                                                  PLOT
!  Model name:  LDDRUM
!  Plotting Block-diagram
!
!  PLOT on:Plotte
!  General plotter drive
!  Options:
!                                                  DIP AUTO
!
```

We have now plotted figure 2.3.

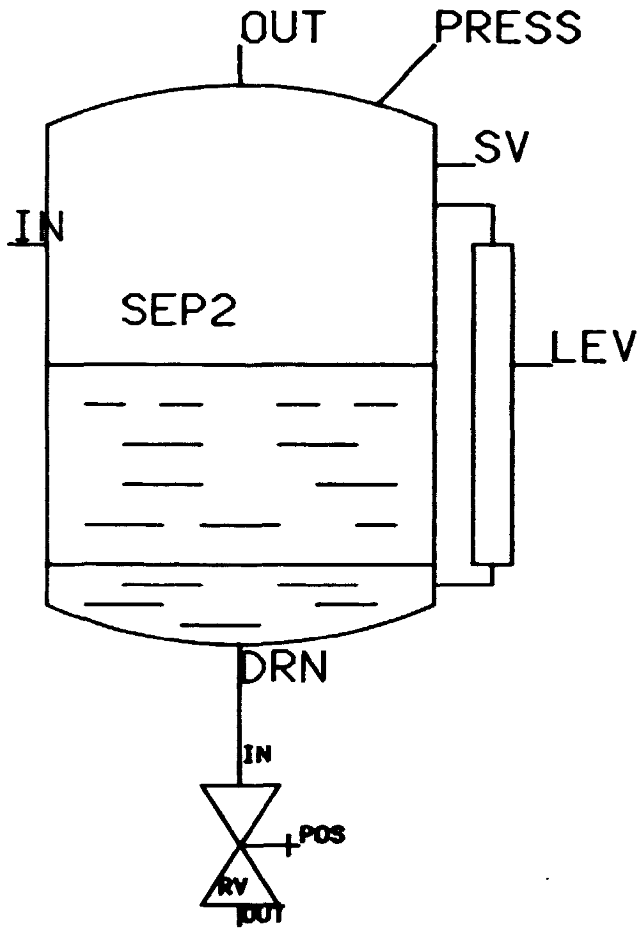


Figure 2.3 First part of a let down system.

We then want to continue our drafting:

```
!
!  What next:                                              DRAFT
!
!   GRACE
!  Interactive drafting system
!  Model name:                                             LDDRUM
!  Old, new or continue:                                   CONTINUE
!  Loading draft.
!
```

The draft is then shown on the screen. The option CONTINUE is allowed, because the draft database has been saved. This database uses the extension *.DIA. (A full list of extensions is found in appendix A).

```
!
!  What now:                                    COMPONENT
!  Type:                                        FORGAC
!  Form:                                        1
!  Component name:                              VPRV1
!
```

The position is then pointed out and the rotation is given and confirmed:

```
!
!                                               3
!                                               E
!
```

The name VPRV1 is chosen as a synonym of Valve Positioner for Regulation Valve 1. The maximum number of characters is 6.

To link the two components it is possible to use the pointing system as described above or to link by names as follows:

```
!
!  What now:                                    LINK
!                                               N
!  From component:                              RV1
!  Port:                                        POS
!  To component:                                VPRV1
!  Port:                                        POS
!                                               F
!
```

The same principle is used to instal a regulation unit connected to the valve positioner. The regulation unit is reading signals from a levelsensor (trough an inverter who invert the out-signal logic from the levelsensor). The screen will now show the draft as seen in figure 2.4:

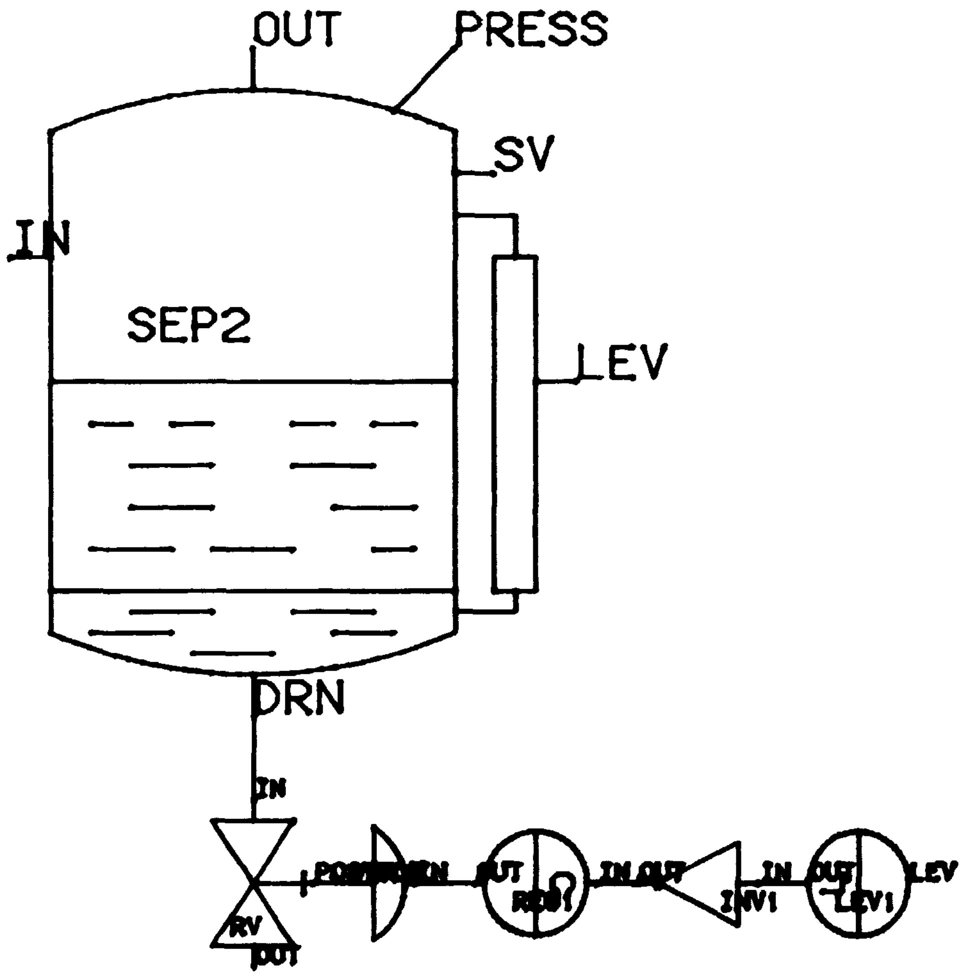


Figure 2.4 Part of a let down system.

We want to connect the levelsensor and the levelport on separator 2. This connection is not a part of the fluid system and we would therefore like to use a dotted line.

The dotted line has no function in relation to the fault tree.

The facility is provided in order to make a higher degree of agreement between a piping and instrument diagram and the model possible.

To draw dotted lines it is nessesary to make a new setup. This is done by writing SETUP and answer YES to the question "Dotted lines(No)?".

The commands to create a good lay out of the dotted line could be as follows:

```
!
!  What now:                                          LINK
!
```

The horizontal and vertical line of sight would now be shown. Sight in center of separator 1 and press Y (and confirm by Y once more), find the level port, the level sensor and the level port on the level sensor and do the same. The text on screen would now be:

```
!
!  Sep1
!  Lev
!  Lev1
!  Lev
!
```

The horizontal line of sight should now be placed trough level gate on separator 1 and the vertical line should be placed at the point where we want the line to change direction (down). A line can be drawn by giving the direction (L:left; R:right; U:up; D:down) from the starting point to the cursor. The correspondance to the RIKKE system is then:

```
!
!                                                     L
!                                                     E
!                                                     D
!                                                     C
!
```

The response from the system is the drawing of the wanted dotted line. It is possible to draw full lines in a setup with "dotted lines" by using the order F for full line instead of L,R,U,D and C for connect (see also table 2.3).

We then continue our drafting by adding supplies and drains to the not connected input and output lines on the separators and valves. The idea is simply to define the border of our system and to make sure that disturbances from outside your system (build into the supplies and drains) is taken into account.

The following components has in total been added:

SEP1	SEPARATOR
SEP2	
RV	REGULATION VALVE
RV2	
RV3	
VPRV1	FORGAC
VPRV2	
VPRV3	
REG1	REGULATOR
REG2	
REG3	
LEV1	LEVEL SENSOR
LEV2	
INV1	INVERTER
INV2	
INV3	
TRA2	TRANSA
SV2	SAFETY VALVE
1	DRAIN
2	
3	
4	
2	SUPPLY

The full drawing is seen in figure 2.5.

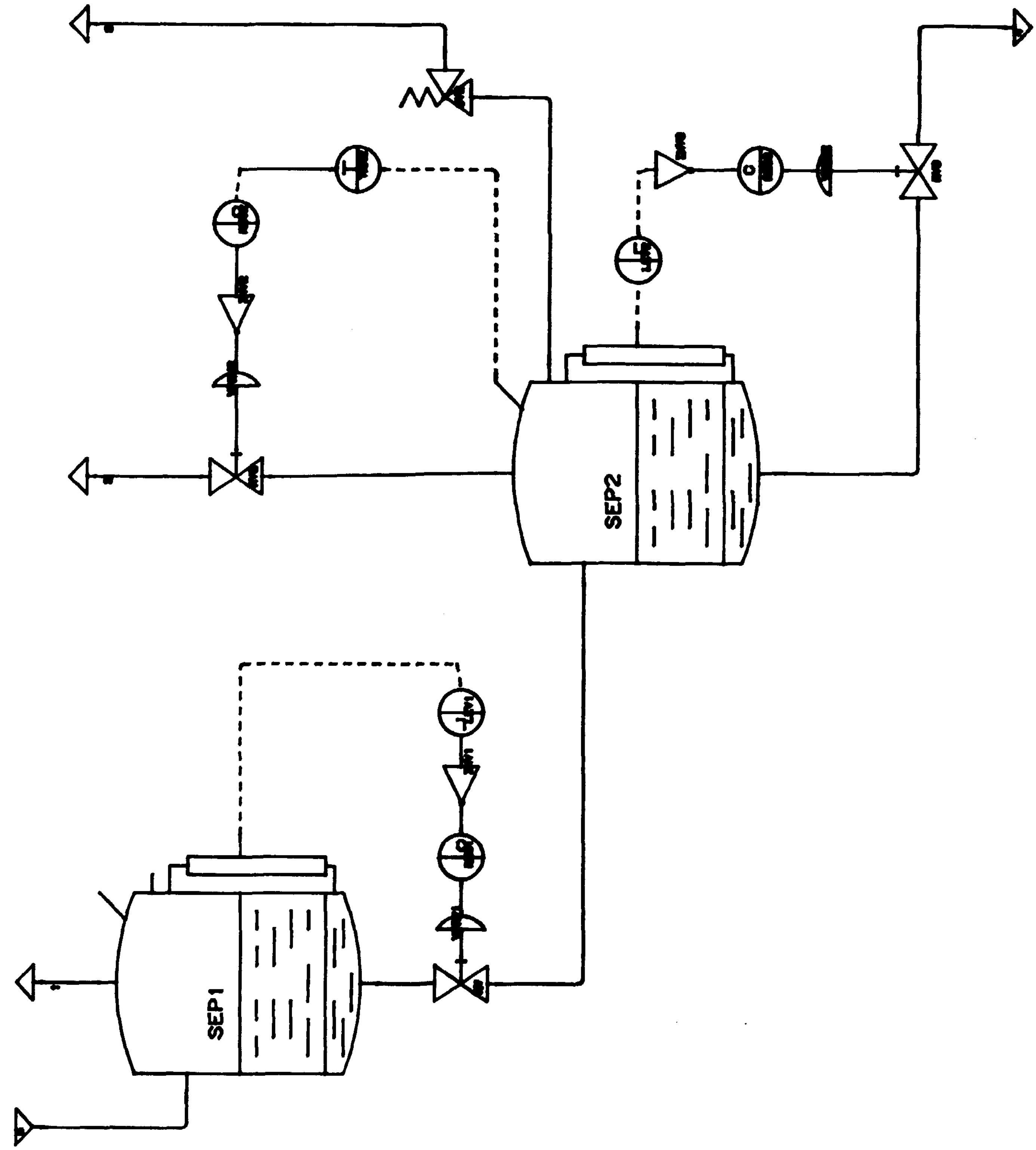


Figure 2.5 The final let down system.

2.2 How to make a plant failure model.

After finishing the model, we are interested in making a plant failure model. In the RIKKE monitor we use the command MAKE. The input data for the plant failure model generator is the block diagram file, with the extension *.BLK, just created by the DRAFT command.

The command will work independent on whether the model has been plotted or seen on the screen.

Nevertheless it is a good idea always to have a plot of your model in front of you, when you make the fault tree. The plant failure model consists of a list of components, their failure models and their connections. The plant failure model uses the extension *.PFM.

```
!
! What next:                                        MAKE
!
!       -RIKKE-
! Plant description Linker
!
! Model name:LDDRUM
!
! LIBRARY:  FTLIB3
! COMPONENT:  SEP2   - NEW TYPE:  SEPARA
! COMPONENT:  RV     - NEW TYPE:  REGVLV
! COMPONENT:  VPRV1 - NEW TYPE:  FORGAC
! CONNECT:  RV - PORT:  POS TO:  VPRV1 - PORT:  POS
! COMPONENT:  REG1 - NEW TYPE:  REG
! CONNECT:  VPRV1 - PORT:  IN TO:  REG1 - PORT:  OUT
! COMPONENT:  INV1 - NEW TYPE:  INVERT
! CONNECT:  REG1 - PORT:  IN TO:  INV1 - PORT:  OUT
! COMPONENT:  LEV1 - NEW TYPE:  LEVSNS
! CONNECT:  INV1 - PORT:  IN TO:  LEV1 - PORT:  OUT
! COMPONENT:  SEP1 - TYPE:  SEPARA
! CONNECT:  SEP1 - PORT:  DRN TO:  RV - PORT:  IN
! CONNECT:  SEP1 - PORT:  LEV TO:  LEV1 - PORT:  LEV
! CONNECT:  SEP2 - PORT:  IN TO:  RV - PORT:  LEV
! COMPONENT:  RV2 - TYPE:  REGVLV
! COMPONENT:  VPRV2 - TYPE:  FORGAC
! COMPONENT:  INV2  - TYPE:  INVERT
! COMPONENT:  REG2 - TYPE:  REG
! CONNECT:  RV2 - PORT:  IN TO:  SEP2 - PORT:  OUT
! CONNECT:  CRV2 - PORT:  POS TO:  VPRV2 - PORT:  POS
! CONNECT:  VPRV2 - PORT:  IN TO:  INV2 - PORT:  OUT
! CONNECT:  INV2 - PORT:  IN TO:  REG2 - PORT:  OUT
! COMPONENT:  TRA2 - NEW TYPE:  TRANSA
! CONNECT:  TRA2 - PORT:  OUT TO:  REG2 - PORT:  IN
! COMPONENT:  SV2 - NEW TYPE:  SV
! COMPONENT:  LEV2 - TYPE:  LEVSNS
! COMPONENT:  INV3 - TYPE:  INVERT
! COMPONENT:  REG3 - TYPE:  REG
! CONNECT:  SEP2 - PORT:  LEV TO:  LEV2 - PORT:  LEV
! CONNECT:  LEV2 - PORT:  OUT TO:  INV3 - PORT:  IN
! COMPONENT:  VPRV3 - TYPE:  FORGAC
! CONNECT:  TRA2 - PORT:  IN TO:  SEP2 - PORT:  PRESS
! COMPONENT:  RV3 - TYPE:  REGVLV
! CONNECT:  INV3 - PORT:  OUT TO:  REG3 - PORT:  IN
! CONNECT:  REG3 - PORT:  OUT TO:  VPRV3 - PORT:  IN
! CONNECT:  VPRV3 - PORT:  POS TO:  RV3 - PORT:  POS
! CONNECT:  RV3 - PORT:  IN TO:  SEP2 - PORT:  DRN
! CONNECT:  SEP2 - PORT:  SV TO:  SV2 - PORT:  IN
! COMPONENT:  1 - NEW TYPE:  DRAIN
! COMPONENT:  2 - TYPE:  DRAIN
! COMPONENT:  3 - TYPE:  DRAIN
! COMPONENT:  4 - TYPE:  DRAIN
! CONNECT:  4 - PORT:  IN TO:  RV3 - PORT:  OUT
! CONNECT:  SV2 - PORT:  OUT TO:  3 - PORT:  IN
! CONNECT:  RV2 - PORT:  OUT TO:  2 - PORT:  IN
! CONNECT:  SEP1 - PORT:  OUT TO:  1 - PORT:  IN
! COMPONENT:  2 - NEW TYPE:  SUP
! CONNECT:  5 - PORT:  OUT TO:  SEP1 - PORT:  IN
!
```

When this plant failure model has been made, you are ready to generate the fault tree.

2.3 How to generate a fault tree.

All the components and the connections between them are now prepared for making a fault tree. To make a fault tree we use the command: FAULT. A number of options are possible. These are shown in table 2.4.

Table 2.4 Options in command FAULT.

Option	Meaning
B	Break
I	Internal
D	Depth
T	Time
L	Loop-stop
E	Event list
S	Show
C	Continue

To solve our first small problem we have chosen the option D(epth).

The syntax for specifying the TOP event is

<variable name> BECOMES <value>

for example

OUT BECOMES ON

```
!
!  What next:                                          FAULT OPTION
D
!
!   -RIKKE-
!  Fault-tree Generator
!  Model name:  LDDRUM
!  Top-Event occurs in Component:                       SEP2
!  Top-Event:                                          DRUM -> BURST
!  Break evaluation at fault-tree level:                2
!   START AT 11:32:11
!  FINISH AT 11:32:19
!  THE CALCULATION TOOK
!          6 SECONDS
!  PROBLEM SIZE - MODE 1:
!           3 - MODE 2:   2
!
```

The fault tree is now generated. The resulting files have the extensions *.FTR (structure), *.FTX (text) and *.FTN (numeric text code).

In order to do the calculations faster, the computer works with the text stored in one database and numbers specifying the text elsewhere. It is therefore necessary to add readable text to your fault tree using the command FTTEXT:

```
!
!  What next:                                        FTTEXT
!
!
!   -RIKKE-
!  F-T or C-D Texter
!  Model name:LDDRUM
!
```

We now want to plot the fault tree. Two different commands are available, namely FTSUPER_PLOT and FTPLOT. The FTSUPER_PLOT produce one large drawing of the fault tree, whereas FTPLOT devides the fault tree into A4 pages. We have chosen the command FTSUPER_PLOT. The resulting fault tree is stored in the file with extension *.HCP.

```
!
!  What next:                                        FTSUPER_PLOT
!
!   -RIKKE-
!  Cause-Consequence-Diagram Plotter
!  Model name:LDDRUM
!
!
!
!   -RIKKE-
!  CCD and Fault-tree plot
!
!   Plot name:LDDRUM
!  BLOAD
!  BSUCC
!  LVLASS
!  BALANC
!  BSHOW
!  BMOVE
!      11      9
!  DRAW
!  ADDTXT
!  FINISH
!
!
!  What next:                                        PLOT
!
!         RIKKE
!  General Plotter Driver
!  Model name:LDDRUM
!  Plotting Fault-tree
!          or Block-diagram ?                        FAULT-TREE
!  PLOT ON:  PLOTTE
!  General plotter drive
!  Options:                                          DIP AUTO
!  Please change paper on plotter - DONE
!
!  What next:                                        STOP
!
!  goodbye
!
```

Many of the programs provide prompts, describing the input which is required next. e.g. in the VIEW program, "Fault tree, or Block diagram". For these prompts the capital letters in the prompt, introducing the words describing alternatives, are acceptable responses. In the example, a response "B" will allow a block diagram to be plotted.

The resulting fault tree is seen in figure 2.6.

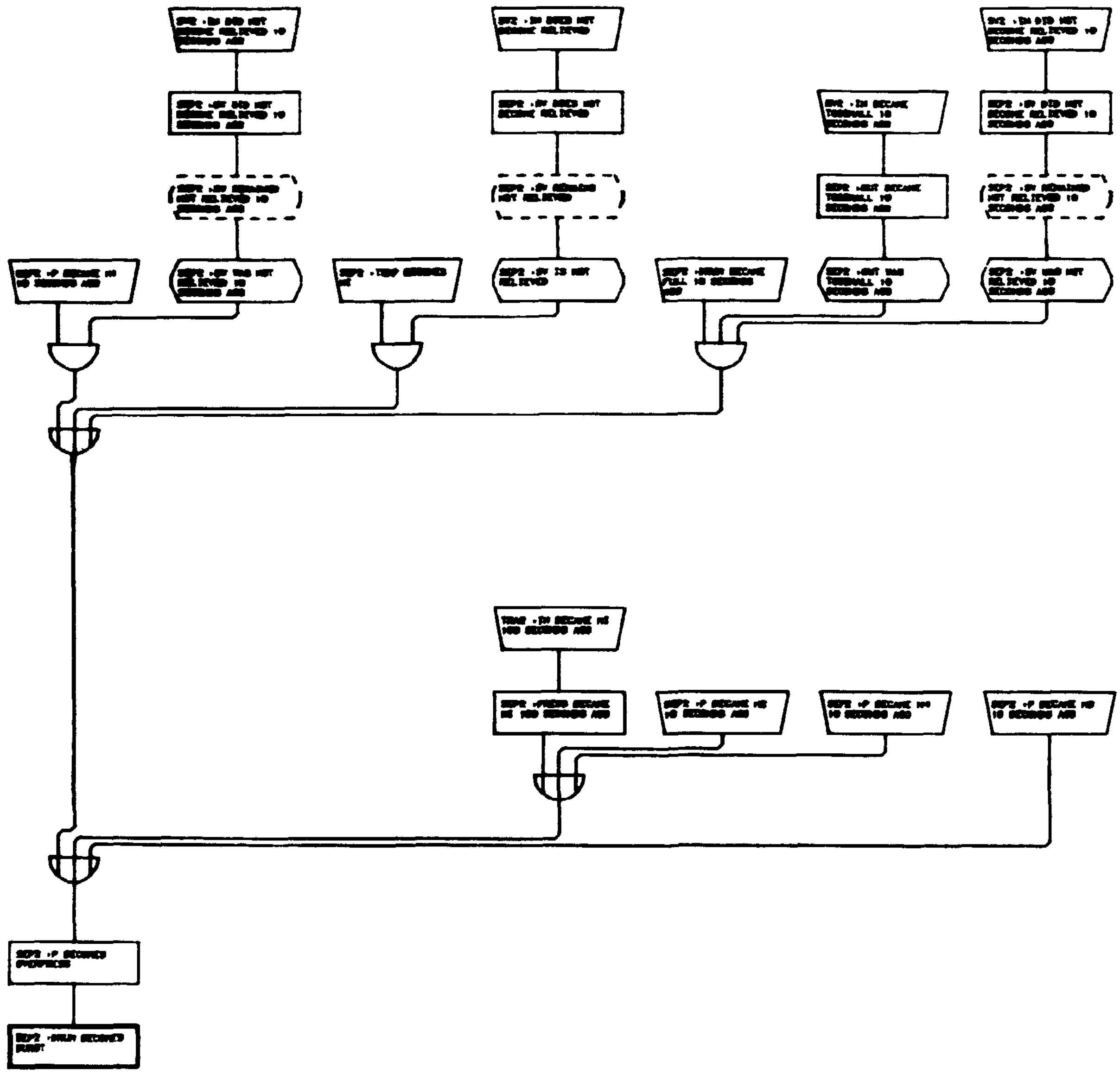


Figure 2.6 A fault tree for the event DRUM -> BURST in separator 2.
Model LDDRUM. Library FTLIB3. DEPTH = 2.

2.4 Interactive use of RIKKE

In the RIKKE system you may choose the other options when you generate your fault tree. The following options are possible. The most important are the BREAK option, which together with the component specification ALL, convert the program from an automatic fault tree generating program to a very powerful interactive tool. By using this command you yourself can decide how far a given branch of the fault tree is to be analysed. This means that it is possible to combine the logic in the computer, with your engineering judgement during the generation of the fault tree. This will reduce the size of the fault tree, and you can therefore analyse larger systems, or use more complicated models as you wish.

In the VAX version 2.8 of the RIKKE system (Not released yet) a further sophisticated option can be used. The Option permits the user to follow the development of the fault tree on one screen, while another shows the piping and instrumentation diagram, and indicates where the generation is at the moment. This option (SEND) can already be used with two PDP-11 computers.

In the following an example of the interactive use is shown. The possible commands is shown in table 2.5.

Table 2.5 Commands in option Break All

B	Break or
H	Halt - stop analysis here - take next alternative
T	This event is always TRUE
F	This event is always FALSE
S	Stop analysis here and on all following break-points
C	and any other response - continue analysis

