Peter B. Ladkin, Bernd Sieker and Joachim Weidner
Faculty of Technology, University of Bielefeld

June 19, 2001

Report RVS-Occ-01-04

Abstract

We show how to generate fault trees algorithmically from Causal Influence Diagrams (CIDs), and report on the implementation of such a facility in the CID-drawing tool `cid2dot`.

1 Some Considerations on Fault Trees

Fault trees are a widely-used method, standardised in many countries, of cataloguing in a structured manner the myriad ways that a system can go wrong. Fault trees have been used in the engineering of safety-critical systems for a half century, starting with the Minuteman ICBM system and the nuclear power industry [VGRH81] in the USA. In many industries they remain the prime method of assessing the safety properties of a system design in advance of building the system. They can also be used during and after an incident for finding the source of failure, in other words for high-level "debugging".

Fault trees have two fundamental theoretical aspects: logical, and probabilistic. The logical aspect is fundamental, since the probabilistic features

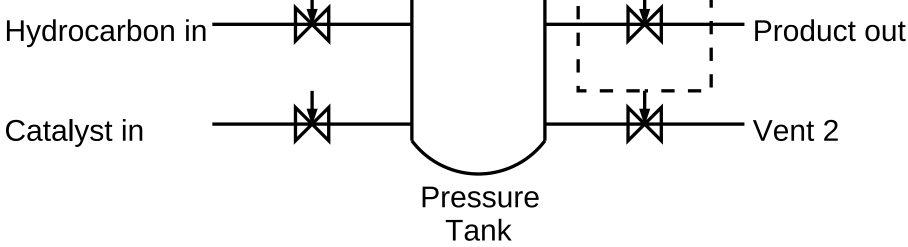


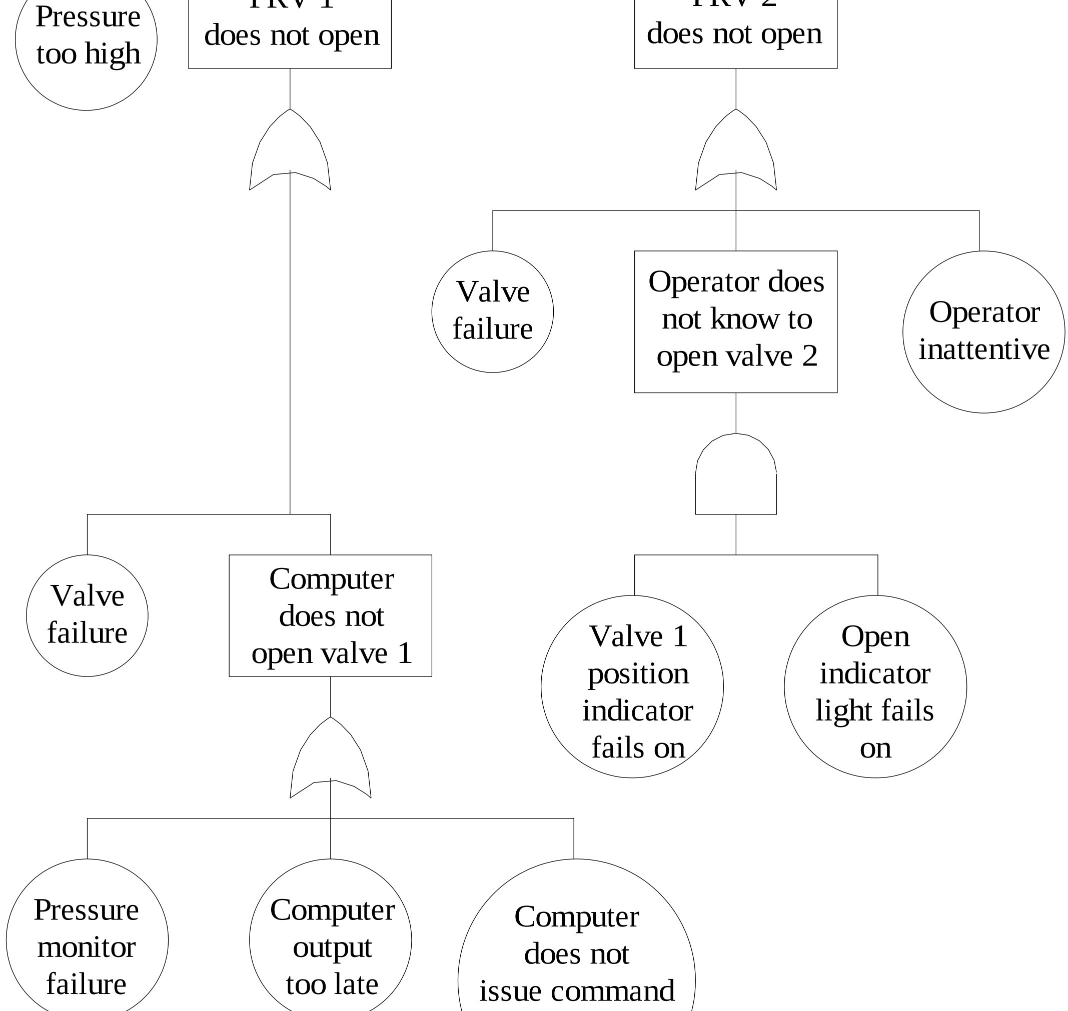
Figure 1: The Pressure Tank With Overpressure Vents

1.1 How Fault Trees Look

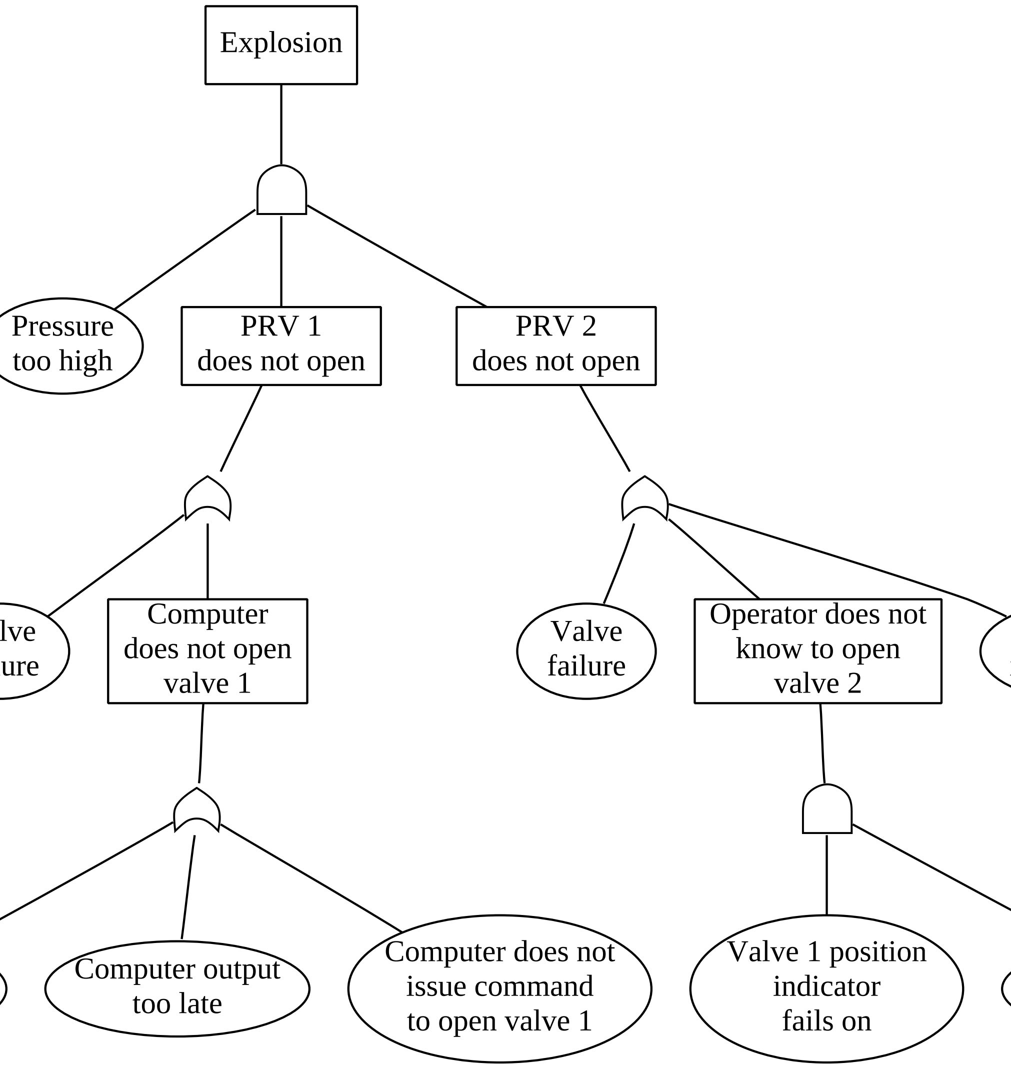
Figure 1 shows a pressure tank, and Figure 2 shows a fault tree for the pressure tank (at this high level of design) taken from [Lev95]. The pressure tank example in various versions is a "classic" amongst fault tree explanatory examples, likely because of its appearance as such in the Fault Tree Handbook [VGRH81], written by some of the original developers of fault tree methodology in the nuclear power industry.

We have implemented a postprocessor to our program for displaying Causal Influence Diagrams (CIDs), `cid2dot`, for automatically generating fault trees according to the techniques explained here. Since `cid2dot` uses the `dot` graph-drawing tool, which draws straight edges between nodes, rather than the horizontal-vertical cornered edges used in conventional fault tree display, we shall display the fault tree in Figure 2 as in Figure 3.

1.2 The Logical Structure of Fault Trees



Pressure too high
does not open
does not open
Valve failure
Operator does not know to open valve 2
Operator inattentive
Valve failure
Computer does not open valve 1
Valve 1 position indicator fails on
Open indicator light fails on
Pressure monitor failure
Computer output too late
Computer does not issue command


Explosion
Pressure too high
PRV 1 does not open
PRV 2 does not open
Computer does not open valve 1
Valve failure
Operator does not know to open valve 2
Computer output too late
Computer does not issue command to open valve 1
Valve 1 position indicator fails on

the labels appearing in the nodes of the tree. A *positive Boolean expression* is a Boolean formula which uses the logical constants "AND" and "OR" only, or a Boolean formula logically equivalent to such a formula.

If R is the label of the top node of the tree, the assertion represented by the fault tree is that

R is equivalent to (the Boolean formula denoted by the tree)

R is typically a statement of a fault (either a failure of function, or a violation of a safety requirement of some sort). Let *the Boolean formula denoted by the tree* be denoted by B. The fault tree thus asserts that fault R occurs in exactly the circumstances in which B occurs.

We call a fault tree *correct* if and only if the statement *R is equivalent to B* is true. Suppose that (the fault tree corresponding to) B is correct. Suppose further that B' is (the assertion corresponding to) another fault tree for R, and suppose that B' is correct. Then because both R is equivalent to B and R is equivalent to B', B must be equivalent to B'. It follows that all correct fault trees for R with the same atomic formulae are Boolean equivalents of each other. Given the same collection of "faults" (atomic formulae), all correct fault trees for R are then just rewrites of each other.

1.2.2 Logical Relation to Causal Logic

The logic EL for describing system behavior was introduced in [LL98] and forms the logical basis for WBA, which is a part of CSA. EL is a multi-modal logic, including temporal logic as well as Lewis's causal logic. It is well known that, given a first-order logical language sufficient for describing fully

It follows that Boolean formulae over the available state predicates for the system do not suffice in general to express liveness properties. Since some violations of system properties which one might want to represent as fault trees may well be, or imply, liveness properties, if follows further that fault trees over the available state predicates of a system do not suffice for expressing all violations of expressible system properties. If all safety properties of a system may be represented as technical safety properties, then one has a chance to represent the safety properties as fault trees (which we shall show immediately). Whether safety properties of a given system may be represented as technical safety properties depends, upon other things, on what is chosen to be defined as an accident of the system.

A failure corresponding to R leaves the system in a particular state, and all states of the system, as considered in the description language L, which may be a high-level or low-level system description language, can be represented as a conjunction of basic formulae of L (a *basic formula* is an atomic formula in L or the negation of an atomic formula). So the collection of all possible failures corresponding to R may be represented as the disjunction of all such state descriptions (conjunction of basic formulas of L). This is writing R in "disjunctive normal form" (DNF). But while this is theoretically possible, the DNF may be orders of magnitude larger than the most succinct form of writing the same failure mode R as a Boolean expression – indeed, the function converting an arbitrary Boolean formula into a DNF equivalent is of exponential complexity (in time and space!). So this is not very plausible method of obtaining a fault tree in practice, even in this case.

A CID in general says how a system works, physically. That is, in general one could consider it as a form of specification of a system. Since specifications can be logically more complex than Boolean expresssions, it follows methods to come up with fault trees that are not logically equivalent (or even semantically equivalent) to each other. It may be very hard or impossible to decide what the appropriate procedure is in these circumstances. Should one choose a "preferred" fault tree, and run the risk that its debugging and prediction properties are incomplete? Or should one use them all, to achieve a greater level of completeness at the cost of significant repetition? Anwering these questions is beyond the scope of the current work. We wish only to make it clear that generating fault trees remains in general an information-lossy process. Nevertheless, let us go ahead.