```
!                                                   RIKKE2
!  What next:                                       FAULT OPTION B
!
!   - R I K K E -
!  Fault-tree Generator [V4G]
!  Model name:                                      LDDRUM
!  Top-Event occurs in Component:                   SEP2
!  Top-Event:                                      DRUM -> BURST
!
!  Break-Point in Component:                        ALL
!
!   START AT 08:50:53
!  SEP2:  DRUM -> BURST                   0
!  SEP2:  P -> OVERPRESS                  0
!  SEP2:  P -> H1                       -10
!  SEP2:  P -> DH1                      -20
!  SEP2:  OUT -> BLOCKED                -30
!  RV2:   IN -> BLOCKED                 -30 :   C
!
!  RV2:   POS -> FAILCLOSED             -30
!  VPRV2:POS -> FAILCLOSED              -30 :   C
!
!  VPRV2:IN -> FAILHI                   -30
```

```
!  INV2:  OUT -> FAILHI                    -30 :  C
!
!  INV2:  IN -> FAILLO                     -30
!  REG2:  OUT -> FAILLO                    -30 :  C
!
!  REG2:  SET -> ERROR                     -30
!  REG2:  WSTATE -> FAILLO                 -30
!  REG2:  PWR -> FAILOFF                   -30
!  REG2:  IN -> FAILLO                     -30
!  TRA2:  OUT -> FAILLO                    -30 :  C
!
!  TRA2:  WS -> LO_INPUT                   -30
!  TRA2:  WS -> FAILLO                     -30
!  RV2:   OUT -> BLOCKED                   -30
!  2:     IN -> BLOCKED                    -30 :  C
!
!  2:     WS -> BLOCKED                    -30
!  RV2:   WS -> BLOCKED                    -30
!  SEP2:  IN -> HISUPPC                    -30
!  RV:    OUT -> HISUPPC                   -30 :  H
!
!  SEP2:  IN -* SHUTOFF                    -20
!  RV:    OUT -* SHUTOFF                   -20 :  H
!
!  SEP2:  SV -* RELIEVED                   -20
!  SV2:   IN -* RELIEVED                   -20 :  C
!
!  SV2:   IPOS -* OPEN                     -20
!  SV2:   IN -* HISUPP                     -20
!  SEP2:  SV -* HISUPP                     -20 :  H
!
!  SEP2:  SV -* RELIEVED                   -10
!  SV2:   IN -* RELIEVED                   -10 :  H
!
!  SEP2:  TEMP -> HI                         0
!  SEP2:  TX -> DHT1                     -1000
!  SEP2:  TX -> DHT2                     -1100
!  SEP2:  IN -> DISTHIT                  -1110
!  RV:    OUT -> DISTHIT                 -1110 :  H
!
!  SEP2:  IN -* SHUTOFF                  -1100
!  RV:    OUT -* SHUTOFF                 -1100 :  H
!
!  SEP2:  IN -* SHUTOFF                  -1000
!  RV:    OUT -* SHUTOFF                 -1000 :  H
!
!  SEP2:  TX -> DHT1                      -100
!  SEP2:  IOUT -> REVFLO                  -110
!  SEP2:  P -> L3                         -110
!  SEP2:  P -> DL3                        -120
!  SEP2:  IN -> LOSUPPC                   -130
!  RV:    OUT -> LOSUPPC                  -130 :  H
!
!  SEP2:  OUT -* SHUTOFF                  -130
!  RV2:   IN -* SHUTOFF                   -130 :  H
!
!  SEP2:  OUT -* SHUTOFF                  -120
!  RV2:   IN -* SHUTOFF                   -120 :  H
!
!  SEP2:  OUT -* SHUTOFF                  -110
!  RV2:   IN -* SHUTOFF                   -110 :  H
!
```

```
!  SEP2:  OUT -> SUP                        -110
!  SEP2:  P -> DL2                          -110
!  SEP2:  P -> DL1                          -210
!  SEP2:  OUT -> ATM                        -220
!  RV2:   IN -> ATM                         -220 :   H
!
!  SEP2:  IN -> NOSUPP                      -220
!  RV:    OUT -> NOSUPP                     -220 :   H
!
!  SEP2:  IN -> ATM                         -220
!  RV:    OUT -> ATM                        -220 :   H
!
!  SEP2:  IN -> BLOCKED                     -220
!  RV:    OUT -> BLOCKED                    -220 :   C
!
!  RV:    POS -> FAILCLOSED                 -220
!  VPRV1:POS -> FAILCLOSED                  -220 :   C
!
!  VPRV1:IN -> FAILHI                       -220
!  REG1:  OUT -> FAILHI                     -220 :   H
!
!  RV:    IN -> BLOCKED                     -220
!  RV:    WS -> BLOCKED                     -220
!  SEP2:  IN -> DISTLOSUPPC                 -310
!  RV:    OUT -> DISTLOSUPPC                -310 :   H
!
!  SEP2:  OUT -* COMPHIBACKPC               -310
!  RV2:   IN -* COMPHIBACKPC                -310 :   H
!
!  SEP2:  OUT -* SHUTOFF                    -210
!  RV2:   IN -* SHUTOFF                     -210 :   H
!
!  SEP2:  OUT -* SHUTOFF                    -110
!  RV2:   IN -* SHUTOFF                     -110 :   S
!
!  SEP2:  OUT -> SUP                        -110
!  SEP2:  OUT -> HOT                        -110
!  RV2:   IN -> HOT                         -110 :  STOPPED
!  SEP2:  IN -> HIT                         -110
!  RV:    OUT -> HIT                        -110 :  STOPPED
!  SEP2:  IN -* SHUTOFF                     -100
!  RV:    OUT -* SHUTOFF                    -100 :  STOPPED
!  SEP2:  SV -* RELIEVED                       0
!  SV2:   IN -* RELIEVED                       0 :  STOPPED
!  SEP2:  DRUM -> FULL                       -10
!  SEP2:  DRN -> REVFLO                     -110
!  SEP2:  DRN -> BLOCKED                     -20
!  RV3:   IN -> BLOCKED                      -20 :  STOPPED
!  SEP2:  IN -* SHUTOFF                      -20
!  RV:    OUT -* SHUTOFF                     -20 :  STOPPED
!  SEP2:  DRN -> DISTHIBACKP                -110
!  RV3:   IN -> DISTHIBACKP                 -110 :  STOPPED
!  SEP2:  DRN -> HIBACKP                     -20
!  RV3:   IN -> HIBACKP                      -20 :  STOPPED
!  SEP2:  IN -* SHUTOFF                      -20
!  RV:    OUT -* SHUTOFF                     -20 :  STOPPED
!  SEP2:  IN -> HISUPP                      -110
!  RV:    OUT -> HISUPP                     -110 :  STOPPED
!  SEP2:  IN -> DISTHISUPP                  -110
!  RV:    OUT -> DISTHISUPP                 -110 :  STOPPED
!  SEP2:  DRN -* COMPLOBACKP                -110
!  RV3:   IN -* COMPLOBACKP                 -110 :  STOPPED
```

```
!  SEP2:  OUT -> TOOSMALL                        -10
!  SEP2:  SV -* RELIEVED                         -10
!  SV2:   IN -* RELIEVED                         -10  :  STOPPED
!  SEP2:  PRESS -> HI                           -100
!  SEP2:  P -> H2                                -10
!  SEP2:  P -> DH2                               -20
!  SEP2:  OUT -> SHUTOFF                         -30
!  RV2:   IN -> SHUTOFF                          -30  :  STOPPED
!  SEP2:  OUT -> HIBACKPC                        -30
!  RV2:   IN -> HIBACKPC                         -30  :  STOPPED
!  SEP2:  IN -* SHUTOFF                          -20
!  RV:    OUT -* SHUTOFF                         -20  :  STOPPED
!  SEP2:  SV -* RELIEVED                         -20
!  SV2:   IN -* RELIEVED                         -20  :  STOPPED
!  SEP2:  P -> H4                                -10
!  SEP2:  P -> DH4                               -20
!  SEP2:  OUT -> DISTHIBACKPC                    -30
!  RV2:   IN -> DISTHIBACKPC                     -30  :  STOPPED
!  SEP2:  IN -* COMPLOSUPPC                      -30
!  RV:    OUT -* COMPLOSUPPC                     -30  :  STOPPED
!  SEP2:  IN -* SHUTOFF                          -20
!  RV:    OUT -* SHUTOFF                         -20  :  STOPPED
!  SEP2:  SV -* RELIEVED                         -20
!  SV2:   IN -* RELIEVED                         -20  :  STOPPED
!  SEP2:  P -> H3                                -10
!  SEP2:  P -> DH3                               -20
!  SEP2:  IN -> SCUM                             -21
!  RV:    OUT -> SCUM                            -21  :  STOPPED
!  SEP2:  IN -> DISTHISUPPC                      -30
!  RV:    OUT -> DISTHISUPPC                     -30  :  STOPPED
!  SEP2:  OUT -* COMPLOBACKPC                    -30
!  RV2:   IN -* COMPLOBACKPC                     -30  :  STOPPED
!  SEP2:  IN -* SHUTOFF                          -20
!  RV:    OUT -* SHUTOFF                         -20  :  STOPPED
!  SEP2:  SV -* RELIEVED                         -20
!  SV2:   IN -* RELIEVED                         -20  :  STOPPED
!
!  FINISH AT 08:53:11
!  THE CALCULATION TOOK 2 MINUTES 17 SECONDS
!  PROBLEM SIZE - MODE 1:  29 - MODE 2:  3
!
```

Here the events are listed. "->" is interpretated as "becomes" and "-*" means "does not become".

This problem is too large to print in this manual, and the command CUT is therfore used. The command is futher described in section 2.5.

2.5 How to cut a fault tree.

Before plotting fault trees, it may be desirable to prune them of unwanted event types. The CUT command allows this pruning to be performed. When the CUT command is given, the program asks which type of cutting is required. The cutting type is selected by typing a number. This number should be the sum of the code numbers for each type of cutting required. The code numbers are given in table 2.6. A copy of table 2.6 can be obtained by pressing the carriage return key at the point where the type of cutting required is asked by the program.

Table 2.6 CUT code numbers.

1	- Drop remaining states
2	- Drop impossible events
4	- Drop normal conditions
8	- Drop unexpected events
16	- Suppress intermediate events/states
32	- Drop unserviceable states
64	- Drop common-mode events
128	- Drop negative loops
256	- Drop unlinked working states
512	- Drop opened loops
1024	- Assign "TRUE" and "FALSE"

Table 2.7 shows in details what gate types are modified, and what values are assigned at each different cutting mode.

Table 2.7 Values assigned to gates in different modes.

CUT code	Gate type	Assigned value	Tree mode
1	"R" with no inputs	.TRUE.	1
2	"I"	.TRUE.	2
4	"B" or "N" with no inputs	.FALSE.	2
	"A" with no inputs	.TRUE.	1
8	"U"	.FALSE.	1
16	"R","=","#",">","P" and "W" with one input	Value of input	-
32	"P" with no inputs	.TRUE.	2
64	"C"	.FALSE.	1
128	"-" with "."	.FALSE.	2
256	"W" with no inputs	.FALSE.	2
512	"O"	.TRUE.	2
1024	"T"	.TRUE.	-
	"F"	.FALSE.	-

The fault tree build in section 2.4 is pruned as an example of the use of the CUT command. We have chosen to cut all kind of unwanted event types, and the sum of the cut code numbers (the mode) is therefore 2047. The pruned tree is called LD2047.

```
!
!  What next:                                         CUT
!
!   - R I K K E -
!  Fault-tree Cutter [V2F]
!
!  Model name:                                        LDDRUM
!  Mode:                                              2047
!
!  Model-name for the pruned Tree:                    LD2047
!
!  Cutting text-file
!  Cutting text-file [numeric]
!  PRUNING FINISHED [ 571 / 579 ]
!
!  What next:                                         FTSHOW
!
!  FILE:  LD2047.FTR - SYSTEM:LD2047 FROM LDDRUM
!
```

```
!
! -
!  SYSTEM: LD2047 PART:    1
!
!
!                                        1
!                                        =
!                                        !
!                                        3
!                                        +
!                                      !!!!!
!              /----------------------/!!!\-------------------\
!              !          /------------/!\--------------\     !
!              !          !             !               !     !
!              4          5            193             208   225
!              &          &             &               &
!             !!         !!            !!!             !!!
!           /-/\--\    /-/\--\    /----/!\----\   /----/!\----\
!           !     !    !     !    !     !     !   !     !     !
!          13    62   65    156  197   204   207 212   221   224
!           &     H    +     H    +     H     H   &     H     H
!          !!         !!         !!              !!
!          !!         !!         !!              !\-----------\
!          !!         !!         !!              \------\     !
!          !!         !!         !\---------------\     !     !
!          !!         !!         \------------\   !     !     !
!          !!         !\----------------\     !   !     !     !
!          !!         \--------\        !     !   !     !     !
!          !\----------\       !        !     !   !     !     !
!          \------\    !       !        !     !   !     !     !
!                 !    !       !        !     !   !     !     !
!                17   44      66       67    200 201   215   218
!                 +    H        &       &     H   H     H     H
!                !!           !!       !!
!              /-/\--\       /-/\--\   !\-----\
!              !     !       !     !   \\     !
!              !     !       !     !    !     !
!             21    41      70    81   150   153
!              +     H       &     H    H     H
!             !!!           !!
!        /----/!\----\    /-/\--\
!        !     !     !    !     !
!       30    40    24   75    78
!        +     E     E    H     H
!      !!!!
! /-----/!!\-------\
! !    /-/\--\     !
! !    !     !     !
!31   32    33    36
! E    E     E     +
!                !!
!              /-/\--\
!              !     !
!             37    38
!              E     E
!
```

```
!
! -
!  SYSTEM: LD2047 PART:    2
!
!             225
!               &
!              !!!
!         /----/!\----\
!         !     !     !
!        229   241   244
!         +     H     H
!        !!
!      /-/\--\
!      !     !
!     232   231
!      H     &
!           !!
!         /-/\--\
!         !     !
!        235   238
!         H     H
!
```

The text to this fault tree is stored in LD2047.FTX:
LD2047 FROM LDDRUM

```
!
!    1 SEP2 :DRUM BECOMES BURST
!   31 REG2 :SET BECAME ERROR 30 SECONDS AGO
!   32 REG2 :WSTATE BECAME FAILLO 30 SECONDS AGO
!   33 REG2 :PWR BECAME FAILOFF 30 SECONDS AGO
!   37 TRA2 :WS BECAME LO_INPUT 30 SECONDS AGO
!   38 TRA2 :WS BECAME FAILLO 30 SECONDS AGO
!   40 2 :WS BECAME BLOCKED 30 SECONDS AGO
!   24 RV2 :WS BECAME BLOCKED 30 SECONDS AGO
!   41 RV :OUT BECAME HISUPPC 30 SECONDS AGO
!   44 RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!   62 SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!   75 RV :OUT BECAME DISTHIT 1110 SECONDS AGO
!   78 RV :OUT DID NOT BECOME SHUTOFF 1100 SECONDS AGO
!   81 RV :OUT DID NOT BECOME SHUTOFF 1000 SECONDS AGO
!   150 RV :OUT BECAME HIT 110 SECONDS AGO
!   153 RV :OUT DID NOT BECOME SHUTOFF 100 SECONDS AGO
!   156 SV2 :IN DOES NOT BECOME RELIEVED
!   200 RV2 :IN BECAME SHUTOFF 30 SECONDS AGO
!   201 RV2 :IN BECAME HIBACKPC 30 SECONDS AGO
!   204 RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!   207 SV2 :IN DID NOT BECOME RELIEVED 20 SECONDS AGO
!   215 RV2 :IN BECAME DISTHIBACKPC 30 SECONDS AGO
!   218 RV :OUT DID NOT BECOME COMPLOSUPPC 30 SECONDS AGO
!   221 RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!   224 SV2 :IN DID NOT BECOME RELIEVED 20 SECONDS AGO
!   232 RV :OUT BECAME SCUM 21 SECONDS AGO
!   235 RV :OUT BECAME DISTHISUPPC 30 SECONDS AGO
!   238 RV2 :IN DID NOT BECOME COMPLOBACKPC 30 SECONDS AGO
!   241 RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!   244 SV2 :IN DID NOT BECOME RELIEVED 20 SECONDS AGO
!
```

2.6 Use of command files in RIKKE.

When you are familiar with making fault trees and cause-consequence diagrams, you can operate the RIKKE system with a set of command files. You can design your own command files, which contain different combinations of commands to the RIKKE system. Some times you make wish to make only the fault tree in an interactive way, and some times you would like to have both cutsets, tiesets and pruned fault trees. Each command file can contain the commands needed for the different analysis.

As an example we have made three command files: one for the plant failure model building and fault tree generation, one for the cutting of the fault tree and one for the generation of cutsets and tiesets.

```
!DEMEX1.EXE - EXAMPLE OF A
!COMMAND FILE WITH PLANT FAILURE
!MODEL AND FAULT TREE GENERATION
MODEL
DRAFT OLD HAZLB2
PLOT B OPTION 'DIP AUTO'
MAKE
FAULT OPTION D LEVEL ALL
FTTEXT
FTSUPERPLOT
VIEW FT
FTSHOW
CONVERT FT
FTCHECK
```

```
!DEMEX2.EXE - EXAMPLE OF A
!COMMAND FILE WITH CUTTING
CUT
FTSUPERPLOT
VIEW FT
FTSHOW
TYPE
PLOT FT OPTION 'DIP AUTO'
```

```
!DEMEX3.EXE - EXAMPLE OF A
!COMMAND FILE WITH CUTSETS AND TIESETS
CUTSET
EVALUATE
PATHSET
```

As you can see the commands are placed in seperate lines with the attached subcommands in the same line. If a line begin with an exclamation mark, the rest of the text in the line is only viewed as comments which is shown on the screen, but not executed as commands.

When you want to use the command files you enter the RIKKE program and make a draft of the plant in an ordinary way. Then you first make the plant failure model and the fault tree by typing:

```
!
!  What next:                                   EXECUTE DEMEX1
!
```

and the commands in the file DEMEX1 will be executed. You can follow the execution on the screen, since the commands are typed as they are executed. For some of the commands in DEMEX1 we have not given all the information needed for execution of the commands, and we are then asked interactively for the missing information.

```
!
!  Then next:  !DEMEX1.EXE - EXAMPLE OF A
!  Then next:  !COMMAND FILE WITH PLANT FAILURE
!  Then next:  !MODEL AND FAULT TREE GENERATION
!
!  Then next:  MODEL
!  Model name:                                        LGTANK
!  The model LGTANK contains:
!  Block-diagram
!
!  Then next:  DRAFT OLD
!    GRACE
!  Interactive drafting system
!  Model name:  LGTANK
!  Old, new or continue:  OLD
!  Loading draft
!
```

Then the draft is shown on the screen, and if you are satisfied with the draft, you can stop the drafting in the usual way. The commands in the command file DEMEX1 will then continue to be executed until the command STOP is reached, and you leave the RIKKE system.

2.7 How to generate a cause-consequense diagram.

The cause-consequence diagram show the effects of a given event. In the fault tree generation you determined a top event and the RIKKE program found the causes to this event. In the cause-consequence diagram building you choose an event and the RIKKE program will find the possible consequences. For each consequence you can decide whether you think it is reasonable or not; B stops further analysis of the consequence; U stops unwanted consequences; S stops further analysis and C continue the analysis. The program package is activated by the command CONSEQUENCE and need information about the component name and the initial event type. As an example we will make a cause-consequence diagram of the LDDRUM system. The initial event is IN -> HIT and it occurs in SEP2:

```
!
!  What next:                                         CONSEQUENCE
!               - R I K K E -
!  Consequence-Diagram Generator [V3A]
!  Model name:                                        LDDRUM
!  Initial-Event occurs in Component:                 SEP2
!  Initial-Event:                                     IN -> HIT
!
!  Comp.       Event                            Time
!  -------+-------------------------------+-----
!  SEP2:   TEMP -> DISTHI                 10 :       C
!  SEP2:   TX -> DHT1                     10 :       C
!  /--------
!  SEP2:   IN ISNT SHUT                   10
!  \--- conditioning
!  SEP2:   OUT -> DISTHIT                 10 :       C
!  /--------
!  RV3:    POS IS OPEN                    10
!  \--- conditioning
!  SEP2:   T -> DISTHI                    10 :       C
!  RV3:    OUT -> DISTHIT                 10 :       C
!  SEP2:   DRN -> DISTHIT                 10 :       C
!  /--------
!  RV2:    POS IS OPEN                    10
!  \--- conditioning
!  RV2:    OUT -> DISTHIT                 10 :       C
!  SEP2:   TEMP -> HI                    110 :       C
!  /--------
!  SEP2:   SV ISNT RELIEVED              110
!  \--- conditioning
!  SEP2:   OUT -> HIT                    110 :       C
!  /--------
!  RV3:    POS IS OPEN                   110
!  \--- conditioning
!  SEP2:   P -> OVERPRESS                110 :       C
!  RV3:    OUT -> HIT                    110 :       C
!  SEP2:   DRUM -> BURST                 110 :       C
!  SEP2:   T -> HI                       110 :       C
!  SEP2:   DRN -> HIT                    110 :       C
!  /--------
!  RV2:    POS IS OPEN                   110
!  \--- conditioning
!  RV2:    OUT -> HIT                    110 :       C
!
```

The generated cause-consequence diagram is turned into a plot by the plotting commands CDPLOT or CDSUPER_PLOT:

```
!
!  What next:                                       CDSUPER_PLOT
!
!              - R I K K E -
!  Cause-Consequence-Diagram Plotter [V2C]
!  Model name:  LDDRUM
!
!         - R I K K E -
!  CCD & Fault-tree plot [V3B]
!  Plot name:  LDDRUM
!  BLOAD
!  BSUCC
!  LVLASS
!  BALANC
!  BSHOW
!  BMOVE
!    Size of plot:      5    *     13
!  DRAW
!  ADDTXT
!  FINISH
!
```

The text to the plot of the cause-consequence diagram is turned into readable form by the command CDTEXT. The text is stored in a file with extension *.CDX. The text in numeric form is found in the file with extension *.CDN.