2 Why Generate Fault Trees Automatically?

Designing fault trees is an art based on experience. Designing CIDs is an art based on experience. If one needs a fault tree, why would one substitute an indirect method, such as designing a CID or CIDs, followed by an automated or semi-automated step, generating the fault trees from the CIDs, for the direct method of just drawing the fault trees?

We believe, and our experience has shown us, that CIDs are intuitive representations of the way a system functions or fails. System designs are often represented by "functional diagrams" and other diagrammatic devices, and the corresponding CIDs are conformant with, although not identical to, the functional diagrams. This suggests to us that one is less likely to need "Eureka" steps when designing a CID, and more likely to find errors when checking the correctness of a CID. The concept of how things work is more cognitively malleable and more cognitively familiar, than a concept of how things fail. We think correct CIDs are easier to come by than correct fault availing itself of causal mechanisms. The property "always (X implies eventually Y)" is a classical example of an always-eventually property, and along with the property "always eventually X" entails "always eventually Y". Such properties are not always reducible to safety properties. Such properties will be ensured in a teleological system through causal link "X $\rightarrow$ Y" with hysteresis (that is, unspecified time delay), which is represented in the CID with the annotation "TIME" on the arrow linking "X" to "Y". So CIDs have some expressive capabilities for teleological systems which lie near to the system specifications. Fault trees are unable to express always-eventually properties in general.

We have found that CIDs in practice are indeed cognitively closer to system function specifications and important complex system properties of teleological systems, and that designing them is cognitively easier a task than designing fault trees by hand; corresponding errors are reduced and error-detection is enhanced. However, the case for having and using fault trees has been made above and elsewhere; we may take it that we need them. We believe this makes the case for generating fault trees as automatically as possible from hand-designed CIDs over hand-designing fault trees. We propose to show here how fault trees may be automatically generated from CIDs.

3 A Causal Influence Diagram Example

Suppose one has a Causal Influence Diagram (CID), as in Figure 4. The nodes are labelled abstractly. We denote the single alphabetic labels in the

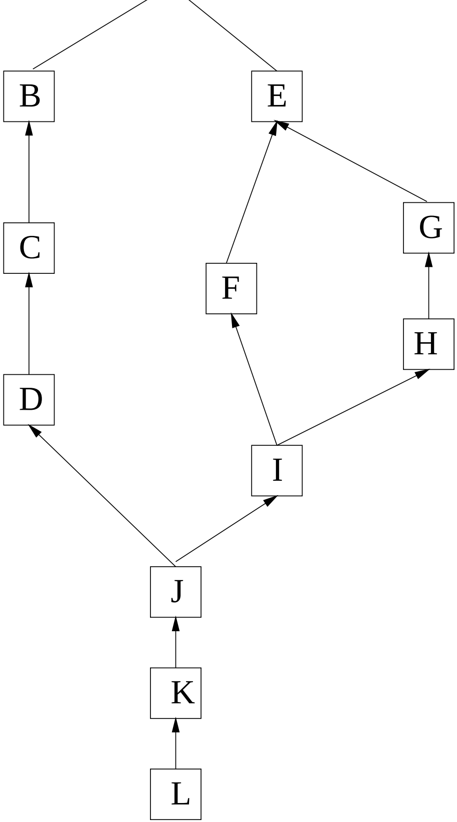


Figure 4: An Example of a CID

a parent node represents a collection of conditions jointly sufficient for the occurrence of the parent situation.

Conditions which are individually necessary for the parent and jointly sufficient we shall call *INJS* factors. The second assumption then says that any node with two or more in-edges has a collection of INJS conditions as children. This assumption entails that one must take particular care in formulating "alternative paths" to a node in a CID, that these alternatives terminate in a set of INJS conditions for their confluence.

4 Denoting "Normal" and "Failure" Conditions

First, we show how to deal with chains. The CID in Figure 4 presents the causal ancestors of a situation "A". We assume that "A" is a condition expected in or consistent with the system specification and the system's safe running, and, correspondingly, that "Not A" represents a fault or failure condition of some kind (either a functional failure, or an accident or a hazard).[2] This assumes that all factors are discrete factors, since, with fluents "X", there is no prima facie meaning to an assertion "Not X": if "X" is "Temperature(Liquid)", for example, it makes no sense to say "Not

[2]CIDs devised to analyse failure conditions or accidents may already have the top node labelled "Not A". The convention we use here is thus important for generating the fault tree. Nodes that represent failure conditions will implicitly include what we represent as a "Not ..." in what follows, whether that negation is explicitly present in the label or not. Another, maybe preferable, way of checking this would be to annotate a formula

is a failure situation: some wished-for causal mechanism has been hindered. Natural language is expressive enough for this failure of a desired causal mechanism to have, maybe many, succinct and evocative representations, but we shall simply use the phrase "X and Not Y" to represent such a failure. In the case in which "Y" has no predecessor in the CID, we shall simply write "Not Y". If this convention may be uniformly followed throughout the fault tree construction; a more succinct representation of the failure condition can be substituted at the end of construction if desired, and meanwhile the label carries its origin with it during construction, enabling more efficient error-checking.

The case of "X $\rightarrow$ Y", where "X" and "Y" are fluents, is largely similar. A normal condition is represented by a particular relation, say "P", between the values of "X" and "Y". We can denote this situation by "P(X,Y)". When this situation does not hold, namely in the case of "Not P(X,Y)", then it is obvious that we can reasonably use the same "Not" annotation as for the discrete case. However, the label of the node itself is not "P(X,Y)", but, rather, "X" or "Y". Further, the description of "P" may be much too involved to want to write in a node label, and, besides, its description is irrelevant to the task of generating a fault tree. We choose to write, as a failure designation, "Y is out of tolerance with respect to X", or, more succinctly, "Y is out of tolerance". We shall denote this in abstract example even more succinctly as "X and Not Y", as we do in the discrete-factor case, since when we are dealing with abstract CIDs, as here, we do not know which factors are discrete and which fluent, and we do not need to know. In concrete examples, we shall use the "out of tolerance" nomenclature.

5 Handling the Individual C

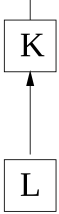


Figure 5: A Chain of Necessary Factors and Ancestors

The relation of being an NCF is not transitive. That is, if X is an NCF of Y and Y is an NCF of Z, this does not necessarily mean that X is an NCF of Z. It might be or it might not be. Thus we show "J" as a necessary causal ancestor (NCA) of "A", meaning that it is one in a chain of necessary causal factors that lead to A, by using a dotted line in Figure 5.

Since all factors shown in Figure 5 are necessary, but not necessarily sufficient, or at least not known to be sufficient, any one could occur without the condition of which it is an NCA occurring. That is, "L" could be the case without "K" occurring, "K" could be the case without "J" occurring, and "J" could be the case without "A" occurring. In a case in which, say, "L" were also to be a sufficient causal factor for "K", then an occurrence of "L" entails that "K" occur also, and it would not then be the case that "L" could occur without "K" occurring. But let us deal first with the situation in which all are known to be either NCFs or NCAs of their successors in Figure 5, as indicated.

The situation in which "L" occurs but "A" does not occur, according to the CID in Figure 5, is represented in Figure 6. The three possibilities for

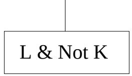


Figure 6: The Fault Tree Arising From the NCA Chain

descriptions in the boxes and is widely used in fault trees.[3]

The case in which a link in the chain is a sufficient causal factor or sufficient causal ancestor is shown in Figure 7. The condition “X” is an SCA of “A” , whereas “Y” is an NCF of “X” and “Z” an NCF of “Y”. So the situation “Z and Not Y” can arise, as can the situation “Y and Not X”, but since “X” is sufficient for “A”, the situation “X and Not A” is impossible, and thus, in contrast to the fault tree in Figure 6, the factor “X and Not A” is not shown. The fault tree resulting from the chain including an SCA is shown in Figure 8.

The case in which one has two NCFs that are INJS conditions, but in which neither is individually sufficient, is shown in Figure 9. In this case, “B and E and Not A” is not possible, since “B and E” entails “A” (this is what it means for “B” and “E” to be jointly sufficient). But each factor indivudually is necessary, although not presumed to be sufficient, so “B and Not E and Not A” is possible, as is “Not B and E and Not A”. The labelling we use on the fault-tree nodes is the pre- and post-conditions of the factors in the NCF/NCA/SCF/SCA relations, so these labels with three factors reduce to “B and Not A” and “E and Not A” for the fault tree representation in Figure 10.

[3] The major difference between the notation we use here and that of fault trees is that

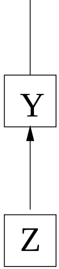


Figure 7: A Chain of NCFs with a SCA

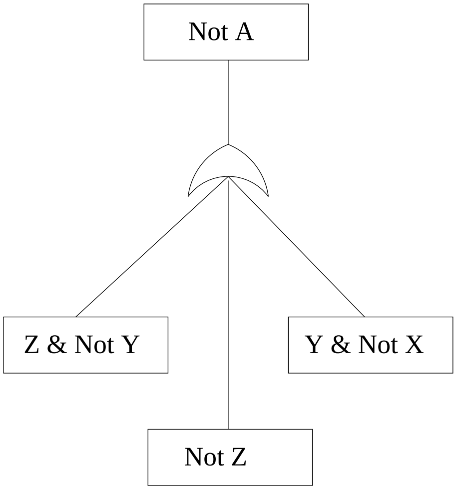


Figure 9: Two Individually Necessary, Jointly Sufficient Factors

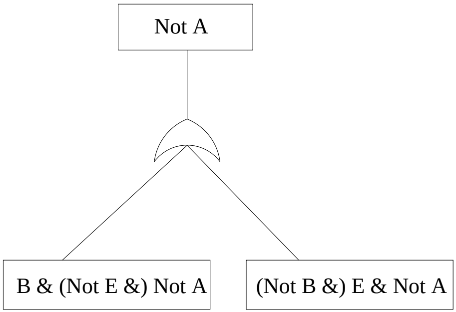


Figure 10: The Corresponding Fault Tree for the INJS Factors

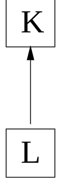


Figure 11: Step 1, After Application of MPRI

6 Putting It All Together

We presume that the CID in Figure 4 is complete, that is, the individual edges indicate the relation of NCF and, when a node has two or more in-nodes, those in-nodes are INJS conditions.

The first step proceeds exactly as indicated above. We repeat it here for completeness.

Multiple Path Reduction (MPR) Where multiple paths connect two nodes in the complete graph, we connect them with an NCA arrow, and eliminate the intervening nodes.

We then apply the MPR operation iteratively:

MPR Iteration (MPRI) Iterate MPR until there are no more multiple paths.

We arrive at the CID in Figure 11.

Now:

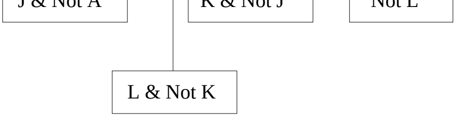


Figure 12: Step 1, After Application of CC

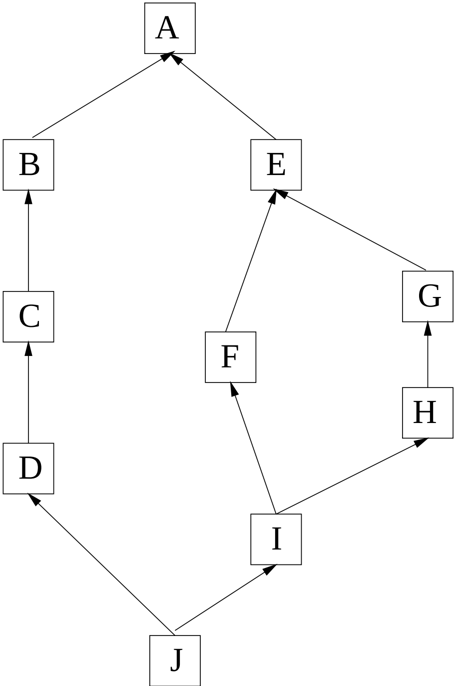


Figure 14: Applying BMP

The "J"–to–"A" fragment of the CID can now be expanded into a fault tree. This fragment is shown in Figure 13.

We first consider the node "A", which has two in-edges. We have seen that this node with its two predecessors can be transformed as above, which we notate as follows:

INJS Resolution (INJSR) Apply the INJS transformation to a specific node

We apply INJSR as follows:

Break Multiple Paths (BMP) When there are multiple paths from node "X" to node "Y", and node "Y" is a confluence node of those paths, apply INJSR to "Y".

We apply BMP thus to "A". The result, as above, is shown in Figure 14.

We are now left with a chain to reduce, denoted by the node "E and Not A", which we recall is "E and Not B and Not A", and a part-chain, part-multipath, graph fragment, denoted by the node "B and Not A", which we recall is "B and Not E and Not A". The chain is shown in Figure 15, and the chain-multipath is shown in Figure 17.

The chain in Figure 15 can be handled as before using CC, and the resulting fault tree is shown in Figure 16.