```
!
!  What next:                                       CDTEXT
!
!       - R I K K E -
!  F-T or C-D Texter [V2B]
!  Model name:  LDDRUM
!
```

The cause-consequence diagram is stored in a file with extension *.CDR and can be shown on the screen by the command CDSHOW:

```
!
!  What next:                                      CDSHOW
!
!  FILE:  LDDRUM.CDR - SYSTEM:LDDRUM
!
!  _
!  SYSTEM: LDDRUM PART:    1
!
!
!                     1
!                     E
!                     !
!                     2
!                     *
!                    !!!
!      /-------------/!\--------\
!      !              !         !
!      3              4         5
!      E              E         #
!      !              !         !
!      6              7        11
!      E              Y         E
!      !              !         !
!     14              8        12
!      #              *         Y
!      !             !!         !
!      !         /---/\---\     !
!      !         !        !     !
!     16         9       10    13
!      E         E        #     #
!      !         !        !     !
!     17        20       24    15
!      Y         *        E     U
!      !        !!        !
!      !      /-/\--\     !
!      !      !     !     !
!     18     21    22    25
!      #      Y     E     Y
!      !      !     !     !
!     19     23    29    26
!      U      E     #     #
!             !     !     !
!            27    30    28
!             U     E     U
!                   !
!                  31
!                   Y
!                   !
!                  32
!                   #
!                   !
!                  33
!                   U
!
!
```

```
!
!  What next:                            TYPE LDDRUM.CDX
!
!  ---------- File:  LDDRUM.CDX ----------
!  LDDRUM
!    1 SEP2 :IN BECOMES HIT
!    3 SEP2 :TEMP BECOMES DISTHI 10 SECONDS AFTER START
!    4 SEP2 :TX BECOMES DHT1 10 SECONDS AFTER START
!    7 SEP2 :IN IS NOT SHUTOFF 10 SECONDS AFTER START
!    5 SEP2 :OUT BECOMES DISTHIT 10 SECONDS AFTER START
!    11 RV3 :IN BECOMES DISTHIT 10 SECONDS AFTER START
!    12 RV3 :POS IS OPEN 10 SECONDS AFTER START
!    6 SEP2 :T BECOMES DISTHI 10 SECONDS AFTER START
!    13 RV3 :OUT BECOMES DISTHIT 10 SECONDS AFTER START
!    15 5 :IN BECOMES DISTHIT 10 SECONDS AFTER START
!    14 SEP2 :DRN BECOMES DISTHIT 10 SECONDS AFTER START
!    16 RV2 :IN BECOMES DISTHIT 10 SECONDS AFTER START
!    17 RV2 :POS IS OPEN 10 SECONDS AFTER START
!    18 RV2 :OUT BECOMES DISTHIT 10 SECONDS AFTER START
!    19 3 :IN BECOMES DISTHIT 10 SECONDS AFTER START
!    9 SEP2 :TEMP BECOMES HI 110 SECONDS AFTER START
!    21 SEP2 :SV IS NOT RELIEVED 110 SECONDS AFTER START
!    10 SEP2 :OUT BECOMES HIT 110 SECONDS AFTER START
!    24 RV3 :IN BECOMES HIT 110 SECONDS AFTER START
!    25 RV3 :POS IS OPEN 110 SECONDS AFTER START
!    23 SEP2 :P BECOMES OVERPRESS 110 SECONDS AFTER START
!    26 RV3 :OUT BECOMES HIT 110 SECONDS AFTER START
!    28 5 :IN BECOMES HIT 110 SECONDS AFTER START
!    27 SEP2 :DRUM BECOMES BURST 110 SECONDS AFTER START
!    22 SEP2 :T BECOMES HI 110 SECONDS AFTER START
!    29 SEP2 :DRN BECOMES HIT 110 SECONDS AFTER START
!    30 RV2 :IN BECOMES HIT 110 SECONDS AFTER START
!    31 RV2 :POS IS OPEN 110 SECONDS AFTER START
!    32 RV2 :OUT BECOMES HIT 110 SECONDS AFTER START
!    33 3 :IN BECOMES HIT 110 SECONDS AFTER START
!
```

3. HOW TO USE PAUNET.

The PAUNET program package calculates cutsets and pathsets/tiesets of fault trees and further allows availability and reliability calculations. It exists as a set of FORTRAN programs which can be activated by issuing commands to the RIKKE monitor. (Andrews (1983)).

For the most part the programs communicate by means of input and output files in a standard 'Fault tree' format. The programs have in some cases parameters, such as, for example the 'name' of the system or fault tree under investigation, or the program execution options. Such parameters are requested by the programs in prompt-response form, unless the information is already available to the system.

The usual progression of a fault tree analysis with PAUNET is as follows.

(1) The fault tree description is written as a file on the disk store in a relatively free format (see appendix C) together with the primary event failure and repair data (see appendix D). Instead of a fault tree a network can be analysed (see appendix C). The fault tree generated by RIKKE is converted to PAUNETs fixed format by the command CONVERT.

(2) The fault tree is used as basis for calculation of minimal cutsets by the command CUTSET or of minimal path/tiesets by the command TIESET.

(3) The generated cutsets or tiesets may now be used for probability calculations using bounding techniques by the command UNAVAILABILITY.

(4) In order to perform an exact probability calculation, the cutsets or tiesets may be decomposed by issuing the command DECOMPOSE, whereafter the 'UNAVAILABILITY DECOMPOSED' command performs the probability calculation.

(5) The resulting modularized cut/tiesets can be completely evaluated by the EVALUATE command, or they may be converted into a pruned fault tree by the TREE command.

(6) The cutsets and tiesets can further on be grouped by the command GROUPING. The grouped sets are stored in a file with the extension *.CSG/*.TSG.

(7) Using a pruned fault tree generated from minimal cut/tie sets as input for another tie/cutset calculation often end up with a set, which is modularized to an even higher degree; ending up with completely modularized minimal cutsets or tiesets.

(8) The final cut/tiesets are found in a file on the disk, from where they may be TYPEd or PRINTed. The names of the files consist of the system name followed by an extension classifying the actual set. As an example LDDRUM.CSR contains the resulting minimal cutsets for the LDDRUM system, while LDDRUM.TSG contains the grouped tiesets for the same system. The total set of file names is listed in appendix A and C.

(9) In general after issuing a command that result in an output on the terminal, a copy of the text will exist on the disk with the file name *.LIS (* stand for the system name). This file may be printed on the typewriter by the PRINT command: e.g. PRINT LDDRUM.LIS.

3.1 How to convert a fault tree into cutsets.

As an example of conversion of a fault tree into cutsets and tiesets we use the fault tree of the LDDRUM model made in section 2.4 and pruned in section 2.5 under the modelname LD2047.

As mentioned in the start of this chapter the fault tree have to be converted by the command CONVERT:

```
!
!  What next:                                          CONVERT
!
!  - R I K K E <=> F A U N E T -
!     Converter Program [V1A]
!
!  Model name:                                         LD2047
!
!  Converting Fault-tree, cutsets or Evaluated cutsets:  F
!
!  Converting System:  LDDRUM
!  Loading events & gate-numbers
!  - last event = 244
!  Comparing events in LD2047.FTX
!  Converting tree
!  - dropped, trying *.FTN
!  Comparing events in LD2047.FTN
!  -     10 matching events
!
!  Save conversion table
!
!  Converting tree
!
```

The fault tree text is stored in readable form in LD2047.FTX and in numeric code in LD2047.FTN.

The fault tree has now been converted into FAUNET form and can be analysed by the command CUTSET:

```
!
!  What next:                                          CUTSET
!
!  CUTSET or TIESET:  CUTSET
!  CUTSET of:  LD2047
!  New or Pruned [NEW]:  NEW
!  SYSTEM:  LDDRUM
!  Extract (Yes/No) [Y]:  YES
!  Highest order wanted:    999
!  Top gate:       0
!  GATE:  1000 SELECTED AS TOP
!  FACTORIZE
!  FACTORIZE
!  FACTORIZE
!  EXTRACT [Y]
!  FACTORIZE
!  FACTORIZE
!  EXTRACT [Y]
```

```
!  FACTORIZE
!  FACTORIZE
!  EXTRACT [Y]
!
!  LOAD LD2047
!  EVALUATE
!  MINIMIZE
!  OVERFLOW
!
!  FINISH LD2047
!  REDUCE
!  OUTPUT
!
!  RESULT OF LDDRUM
!
!  REDUCED CUTSETS:
!    1.  SET OF ORDER 1
!  ------------
!    1.
!
!  EVALUATED CUTSETS:
!   13.  SETS OF ORDER 3
!    2.  SETS OF ORDER 4
!  ------------
!   15.
!
```

The CUTSET command have default NEW fault tree and the answer YES to the question 'extract ?'. If the fault tree is pruned and no extract is wanted the command is CUTSET PRUNED NO. Further on the highest order is default 999 and the top gate 0. If the tree should not be analysed using the first gate in the file as top gate, then another gate number must be assigned in the command.

If you use the command CONVERT again you can convert the cutsets into readable text which is stored in a file with the extension *.LIS:

```
!
!  What next:                                          CONVERT
!
!  - R I K K E <=> F A U N E T -
!     Converter Program [V1A]
!
!  Model name:  LD2047
!  Converting Fault-tree, Cutsets or Evaluated cutsets:  C
!
!  Converting modularized cutsets
!  Text loaded - last event/state =    238
!
!  - R I K K E / F A U N E T -
!     Cutset Printer [V1A]
!
!  The cutset text is stored in:  "LD2047.LIS"
!
```

The content of the file LD2047.LIS is:

```
!
!  What next:                              TYPE LD2047.LIS
!
!  ---------- File:  LD2047.LIS ----------
!       Minimal cutsets found in model:  LD2047 FROM LDDRUM
!
!       Top event in SEP2 :DRUM BECOMES BURST
!
!  Complex Module   1 fails if:
!
!   1)Fault in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!     and in RV :OUT BECAME SCUM 21 SECONDS AGO
!
!  ------------------------
!   2)Fault in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!     and in RV :OUT BECAME DISTHISUPPC 30 SECONDS AGO
!     and in RV2:IN DID NOT BECOME COMPLOBACKPC 30 SECONDS AGO
!
!  ------------------------
!   3)Fault in 2 :WS BECAME BLOCKED 30 SECONDS AGO
!     and in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!
!  ------------------------
!   4)Fault in RV :OUT BECAME HISUPPC 30 SECONDS AGO
!     and in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!
!  ------------------------
!   5)Fault in TRA2 :WS BECAME LOINPUT 30 SECONDS AGO
!     and in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!
!  ------------------------
!   6)Fault in TRA2 :WS BECAME FAILLO 30 SECONDS AGO
!     and in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!
!  ------------------------
!   7)Fault in REG2 :WSTATE BECAME FAILLO 30 SECONDS AGO
!     and in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!
!  ------------------------
!   8)Fault in REG2 :PWR BECAME FAILOFF 30 SECONDS AGO
!     and in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!
!  ------------------------
!   9)Fault in RV2 :WS BECAME BLOCKED 30 SECONDS AGO
!     and in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!
!  ------------------------
!  10)Fault in REG2 :SET BECAME ERROR 30 SECONDS AGO
!     and in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!
!  ------------------------
```

```
!  11)Fault in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!     and in RV2 :IN BECAME DISTHIBACKPC 30 SECONDS AGO
!     and in RV :OUT DID NOT BECOME COMPLOSJPPC 30 SECONDS AGO
!
!  ------------------------
!  12)Fault in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!     and in RV2 :IN BECAME SHUTOFF 30 SECONDS AGO
!
!  ------------------------
!  13)Fault in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!     and in RV2 :IN BECAME HIBACKPC 30 SECONDS AGO
!
!  ------------------------
!  14)Fault in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!     and in RV :OUT BECAME DISTHIT 1110 SECONDS AGO
!
!  ------------------------
!  15)Fault in RV :OUT DID NOT BECOME SHUTOFF 20 SECONDS AGO
!     and in SV2 :IN DID NOT BECOME RELIEVED 10 SECONDS AGO
!     and in RV :OUT BECAME HIT 110 SECONDS AGO
!
!  ------------------------
!
!  Cutsets of  1.  order:
!
!   1)Fault in module #  1
!
!  ------------------------
```

The tiesets are made by the command TIESET:

```
!  What next:                                          TIESET
!
!  CUTSET or TIESET:  TIESET
!  TIESET of:  LD2047
!  New or Pruned [NEW]:  NEW
!  SYSTEM:  LDDRUM
!  Extract (Yes/No) [Y]:  YES
!  Highest order wanted:    999
!  Top gate:      0
!  GATE:  1000 SELECTED AS TOP
!  FACTORIZE
!  FACTORIZE
!  FACTORIZE
!  EXTRACT [Y]
!  FACTORIZE
!  FACTORIZE
!  EXTRACT [Y]
!  FACTORIZE
!  FACTORIZE
!  EXTRACT [Y]
!
!  LOAD LD2047
!  EVALUATE
!  MINIMIZE
!  OVERFLOW
```

```
!
!  FINISH LD2047
!  REDUCE
!  OUTPUT
!
!  RESULT OF LDDRUM
!
!  REDUCED TIESETS:
!         1.  SET  OF ORDER  1
!  ------------
!         1.
!
!  EVALUATED TIESETS:
!         2.  SETS OF ORDER  1
!         4.  SETS OF ORDER 15
!  ------------
!         6.
!
```

As well as with the CUTSET command you can define othe options than the default.

3.2 Analysis of cutsets by FAUNET.

Both the cutsets and the tiesets can be evaluated by the command EVALUATE. The generated cutsets or tiesets are expanded from the complex events to an expression in terms of the original basic events. As an example we have have chosen to evaluate the cutsets (which are default) of the LDDRUM-model (LD2047).

```
!  What next:                                            EVALUATE
!
!  Evaluate complex events in system:  LD2047
!  From CUTSET or TIESET?  CUTSET
!
!  RESULTING EVALUATED CUTSETS IN LDDRUM
!  ------------------------------------
!       13.  CUTSETS OF  3.  ORDER
!        2.  CUTSETS OF  4.  ORDER
!  ------------------------------------
!       15.  CUTSETS IN TOTAL
!
```

The minimal cutsets or tiesets can be converted into a modularised fault tree. By alternating between cutset and tieset calculations on a tree, the tree can be reduced to its smallest form. The command TREE works default on cutsets.

```
!
!  What next:                                            TREE
!
!  Make a fault-tree from CUTSET or TIESET - [CUTSET]:  CUTSET
!  CUTSET result of:  LD2047
!  Grouped, Evaluated or Not (G/E/N) [N]?  N
!  CONVERTING CUTSETS OF LDDRUM INTO A PRUNED TREE
!  PRUNED TREE MADE
!
```

4. HOW TO CREATE OR UPDATE A LIBRARY.

A library useable for the RIKKE system contains both a graphic and a generic library part.

The basic elements in a graphic library are component forms identified by the generic type of the component as used elsewhere in the RIKKE system. Each generic component type may exist in several graphic forms. The actual form is identified by the name (number) of this form.

The graphic libraries uses the extension #.DGL, where the generic libraries uses the name #.GCL. A full list of the available graphic libraries is therefore obtained in VAX or PDP-11 monitor by asking for these extensions:

```
!
!                                                   DIR *.DGL
! 12-sept-84
! LOGIC .DGL    50  07-Feb-82
! FLOW  .DGL    50  04-Nov-81
! DEMO  .DGL    60  22-Dec-81
! HAZLB2.DGL    82  27-Jan-84
!
```

This example shows 4 available graphic libraries named LOGIC, DEMO, FLOW and HAZLB2.

The extension *.DGL is an abreviation for Draft Graphic Library, and the extension *.GCL is an abreviation for Generic Component Library. A list of all extensions used can be found in appendix A.

By typing DIR *.GCL (Generic Component Library) the computer will show all available generic libraries and it will be possible to see whether there is both a graphic and a generic library.

The graphic libraries are maintained by the command: GRAPHIC (programs GRALIB, GLEDIT and GLPLOT). The use of these programs (command: GRAPHIC) is described in section 4.1.

One or more component forms may be extracted from a library or may be created or modified interactively using the GLEDIT program, and later used to update the same or another library. The extract has the file extension *.GML. A completely new library may be created using these extracted forms.

It is possible to draw a set of (or all) graphic forms in a library (command: GRAPHIC, subcommand: PLOT).

The description of handling the graphic and generic files are split into two. Section 4.1 (with subsections) describe the creation and handling of the graphic library, while section 4.2 (with subsections) take care of the generic library.

The existance of both a graphic and generic library does not ensure compatibility. This phenomenon is described in section 4.3.

4.1 How to create a graphic component.

The graphic library is called from the RIKKE monitor by the command GRAPHIC as seen in the following example.

```
!
!  What next:                                          GRAPHIC
!  RIKKE
!  Graphic Librarian
!  Graphic Library name:                               DEMO
!  What now:
!
```

GRAPHIC is now ready for subcommands. The operator may at any step enter a carriage return to force GRAPHIC to print a list of all possible commands at any step.

The legal answers to the "What now:" query is shown in table below:

Table 4.1 Subcommands in GRAPHIC.

Create	- Create a new graphic library from graphic forms.
Update	- Update a graphic library by replacing forms or adding new.
Make	- Make new graphic form (calling GLEDIT).
EDit	- Edit graphic forms (calling GLEDIT).
Plot	- Plot graphic forms (calling GLPLOT).
LISt	- List all graphic forms in library.
EXtract	- Extract graphic forms from library.
Delete	- Delete graphic forms from library.
LIBrary	- Define another library name.
Stop	- Stop execution [return to RIKKE].

A command is activated by entering enough letters for a full identification as indicated by the capitals in the commands listed above. The rest of the word is optional (but it must match). E.g. EX or EXTR or EXTRACT all activate the extraction of forms.

In order to CREATE a new library or UPDATE an old one, we must have separate forms either made by GLEDIT (command: MAKE) or EXTRACTed from elsewhere. The PTLIB3 distribution contains a set of forms for that library.
The commands EDIT, PLOT, EXTRACT or DELETE all ask for identification of the individual components by their generic type and graphic form.

The query "Generic type :" may be answered by the actual generic type name (max. 6 characters, letters or digits). When a type name is entered, GRAPHIC will ask "Graphic Form:". Here the name of the form (max. 6 characters) should be entered, or ALL to indicate all forms of this component. The answer ALL to the query "Generic Type:" will select all components in all forms within the library, while the answer ? will scan the library, and for each possible component and form ask for acceptance or rejection of this particular element.

The acceptance query looks like the following example:

```
!
!  Type:                                                PUMP
!  Form:                                                1
!
```

The response to this question should be Y or YES for accept, N or NO for rejection or S (STOP) for rejection of this and all remaining component forms. After extraction, the name of the file containing the extract is shown on the terminal.

As an example of creating a new graphic form, we will follow the creation of a tank step by step. From the initial sketch (figure 4.1) we can see that we need to make lines, arcs and circles to fullfill the graphic form. In addition we have to specify the ports of the component.

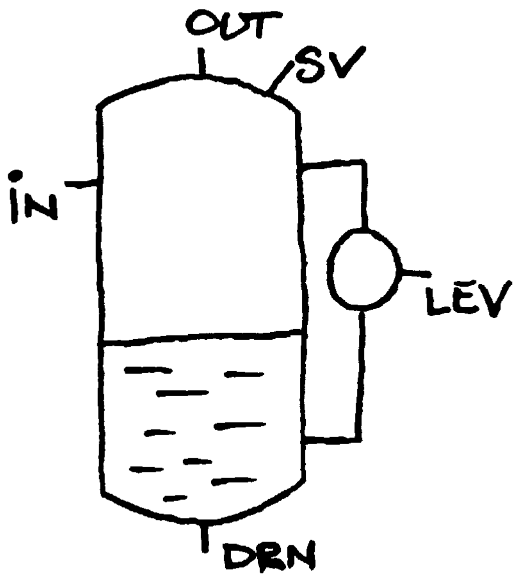


Figure 4.1 Initial sketch of a tank.

The drafting is initiated by the command MAKE. We are prompted for the name of the graphic library file and then get a drawing table on the screen. By the command ADD we get a gleaming sight on the board and can start to draw.

```
!
!  What now:                                            MAKE
!
!                     R I K K E
!  Graphic component editor [V1C]
!  What now:  MAKE
!  Graphic Library-file:                                TANK
!
```

First we point out the center of the picture and mark it with a C (for center). All the possible markers can be shown on the screen by typing a question mark.

The lines are made by positioning the cursor at the first end of the and mark the point by P (for point out), positioning it at the other end and type an L (for line).

The arcs at the end and top of the tank are made by pointing out one of the ends of the arc and type P. Then pointing out a point on the arc, type space, and positioning at the other end of the arc and type A (for arc).

The circle are made by positioning the cursor on the periphery of the circle and type a space. The cursor are moved to the center of the circle, and we type an O.

When the drawing is finished we need to add ports. We position the cursor, where the first port should be and type a number according to the orientation of the as you can see it in figure 4.2. We are then prompted for the name of the port.

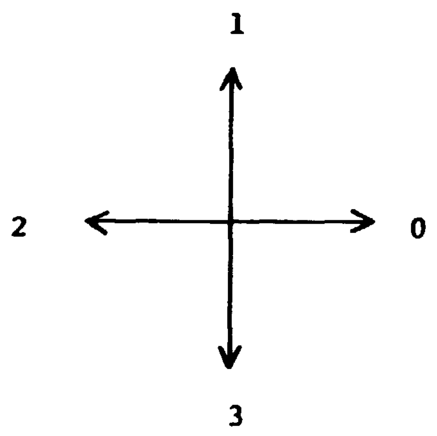


Figure 4.2 Orientation of the ports.

When the drawing session is finished we type X to exit from the drawing table. We save the graphic form by the command SAVE and are prompted for the type and form of the component.

```
!
!                                                       SAVE
!  Type:                                                TANK
!  Form:                                                1
!
```

The position of the component name on the draft is pointed out and confirmed by typing E. When the saving is finished we get a new drawing table, but the drawing can be terminated by typing X and END.

4.1.1 How to edit a graphic component.

```
!
!                                                    EDIT
!  Generic type:                                     TFTANK
!  Graphic form:                                     1
!
```

And we see the existing graphic image of the component TFTANK on the screen. We want to add some new ports to an existing graphic form. First we add a level sensor. The step size on the component drawing is too big, and we want to make it smaller. The standard step size is 10 and we change it to 2 by typing:

```
!
!                                                    STEP
!  Grid/step size [10]:                              2
!
```

We then redraw the component with the smaller step size by typing:

```
!
!                                                    SHOW
!
```

To add the level sensor we type:

```
!
!                                                    ADD
!
```

and we get a sight on the screen. The gate from the tank to the level sensor is marked by typing P. The length of the gate is determined and the line is drawn by typing L. The level sensor itself is drawn in a similar way. The port from the level sensor is defined by the sight and the desired orientation of the port is chosen by typing 0, 1, 2 or 3 (according to the direction shown in figure 4.2).

Table 4.2 Sub-subcommands in Graphic Editor.

Add	- Enter interactive graphic vector mode.
CEnter	- Change center of figure.
ENd	- Finish.
ERase	- Erase area.
EXit	- Finish - present figure not added.
FOrm	- Change name of form.
Grid	- Specify grid (& step).
NAme	- Change position of component name (type).
OK	- Accept this figure - take next.
Quit	- Drop all.
READ	- Load next from input.
REDraw	- Repeat the figure as it will be stored.
REMove	- Remove a port.
RESt	- Accept this & rest of input.
REWind	- Rewind input for repeated entrance.
SAve	- Accept this figure - without taking another.
SCale	- Change sealing of figure.
SHow	- [=REDraw].
SKip	- Drop this figure - take next.
STep	- Specify steps for addressable points.
TYpe	- Change figure type.

The program answer:

```
!
!  Portname:                                                  LEV
!
```

We have now made the desired change in the graphic component and want to exit from the adding system. We type

```
!
!                                                             X
!  :                                                          OK
!                                                             END
!         RIKKE
!  Graphic Component Editor
!  What now:                                                  UPDATE
!         RIKKE
!  Graphic Librarian
!  Graphic Library Name:                                      PTLIB3
!  What now:                                                  UPDATE
!  From graphic file:                                         PTTANK
!  Reading type:  "PTTANK" - Form:  "1"
!  What now:                                                  STOP
!
```

4.1.2 How to include a graphic component.

In order to make a new graphic library or update an existing with graphic components from other libraries, you use the facilities EXTRACT and UPDATE. The first step is to enter the graphic library from which you want to extract the graphic component, and then use the command EXTRACT:

```
!
!  What next:                                              GRAPHIC
!
!   - R I K K E -
!  Graphic Library Editor [V1A]
!
!  Library name:                                           PTLIB3
!
!
!  What now:                                               EXTRACT
!
!  Extract Component type:                                 COLUMN
!  Extract Graphic form:                                   1
!
!  EXTRACTING:  COLUMN
!  Component extracted [in COLUMN.GML]
!  Extract Component type:                                 <CR>
!
```

You have now extracted the graphic component COLUMN from PTLIB3, and the informations are stored in a file named COLUMN.GML. The next step in including the component to the new library CHELIB is to change library and then use the command UPDATE.

```
!
!  What now:                                               LIB
!
!  Library name (PTLIB3):                                  CHELIB
!
!  What now:                                               UPDATE
!
!  Input file type Lib, Gml or New [NEW]:                  GML
!
!  Read from file:                                         COLUMN
!
!  Expanding Database
!  READING TYPE:  COLUMN - - - 1009 -
!          TYPE:  COLUMN - INSERTED @ 58
!  Read from file:                                         <CR>
!
!  What now:                                               STOP
!
```

4.2 How to create a generic component.

In the RIKKE system you are able to make your own components, and just as well as the program needs a graphic model of the components, it needs a generic part, which tells what happens when the conditions are changed.

The generic part consist of a definition of the ports of the component, several small fault trees, a list of spontaneous events and possible working states.

All the attributes in the generic component is listed in table 4.4.

To make a new component you need to define the ports and the transfer functions. The variable list is generated automatically, when you use new variables in the transfer function, and it serves as a control list. The other attributes are used when nessesary.

You call the generic library editor with the command EDIT:

```
!
!  What next:                                   EDIT
!         -RIKKE-
!  Gereric Library Editor
!
!  Library name:                                CHELIB
!
```

The subcommands are shown in table 4.3.

Table 4.3 Subcommands in EDIT of generic library.