Finally, the "J" to "Not E" fragment looks like Figure 17, to which MPR

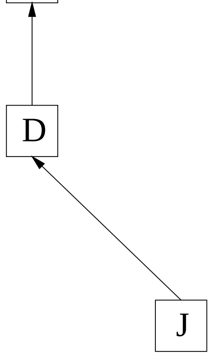
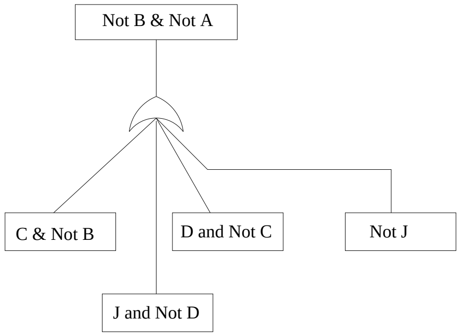


Figure 15: The J-to-Not-B Fragment



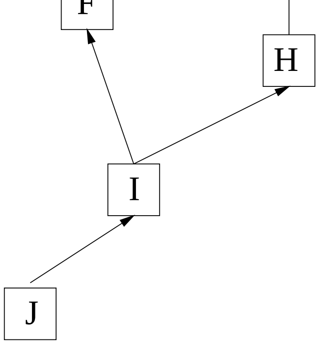
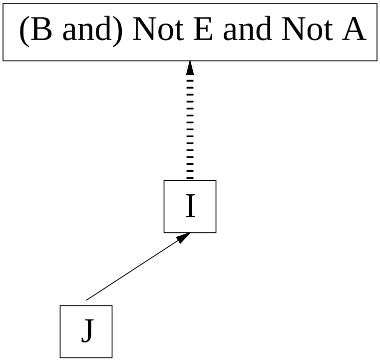


Figure 17: The J-to-Not-E Fragment



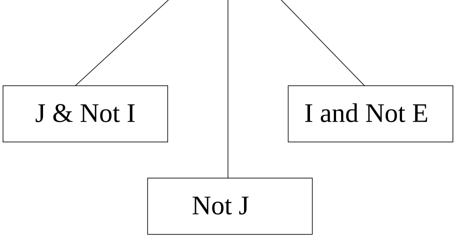
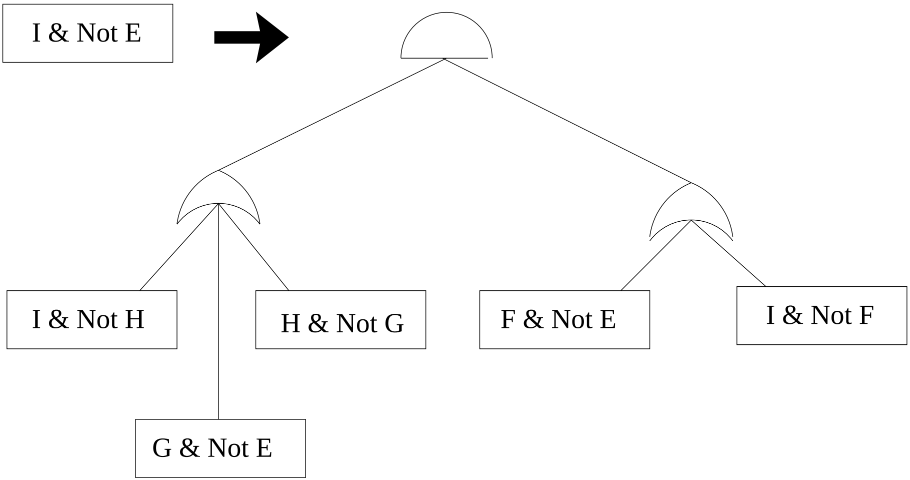


Figure 19: The J-to-Not-E Fault Tree After BMP



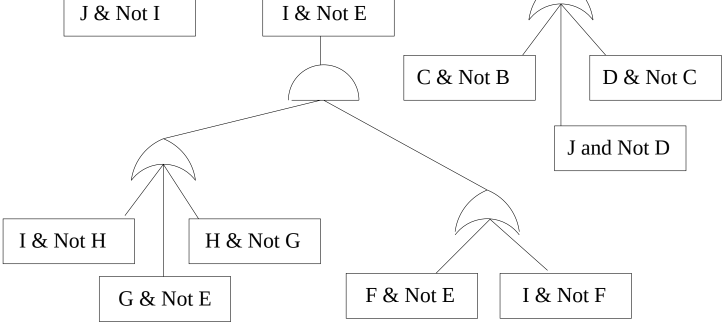


Figure 21: The Complete Fault Tree

The final result of putting all these steps together is shown in Figure 21. Here, the uppermost "OR" node has five children, which arise from fusing the two "OR" nodes in Figure 12 and Figure 14, which corresponds to a Boolean identity arising from the fact that "OR" is an associative and commutative operation. A final point to note is that the "Not J" node in Figure 16 as well as a similarly-labelled node in Figure 19 have been identified with the node labelled "K and Not J" under the top-level "OR" gate. This operation is as follows:

Node Fusion Identify nodes in component sub-trees with any node containing the same "Not"-assertion, when the labels in the node which do not occur in the component sub-tree are omitted.

Using mode fusion, we observe that node "K" appears neither in Figure 16 continuing work on the automatic generator explores various efficiency and other questions of algorithm engineering.

7 A Simplified Fault Tree

We have shown how rigorously to derive a fault tree from a CID. The primary way in which such a generated fault tree may be simplified is through observing various of the intermediate nodes in Figure 21 to be superfluous. That is, they describe a situation which has a trivial logical relation to situations described in other nodes in the fault tree. This simplification procedure remains a human operation for us (in principle one could use some sort of logic checker to test the logical relations between the various faults. We would guess, however, that this would be too much work for the potential benefits it might reap).

7.1 Handling Superfluous Nodes

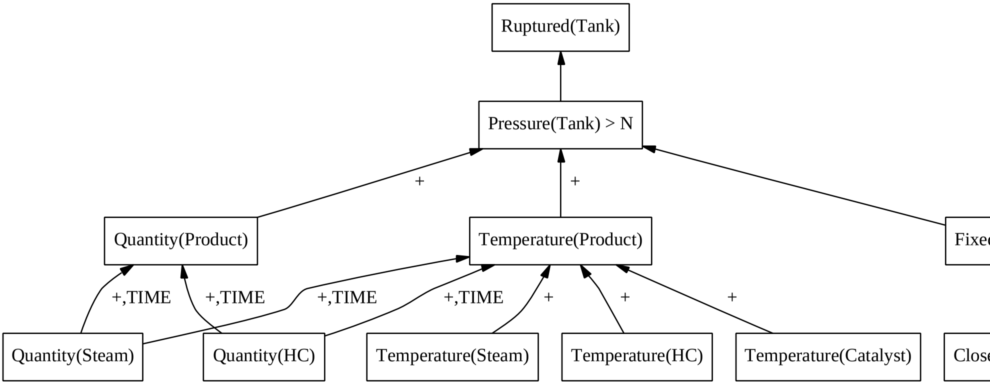
When nodes are observed to be superfluous, they can simply be omitted from the fault tree. It may also be that it is convenient for semantical purposes to leave them in. Thus we prescribe no algorithmic method for dealing with such superfluities. We suggest it is best to inspect the generated fault tree by hand, identify potential superfluities by inspection, and eliminate them if it is felt appropriate to do so.