```
  EDitor  - Call the interactive editor.
  LISt    - List content of library.
  PRint   - Print component(s) formatted.
  Type    - Type component(s) directly on console.
  EXTract - Extract one or more components from library.
  Update  - Update a library by
               replacing components or adding new.
  INSert  - Add new components to
               library unless already existing.
  REPlace - Replace old components in library by a new one.
  PAck    - Extract in packed form.
  DELete  - Delete components from library.
  CHange  - Change types of components in a library.
  COPy    - Copy one component changing its generic type.
  WHAt    - Tell about editors work-copy and free space.
  ROom    - Tell how much free space in database.
  CLaim   - Claim extra workspace in database.
  LIBrary - Specify library.
  CReate  - Create a new library from source.
  INItial - Create a new (empty) library.
  Stop    - Stop execution - back to monitor.
```

We now want to make an entirely new component. We use the subcommand EDIT, which allows us to make the new components interactively:

```
!
!                                                       EDIT
!
!  (EDITOR) Make, Get, Copy, REStore,
!               List, What or Exit:                     MAKE
!  Make initial work copy af new generic type:          VALVE
!  Initial (empty) work copy made -
!          ready to MOdify.
!
!  Editor is working on :  VALVE in block:  25
!
!  (EDITOR) Make, Get, Copy, REStore, EDit,
!              REMove, List, What or EXit:              EDIT
!  (COMP.) Attribute:
!
```

We have now entered the editor, made a work copy for a component called VALVE, and are ready to specify ports, spontaneous events etc. All the possible attributes to a component are shown by typing <CR>:

Table 4.4 Legal attributes of generic models.

VL	Variable list
PL	Port list
TF	Transfer functions (Mini-fault-trees)
NS	Normal states
IS	Initial states
WS	Working states
PS	Possible states
SE	Spontaneous events
LF	Latent failures

We start to define the ports of the VALVE by typing PL. The ports of the VALVE are named IN, OUT and POS:

```
!
!  (COMP.) Attribute:                            PL
!
!  (Attr:  PL)-Add-What-End:                     ADD
!  -
!  Port:                                         IN (IN)
!
!  -
!  Port;                                         OUT (OUT)
!
!  -
!  Port:                                         POS (POS)
!
!  -
!  Port:                                         <CR>
!
!  (Attr:  PL)-Add-Mod-Print-Last-What-End:      PRINT
!     1:  (IN      (IN     ))
!     2:  (OUT     (OUT    ))
!     3:  (POS     (POS    ))
!
!  (Attr:  PL)-Add-Mod-Print-Last-What-End:
!
```

In the paranthesis we have written the same names as the port names. But if we want to give some of the ports other variable names in the generic system and still have the graphic name saved to fit with the graphic component, we write the original name first and the variable name in the paranthesis.

To return to the editor we write END and EDIT, and we are then ready to create the transfer functions of the VALVE:

```
!
!                                                END
!
!  (EDITOR) SAve, SWap, EDit, REMove,
!               List, What or EXit:              EDIT
!  (COMP.) Attribute:                            TF
!
!  (Attr:  TF)-Add-What-End:                     ADD
!
!  - Transfer Function - Cause -
!  Event:                                     IN -> HIGHPRES
!  Condition -
!  State:                                        POS IS OPEN
!  State:                                        <CR>
!  Delay:                                        0
!  Effect -
!  Event:                                    OUT -> HIGHPRES
!  Event:                                        <CR>
!
!  - Transfer Function - Cause -
!  Event:
!
```

As you can see the program first ask for a cause event, then about which conditions must be fulfilled before the effect event happens, and finally about the effect events. If there is no condition you just give a <CR>. The program also ask for a time delay. You can define several condition states and effect events. When you have finished defining all transfer functions you type <CR>.

```
!
!                                                      <CR>
!
! (Attr:  TF)-Add-Mod-Print-
!                Last-What-End:                        PRINT
!    1:  ((IN -> HIGHPRES)((POS IS OPEN ))
!              (O )((OUT -> HIGHPRES )))
! (Attr:  TF)-Add-Mod-Print-
!              Last-What-End:                          END
! (EDITOR) SAve, SWap, EDit, REMove,
!              List, What or EXit:
!
```

When we have finished making the generic model we save the work-copy:

```
!
!
!                                                      SAVE
! Saving  -    17 -
! Done
! (EDITOR) SAve, SWap, EDit, REMove,
!              List, What or EXit:                     EXIT
! What now:                                            STOP
!
```

The generic editor is always working on a work copy separate from the copy of the component found in the library. This means that it is necessary to save a work copy before the new component (or new version) is active in the library. If the editing is interrupted and the work copy is not saved, the editor keeps the work copy.

When a work copy is saved, the former version is stored as backup copy. The backup copy can be recovered by using the command RESTORE in the editor.

4.2.1 How to edit a generic component.

In the RIKKE program you are able to modify an existing component by a similar procedure as the one used in making new components:

```
!
!  What next:                                              EDIT
!        -RIKKE-
!  Generic Library Editor
!
!  Library name:                                           CHELIB
!  What now:                                               EDIT
!
!  (EDITOR) Make, Get, Copy, REStore, List,
!              What or Exit:                               GET
!  Get component type:                                     VALVE
!  Copying Component
!  New Edition:  2
!
!  (EDITOR) Make, Get, Copy, REStore, List,
!              What or Exit:                               EDIT
!  (COMP.) Attribute:                                      PL
!
!  (Attr:  PL)-Add-Mod-Print-Last-What-End:                PRINT
!     1:  (IN      (IN     ))
!     2:  (OUT     (OUT    ))
!     3:  (POS     (POS    ))
!
!  (Attr:  PL)-Add-Mod-Print-Last-What-End:
!
```

We can now ADD ports, and we can MODIFY the existing ports:

```
!
!                                                          MOD
!
!  (Modify:  PL)-DElete-DUplicate-Replace-
!   Change-Print-First-Last-Next-etc.
!  :
!
```

If we want to DELETE a port, we type DELETE and the number of the port:

```
!
!                                                          DELETE 4
!  Do you really want to delete attribute
!   "PL" - element 4
!  ?                                                       YES
!  Done
!
!  (Modify:  PL)-DElete-DUplicate-Replace-
!   Change-Print-First-Last-Next-etc.
!  :
```

Another possibility is to REPLACE a port by a new port by the command REPLACE. The same command is used when you are changing transfer functions:

```
!
!  (EDITOR) Make, Get, Copy, REStore, List,
!   What or Exit:                                      EDIT
!  (COMP.) Attribute:                                  TF
!
!  (Modify:  TF)-DElete-DUplicate-Replace-
!   Change-Print-First-Last-Next-etc.
!   :                                                  PRINT 3
!   3:  ((IN -> LOWTEMP )
!   ((OUT ISNT COMPLOWTEMP )) (0)((OUT -> LOWTEMP )))
!
!  (Modify:  TF)-DElete-DUplicate-Replace-
!   Change-Print-First-Last-Next-etc.
!   :                                                  REPLACE 3
!
!  Modifying element 3
!
!  Replace variable:                                   <CR>
!
!  Replace value:                                      LOWTEMP
!              by:                                     HIGHTEMP
!  Replace value:                                      <CR>
!
!  Replacing 0 variable, and 1 value - ok ?            YES
!  Copying 3 to 1
!
!  (Modify:  TF)-DElete-DUplicate-Replace-
!   Change-Print-First-Last-Next-etc.
!   :
!
```

By using the other commands in the modify system you are able to DELETE ports or transfer functions from the generic model, DUPLICATE whole parts or REPLACE elements of the attributes.

4.2.2 How to include a generic component.

When we make a new library, we may often wish to use old components from other libraries, and just change them or add a few new components. By using the EDIT command we can EXTRACT generic forms from existing libraries and INCLUDE them in new libraries. The first step in this routine is to EXTRACT the generic forms.

```
!
!  What next:                                          EDIT
!        -RIKKE-
!  Generic Library Editor
!
!  Library name:                                       FTLIB3
!  What now:                                           EXTRACT
!  Extract Component type:                             VALVE
!  EXTRACTING:  VALVE
!  Component extracted [in VALVE .CMP]
!  Extract Component type:                             <CR>
!  What now:
!
```

To INSERT the EXTRACTed component type, we change the library to the new home in EDIT and use the command INSERT:

```
!
!                                                      LIBRARY
!
!  Library name [FTLIB3]:                              CHELIB
!  What now:                                           INSERT
!  Input file type Lib, Cmp or New [NEW]:              CMP
!  Read from file:                                     VALVE
!  Expanding Database
!  READING TYPE:  VALVE  -  -  -   797 -
!          TYPE:  VALVE  - INSERTED @   8
!  Read from file:                                     <CR>
!  What now:                                           STOP
!
```

We have now EXTRACTed the component type VALVE and INCLUDEd it in the new library.

Note that it is very important to ensure that the values used for the different variables are compatible with the new library.

4.3 How to check a library.

In the RIKKE system a command named CHECK is found. This command is used for checking a library in respect to compatibility between the graphic and generic forms. The command checks that all ports on the graphic component, the ones we use in drawing plant on screen, is defined in the generic system.

From the list of available libraries we decide which one to use, and start running the RIKKE system:

```
!
!                                                    RIKKE2
!
!
!  Welcome to RIKKE2
!
!  What next:
!
!                                                    CHECK
!
!
!       RIKKE
!  LIBRARY CROSSCHECK [V1A]
!
!  Library name:
```

!.b2;Now we print the chosen library name (without the extension).

```
!
!                                                    FTLIB3
```

!.b2;An example of a test responce is:

```
!
!  Generic type:  DUMMY, has no graphic equivalence
!
!  Generic type:  FLPFLP, has no graphic equivalence
!
!  Graphic type:  AIRBRN, has no generic equivalence
!
!  Port mismatch in component:  CHECKV - form:  1
!  Generic Graphic - ports without match:
!  "F     "    "------"
!  "T     "    "------"
!  "P     "    "------"
!
!  The Libraries are incompatible
!
```

The first message concerns a component called DUMMY. This component has no graphic equivalence. For a normal component this would be a failure, but because DUMMY is a "dummy component" for the generic libraries, there is no need for a graphic equivalence. The purpose of DUMMY is to serve as starting point for new components.

The next three messages is on specific components and indicates that component FLPFLP and AIRBRN are unknown to the generic and the graphic library respectively. Component CHECKV can not be used because the ports does not match between the generic and the graphic part of the library. These components MUST NOT BE USED in any work including this library before the incompatibilities are repaired.

A mismatch between ports in graphic and generic systems would result in INCOMPATIBILITY between libraries. The reason for incompatibility in the example above is that the component CHECKV does not have graphic ports with the names of "F", "T" and "P".

5. COMMANDS IN RIKKE SYSTEM.

The most common RIKKE commands are

MODEL	- define or change model name
WHAT	- ask for current model
STOP	- stop execution of RIKKE session
DRAFT	- activate model drafting
MAKE	- build up a plant model
FAULT	- produce a fault tree
TEXT	- add readable text to fault tree
FTPLOT	- produce a plotting file / fault tree (A4 sheets)
FTSUPER	- produce a plotting file / fault tree on one sheet
PLOT	- send plotting file to actual plotter
VIEW	- send plotting file to graphic display screen
FTSHOW	- plot a fault tree on typewriter
CUT	- prune fault tree of unwanted event types

An information about all of the commands in the main RIKKE system can be obtained by typing HELP, when you are in the RIKKE monitor. At the following pages you have a short description of these commands.

ANALYZE

The command is used to analyse a fault tree.
The syntax of the command is:

```
ANALYZE [ITEM,ELEMENT] <item> [MODEL] <model name>
```

CALL

The CALL command is used to call and execute a module in the RIKKE package with a model name.
The syntax of the command is:

```
CALL [PROGRAM] <program name> [MODEL] <model name>
```

CCPLOT

The general plotter program used by both FTPLOT and CDPLOT.
The syntax of the command is:

```
CCPLOT [MODEL] <model name>
```

CCSUPER_PLOT

The general plotter program used by both FTSUPER_PLOT and CDSUPER_PLOT.
The syntax of the command is:

```
CCSUPER_PLOT [MODEL] <model name>
```

CDCOMBINE

Combines two cause-consequence diagrams.
The syntax of the command is:

```
CDCOMBINE <new name> [MODEL,ROOT] <name of root>
```

CDPLOT

Plot the generated cause-consequence diagram in A4 sheets.
The syntax of the command is:

    CDPLOT [MODEL] <model name>

CDSHOW

Show the generated cause-consequence diagram on the typewriter.
The syntax of the command is:

    CDSHOW [MODEL] <model name>

CDSUPER_PLOT

Produce a plotting file for a cause-consequence diagram on one sheet (not broken in A4 sheets).
The syntax of the command is:

    CDSUPER_PLOT [MODEL] <model name>

CDTEXT

Add readable text to cause-consequence diagrams.
The syntax of the command is:

    CDTEXT [MODEL] <model name>

CHECK

Check the compatibility between the generic and the graphic part of a Library.
The syntax of the command is:

    CHECK [LIBRARY] <library name>

CODE

The syntax of the command is:

```
CODE (WANT[COMMAND,KEYWORD]=ALL:A30,
               ALL[ALL]=KEYWORDS:A30,ON[ON]=TT,
               WHAT=CODE:-,FOR[FOR=FOR]:-)
```

COMBINE

General combination program for both FTCOMBINE and CDCOMBINE.
The syntax of the command is:

    COMBINE <new name> [MODEL,ROOT] <name of root>

CONVERT

Converts a fault tree in RIKKE form to FAUNET form.
The syntax of the command is:

```
CONVERT <item> [MODEL] <model name>
```

legal items are: FT for fault tree
CS for cutsets
EV for evaluated cutsets

CONSEQUENCE

The consequence command is used to generate a cause-consequence diagram.
The syntax of the command is:

```
CONSEQUENCE [COMPONENT] <component name> [EVENT] <event>
               [MODEL] <model name>
```

CUT

The CUT command allows pruning of unwanted event types in the fault trees before plotting. When the CUT command is given the program asks which types of cutting are required. A detailed description is found in section 2.5.
The syntax of the command is:

```
CUT [MODE] <mode number> [MODEL] <model name>
```

DEBUG

Give an axplanaision of the commands. The facility is resetted by a carriage return.
The syntax of the command is:

```
DEBUG
```

DRAFT

Activate model drafting. Further descriptions of the subcommands can be found in chapter 2 and in (Larsen, 1982).
The syntax of the command is:

```
DRAFT <type> [LIBRARY] <library name>
               [MODEL] <model name>
```

Legal types are: OLD for old draftings
NEW for making new drafts.
CONTINUE for working on a draft data base.

EDIT

The EDIT command envokes the program GENLIB and permits editing in the generic models from a given library.
The syntax of the command is:

```
EDIT [LIBRARY] <library name>
```

EXECUTE

The execute command permits execution of DAPHNE command files. A further description is found in section 2.6.
The syntax of the command is:

```
EXECUTE [FILE] <file name> [MODEL] <model name>
```

EXTRACT

Extract forms from a Library in a separate file.
The syntax of the command is:

```
EXTRACT [TYPE] <generic type> [LIBRARY] <library name>
```

FAULT

The command FAULT generates fault trees. The command is appended by the name of the component in which the event happens, and the type of event.
The syntax of the command is:

```
FAULT [COMPONENT] <component name> [EVENT] <event>
                  [OPTION] <option type>
```

FIX

Repair an uncomplete fault tree. Legal types are:

B	for Break
C	for Continue
D	for max. Depth before break
E	for Event list
L	for Loop stop
N	for None (default)
S	for Show (not on VAX)
T	for Time

The syntax of the command is:

```
FIX [MODEL] <model name>
```

FTCOMBINE

The FTCOMBINE command is used to combine two fault trees.
The syntax of the command is:

```
FTCOMBINE [AS] <new name> [MODEL,ROOT] <name of roots>
```

FTEDIT

The command FTEDIT permits editing of a fault tree. You can cut out a piece or find certain events in the tree.
The syntax of the command is:

```
FTEDIT A[AS],W[DO],K[ON],[MODEL] <model name>
```

FTPLOT

Produce a plotting file for a fault tree on A4 sheets.
The syntax of the command is:

```
FTPLOT [MODEL] <model name>
```

FTSHOW

Show the generated fault tree on the typewriter.
The syntax of the command is:

```
FTSHOW [MODEL] <model name>
```

FTSUPER_PLOT

Produce a plotting file for a fault tree on one sheet (not broken in A4 sheets).
The syntax of the command is:

```
FTSUPER_PLOT [MODEL] <model name>
```

FTTEXT

FTTEXT changes the text form of a fault tree from numeric form to text form.
The syntax of the command is:

```
FTTEXT [MODEL,IN] <model name>
```

GRAPHIC

The GRAPHIC command permits graphic editing of graphic libraries.
The syntax of the command is:

```
GRAPHIC [LIBRARY] <library name> [TYPE] <component name>
                  [MODEL] <model name>
```

Legal component names are all components in library and the type ALL which extracts all types.

HELLO

HELLO displays the initial welcome screen.
The syntax of the command is:

```
HELLO
```

HELP

Gives information about which commands you can use, and what syntax they use.
The syntax of the command is:

```
HELP [ABOUT] <name of command>
```

HOPSA

HOPSA is short for Human Operator Safety Analysis and permits an analysis of start up/shut down procedures or other procedures. This program has not been released.
The syntax of the command is:

```
HOPSA
```

LIBRARY

Defines or redefines the name of the Library you want to work with or on.
The syntax of the command is:

```
LIBRARY [LIBRARY] <library name> [TYPE] <generic type>
```

LIST

The LIST command enables you to see one or more files on the screen.
The syntax of the command is:

```
LIST [FILE,FILES] <file name(s)>
```

MAKE

The MAKE command initiates the building of a plant model.
The syntax of the command is:

```
MAKE [MODEL] <model name>
```

MINI_FAULT_TREE_PLOT

Plots the mini fault trees of a component.
The syntax of the command is:

```
MINI_FAULT_TREE_PLOT [LIBRARY] <library name>
                     [COMPONENT,COMPONENTS] <component name(s)>
```

MODEL

Defines or redefines the model you are working with.
The syntax of the command is:

```
MODEL [NAME] <name of model>
```

NUMBER

Renumber the events in a fault tree.

The syntax of the command is:

```
NUMBER [MODEL] <model name>
```

PLOT

Send plotting file to the plotter.
The syntax of the command is:

```
PLOT [MODEL] <model name>
```

PRINT

The PRINT command activates typing of one file or more on a printer.
The syntax of the command is:

```
PRINT [FILE,FILES] <file name(s)>
```

RT11

Returns you to the PDP-11 monitor for one command.

Same command as the the VMS command, but only used on the PDP-11.

RUN

Call a seperate program for execution, return to RIKKE on exit.
The syntax of the command is:

```
RUN [PROGRAM] <program name>
```

STOP

The STOP command stops the execution of the RIKKE program.

string handling

The string manipulating commands may be used in connection with more generel command files. They were primarily developed in connection with the DAPHNE code facility (not released with RIKKE-II).

APPEND

The function APPEND is a string manipulator which appends the argument of the function to the storage called RESULT. A space will be imbedded.
The syntax of the command is:

```
FUNCTION APPEND [NEW]
```

CONCAT

Store an argument in the RESULT buffer. If CONCAT have two arguments they will be combined.
The syntax of the command is:

```
FUNCTION CONCAT [WITH]
```

FIRST_PART

Scan the input argument (default RESULT) for the first space and the first "half" is stored as new result.
The syntax of the command is:

    FUNCTION FIRST_PART [OF]

PUSH

Save the argument in the result buffer (POP).
The syntax of the command is:

    PUSH

PROMPT

Accept one argument. The value is used as prompt for a query on the screen. An answer is expected from the keyboard. The answer is stored in the result buffer.
The syntax of the command is:

    FUNCTION PROMPT

QUIET

Suppress "unnescessary" output prompts, where the value is already supplied, for a limited number of steps.
The syntax of the command is:

    QUIET <integer>

REST

Act like FIRSTPART except that the final result is the second "half" of the text.
The syntax of the command is:

    FUNCTION REST

WRITE

Type the argument text on the console.
The syntax of the command is:

    WRITE

SUPER_PLOT

Plots the model in one large drawing.
The syntax of the command is:

    SUPER_PLOT [MODEL] <model name>

SYNTAX

Gives the information about the syntax of a given command.
The syntax of the command is:

SYNTAX [FOR] <name of command>

TEXT

Transform fault tree text from numeric to readable form, and add it to the fault tree or cause-consequence diagram.
The syntax of the command is:

TEXT [MODEL] <model name>

TYPE

Types a file on the screen.
The syntax of the command is:

TYPE [FILE,FILES] <file name(s)>

UPDATE

Update a Library by replacing forms or adding new.
The syntax of the command is:

UPDATE [LIBRARY] <library name> [TYPE] <generic type>

VIEW

Send plotting file to graphic display screen.
The syntax of the command is:

VIEW [MODEL] <model name>

WHAT

Ask for the name and information about the current model.

VMS

Permits one command to be executed in the monitor on the VAX computer (VMS). When this command has been executed you are returned to the RIKKE session in hand.

6. HOW TO GET HELP.

In the RIKKE Monitor the command HELP produces the following information:

```
!
!  The most common RIKKE commands are
!  ----------------------------------------
!  MODEL   - define or change model name
!  WHAT    - ask for current model
!  STOP    - stop execution of RIKKE session
!  ----------------------------------------
!  DRAFT   - activate model drafting
!  MAKE    - build up a plant model
!  FAULT   - produce a fault tree
!  TEXT    - add readable text to fault tree
!  FTPLOT  - produce a plotting file / fault tree (A4 sheets)
!  FTSUPER - produce a plotting file / fault tree on one sheet
!  PLOT    - send plotting file to actual plotter
!  VIEW    - send plotting file to graphic display screen
!  FTSHOW  - plot a fault tree on typewriter
!  CUT     - prune fault tree of unwanted event types
!  DIAGRAM - create or modify Block Diagram
!
!  Information available:
!
!     ANALYZE   CALL      CDPLOT    CCSUPER_PLOT
!     CDCOMBINE CDPLOT    CDSHOW    CDSUPER_PLOT
!     CDTEXT    CHECK     CODE      COMBINE   CONVERT
!     CONSEQUENCE         CUT       DRAFT     EDIT
!     EXECUTE   EXTRACT   FAULT     FIX       FTCOMBINE
!     FTEDIT    FTPLOT    FTSHOW    FTSUPER_PLOT
!     FTTEXT    GRAPHIC   HELLO     HELP      HOPSA
!     LIBRARY   LIST      MAKE      MINI_FAULT_TREE_PLOT
!     MODEL     NUMBER    PLOT      PRINT     RUN
!     STOP      string handling     SUPER_PLOT
!     SYNTAX    TEXT      TYPE      UPDATE    VIEW
!     WHAT      VMS
!
!  Topic:
!
```

You can type the command, you wish to know more about, and the HELP facility answer:

```
!
!                                    DRAFT
!
!  Activate model drafting.
!  A complete description of the
!  subcommands can be found in:
!  GRACE USER MANUAL (RISO-M-2343)
!  Call GRACE
!
!  Syntax for the command:
!
!  DRAFT <type> [LIBRARY] <library name>
!   [MODEL] <model name>
!
!  Legal types are:  OLD for old draftings
!                    NEW for making new drafts.
!                    CONTINUE for working on a draft data base.
!
!  Additional information available:
!
!  Parameters Qualifiers
!  /ALL
!  /ALTER
!  /COMPONENT
!  /DRAW
!  /DUPLICATE
!  /ERASE
!  /FIND
!  /GRID
!  /IN
!  /LIBRARY
!  /LINK
!  /MOVE
!  /OUT
!  /QUIT
!  /REDRAW
!  /RELINK
!  /SAVE
!  /SETUP
!  /SHIFT
!  /STOP
!  /TEXT
!  /UNLINK
!  /WINDOW
!
!  DRAFT Subtopic:
!
```

If you want to know about one of the subtopics, you type the name. If you don't, you just press the carriage return, and the HELP system returns to the maingroup of topics. If you don't want to know anything further, you press the carriage return until the RIKKE monitor answers "What next".

In all other parts of the RIKKE system a question mark or a carriage return will produce a list of information about the available commands and their use.

It is the intention that the RIKKE system should be a self teaching system. The program gives prompts indicating when it needs control inputs, and many of these are indicative of the input required. In the case where prompts are uninformative, such as the prompt "What next:", pressing the return key will result in a listing of the possible commands which can be given. When in the RIKKE monitor, typing HELP results in a listing of the available commands for users to learn to use the RIKKE system with no help at all but the help provided by the computer itself.

In general, if in doubt, press the carriage return key. This will either take you back to an earlier stage of command input, or will produce some comment intended to help you out of the difficulties.

7. THE LIBRARIES.

Each library consists of a number of components with a generic and a graphic equivalence. In a library we have certain rules for the levels (or values) of the variables and certain specified names of the failure modes of the components. This is to certify that a level of a variable in one component can be recogniced in the other components and that the failure modes are understood.

The libraries FTLIB3 and HAZLB2 do not have the same sets of levels and failure modes. This means that a component in one library do not match the components in the other library. In the following sections (7.1 and 7.2) we will describe the libraries and give an example of a component in each library so the difference may be seen more clearly.

7.1 FTLIB3.

In the library FTLIB3 a range of 63 components was made:

Table 7.1 Components in FTLIB3.

Component:	Used for:	Ports:
ACTUAT	Actuater	pos, in
AIRREG	Airregulator	in, out, air, set
AND	And gate	in1, in2, out
CCONT	Normaly closed contact	in, out, x
CHECKV	Check valve	in, out
COLUMN	Column	c, p, f, ref, in, out
CONDEN	Condenser	drn, in1, in2, out1, out2
CTANK	Tank	lev, of, in, out
CVALVE	Checkvalve	in, out
DELAY	Time delay	in, out
DIV	Divider	in, out1, cut2
DIVVLV	Divider valve	pos, in, out1, out2
DRAIN	Drain	in
DWNCMR	Revers riser	in, out
EVAP	Evaporator	py, ty, lev, heat, in, out
EVAPD	Evaporator	py, ty, lev, heat, drn, in, out
FLPFLP	Flip flop	s, r, q, nq
FORGAC		pos, in
FURN	Furnace	air, monito, pilot, in1, in2, out1, out2
HEATER	Heater	heat, in, out
HEX	Heat exchanger	in1, in2, out1, out2
HW		pos
INVERT	Inverter	in, out
KODRUM	Knockout drum	press, sv, lev, drn, in, out
LEVSNS	Level sensor	lev, out
LIQFRN	Liquid furner	in1, in2, in3, in4, in5, in6, in7, in8, in, out
LOAD	Load	in, out
MANU		
MIX	Mixer	in1, in2, out
MIXVLV	Mixer valve	pos, in1, in2, out
NOT	Not gate	in, out
NOZZLE	Nozzle	in, out
NREAC		p, in
OCONT	Normaly open contact	in, out, x
OFTANK	Over flow tank	lev, dr, of, in, out
OR	Or gate	in1, in2, out
PIPE	Pipe	c, p, f, t, in, out
PSH	Pressure sensor high	in, out
PSL	Pressure sensor low	in, out
PSN		in, out
PUMP	Pump	pwr, in, out
PUSHER	Push contact	pos, in, out
PV	Pressure vessel	p, sv, in, out
PWRSUP	Power supply	out
REG	Regulator	in, out

REGVLV	Regulation valve	pos, in, out
RISER	Riser	in, out
SBYPMP	Standby pump	pwr, in, out
SEPARA	Separator	press, sv, lev, drn, in, out
SH	Sensor high	in, out
SIGDIV	Signal divider	in, out1, out2
SL	Sensor low	in, out
SPLIT	Splits flow into two	in, out1, out2
STRAP		in, out, drn
SUP	Supply	out
SUPTNK	Supply tank	lev, of, dr, in, out
SV	Safety valve	in, out
TANK	Tank	lev, in, out
TFTANK	Transfer tank	lev, dr, of, in, out
TRANSA	Transformer	in, out
TURBIN	Turbine	pwr, in, out
VALVE	Valve	pos, in, out
XLI		in, out

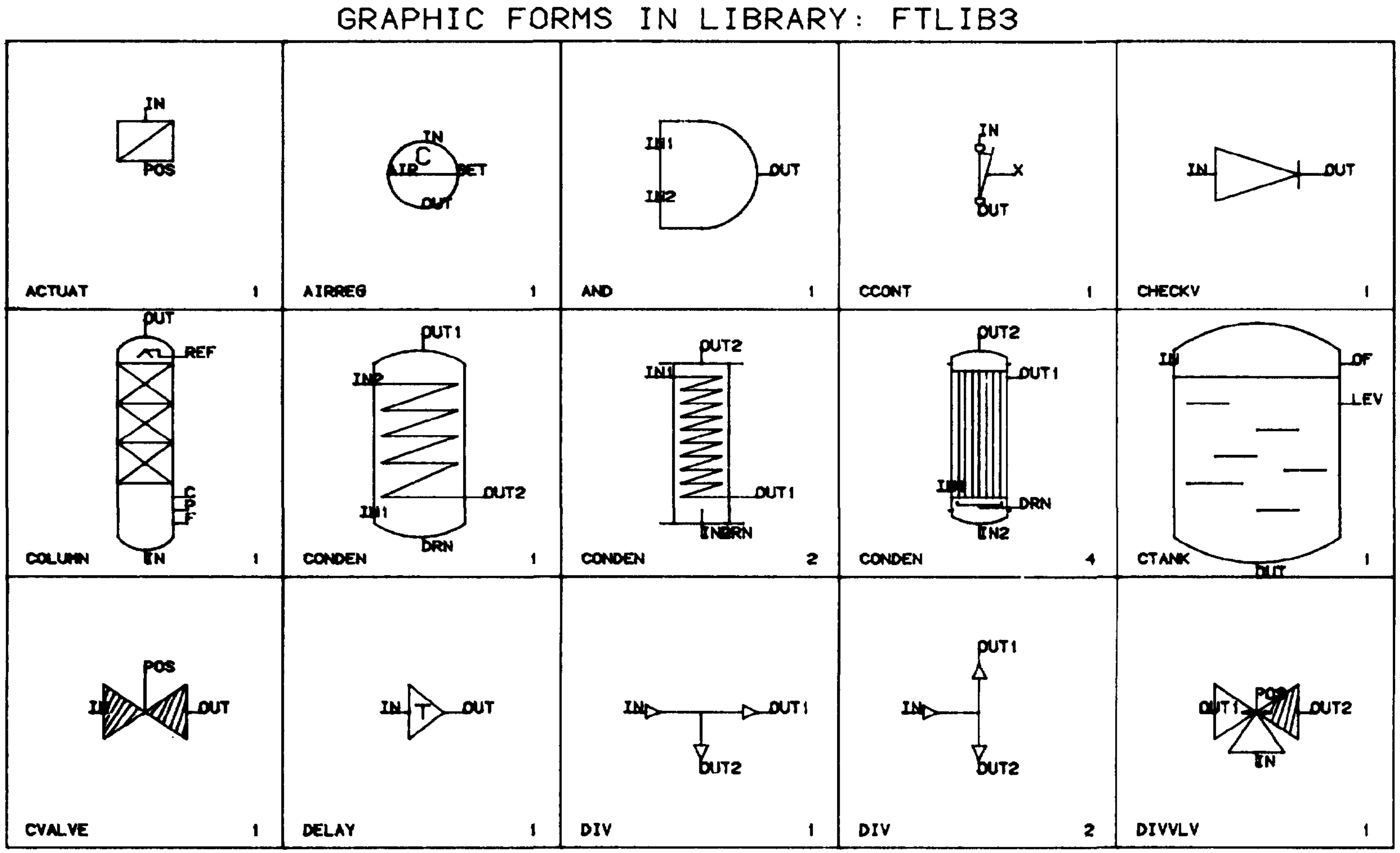


Figure 7.1 Graphic components in FTLIB3.

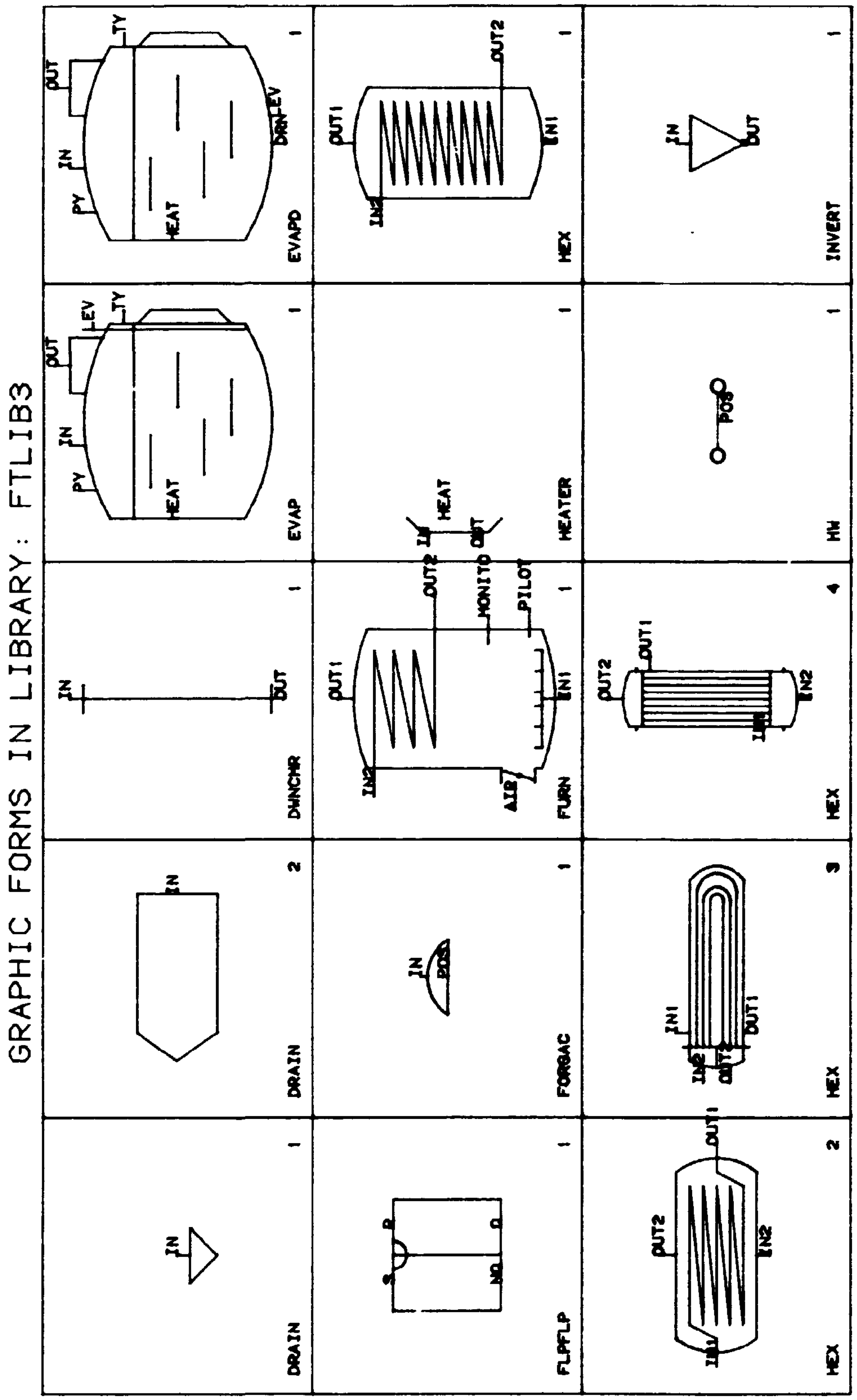


Figure 7.1 Graphic components in FTLIB3 continued.

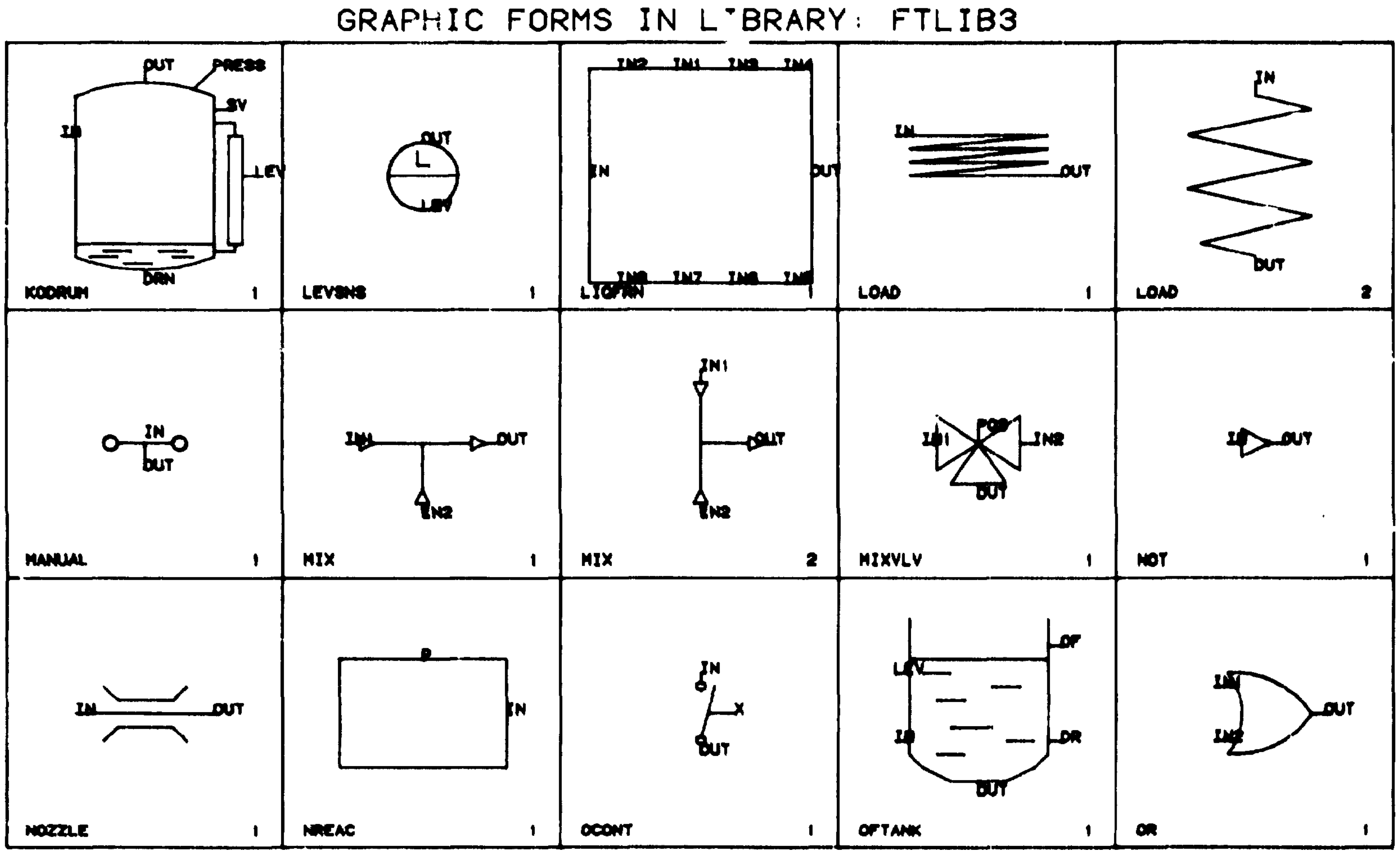


Figure 7.1 Graphic components in FTLIB3 continued.

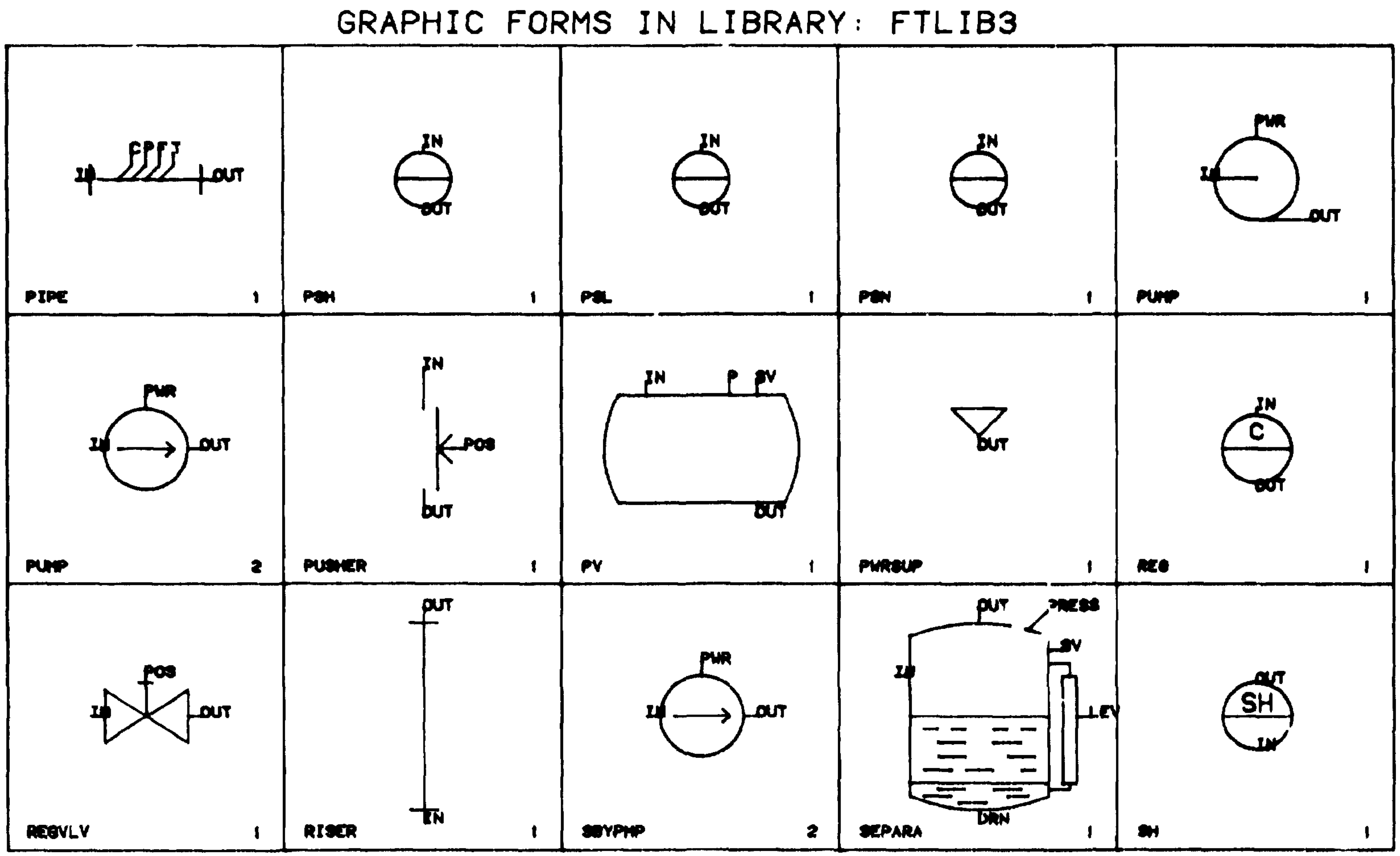


Figure 7.1 Graphic components in FTLIB3 continued.

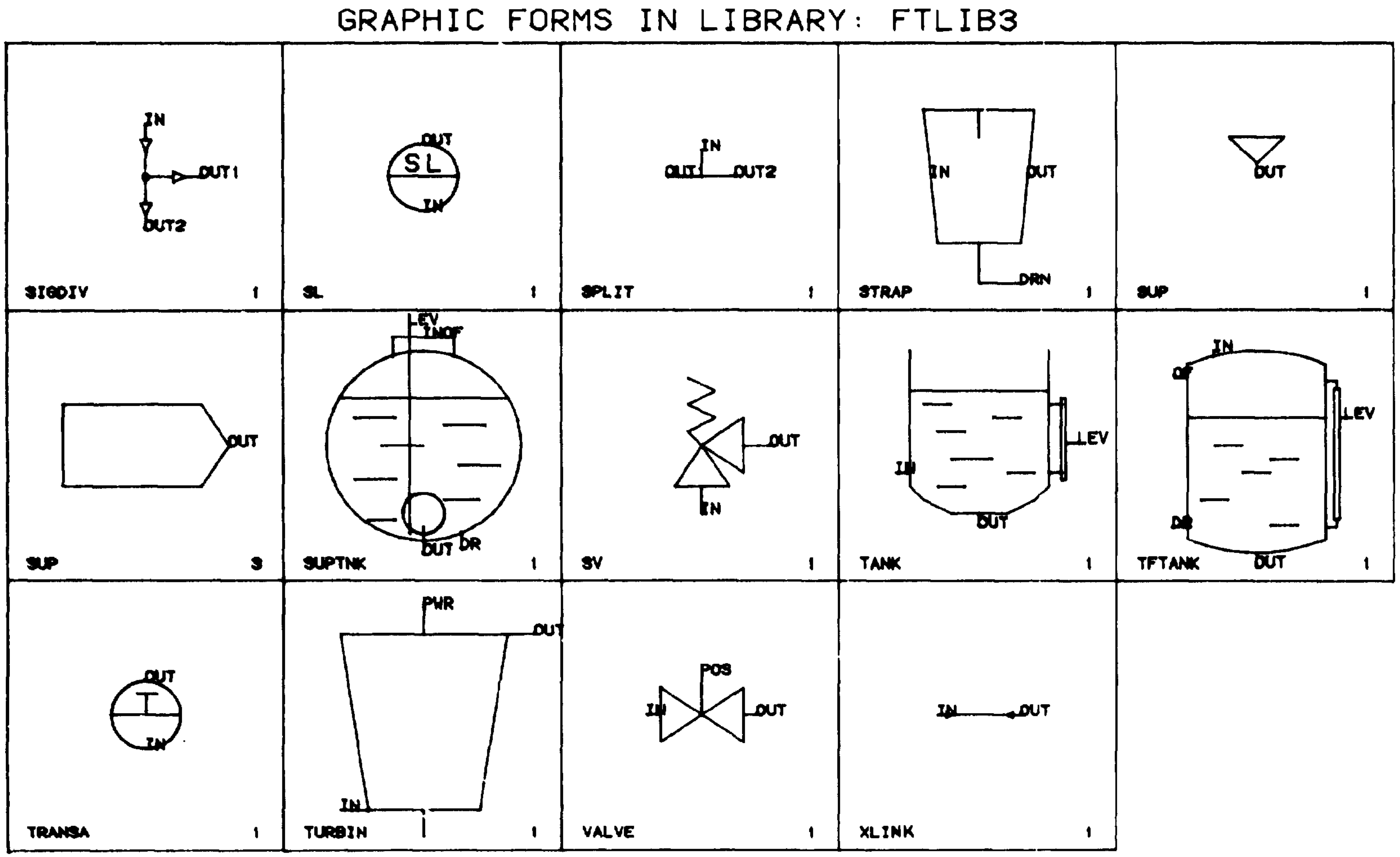


Figure 7.1 Graphic components in FTLIB3 continued.

The discretisation levels for disturbances used in the FTLIB3 construction process are based on the following:

Table 7.2 Discrete levels in FTLIB3.

VHI	Very high - so high that no compensation is possible, e.g., VHIP = very high pressure.
HI	So high that the disturbance can only be compensated by shutdown.
DISTHI	High enough to cause an accident, not so high that a compensation is impossible.
DISTLO LO	Defined analogously.
ZERO	Disturbances resulti ıg in valves indistinguishable from zero.
REV	Reversal of flow.

Corresponding failure modes that can be distinguished in flow system is:

Table 7.3 Failure modes in FTLIB3.

BLOCKED	causing zero flow.
BURST	causing zero pressure.
LEAK	causing low.
SMALL LEAK	causing DISTLO pressure.
PARTIALLY	
SMALL BLOCKAGE	
LOW RESISTANCE	
SLIGHTLY LOW RESISTANCE	
NO RESISTANCE	

7.1.1 Example of a component in FTLIB3.