8 Implementing Fault-Tree Generation

The fault tree in Figure 24 has a number of features worth remarking. First, it is customary in fault-tree generation to label with nodes with short natural-language descriptions of the fault. We do not do this. Instead, we label the nodes wiht a sentence of logic, derived from the formalised language used for describing the CID. We have a simple logical algorithm for generating the labels. Short, intuitive labels for the annunciated faults may be defined through a *glossary*, which uses a series of definitions such as illustrated in the table Figure 25. The labels are best generated by hand, because there is likely to be no simple automatic way to generate an intuitively appropriate name for a particular type of failure. Engineering practice already has conventional names for failures, but to our knowledge there is no algorithmic relation between the conventional labels and the logical form of our automatically generated labels. An algorithm for generating short labels is thus inappropriate, and short labels are best generated through a *glossary* as in Figure 25, because

- engineers are likely to use the intuitive failure labels in their everyday practice, and
- it is cognitively more efficient if the fault tree labels correspond to those used by the engineers in their everyday practice; and
- for reasons illustrated in the formal WBA in [Lad00, Part III], it is appropriate to have able formal logical definitions of everyday terms, for the purposes of reasoning about the faults.

There is no reason to restrict the labelling procedure to the fault tree. It can well apply to the CID, also, and we have in fact applied it, by defining



Ruptured(Tank)
Pressure(Tank) > N
+
+
Quantity(Product)
Temperature(Product)
Fixe
+,TIME
+,TIME
+,TIME
+,TIME
+
+
+
Quantity(Steam)
Quantity(HC)
Temperature(Steam)
Temperature(HC)
Temperature(Catalyst)
Clos

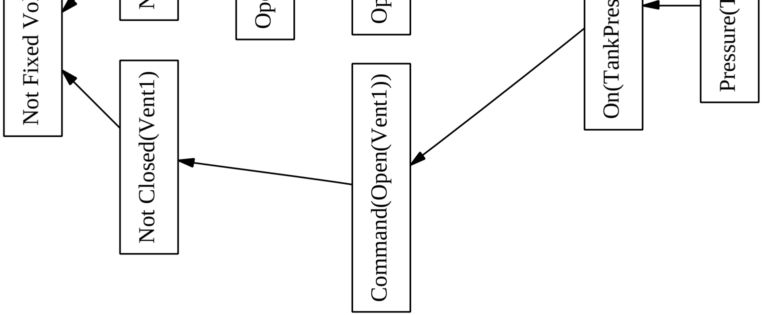
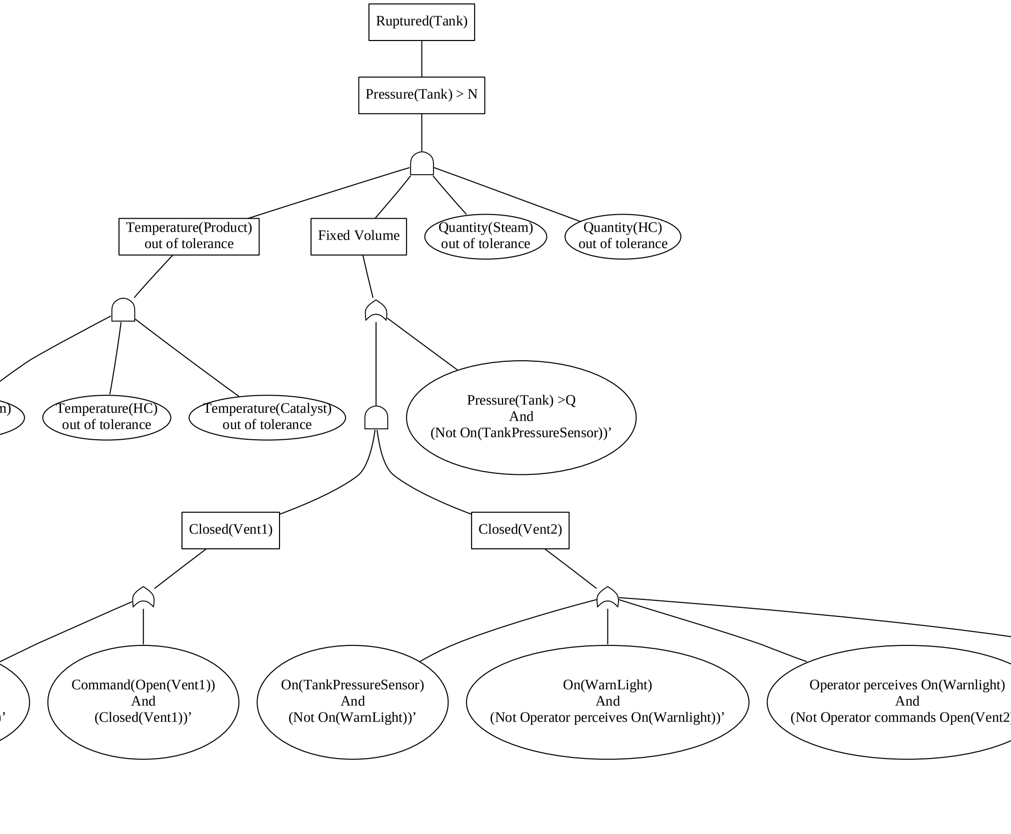


Figure 23: The CID for the Vent Subsystem

with the predicates *Not Closed(Vent1)* and *Not Closed(Vent2)*.[4] Such inter-translation of predicates is precisely the same as choosing labels: logically, one is defining new primitive predicates in terms of other, previously defined, predicates. Thus we enter these into the definition table also, as in Figure 25. The glossary file created as input into the graph-drawing program is shown in Figure 26. The result of substituting the label definitions in the fault tree of Figure 24 is shown in Figure 27.

8.2 The Logical Generation of Labels

There is a straightforward logical procedure (one hesitates to call it an algorithm, because of its simplicity) for generating the precise labels in the fault tree. Each CID diagram, say Figure 23, is already equipped with node labels. Consider a transition, say from *Command(Open(Vent1))* to *Open(Vent1)* in



Ruptured(Tank)
Pressure(Tank) > N
Temperature(Product) out of tolerance
Fixed Volume
Quantity(Steam) out of tolerance
Quantity(HC) out of tolerance
Temperature(HC) out of tolerance
Temperature(Catalyst) out of tolerance
Pressure(Tank) >Q And (Not On(TankPressureSensor))’
Closed(Vent1)
Closed(Vent2)
Command(Open(Vent1)) And (Closed(Vent1))’
On(TankPressureSensor) And (Not On(WarnLight))’
On(WarnLight) And (Not Operator perceives On(Warnlight))’
Operator perceives On(Warnlight) And

Indicator Failure	*On(TankPressureSensor) And Not On(WarnLight)*
Operator Fails to See Indicator	*On(WarnLight) And Not Operator perceives ON(WarnLight)*
Operator Failure to follow procedure	*Operator perceives ON(WarnLight) And Not Operator commands (Open(Vent2)*
Vent2 Failure	*Operator commands Open(Vent2) And Not Not Closed(Vent2)*
Open(X)	*Not Closed(X)*

Figure 25: Label Definitions for the Fault Tree

Figure 23. The CID shows that this transition is expected to take place through the system design. A failure of this intended transition would mean that the postcondition, *Open(Vent1)*, is not achieved, even though the precondition *Command(Open(Vent1))* is fulfilled.

This failure is thus correctly described by saying that the precondition was fulfilled, but the required postcondition did not come to pass. The failure event, then is described by the action formula

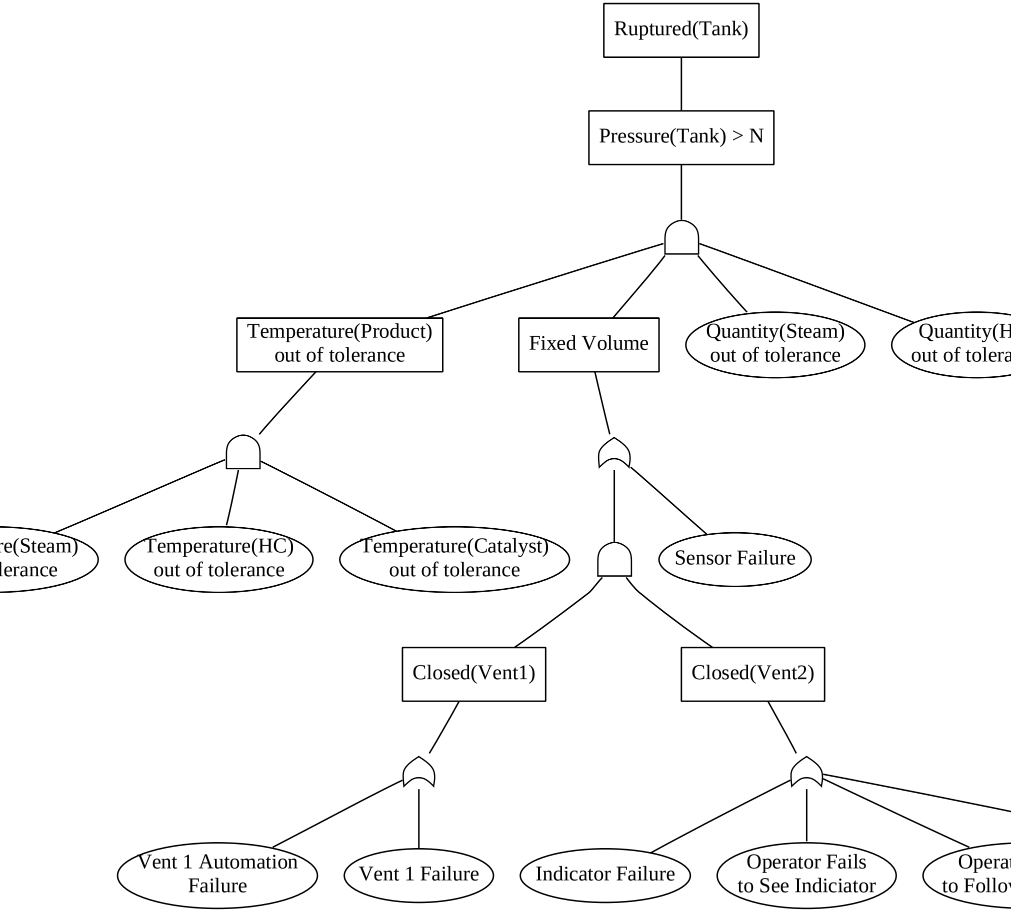
$$Command(Open(Vent1))\ And\ Not\ Open'(Vent1)$$

using the prime notation.

The fault tree generated from this simple example may seem for the example somewhat complicated, especially in comparison with that from [Lev95] in Figure 2. However, it is easy to make mistakes (mainly omissions) when generating a fault tree by hand, and the automatically generated fault

```
# Glossary file automatically created by cid2ft.pl, Version 1.0
#
# Each line consists of two parts, the left one containing the automatically
# generated pre-post conditions, the right one (after the "-->" seperator)
# should be filled in manually with appropriate replacements.
#
# If the part left of the "-->" seperator is changed in any way, the
# automatic replacement will not work.
#
# Lines beginning with # are ignored.
#
# the use of '/n' for line feed is possible.
#
#
# CI-file (main input file): ../Leveson/Leveson_Tank.ci
# glossary file (this file): ../Leveson/Leveson_Tank.gls
#

On(TankPressureSensor) And (Not Command(Open(Vent1)))' -->
Command(Open(Vent1)) And (Closed(Vent1))' -->
Pressure(Tank) >Q And (Not On(TankPressureSensor))' -->
On(TankPressureSensor) And (Not On(WarnLight))' -->
On(WarnLight) And (Not Operator perceives On(Warnlight))' -->
Operator perceives On(Warnlight) And (Not Operator commands Open(Vent2))' -->
Operator commands Open(Vent2) And (Closed(Vent2))' -->
```

Figure 26: Glossary File for the Fault Tree



Ruptured(Tank)
Pressure(Tank) > N
Temperature(Product) out of tolerance
Fixed Volume
Quantity(Steam) out of tolerance
Temperature(HC) out of tolerance
Temperature(Catalyst) out of tolerance
Sensor Failure
Closed(Vent1)
Closed(Vent2)
Vent 1 Automation Failure
Vent 1 Failure
Indicator Failure
Operator Fails to See Indiciator

have the effect that an overpressure situation exists but is not annunciated, and this is likely to lead to the situation in which *PRV 2* does not open, which, in conjunction with a situation in which *PRV 1* does not open, could lead to an explosion, according to the fault tree.

If the pressure monitor for the *Valve 2* subsystem is identical with that for the *Valve 1* subsystem, as in our example in Figure 1, then this is a common-cause failure of both valve subsystems. Such common-cause failures are recognised by the automatic fault-tree generation algorithm, and denoted just once in the appropriate place (in Figure 24, at a disjunctive node above the individual failure nodes *Closed(Vent1)* and *Closed(Vent2)*; of course the failure of the pressure sensor entails both of these predicates, but although they are true, they are not true because of failures of *Vent1* and *Vent2*, which the nodes *Closed(Vent1)* and *Closed(Vent2)* are intended to convey).