```
 ========================

  RIKKE - Library: FTLIB3

 ------------------------

Generic Component: REGVLV
  19-Sep-84 11:07:25

Attribute: VL - Variable List

(IN FV )
(REG FV )
(OUT FV )
(WS FV )
(STA FV )
(IIN FV )
(IOUT FV )
(F FV )
(POS FV )
(VALVE FV )

Attribute: PL - Port List

(POS (POS))
(OUT (OUT))
(* )
(IN (IN))

Attribute: TF - Transfer Functions (Mini-fault-trees)

((IN -> HSPR)((POS IS OPEN))(0)((OUT -> HSPR)(OUT -> HSPC)))
((OUT -> HBPR)((POS IS OPEN))(0)((IN -> HBPR)(IN -> HBPC)))
((IN -> HSPR) TRUE (0)((IIN -> HP)))
((IN -> HSPR)((POS IS OPEN)(OUT IS R))(0)((IOUT -> HP)))
((OUT -> HBPR) TRUE (0)((IOUT -> HP)))
((OUT -> HBPR)((POS IS OPEN)(IN IS R))(0)((IIN -> HP)))
((IN -> HSPR)((OUT ISNT BLOCKED)(POS ISNT FAILCLOSED))(0)
  ((F -> HF)))
((OUT -> HBPR) TRUE (0)((F -> LF)))
((IN -> COMPLOSUPPR)((POS IS OPEN))(0)((OUT -> COMPLOSUPPR)
  (OUT -> COMPLOSUPPC)))
((OUT -> COMPLOBACKPR)((POS IS OPEN))(0)((IN -> COMPLOBACKPR)
  (IN -> COMPLOBACKPC)))
((IN -> COMPLOSUPPR) TRUE (0)((IIN -> COMPLOP)))
((IN -> COMPLOSUPPR)((POS IS OPEN)(OUT IS R))(0)
  ((IOUT -> COMPLOP)))
((OUT -> COMPLOBACKPR) TRUE (0)((IOUT -> COMPLOP)))
((OUT -> COMPLOBACKPR)((POS IS OPEN)(IN IS R))(0)
  ((IIN -> COMPLOP)))
((OUT -> COMPLOBACKPR) TRUE (0)((IOUT -> COMPHIFLO)))
((IN -> COMPHISUPPR)((POS IS OPEN))(0)((OUT -> COMPHISUPPR)
  (OUT -> COMPHISUPPC)))
((OUT -> COMPHIBACKPR)((POS IS OPEN))(0)((IN -> COMPHIBACKPR)
  (IN -> COMPHIBACKPC)))
((IN -> COMPHISUPPR) TRUE (0)((IIN -> COMPHIP)))
((IN -> COMPHISUPPR)((POS IS OPEN)(OUT IS R))(0)
  ((IOUT -> COMPHIP)))
((OUT -> COMPHIBACKPR) TRUE (0)((IOUT -> COMPHIP)))
```

```
((OUT -> COMPHIBACKPR)((POS IS OPEN)(IN IS R))(0)
  ((IIN -> COMPHIP)))
((OUT -> COMPHIBACKPR) TRUE (0)((IOUT -> COMPLOFLO)))
((IN -> DISTHISUPPR)((POS ISNT COMPLO))(0)((OUT -> DISTHISUPPR)
  (OUT -> DISTHISUPPC)))
((OUT -> DISTHIBACKPR)((POS ISNT COMPHI))(0)((IN -> DISTHIBACKPR)
  (IN -> DISTHIBACKPC)))
((IN -> DISTHISUPPR) TRUE (0)((IIN -> DISTHIP)))
((IN -> DISTHISUPPR)((POS ISNT COMPLO)(OUT IS R)
  (OUT ISNT COMPLOBACKPR))(0)((IOUT -> DISTHIP)))
((OUT -> DISTHIBACKPR) TRUE (0)((IOUT -> DISTHIP)))
((OUT -> DISTHIBACKPR)((POS ISNT COMPHI)(IN IS R))(0)
  ((IIN -> DISTHIP)))
((IN -> DISTHISUPPR)((OUT ISNT SHUTOFF)(POS ISNT COMPLO)
  (OUT ISNT COMPHIBACKPR))(0)((IOUT -> DISTHIFLO)))
((OUT -> DISTHIBACKPR)((POS ISNT COMPHI)(IN ISNT COMPHISUPPR))(0)
  ((IOUT -> DISTLOFLO)))
((IN -> DISTLOSUPPR)((POS ISNT COMPHI))(0)((OUT -> DISTLOSUPPR)
  (OUT -> DISTLOSUPPC)))
((OUT -> DISTLOBACKPR)((POS ISNT COMPLO))(0)((IN -> DISTLOBACKPR)
  (IN -> DISTLOBACKPC)))
((IN -> DISTLOSUPPR) TRUE (0)((IIN -> DISTLOP)))
((IN -> DISTLOSUPPR)((POS ISNT COMPHI)(OUT IS R)
  (OUT ISNT COMPHIBACKPR))(0)((IOUT -> DISTLOP)))
((OUT -> DISTLOBACKPR) TRUE (0)((IOUT -> DISTLOP)))
((OUT -> DISTLOBACKPR)((POS ISNT COMPLO)(IN IS R)
  (IN ISNT COMPHISUPPR))(0)((IIN -> DISTLOP)))
((IN -> DISTLOSUPPR)((OUT ISNT COMPLOBACKPR)(POS ISNT COMPHI))(0)
  ((IOUT -> DISTLOFLO)))
((OUT -> DISTLOBACKPR)((POS ISNT DISTLO)(IN ISNT COMPLOSUPPR))(0)
  ((IOUT -> DISTHIFLO)))
((IN -> LOSUPPR) TRUE (0)((OUT -> LOSUPPR)(OUT -> LOSUPPC)))
((OUT -> LOBACKPR)((POS IS OPEN))(0)((IN -> LOBACKPR)
  (IN -> LOBACKPC)))
((IN -> LOSUPPR) TRUE (0)((IIN -> LOP)))
((IN -> LOSUPPR)((OUT IS R))(0)((IOUT -> LOP)))
((OUT -> LOBACKPR) TRUE (0)((IOUT -> LOP)))
((OUT -> LOBACKPR)((POS IS OPEN)(IN IS R))(0)((IIN -> LOP)))
((IN -> LOSUPPR) TRUE (0)((IOUT -> LOFLO)))
((OUT -> LOBACKPR)((IN ISNT SHUTOFF)(POS ISNT CLOSED))(0)
  ((IOUT -> HIFLO)))
((IN -> HISUPPR)((POS IS OPEN))(0)((OUT -> HISUPPR)
  (OUT -> HISUPPC)))
((OUT -> HIBACKPR)((POS IS OPEN))(0)((IN -> HIBACKPR)
  (IN -> HIBACKPC)))
((WS -> R) TRUE (0)((IN -> R)(OUT -> R)))
((WS -> BURST) TRUE (0)((IN -> C)(OUT -> C)))
((IN -> HISUPPR) TRUE (0)((IIN -> HIP)))
((IN -> HISUPPR)((POS IS OPEN)(OUT IS R))(0)((IOUT -> HIP)))
((OUT -> HIBACKPR) TRUE (0)((IOUT -> HIP)))
((OUT -> HIBACKPR)((POS IS OPEN)(IN IS R))(0)((IIN -> HIP)))
((IN -> HISUPPR)((OUT ISNT SHUTOFF)(POS IS OPEN))(0)
  ((IOUT -> HIFLO)))
((OUT -> HIBACKPR) TRUE (0)((IOUT -> LOFLO)))
((IN -> NOSUPP) TRUE (0)((IIN -> X)))
((IIN -> X)((OUT IS NOBACKPR))(0)((IOUT -> NOP)))
((OUT -> NOBACKP) TRUE (0)((IOUT -> X)))
((WS -> BLOCKED) TRUE (0)((IIN -> X)(IOUT -> X)))
((IN -> NOSUPP)((IOUT IS X))(0)((IIN -> NOP)))
((IN -> ATM) TRUE (0)((IIN -> NOP)))
((OUT -> ATM) TRUE (0)((IOUT -> NOP)))
((WS -> BLOCKED) TRUE (0)((IN -> NOBACKP)(IN -> NOBACKPR)
```

```
  (OUT -> NOSUPP)(OUT -> NOSUPPR)))
((POS -> FAILCLOSED) TRUE (0)((IN -> NOBACKP)(IN -> NOBACKPR)
  (OUT -> NOSUPP)(OUT -> NOSUPPR)))
((WS -> BLOCKED) TRUE (0)((IIN -> NOFLO)(IOUT -> NOFLO)))
((POS -> FAILCLOSED) TRUE (0)((IIN -> NOFLO)(IOUT -> NOFLO)))
((IN -> NOSUPFLO)((OUT IS NOBACKFLO))(0)((IIN -> NOFLO)
  (IOUT -> NOFLO)))
((IN -> NOSUPP)((OUT IS BACKFLO)(POS IS OPEN))(0)((IIN -> REVFLO)
  (IOUT -> REVFLO)))
((OUT -> BACKFLO)((POS IS OPEN))(0)((IN -> BACKFLO)))
((WS -> BURST) TRUE (0)((IN -> NOBACKPR)(IN -> ATM)(IIN -> NOP)
  (IOUT -> NOP)(OUT -> ATM)(OUT -> NOSUPPR)))
((IN -> NOSUPPR) TRUE (0)((OUT -> NOSUPP)(OUT -> NOSUPPR)))
((OUT -> NOBACKPR) TRUE (0)((IN -> NOBACKPR)(IN -> NOBACKP)))
((WS -> BLOCKED) TRUE (0)((IN -> BLOCKED)(OUT -> BLOCKED)))
((IN -> BLOCKED) TRUE (0)((OUT -> BLOCKED)))
((OUT -> BLOCKED) TRUE (0)((IN -> BLOCKED)))
((OUT -> BACKFLO)((POS IS OPEN))(0)((IN -> BACKFLO)))
((IN -> SUP)((POS IS OPEN))(0)((OUT -> SUP)))
((POS -> FAILCLOSED) TRUE (0)((IN -> NOTATM)(OUT -> NOTATM)))
((IN -> NOTATM) TRUE (0)((OUT -> NOTATM)))
((OUT -> NOTATM) TRUE (0)((IN -> NOTATM)))
((OUT -> NOBACKFLO) TRUE (0)((IN -> NOBACKFLO)))
((IN -> NOSUPFLO) TRUE (0)((OUT -> NOSUPFLO)))
((WS -> BURST) TRUE (0)((OUT -> NOSUPFLO)(IN -> NOBACKFLO)))
((IN -> NOSUPFLOTR) TRUE (0)((OUT -> NOSUPFLOT)
  (OUT -> NOSUPFLOTR)))
((OUT -> NOBACKFLOTR) TRUE (0)((IN -> NOBACKFLOT)
  (IN -> NOBACKFLOTR)))
((WS -> BURST) TRUE (0)((IN -> NOBACKFLOTR)(OUT -> NOSUPFLOTR)))
((POS -> FAILCLOSED) TRUE (0)((IN -> BLOCKED)(OUT -> BLOCKED)))
((IN -> ON)((POS IS OPEN)(VALVE IS NOTBLOCKED)
  (VALVE IS NOTBURST))(0)((OUT -> ON)))
((IN -> OFF) TRUE (0)((OUT -> OFF)))
((IN -> LIQUID)((POS IS OPEN))(0)((OUT -> LIQUID)))
((IN -> GAS)((POS IS OPEN))(0)((OUT -> GAS)))
((IN -> CONTAMINATED)((POS IS OPEN))(0)((OUT -> CONTAMINATED)))
((IN -> SCUM)((POS IS OPEN))(0)((OUT -> SCUM)))
((IN -> HIT)((POS IS OPEN))(0)((OUT -> HIT)))
((IN -> DISTHIT)((POS IS OPEN))(0)((OUT -> DISTHIT)))
((IN -> DISTLOT)((POS IS OPEN))(0)((OUT -> DISTLOT)))
((IN -> LOT)((POS IS OPEN))(0)((OUT -> LOT)))
((IN -> COMPHIT)((POS IS OPEN)(VALVE IS NOTBLOCKED)
  (VALVE IS NOTBURST))(0)((OUT -> COMPHIT)))
((IN -> COMPLOT)((POS IS OPEN)(VALVE IS NOTBLOCKED)
  (VALVE IS NOTBURST))(0)((OUT -> COMPLOT)))
((IN -> HICONC)((POS IS OPEN))(0)((OUT -> HICONC)))
((IN -> DISTHICONC)((POS IS OPEN))(0)((OUT -> DISTHICONC)))
((IN -> DISTLOCONC)((POS IS OPEN))(0)((OUT -> DISTLOCONC)))
((IN -> LOCONC)((POS IS OPEN))(0)((OUT -> LOCONC)))
((IN -> COMPHICONC)((POS IS OPEN)(VALVE IS NOTBURST)
  (VALVE IS NOTBLOCKED))(0)((OUT -> COMPHICONC)))
((IN -> COMPLOCONC)((POS IS OPEN)(VALVE IS NOTBLOCKED)
  (VALVE IS NOTBURST))(0)((OUT -> COMPLOCONC)))
((IN -> SUBST1PRESENT)((POS IS OPEN))(0)((OUT -> SUBST1PRESENT)))
((IN -> SUBST1HI)((POS IS OPEN))(0)((OUT -> SUBST1HI)))
((IN -> SUBST1LO)((POS IS OPEN))(0)((OUT -> SUBST1LO)))
((IN -> SUBST2HI)((POS IS OPEN))(0)((OUT -> SUBST2HI)))
((IN -> SUBST2PRESENT)((POS IS OPEN))(0)((OUT -> SUBST2PRESENT)))
((IN -> SUBST2LO)((POS IS OPEN))(0)((OUT -> SUBST2LO)))
((IOUT -> REVFLO)((OUT IS HOT))(0)((IIN -> HIT)(IN -> HOT)))
((IOUT -> REVFLO)((OUT IS COLD))(0)((IIN -> LOT)(IN -> COLD)))
```

```
((IOUT -> REVFLO)((OUT IS SUBST1PRESENT))(0)((IN -> SUBST1PRESENT)
  (IIN -> SUBST1PRESENT)))
((IOUT -> REVFLO)((OUT IS SUBST2PRESENT))(0)((IN -> SUBST2PRESENT)
  (IIN -> SUBST2PRESENT)))
((IOUT -> REVFLO)((OUT IS LIQUID))(0)((IN -> LIQUID)
  (IIN -> LIQUID)))
((IOUT -> REVFLO)((OUT IS GAS))(0)((IN -> GAS)(IIN -> GAS)))
((ICUT -> REVFLO)((OUT IS DIRTY))(0)((IN -> DIRTY)(IIN -> DIRTY)))
((IOUT -> REVFLO)((OUT IS GRITTY))(0)((IN -> GRITTY)
  (IIN -> GRITTY)))
((IOUT -> REVFLO)((OUT IS CONTAMINATED))(0)((IN -> CONTAMINATED)
  (IIN -> CONTAMINATED)))
((POS -> FAILHI)((IN ISNT SHUTOFF))(0)((OUT -> HISUPPC)
  (OUT -> HISUPPR)(OUT -> HISUPP)(IN -> LOBACKPC)(IN -> LOBACKPR)
  (IN -> LOBACKP)))
((POS -> FAILHI)((OUT IS R)(IN ISNT SHUTOFF))(0)((IOUT -> HIP)))
((POS -> FAILHI)((IN ISNT SHUTOFF)(OUT ISNT SHUTOFF))(0)
  ((IIN -> HIFLO)(IOUT -> HIFLO)))
((POS -> FAILHI)((IN IS R)(OUT ISNT SHUTOFF))(0)((IIN -> LOP)))
((POS -> FAILLO) TRUE (0)((OUT -> LOSUPPC)(OUT -> LOSUPPR)
  (OUT -> LOSUPP)(IN -> HIBACKPC)(IN -> HIBACKPR)(IN -> HIBACKP)))
((POS -> FAILLO)((OUT IS R)(OUT ISNT SHUTOFF))(0)((IOUT -> LOP)))
((POS -> FAILLO) TRUE (0)((IIN -> LOFLO)(IIN -> LOFLO)))
((POS -> FAILLO)((IN IS R)(IN ISNT SHUTOFF))(0)((IIN -> HIP)))
((POS -> DRIFTHI)((IN ISNT COMPLOSUPPR))(0)((OUT -> DISTHISUPPC)
  (OUT -> DISTHISUPPR)(OUT -> DISTHISUPP)))
((POS -> DRIFTHI)((OUT ISNT COMPHIBACKPR))(0)((IN -> DISTLOBACKPC)
  (IN -> DISTLOBACKPR)(IN -> DISTLOBACKP)))
((POS -> DRIFTHI)((IN ISNT COMPLOSUPPR)(IN ISNT SHUTOFF)
  (OUT IS R))(0)((IOUT -> DISTHIP)))
((POS -> DRIFTHI)((IN ISNT COMPLOSUPPR)(OUT ISNT COMPHIBACKPR))(0)
  ((IIN -> DISTHIFLO)(IOUT -> DISTHIFLO)))
((POS -> DRIFTHI)((IN IS R)(IN ISNT COMPHISUPPR)
  (OUT ISNT COMPHIBACKPR))(0)((IIN -> DISTLOP)))
((POS -> DRIFTLO)((IN ISNT COMPHISUPPR))(0)((OUT -> DISTLOSUPPC)
  (OUT -> DISTLOSUPPR)(OUT -> DISTLOSUPP)))
((POS -> DRIFTLO)((OUT ISNT COMPLOBACKPR))(0)((IN -> DISTHIBACKPC)
  (IN -> DISTHIBACKPR)(IN -> DISTHIBACKP)))
((POS -> DRIFTLO)((IN ISNT COMPHISUPPR)(IN ISNT SHUTOFF)
  (OUT IS R))(0)((IOUT -> DISTLOP)))
((POS -> DRIFTLO)((IN ISNT COMPHISUPPR)(OUT ISNT COMPLOBACKPR))(0)
  ((IIN -> DISTLOFLO)(IOUT -> DISTLOFLO)))
((POS -> DRIFTLO)((IN IS R)(IN ISNT COMPLOSUPPR)
  (OUT ISNT COMPLOBACKPR))(0)((IIN -> DISTHIP)))
((POS -> COMPHI) TRUE (0)((OUT -> COMPHISUPPC)(OUT -> COMPHISUPPR)
  (OUT -> COMPHISUPP)(IN -> COMPLOBACKPC)(IN -> COMPLOBACKPR)
  (IN -> COMPLOBACKP)))
((POS -> COMPLO) TRUE (0)((OUT -> COMPLOSUPPC)(OUT -> COMPLOSUPPR)
  (OUT -> COMPLOSUPP)(IN -> COMPHIBACKPC)(IN -> COMPHIBACKPR)
  (IN -> COMPHIBACKP)))
((POS -> DRIFTHI) TRUE (0)((OUT -> DRIFTHISUPPC)
  (OUT -> DRIFTHISUPPR)(OUT -> DRIFTHISUPP)(IN -> DRIFTLOBACKPC)
  (IN -> DRIFTLOBACKPR)(IN -> DRIFTLOBACKP)))
((POS -> DRIFTLO) TRUE (0)((OUT -> DRIFTLOSUPPC)
  (OUT -> DRIFTLOSUPPR)(OUT -> DRIFTLOSUPP)(IN -> DRIFTHIBACKPC)
  (IN -> DRIFTHIBACKPR)(IN -> DRIFTHIBACKP)))
((POS -> CLOSED) TRUE (0)((IN -> SHUTOFF)(OUT -> SHUTOFF)))
((IN -> SHUTOFF) TRUE (0)((OUT -> SHUTOFF)))
((OUT -> SHUTOFF) TRUE (0)((IN -> SHUTOFF)))
((POS -> FAILCLOSED) TRUE (0)((IIN -> X)))
((OUT -> BLOCKED) TRUE (0)((IOUT -> HIP)))
((IN -> BLOCKED)((OUT IS SUP))(0)((IOUT -> LOP)))
```

```
((IN -> BLOCKED) TRUE (O)((IIN -> NOFLO)(IOUT -> NOFLO)))
((OUT -> BLOCKED) TRUE (O)((IIN -> NOFLO)(IOUT -> NOFLO)))
((WS -> BURST) TRUE (O)((OUT -> NOSUPP)(OUT -> NOSUPPR)
  (OUT -> NOSUPPC)(IN -> NOBACKP)(IN -> NOBACKPR)
  (IN -> NOBACKPC)))
((WS -> BURST) TRUE (O)((IN -> LOBACKPR)))
((IN -> GAS)((POS IS OPEN))(O)((OUT -> HISUPPR)))
((IN -> GAS)((POS IS OPEN))(O)((OUT -> HISUPPC)))
```

Attribute: NS - Normal States

```
((POS IS OPEN)((POS -> CLOSED)))
((POS ISNT COMPHI)((POS -> COMPHI)))
((POS ISNT COMPLO)((POS -> COMPLO)))
((OUT ISNT SHUTOFF)((OUT -> SHUTOFF)))
((IN ISNT SHUTOFF)((IN -> SHUTOFF)))
((IN ISNT COMPHISUPPR)((IN -> COMPHISUPPR)))
((IN ISNT COMPLOSUPPR)((IN -> COMPLOSUPPR)))
((OUT ISNT COMPHIBACKPR)((OUT -> COMPHIBACKPR)))
((OUT ISNT COMPLOBACKPR)((OUT -> COMPLOBACKPR)))
((WS IS R))
```

Attribute: SE - Spontaneous Events

```
(WS -> BLOCKED)
(WS -> BURST)
```

Attribute: WS - Working States

```
((POS IS OPEN)((POS IS CLOSED)))
((VALVE IS NOTBLOCKED)((VALVE IS BLOCKED)))
((VALVE IS NOTBURST)((VALVE IS BURST)))
((VALVE ISNT BLOCKED)((VALVE IS BLOCKED)))
```

Attribute: LF - Latent Failures

```
(VALVE IS BLOCKED)
(VALVE IS BURST)
```

7.2 HAZLB2

The library HAZLB2 has 26 components. The names and uses are shown in table 7.4.

Table 7.4 Components in HAZLB2.

Component:	Used for:	Ports:
AIRREG	Air regulator	set, air, in, out
BFTANK	Buffer tank	lev, drn, sv, of, t, p, in, out
CCN	Normaly closed contact	act, in, out
CV	Check valve	in, out
CVALVE	Check valve	in, out, pos
DIV	Divider	in, out1, out2
EVAP	Evaporator	in, out, drn, heat, sv, lev, p, t
HEX	Heat exchanger	hin, hout, in, out
INVERT	Inverter	in, out
LGTANK	Liquid/gas tank	sv, p, lev, drn, in, out
LOAD	Load	in, out
MIX	Mixer	in1, in2, out
OCN	Normaly open contact	in, out, act
PIPE	Pipe	v, c, p, f, t, in, out
PORT	External connection	port
PUMP	Pump	pwr, in, out
RVALVE	Regulation valve	pos, in, out
SEP	Separator	p, sv, lev, drn, in, out
SH	Sensor high	in, out
SIGDIV	Signal divider	in, out1, out2
SIGMIX	Signal divider	in1, in2, out
SL	Sensor low	in, out
SV	Safety valve	in, out
TFTANK	Transfer tank	drn, of, in, out
TRANSA	Transformer	in, out
VALVE	Valve	pos, in, out

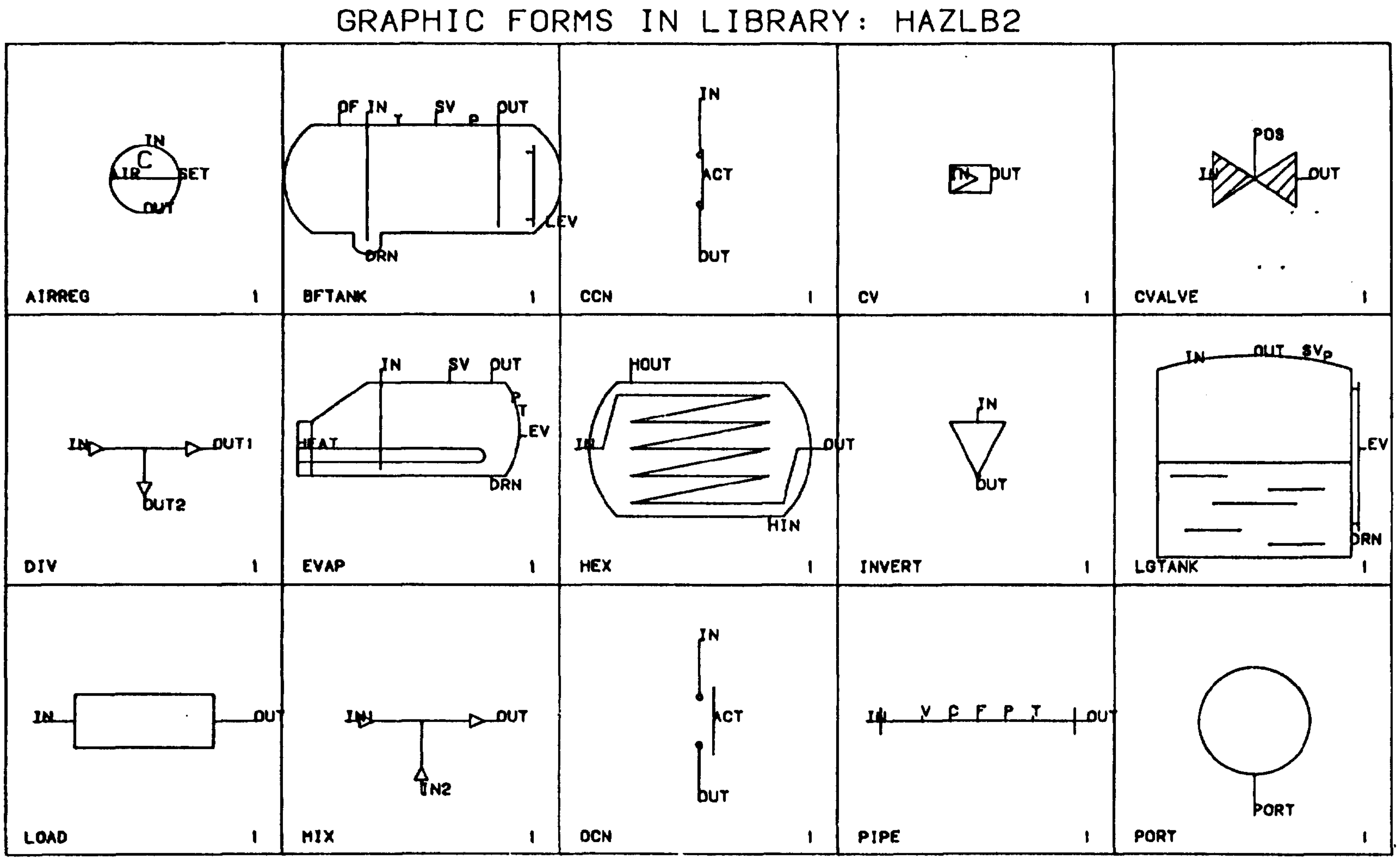
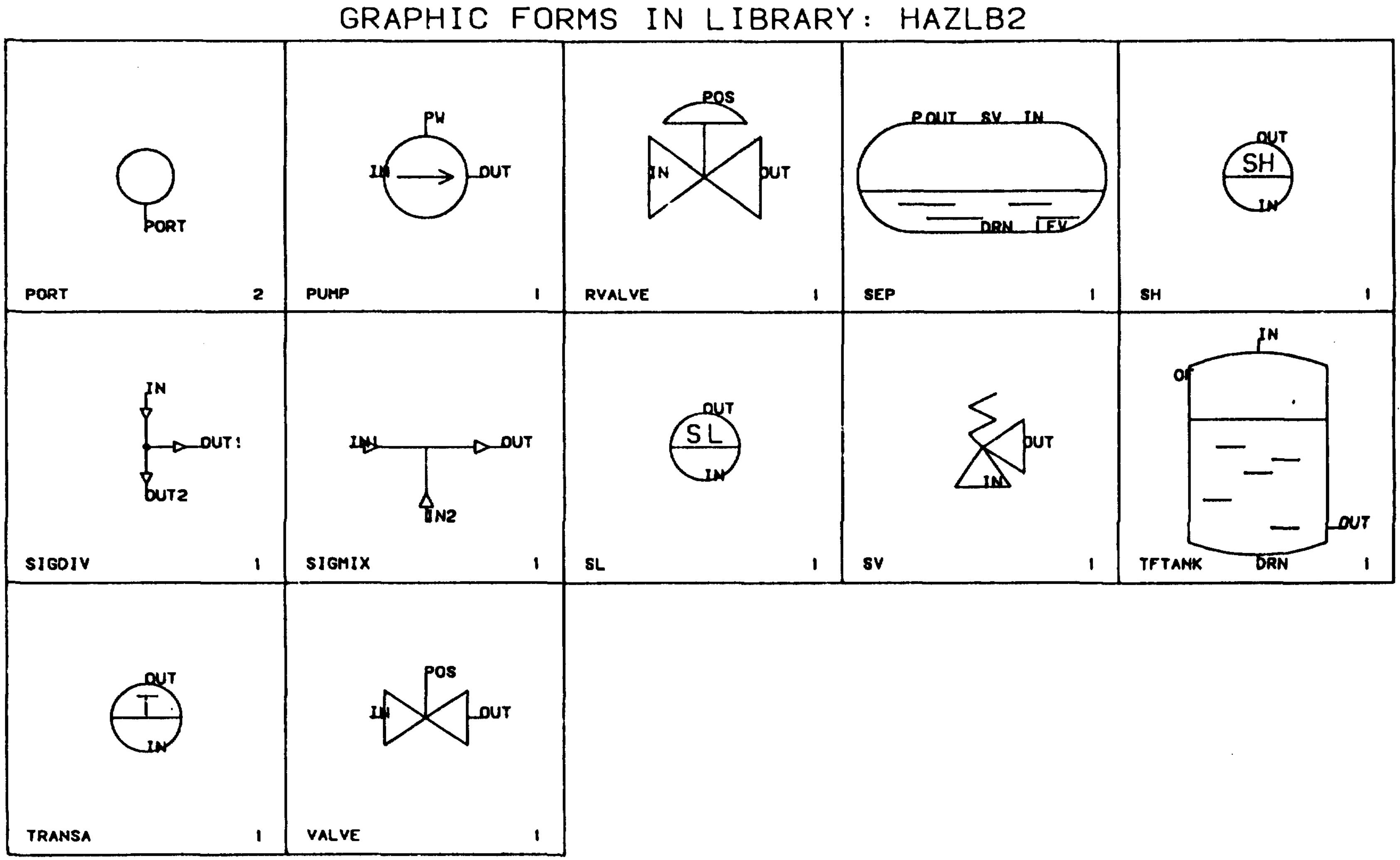


Figure 7.2 Graphic components in HAZLB2.



lgure 7.2 Graphic components in HAZLB2 continued.

In HAZLB2 a special failure generating component (PORT) is available.

This is a contracted component used for generating possible external disturbances that could be let into the system represented by a draft.

If a system includes open ports, as supply, drain and power ports, this component would assure the generation of possible disturbances from the open port while closing it by connection.

Compared to the FTLIB3 this component is a replacement of the drain, supply, power, etc. components, and should be used as such.

The discretisation levels for disturbances used in HAZLB2 are based on the following:

Table 7.5 Discrete levels in HAZLB2.

HI	So high that the disturbance can only be compensated by shutdown.
DISTHI	High enough to cause an accident, not so high that a compensation is impossible.
HISUP	High disturbances in the supply pipe.
HIBACK	High disturbances reverse from the outlet.
COMPHI	Compensation of disturbances.(Regulation)
DISTLO LO LOSUP LOBACK COMPLO	Defined analogously.
ZERO	Disturbances resulting in valves indistinguishable from zero.
REV	Reversal of flow.

Corresponding failure modes that can be distinguished in flow system is:

Table 7.6 Failure modes in HAZLB2.

BLOCKED	causing zero flow.
BURST	causing zero pressure.
LEAK	causing low.
SUPPLIED	
DRAINED	
RELIEVED	
SHUTOFF	
CONTAMINATED	
ON	
OFF	
FAILON	
FAILOFF	

In the following an example of the component RVALVE is shown.

7.2.1 Example of a component in HAZLB2.

=========================

RIKKE - Library: HAZLB2

```
Generic Component: RVALVE
    1-Mar-84 13:04:46

Attribute: VL - Variable List

(IN FV )
(VALVE FV )
(OUT FV )
(WS FV )
(POS FV )
(DE FV )
(R FV )
(O FV )

Attribute: PL - Port List

(IN (IN))
(OUT (OUT))
(POS (POS))

Attribute: TF - Transfer Functions (Mini-fault-trees)

((IN -> HISUPP) TRUE (1)((IN -> AHIP)))
  ((IN -> AHIP)((IN ISNT SHUTOFF)(IN ISNT COMPLOP)
(OUT ISNT COMPLOBACKP))(O)((IN -> HIP)))
  ((IN -> AHIP)((IN ISNT SHUTOFF)(IN ISNT COMPLOP)(OUT ISNT
COMPLOBACKP)
(VALVE IS OPEN))(O)((OUT -> HIP)))
  ((IN -> AHIP)((IN ISNT SHUTOFF)(OUT ISNT SHUTOFF)(IN ISNT
COMPLOP)
(OUT ISNT COMPHIBACKP)(OUT ISNT SHUTOFF)(VALVE IS OPEN))(O)
  ((IN -> HIFLO)(OUT -> HIFLO)))
((OUT -> HIBACKP) TRUE (1)((OUT -> AHIP)))
((OUT -> AHIP)((OUT ISNT SHUTOFF)(OUT ISNT COMPLOBACKP)
  (VALVE IS OPEN)(IN ISNT SHUTOFF)(IN ISNT COMPLOP))(O)((IN ->
HIP)))
((OUT -> AHIP)((OUT ISNT COMPLOBACKP)(IN ISNT COMPHIP))(O)
  ((OUT -> LOFLO)(IN -> LOFLO)))
((OUT -> AHIP)((IN ISNT SHUTOFF)(IN ISNT COMPLOP)
  (OUT ISNT SHUTOFF)(VALVE IS OPEN)(OUT ISNT COMPLOBACKP))(O)
  ((OUT -> HIP)))
((OUT -> AHIP)((OUT IS SUPPLIED)(OUT ISNT SHUTOFF)
  (IN ISNT SHUTOFF)(VALVE IS OPEN))(O)((OUT -> REVFLO)(IN ->
REVFLO)))
((IN -> LOSUPP) TRUE (1)((IN -> ALOP)))
((OUT -> LOBACKP) TRUE (1)((OUT -> ALOP)))
((IN -> ALOP)((IN ISNT COMPHIP)(OUT ISNT COMPHIBACKP)
  (OUT ISNT SHUTOFF)(VALVE ISNT CLOSED))(O)((IN -> HIP)
  (OUT -> HIP)))
((IN -> ALOP)((IN ISNT COMPHIP)(OUT ISNT COMPLOBACKP))(O)
  ((IN -> LOFLO)(OUT -> LOFLO)))
((OUT -> ALOP)((OUT ISNT COMPHIBACKP)(OUT ISNT SHUTOFF))(O)
  ((OUT -> LOP)))
((OUT -> ALOP)((OUT ISNT COMPHIBACKP)(IN ISNT COMPLOSUPP)
  (IN ISNT SHUTOFF)(OUT ISNT SHUTOFF)(VALVE IS OPEN))(O)
```

```
  ((IN -> HIFLO)(OUT -> HIFLO)))
((OUT -> ALOP)((OUT ISNT COMPHIBACKP)(OUT ISNT SHUTOFF)
  (VALVE IS OPEN)(IN ISNT COMPHIP))(O)((IN -> LOP)))
((OUT -> AHIP)((VALVE IS OPEN)(OUT IS SUPPLIED)(IN ISNT SHUTOFF)
  (OUT ISNT SHUTOFF))(O)((IN -> REVFLO)(OUT -> REVFLO)))
((IN -> PDHIFLO)((VALVE IS OPEN))(O)((OUT -> PDHIFLO)))
((IN -> PDHIFLO)((VALVE IS CLOSED))(O)((IN -> AHIP)(WS -> BURST)))
((VALVE -> FAILCLOSED) TRUE (O)((VALVE -> CLOSED)))
((POS -> CLOSED)((VALVE ISNT STUCK))(O)((VALVE -> CLOSED)))
((VALVE -> CLOSED) TRUE (O)((IN -> BLOCKED)(OUT -> BLOCKED)))
((POS -> CLOSED)((VALVE ISNT STUCK))(O)((IN -> SHUTOFF)
  (OUT -> SHUTOFF)))
((POS -> OPEN)((VALVE ISNT STUCK))(O)((VALVE -> OPEN)))
((VALVE -> FAILOPEN)((IN IS SUPPLIED))(O)((OUT -> HISUPP)
  (OUT -> AHIP)))
((IN -> SUPPLIED)((VALVE IS OPEN))(O)((OUT -> SUPPLIED)))
((OUT -> SUPPLIED)((VALVE IS OPEN))(O)((IN -> SUPPLIED)))
((IN -> BLOCKED) TRUE (O)((OUT -> BLOCKED)))
((OUT -> BLOCKED) TRUE (O)((IN -> BLOCKED)))
((IN -> SHUTOFF) TRUE (O)((OUT -> SHUTOFF)))
((OUT -> SHUTOFF) TRUE (O)((IN -> SHUTOFF)))
((IN -> NOSUPP)((OUT ISNT SUPPLIED))(O)((IN -> NOP)(OUT -> NOP)))
((IN -> BLOCKED)((OUT IS SUPPLIED))(O)((OUT -> LOP)(IN -> LOP)))
((IN -> BLOCKED) TRUE (O)((IN -> NOFLO)(OUT -> NOFLO)))
((OUT -> BLOCKED) TRUE (O)((IN -> NOFLO)(OUT -> NOFLO)))
((VALVE -> CLOSED) TRUE (O)((IN -> NOFLO)(OUT -> NOFLO)))
((VALVE -> CLOSED) TRUE (O)((IN -> HIBACKP)))
((VALVE -> BLOCKED) TRUE (O)((IN -> BLOCKED)(OUT -> BLOCKED)
  (IN -> HIBACKP)))
((VALVE -> BLOCKED) TRUE (O)((OUT -> NOSUPP)))
((VALVE -> CLOSED) TRUE (O)((OUT -> NOSUPP)))
((IN -> NOSUPP) TRUE (O)((OUT -> NOSUPP)))
((IN -> ATM)((VALVE IS OPEN))(O)((OUT -> ATM)))
((OUT -> ATM)((VALVE IS OPEN))(O)((IN -> ATM)))
((VALVE -> BURST) TRUE (O)((IN -> ATM)(OUT -> ATM)))
((IN -> ATM) TRUE (1)((IN -> ANOP)))
((OUT -> ATM) TRUE (1)((OUT -> ANOP)))
((IN -> ANOP)((VALVE IS OPEN))(O)((IN -> NOP)))
((IN -> ANOP) TRUE (O)((OUT -> NOP)))
((OUT -> ANOP)((OUT ISNT SHUTOFF))(O)((OUT -> NOP)))
((OUT -> ANOP)((VALVE IS OPEN)(OUT ISNT SHUTOFF))(O)((IN -> NOP)))
((IN -> HIT)((VALVE IS OPEN))(O)((OUT -> HIT)))
((IN -> LOT)((VALVE IS OPEN))(O)((OUT -> LOT)))
((IN -> HICONC)((VALVE IS OPEN))(O)((OUT -> HICONC)))
((IN -> LOCONC)((VALVE IS OPEN))(O)((OUT -> LOCONC)))
((IN -> LIQUID)((VALVE IS OPEN))(O)((OUT -> LIQUID)))
((IN -> GAS)((VALVE IS OPEN))(O)((OUT -> GAS)))
((IN -> CONTAMINATED)((VALVE IS OPEN))(O)((OUT -> CONTAMINATED)))
((IN -> COMPHIP)((VALVE IS OPEN))(O)((OUT -> COMPHIP)))
((IN -> COMPLOP)((VALVE IS OPEN))(O)((OUT -> COMPLOP)))
((IN -> COMPHIT)((VALVE IS OPEN))(O)((OUT -> COMPHIT)))
((IN -> COMPLOT)((VALVE IS OPEN))(O)((OUT -> COMPLOT)))
((IN -> COMPHICONC)((VALVE IS OPEN))(O)((OUT -> COMPHICONC)))
((IN -> COMPLOCONC)((VALVE IS OPEN))(O)((OUT -> COMPLOCONC)))
((OUT -> COMPHIBACKP)((VALVE IS OPEN))(O)((IN -> COMPHIBACKP)))
((OUT -> COMPLOBACKP)((VALVE IS OPEN))(O)((IN -> COMPLOBACKP)))
((IN -> HIVAC)((VALVE IS OPEN))(O)((OUT -> HIVAC)))
((IN -> LOVAC)((VALVE IS OPEN))(O)((OUT -> LOVAC)))
((OUT -> HIVAC)((VALVE IS OPEN))(O)((IN -> HIVAC)))
((OUT -> LOVAC)((VALVE IS OPEN))(O)((IN -> LOVAC)))
((IN -> HIVAC)((VALVE IS OPEN)(OUT ISNT SHUTOFF))(O)
  ((IN -> REVFLO)(OUT -> REVFLO)))
```

```
((OUT -> HIVAC)((VALVE IS OPEN)(IN ISNT SHUTOFF))(O)
  ((IN -> HIFLO)(IN -> LOFLO)))
((IN -> HISUPP)((VALVE IS OPEN)(POS ISNT COMPLO))(O)
  ((OUT -> HISUPP)))
((OUT -> HIBACKP)((VALVE IS OPEN)(VALVE ISNT COMPLO))(O)
  ((IN -> HIBACKP)))
((IN -> LOSUPP)((VALVE ISNT COMPHI))(O)((OUT -> LOSUPP)))
((OUT -> LOBACKP)((VALVE IS OPEN)(VALVE ISNT COMPLO))(O)
  ((IN -> LOBACKP)))
((POS -> COMPHI)((VALVE ISNT STUCK))(O)((VALVE -> COMPHI)))
((POS -> COMPLO)((VALVE ISNT STUCK))(O)((VALVE -> COMPLO)))
((IN -> DRAINED)((VALVE IS OPEN))(O)((OUT -> DRAINED)))
((OUT -> DRAINED)((VALVE IS OPEN))(O)((IN -> DRAINED)))
((IN -> RELIEVED)((VALVE IS OPEN))(O)((OUT -> RELIEVED)))
((OUT -> RELIEVED)((VALVE IS OPEN))(O)((IN -> RELIEVED)))
```

Attribute: NS - Normal States

```
((VALVE IS OPEN)((VALVE -> CLOSED)))
((OUT ISNT SHUTOFF)((OUT -> SHUTOFF)))
((IN ISNT SHUTOFF)((IN -> SHUTOFF)))
((IN ISNT COMPLOP)((IN -> COMPLOP)))
((IN ISNT COMPHIP)((IN -> COMPHIP)))
((OUT ISNT COMPLOBACKP)((OUT -> COMPLOBACKP)))
((OUT ISNT COMPHIBACKP)((OUT -> COMPHIBACKP)))
((VALVE ISNT STUCK)((VALVE -> STUCK)))
((VALVE ISNT COMPLO)((VALVE -> COMPLO)))
((VALVE ISNT COMPHI)((VALVE -> COMPHI)))
```

Attribute: SE - Spontaneous Events

```
(VALVE -> BURST)
(VALVE -> BLOCKED)
(VALVE -> FAILCLOSED)
(VALVE -> FAILOPEN)
```

Attribute: WS - Working States

```
((VALVE ISNT STUCK)((VALVE IS STUCK)))
((VALVE IS OPEN)((VALVE IS CLOSED)(VALVE IS BLOCKED)
  (VALVE IS BURST)))
```

Attribute: LF - Latent Failures

```
(VALVE IS STUCK)
(VALVE IS BLOCKED)
(VALVE IS BURST)
(VALVE IS FAILCLOSED)
```

----

8. FILOSOPHY OF GENERIC MODELLING

The automatic fault tree generation almost has reached a point where it can be used routinely. A well recognised problem, though, is that of creating the component models to be used. This is the work of the domain expert.

The considerations are how the component modelling process should be, and what sizes of fault trees results from different kind of models. This is an important question because the trees grow very rapidly, if you insist on making them at the same time very thorough.

For the modelling work described here, three criteria were established:

(1) The models should be universal, in the sense that, given a model library, the only work required in constructing a new tree should be to draw a flow sheet, piping diagram, or wiring diagram and input of the relevant top event.

(2) The event sequences placed in the tree should be a proper physical description of the dynamic behavior of the plant.

(3) The models should have a well defined scope and within the scope of the disturbance types and failure modes treated, the fault trees should be complete.

These are quite ambigious goals, when applied to process plants or electrical systems. They are considered important, when using fault tree analysis as a design aid however; the first because otherwise the time taken for automatic analysis is longer than for manual; the second and third because mistakes are otherwise easily made and reduce all confidence in results.

Shafaghi (1982) distinguishes between pure logic or predictive models, which aim at producing fault tree results directly via a pattern matching process, and descriptive models, which explain the physical processes occuring. The problem with pure logic models is that all possible patterns must be predicted beforehand, and there is often controversy concerning the correct form of the results (Henley and Kumamoto, 1977 ;Locks, 1979). Descriptive models can be used to analyse component configurations, which have never yet been seen, since the physical processes involved are constant.

Most published models fall between the extremes of pure logic models and descriptive models. The models described here are entirely descriptive.

In (Taylor, 1973) a model construction method was described which fulfills the three criterions mentioned earlier and the following two requirements:

(1) It is necessary to distinguish between disturbances of flow (current), disturbances of pressure (voltage), and disturbances of variables such as temperature, concentration, phase etc., since these have different causal structures.

(2) It is necessary to take account of disturbances which spread upstream as well as downstream in an energy flow system.

Briefly, the model construction is as follows:

(1) A range of components is chosen, and variables to describe their states.

(2) A set of discrete variable values is chosen.

Then for each component:

(3) A set of functional and failure modes is described.

(4) Equations are written to describe functioning and failure.

(5) An equation bigraph is drawn in which squares represent equations, circles represent variables.

(6) All possible causal relationships are drawn on the bigraphs.

(7) Signal flow graph fragments are extracted from the graphs.

(8) For each signal flow graph fragment, an input (x) state -> output table is drawn.

(9) Mini fault trees are written for each entry in the table.

8.1 Model simplification.

In most risk analysis of process plants and electric circuits the fault trees generated are rather big with many branches and loops. To handle the fault trees simplifications are necessary.

(1) A fault tree should be generated, so that propagation of disturbances is completely described, while duplications are eliminated.This pattern constitutes the first simplification of the models.

(2) When plotting the propagation of a distubance such as HIGHPRESSURE, its effect at the output of a component will depend on the back pressure or downstream resistance. At each step along a chain of components, the question must be asked "what is the resistance downstream". This leads to a fault tree structure, which corresponds to an approximate solution of flow equations at each component. Fortunately, such a work is not necessary. If instead of searching for disturbances, a search is made for potential causes of disturbances, such as HIGH SUPPLY PRESSURE, and HIGH BACKPRESSURE, a simpler structure can be achieved.

(3) A third simplification in mini fault trees is deletion of normal conditions. If event A causes event B under condition C, and C is a condition which is normally fulfilled, and there is nothing in the cause of A which can invalidate C, C may be deleted from the mini fault tree. The justification for this is that the probability of a normal condition is close to 1. Deletion of such a condition will not affect the fault tree calculation significantly, and will improve its clarity.

(4) If an event produces the same effect under all conditions, the conditions may be deleted, in a form of "don't care" simplification. The subsumption rule of logic can be used to simplify models. If event A causes B irrespective of C, the mini fault tree involving A, B and C may be deleted. This is a particularly effective simplification in combination with normal state deletion.

(5) Logical inversion of conditions is often useful. If a valve has positions CLOSED, SLIGHTLY OPEN, HALF OPEN, FULLY OPEN, the condition NOT-CLOSED can serve a three fold branching until the "leaves" of the tree are reached.

(6) Cutset to tieset transformation can reduce branching in fault trees. If event X in component type K causes event Y under condition A, and also under condition B and C, then with models in cutset form, branching increases the size of the tree six fold for every instance of type K. By conversion to tieset form, branching is reduced to four fold.

(7) By using complex conditions, branching can be reduced even further. Seperate conditions A, B and C can be reduced to an equivalent complex condition D.

(8) A transformation called sequence splitting is very useful particularly in the analysis of operating procedures. An event X which can lead to events Y and Z under condition A, and to event Y and W under condition B, will lead to a two fold branching if the cause of Y is sought. By splitting into X -> Y, X & A -> Z, X & B -> W, this branching is avoided.

So far the simplifications have preserved the logic of the models. The remaining simplifications involve approximations which are generally, but not always conservative.

(9) Possible compensating conditions can be included in mini fault trees. But if the compensation results in a worse disturbance in the same direction, the compensating condition may reasonably be deleted, on the assumption that a fault tree for worse disturbance will be constructed. For example, in the mini fault tree for a valve IN -> LOWPRESSURE, VALVE IS NOT CLOSED => OUT -> LOWFLOW the condition VALVE IS NOT CLOSED may be omitted, since it will result in OUT -> NOFLOW, a worse disturbance. This may be termed "worse effect deletion".

(10) In the theory described in (Taylor, 1982) a distinction is made between event sequences AB and BA. This is important if there is a difference in consequences for the two sequences. This is often the case if for example A initiates a safety action which takes some time to come into effect and prevent the results of events A and B. All cases where sequence is important though involve loops. In the absence of loops it is permissible to consolidate the sequences, so that AB and BA are treated together.

(11) It was pointed out (Taylor, 1982) that a disturbance LOW at the input to a component can cause a disturbance LOW at the output (can be corrected by shutdown) or DISTURBEDLOW (can be corrected by regulation.
The DISTURBEDLOW transition may be deleted provided that it is known that the larger LOW disturbance is always worst, and that fault trees will be drawn for the worst disturbance.

(12) In some cases a failure can prevent an accident, e.g. an instrument failure causing a trip "just in time" to prevent a serious incident. Such "miracle" effects can generally be deleted from models.

(13) It is generally advisable to distinguish between disturbances caused by failures, and intentional disturbances caused by control devices, e.g. distinguish FAILHIGH and CONTROLHIGH disturbances. Otherwise, algoritms will search in failure structures for sources of potential control actions.

(14) In components which accumulate energy or mass, such as a tank, a small, large or very large disturbance in flow can cause a small disturbance in level either at input or output. The same six disturbances can carry the level disturbance to high and then to extreme levels. The result, in a complete model, is a not very informative

216 fold branching in the fault tree; a kind of "momentum principle", in which a disturbance, once started, continues, requires only that level disturbances are coded according to their origin. Branching is reduced to six fold.

(15) With two storage components connected together, a high level in one causes a high pressure, causing a high outflow, which in turn can cause a high level in the second, a reduced inflow, and an equalisation of levels. Such event sequences simulate level transients in multiple storage systems, but are not particularly enlightening from the point of view of failure analysis. Specific coding of level variations according to cause can restrict such "ping-pong" event sequences between storages, so that event sequences propagate either upstream or downstream, but not back and forth.

8.2 Size versus completeness of fault trees.

The size of a fault tree is best measured for our purposes in terms of the number of branches at the highest level of the tree (i.e., at primary failure).

If models are build according to the principles mentioned above and the simplifications 1-15, then a fault tree for a linear system (a single pipe line) will have a size which grows linearly with the number of components. If sequence simplification is not applied, then the number of branches in the tree will double at every component, giving which is at most $K*2^N$ where K is a constant, and N is the number of components. With some 20 components, this gives several million branches. It is obvious that simplification which is not necessarily conservative, must, for practical purpose, be applied.

Without simplification, there is an additional doubling of fault tree size for some disturbances at every resistive component.

Models which are build following the pattern in section 8.1 may be termed "fully physically conditioned". At the top event they will generate up to size branches, and at every Y junction a four fold branching will follow. The size of a tree for which simplifications are applied, but which are nevertheless fully physically conditioned, is therefore less than K_2*4^M where M is the number of Y junctions, and K_2 is a constant. With 10 Y junctions, this gives a total of area 1 m branches.

Deletion of the resistance conditions and downstream compensations yields models similar to those of Martin Solil et al. (1978). Further deletion of the distinction between flow and pressure disturbances produces models similar to those published by Amendola et al. and Berg et al. Further deletion of transfers of information in two-stream directions produces models similar to those published by Wu et al. (1977).

Of these simplifications, the first, deletion of resistance conditions, is the most effective, since it reduces the number of branches in the tree to a number proportionally to the number of components in the system analysed.

One might think that the deletion of resistance conditions is conservative because cutset sizes are reduced, and generally it is so. However, in the absence of conditioning, it might be thought that a safety device would work, when in fact a pressure signal could not be transmitted past a resistance or past a Y junction, because, for example, a valve had failed open. In such cases, the simplification is definitely not conservative.

On reaching a control component (such as a regulating or shut off valve), component by component algorithm give a branching in the fault tree, with one branch for the disturbance, and one branch for failures in any potential control action. For simple loops such branches soon terminate. But for cascade loops, and loops with two way flow of information, some branches will not terminate directly, and lead to a global search of almost the whole system, looking for signals which might activate the safety action. This corresponds to a global search for negative loops in Lapp and Powers algorithm. Fortunately, most of the "compensation" branches of the fault tree terminate without loop closure, and can be pruned from the trees.

The many branches of a fully physically conditioned tree involve many repeated subtrees. An effective strategy is to store the fault tree as it is generated, and to make a cross link between parts of the tree when such repetitions are found. The value of this strategy was noted by (Lapp and Powers, 1977). this strategy imposes limitations on the size of fault tree which can be produced however, because of the storage required during construction. There is also an insisious pitfall inherent in the strategy, if it is applied to two alternative (OR gate) branches of a tree. The branches may involve different timings, or alternative conditions, in the physical system so that a potential safety action, found in a repeated branch, is not compatible with all disturbances requiring that safety action. Use of the repetition detection strategy may be applied at any time above an AND gate, but should be applied only with care above an OR gate.

Fault tree sizes close to the above bounds are achieved in practice. For example, the pressurised water reactor high pressure soolant injection system of (Rasmussen, 1975) gives a fault tree for loss of flow with branches.

Systems with up to six or seven Y junctions can be treated on a small computer (128 K bytes) and with perhaps ten Y junctions on a large computer (2 M bytes). To treat parts of the fault tree corresponding to each are later interconnected. In this way, fully physically conditioned fault trees of unlimited size can be constructed. The repetition strategy can be applied under close control by analyst.

A useful strategy would be to apply cut off rules to the true construction, so that, for example fourth or fifth order cutsets were omitted. This can be done interactively, but automatical use requires a distinction between possible "normal state" and "unusual disturbance" branches of an OR gate.

9. REFERENCES.

Amendola, A.; Pouchet, A.; Contini, S.; Squellati, G.; Mongellunzzo, R.
Component modelling and computer aided fault tree construction
To be published.

Andrews, J.D.
A user guide to the fault tree and network evaluation program faunet.
Midlands Research Station, England, november 1983, proj M45

Berg, U.; Hellstrom, P.; Lydell, B.
Fault tree synthesis using the CAT algorithm.
Report PSA 02-81 Swedish Nuclear Power Inspectorate.

Larsen, P. Dines
Grace user manual.
Riso National Laboratory, april 1982, Riso-M-2343.

Larsen, P. Dines; Olsen, J.V.
A standardized device-independent graphics system.
Interfaces in Computing, 2, 167-179, 1984.

Olsen, J.V.
A data-base management system for FORTRAN-IV on PDP-11.
Riso National Laboratory, Electronics Department,
Internal note to the system. [DBFOR.MEM]

Olsen, J.V.
A device independent graphic language for minicomputers like PDP-11 or PDP-8.
Riso National Laboratory, Electronics Department,
Internal note to the system. [HCOPY.MEM]

Olsen, J.V.
A device independent graphic package in FORTRAN for PDP-11 under RT11.
Riso National Laboratory, Electronics Department,
Internal note to the system. [GRPLOT.MEM]

Olsen, J.V.
RIKKE - viewed as an expert system.
Riso National Laboratory, Electronics Department.
Internal note to the system. [EXPERT.MEM]. 1984.

Olsen, J.V.; Taylor, J.R.; Nielsen, F.
Use of automatic fault tree and cause consequence analysis methods in the analysis of a chlorine evaporator. Computers in chemical engineering - case studies in design and control.
A symposium organised by the London and South-Eastern Branch of
the Institution of Chemical Engineers. London, june 3rd 1980.

Platz, O.; Olsen, J.V.
FAUNET: A program package for evaluation of fault trees and networks.
Research Establisment Riso, Electronics Department, september 1976. Riso Report no. 348.

Platz, O.; Olsen, J.V.
FAUNET: A program package for fault tree and network calculations.
in Proceedings of the topical meeting, Probabilistic Analysis of Nuclear Reactor Safety, may 8-10 1978, Newport Beach, California USA.

Platz, O.; Olsen, J.V.
Calculating the number and size of prime implicants for a modularized fault tree.
in Lauger, E.; Moltoft, J.(Eds.): Reliability in electrical and electronic components and systems.
North-Holland Publishing Company, 1982

Rasmussen, N.
Reactor Safety Study. An assessment of accident risks in U.S. commercial power plants.
WASH-1400, NUREG-75/014, 1975.

Shafagi, A.
Component modelling for fault tree analysis. Doctoral thesis. Loughborough University. Dept. Chem. Engineering 1982.

Taylor, J.R.
A formalisation of failure mode analysis of control systems.
Riso National Laboratory, october 1973, Riso-M-1654.

Taylor, J.R.
An algorithm for fault tree construction.
Riso National Laboratory, Electronics Department, internal report april 1980, N-19-80.
Preliminary work for:

Taylor, J.R.
An algorithm for fault-tree construction.
IEEE Transactions on Reliability. vol R-31, N 2, june 1982.

Taylor, J.R.
Automated hazard analysis - pitfalls, perspective and prospects.
International conference on Risk Analysis, London. OYEZ.

Taylor, J.R.
Automatic fault tree construction with RIKKE - A compendium of examples, volume 1 basic models.
Riso National Laboratory, september 1981. Riso-M-2311.

Taylor, J.R.
Automatic fault tree analysis of large systems using RIKKE.
Riso National Laboratory, Electronics Department, internal report, may 1982. N-13-82.

Taylor, J.R.
Automatic fault tree construction with RIKKE - A compendium of examples, volume 2 control and safety.
Riso National Laboratory, february 1982. Riso-M-2311.

Taylor, J.R.
Generality of component models used in automatic fault tree synthesis.
Riso National Laboratory, march 1979. Riso-M-2162.

Taylor, J.R.; Hollo, E.
A program for plotting cause consequence diagrams.
Research Establishment Riso, Electronics Department. april 1977, Riso Report M-1932.

Taylor, J.R.; Olsen, J.V.
Treatment of operator error in RIKKE-II.
Riso National Laboratory, Electronics Department, internal report, august 1983. N-22-83.

Taylor, J.R.; Olsen, J.V.
A comparison of automatic fault tree construction with manual methods of hazard analysis.
4'th Int. Symp. on Loss Prev. and Safety Prom. in the Proc. Ind., Harrogate, England, september 12-16 1983.
EFCE Publ series, N 33 vol 1, Pergamon Press.

Wu, J.S.; Salem, S.L.; Apostolakis, G.E.
Use of Decision Talks in Systematic Construction of Fault Trees.
in Fussel, J.B.; Burdick, G.R. (Eds.): Nuclear Systems Reliability Engineering and Risk Analysis, SIAM 1977.

LIST OF TABLES.

LIST OF FIGURES.

APPENDIX A: FILES IN RIKKE AND FAUNET.

Table A.1 List of file extensions.

Filename	Content of file
*.BLK	Block Diagram / Draft Description
*.DIA	Draft database
#.GCL	Genetic Component Library
#.CMP	Extracted (Packed) Component Model
#.LIB	Extracted (Packed) Component Library
#.DGL	Graphic Component Library
#.GML	Extracted Graphic Form(s)
*.PFM	Plant Function/Failure Model
*.FTR	Fault Tree Structure
*.FTX	Fault Tree Text
*.FTN	Fault Tree Text (numeric code)
*.FDA	Failure and Repair Data (for FAUNET calculations)
*.CDR	Consequence Diagram Structure
*.CDX	Consequence Diagram Text
*.CDN	Consequence Diagram Text (numeric code)
*.ETR	Event tree, from FIND
*.ETX	Event tree text
*.HCB	Flow Sheet (graphic code)
*.HCF	Fault Tree (graphic)
*.HCD	Consequence Diagram (graphic)
*.HCC	Cause Consequence Diagram (graphic)
*.HCM	Mini Fault Trees (graphic)
*.HCO	Optional Graphic File
*.PDA	Picture Data (intermediate)
*.PTE	Picture Text (intermediate)
*.MOU	Picture Log (intermediate)
*.LST	Listing (intermediate)
*.TMP	Temporary file used by varioues routines
*.CON	RIKKE <=> FAUNET Conversion Table
*.DAT	Fault tree in free format
*.FLT	Fault tree (FAUNET form
*.CPX	Complex Events
*.PRT	Pruned Fault Tree / Reduced tree
*.ITR	Input Tree (intermediate)
*.RES	Partial Result (intermediate)
*.CSR	CUTSET - Result File
*.TSR	TIESET - Result File
*.EDA	Event Failure and Repair Data
*.CSG	CUTSET - Grouped
*.TSG	TIESET - Grouped
*.CSD	CUTSET - Decomposed
*.TSD	TIESET - Decomposed
*.CSE	CUTSET - Evaluated
*.TSE	TIESET - Evaluated
*.NET	Network description

Note: * stands for Model or System-name
stands for Library/Component name

APPENDIX B: FAULT TREE FILE CODES IN RIKKE.

Table B.1 List of different gate types.

Code	Meaning	Graphic type
A	Normal event ('A PRIORI')	1
B	Normal event in mode 2 ('BAD')	1
C	Common-mode event	9
E	Spontaneous event	1
F	.FALSE.	4
G	Good state (latent failure in mode-2)	22
H	Halt on break-point	31
I	Impossible event (unlinked port in mode-2)	9
L	Latent failure	22
N	Normal state	22
O	Opened mode-2 loop	9
P	Positive state	22
R	Remaining state	26
T	.TRUE.	4
U	Unexpected event (unlinked port)	9
W	Working state	22
X	AND-gate (in mode-2)	11
&	Priority AND-gate	11
+	OR-gate	12
/	Priority OR-gate (in mode-2)	12
=	Internal event	1
#	External event	1
>	State caused by event	22
-	NOT (negation of state)	4
.	Dot (loop indicator)	27
?	Unspecified input (incomplete tree, but fixed)	28
$	END OF FILE	

APPENDIX C: FILES IN FAUNET.

Table C.1 Input files for the FAUNET system.

Filename	Content of file
*.DAT	Fault tree in free format
*.FLT	Fault tree
*.EDA	Event Failure and Repair Data
*.NET	Network description

Table C.2 Files generated by FAUNET.

Filename	Content of file
*.CPX	Complex Events
*.PRT	Pruned Fault Tree / Reduced tree
*.ITR	Input Tree (intermediate)
*.RES	Partial Result (intermediate)
*.CSR	CUTSET - Result File
*.TSR	TIESET - Result File
*.CSG	CUTSET - Grouped
*.TSG	TIESET - Grouped
*.CSD	CUTSET - Decomposed
*.TSD	TIESET - Decomposed
*.CSE	CUTSET - Evaluated
*.TSE	TIESET - Evaluated

Note: * stands for System-name

C.1 Free format fault tree file (*.DAT).

The fault tree file consist of three parts:

(1) The header record, containing the system identifier, max. 6 characters (needs not to be identical to the file-name).
(2) A list of records, one for each gate in the tree. The top-gate is normally entered first.
(3) Finally an end of data marker.

An example of a fault tree file is shown in figure C.1. Here the header contains the system-identifier "CADI".

The following records each define a gate, starting with the top of the tree. The first character in the record is the gate type. Valid gate types are listed in Table C.1. Immediately following the gate type comes the gate-name. All gates are indexed from 1000 to 2000, while events are indexed from 1 to 999.

The second number in the record counts the number of inputs to the gate. This number is limited to 12 (twelve), which means that in practical examples, where more than 12 inputs are wanted in a gate, then the gate must be split into two or more smaller gates of the same type.

Following the input count comes a list of inputs to this gate. The inputs may be events (number < 1000) or other gates (number > 999). All the numbers in the gate record must be separated by comma.

The "$" sign in the last record indicates the end of the file.

```
CADI
+1000,5,1034,1035,1036,1037,1038
X1034,3,1029,2,16
+1035,3,1030,1031,1024
X1036,3,7,20,1032
X1037,2,2,1033
X1038,5,16,17,21,1028,22
+1029,2,3,5
X1030,2,1023,20
X1031,2,7,19
+1032,3,2,1025,4
+1033,2,1026,1027
+1023,3,1,8,10
X1024,2,4,6
X1025,4,7,13,1518
X1026,2,11,12
+1027,2,16,21
+1028,2,2,7
$
```

Figure C.1 A fault tree file in free format.

Table C.3 Legal gate types in free format files.

Gate type	Meaning	
+	OR gate	preferred
0	OR gate	(*
X	AND gate	preferred
x [small "x"]	AND gate	(*
A	AND gate	(*
-	NAND gate (may be used as a NOT gate)	
M	Majority gate	(see below).
	Special plot-marker (plotting postponed)	

Note: These forms are converted to the preferred one.

C.1.1 Majority gates.

It is possible in the free format file to define a majority gate collecting n out of m events as in the following example.

```
M2,1000,3,1,2,3
```

The number n must follow the type "M". Then comes the gate number, the number m and finally the list of m inputs. The gate 1000 in the example represents any (or-ed) combination of 2 out of 3 of the input events and-ed together. The line above is equivalent to the following.

```
+1000,3,1010,1011,1012
X1010,2,1,2
X1011,2,1,3
X1012,2,2,3
```

The evaluation of the majority gate above.

The program FREEIN (command: FREE_FORM) will convert any fault tree in free format into the fixed format needed by the following programs in the FAUNET package. During the .onversion all alternate gate types will be translated into their preferred form, and the majority gates will be evaluated.

The special plot marker, which consist of the character "" followed by a gate number is used as an indicator to the tree plotter (command: PLTSHOW). This marker is skipped by all other FAUNET programs. It should occur in the file before the gate itself is defined, and will in a tree plot postpone the plotting of the gate from its first reference in the tree to a later one or printed by itself. Hereby a fault tree occupying more than one page may be well formed.

As an example we can refer to an example, where it was necessary to enter 1055 as well as 1057 twice in the Dresden-3 fault tree in order to plot it as shown on the pages 24 to 26.

C.2 Fixed format fault tree files (*.FLT).

The fault tree file in fixed format has the same structure as the free format file. It equals the first record contains the system identifier, maximum 6 characters.

The following gate-records are written in the FORTRAN-format (A1,14I4). The last record in the file starts with a "$"-sign, optionally followed by a 4-digit number telling the highest number allowed for internally created gates. We recommend the user to omit this number, leaving the "$"-sign alone in the record.

The set of legal gate types in a fixed format file is limited to the following set:

Table C.4 Legal gate types in fixed format files.

Gate type	Meaning
+	OR gate
X	AND gate
-	NAND gate (may be used as a NOT gate)
	Special plot-marker (plotting postponed)

The fault tree file (CADI.DAT) in figure C.1 may be converted to fixed format by the command:

FREEFORM SYSTEM CADI

The resulting file (CADI.FLT) is shown below:

```
CADI
+1000   51034103510361037 1038
X1034   31029   2  16
+1035   3103010311024
X1036   3   7  201032
X1037   2   21033
X1038   5  16  17  211028  22
+1029   2   3   5
X1030   21023  20
X1031   2   7  19
+1032   3   21025   4
+1033   210261027
+1023   3   1   8  10
X1024   2   4   6
X1025   4   7  131518
X1026   2  11  12
+1027   2  16  21
+1028   2   2   7
$
```

Figure C.2 A fault tree file in fixed format.

C.3 Event data file (*.EDA).

The Event Failure and Repair Data file is format free. It consist of three parts:

(1) A header record containing the the system identifier, maximum 6 characters. It must be identical to the identifier in the fault tree file for the actual problem.

(2) A list of records containing: The component (event) number, calculation type, failure data, mean repair time and test interval etc. All the numbers are separated by comma (","). A list of possible calculation types is shown in table C.5.

(3) Finally an empty record, or a record containing a "0" acting as an end-of-file indicator.

Table C.5 Calculation Types and their Input Data.

Calculation type	Meaning	Inputs
1	Constant Failure Probability (A)	$A*10^6$
2	Exp. Fail. Distribution (rate=A) and Exp. Repair Distr. (mean=B)	$A*10^6$ and B
3	Exp. Fail. Distr.(rate=A) and Const. Repair Time (B)	$A*10^6$ and B
4	Exp. Fail. Distr.(rate=A) with Const. Repair Time (B) and Constant Test Interval (C)	$A*10^6$, B and C

The following figure shows an example of an Event Data file.

```
ESS
11,2,2.,50.
12,2,2.,50.
13,2,2.,50.
14,2,2.,50.
15,2,2.,50.
21,2,2.,50.
22,2,2.,50.
23,2,2.,50.
24,2,2.,50.
25,2,2.,50.
26,2,2.,50.
31,2,2.,50.
32,2,2.,50.
33,2,2.,50.
34,2,2.,50.
51,2,2.,50.
52,2,2.,50.
71,2,2.,50.
110,2,2.,50.
120,2,2.,50.
210,2,0.1,20.
220,2,0.1,20.
230,2,0.1,20.
240,2,0.1,20.
250,2,0.1,20.
260,2,0.1,20.
361,2,0.5,2000.
362,2,10.,200.
363,2,0.5,2000.
364,2,10.,200.
371,2,0.5,2000.
372,2,10.,200.
381,2,0.5,2000.
382,2,10.,200.
383,2,0.5,2000.
384,2,10.,200.
411,2,0.1,10.
412,2,0.1,10.
413,2,0.1,10.
414,1,10000.
421,2,0.1,10.
422,2,0.1,10.
423,2,0.1,10.
424,1,10000.
431,2,0.1,10.
432,2,0.1,10.
433,2,0.1,10.
434,1,10000.
610,2,10.,1.
620,2,10.,1.
710,1,100000.
811,2,0.1,100.
812,2,0.5,1.
821,2,0.1,100.
822,2,0.5,1.
831,2,0.1,100.
832,2,0.5,1.
841,2,0.1,100.
842,2,0.5,1.
851,2,0.1,100.
852,2,0.5,1.
861,2,0.1,100.
862,2,0.5,1.
871,2,0.1,100.
872,2,0.5,1.
901,2,0.1,100.
902,2,0.5,1.
0
```

Figure C.3 Event Failure and Repair Data file. (From Platz and Olsen, 1978).

C.4 Network description (*.NET).

A network is described in a (format-free) network description file. This file consist of three parts:

(1) A header record containing the the system identifier, maximum 6 characters.
(2) A list of records defining the network by its links. A bidirectional link is described by the link-number followed by the numbers of the connected nodes (separated by commas). A unidirectional link is described by a minus ("-") followed by the link-number, the number of the outgoing node and finally the number of the incoming node.
(3) Finally an empty record, or a record containing a "0" as an end-of-file indicator.

The link-numbers as well as the node-numbers are used as component (event) numbers in the fault tree produced as a description of the wanted cuts or paths in the network. We therefore recommend the user to specify different numbers for nodes and links. This a "must" in the case, where both nodes and links are included in the analysis.

As an example, figure C.4 shows two network-files.

```
NBBEX2          JBFIG1
-1,20,21        10,1,3
-2,20,21        11,3,4
-3,20,22        12,4,7
-4,21,23        13,7,8
-5,21,23        14,1,2
-6,22,23        15,2,5
-7,23,24        16,5,6
-8,23,24        17,6,8
-9,24,25        18,3,5
-10,23,25       19,5,7
-11,25,27       0
-12,25,27
-13,25,27
-14,23,26
-15,26,27
-16,26,27
0
```

Figure C.4 Examples of network description files.
(NBBEX2.NET and JBFIG1.NET).
(From Platz and Olsen, 1976).

APPENDIX D: EVENT FAILURE AND REPAIR DATA USED IN FAUNET.

Kind: 1 Constant failure probability p.

Form: <event>,1,p*10^6

Kind: 2 Exponential failure distribution with failure rate lambda and exponential repair distribution with mean repair time r.

Form: <event>,2,lambda*10^6,r

Kind: 3 Exponential failure distribution with failure rate lambda and constant repair time r.

Form: <event>,3,lambda*10^6,r

Kind: 4 Exponential failure distribution with failure rate lambda, constant repair time r and constant test interval i.

Form: <event>,4,lambda*10^6,r,i

<event> stands for the actual event number (integer), while the arguments p, lambda, r and i are all real numbers.

Note that probabilities and failure rates are multiplied by 10^6.

The data file (*.EDA or *.FDA) contains:

(1) The system (model) name.
(2) One record of data for each basic event.
(3) Finally an empty record (or a 0) indicating the end of the list.

Example:

```
BMFT4
1,3,100.,0.5,50.55
2,1,100000.
3,4,80.,10.,672.,27415.3
...........
..........
35,4,10.,100.,672.,3467.3
0
```

APPENDIX E: RIKKE COMMANDS AT A GLANCE.

Command	Program called	Purpose
MODEL	none	Allows user to define or redefine which model the system is to construct or make use of.
WHAT	none	To find the name of the plant model currently being used.
STOP	none	Stops execution of RIKKE and terminates a RIKKE session
DRAFT	GRACE	To activate the drafting input program.
MAKE	LNKMOD	To build up a plant functional and failure model.
FAULT	FTGEN	To produce a fault tree.
TEXT	TEXTER	To transform fault tree text from numeric form to a readable form.
FTPLOT	CCPLOT	To produce a plotting file containing a fault tree as a series of A4 pages.
FTSUPER	CCPLOT	To produce a plotting file containing a fault tree (not broken into A4 pages).
PLOT	PLOT	To send a plotting file to the actual plotting device.
VIEW	PLOT	To send a plotting file to a graphic display screen.
FTSHOW	TTTREE	To plot a fault tree on the typewriter.
CUT	FTCUT	To prune a fault tree of unwanted event types.

APPENDIX F: PAUNET COMMANDS AT A GLANCE.

Command	Program called	Purpose
SYSTEM	PAUNET	Allows the user to define or redefine the system file name for which the PAUNET calculations are to be evaluated. Tells which files are available for this system.
PAUNET	PAUNET	Tell the system file name and which files are precently available for this system.
CUTSET	CUT	Calculate minimal cutsets of a fault tree.
TIESET	CUT	Calculate minimal tiesets of a fault tree.
PATHSET	CUT	Equivalent to the command: TIESET.
CUTSET PRUNED	CUT	Calculate minimal cutsets using a previousely pruned fault tree as input.
TIESET PRUNED	CUT	Calculate minimal tiesets using a previousely pruned fault tree as input.
PRUNE	CUT	Perform a modularisation of a fault tree and output the pruned fault tree together with its list of complex events.
RESULT	CUTRES	Show the result (count of cutsets) from a previous calculation.
RESULT OF TIESET	CUTRES	Show count of minimal tiesets previousely calculated.
DECOMPOSE	CUTPIV	Perform a pivotal decomposition of the minimal cutsets previously calculated.
DECOMPOSE TIESET	CUTPIV	Perform a pivotal decomposition of the minimal tiesets previously calculated.
TREE	CUTREE	Convert minimal cutsets into a pruned fault tree.
TREE FROM TIESET	CUTREE	Convert minimal tiesets into a pruned fault tree.
UNAVAILABILITY [USING TIESET] [DECOMPOSED] [REPAIR]	UNAVA	Calculate unavailabilities, and optionally failure intensities from

		cutsets or tiesets using failure data for the primary events.
	Note:	Arguments in brackets are optional.
CHECK [DUAL]	TREECH	Check consistency of a fault tree file and calculate the maximum number of cut/tiesets.
NETPATH [LINKS/NODES]	TIENET	FROM a TO b Calculate paths in a network (directed or not) from node a to node b (both entered as numbers) and optionally output either the links passed, the nodes passed or both links and nodes (default).
	example:	NETPATH LINKS FROM 5 TO 6 Calculate the set of links passed in all possible paths from node 5 to node 6. The output is formed as a fault tree.
FREEFORM [DUAL]	FREEIN	Convert a faunet fault tree written in free format to fixed format form, optionally producing the dual tree.
PLTSHOW	TTTREE	Plot a FAUNET fault tree on the typewriter.
PRTSHOW	TTTREE	Plot a pruned FAUNET fault tree on the typewriter.
EVALUATE [TIESET]	CUTEV	Evaluate the modularized cutsets (default) ot tiesets completely and sort the result.
GROUPING [TIESET]	CUTGRP	To divide the calculated cut/tiesets into independant groups.
PRINT	RIKUTL	May be used to print the calculated cut/tiesets on the typewriter.
Subcommand: FILE-NAME		Specify the wanted result by combining the system name and the file type into a file name. Example: LDDRUM.CSR

INDEX

Risø National Laboratory **Risø - M - 2480**

Risø - M - 2480

Title and author(s)

RIKKE

User's Manual

P. Haastrup, J.V.Olsen, J.R.Taylor,
Axel Damborg and N.K.Vestergaard

Date
February 1985

Department or group

Group's own registration number(s)

133 pages + tables + illustrations

Abstract

RIKKE is a computer program for reliability and safety analysis of process plants, electrical systems ets. The program is available in a PDP-11 and a VAX varsion. The manual gives a description of the use of the program as a tool in the hazard analysis of an actual process plant. Furthermore the manual gives a summary of the principles of building new components as parts of the existing libraries.

Copies to

Available on request from Risø Library, Risø National Laboratory (Risø Bibliotek), Forsøgsanlæg Risø), DK-4000 Roskilde, Denmark
Telephone: (0) 37 12 12, ext. 2262. Telex: 43116