If the pressure monitors for the two valve subsystems are different, they would be differently denoted, and failures of each subsystem could well be assumed to be independent, in which case one would place two separate nodes under the respective valve failure nodes *Closed(Vent1)* and *Closed(Vent2)*, as indeed the fault tree in Figure 2 has under the one, but not the other, subsystem failure.

We feel that the automatic raising in such circumstances of common-cause failure is always an appropriate cognitive optimisation to make in a fault tree, for the following reasons:

1. The resulting fault tree makes a logically equivalent statement to the tree with an unraised common-cause failure;

2. The common-cause failure is not denoted through its relations to its causal consequences, but is emphasised early on when descending through

8.3 A Second Example

For a second illustration of the use of this tool, we chose the pressure tank example in Figure 28, taken from the textbook [KH99]. The fault tree given in [KH99] is reproduced in Figure 29. Our development reveals obvious lacunae in this proposed fault tree. The CI-Script for the whole system, for the relief valves, and for the inlet steam pipe, is similar to that in [Lad00, Chapter 7, Causal Analysis of a Pressure Tank], and is shown in Figure 30, Figure 31, and Figure 32, respectively. The CIDs for the whole system, for the relief valves, and for the inlet steam pipe, are shown in Figure 33, Figure 34, and Figure 35, respectively.

The automatically-generated fault tree for the system is shown in Figure 36, and the fault tree with defined labels using the glossary in Figure 37 is shown in Figure 38. This may be compared with the fault tree from [KH99] in Figure 29, to see the difference between a carefully generated fault tree from a believable CID, and a fault tree generated by hand as a toy example. We do not regard the reduced size of the latter as any advantage at all, compared with the extent of the information loss. We found much to criticise in the fault tree in Figure 29, and we think the reader will also.

References

[KH99] Daniel M. Kammen and David M. Hassenzahl. *Should We Risk It?: Exploring Environmental, Health, and Technological Problem Solving.* Princeton University Press, Princeton, N.J., 1999.

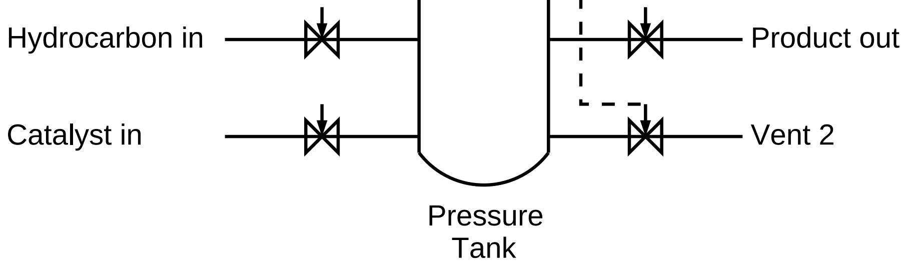
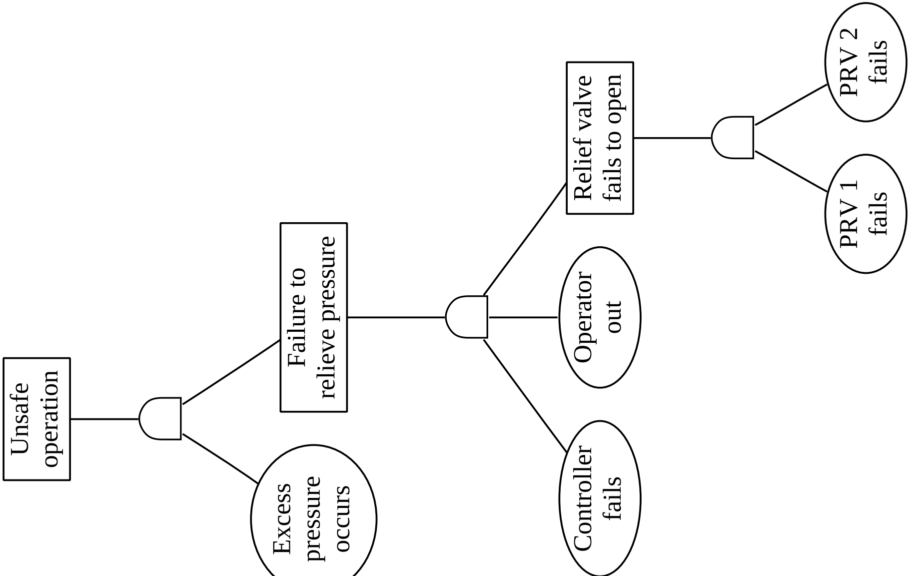


Figure 28: Another Pressure Tank



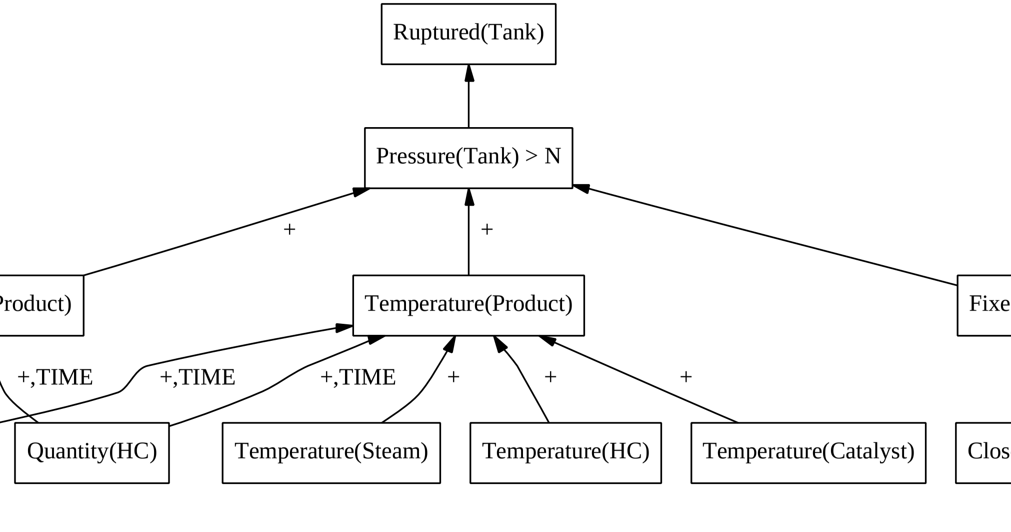
```
[1] /\ [-.1] /* Quantity(Product) // + */
    /\ [-.2] /* Temperature(Product) // + */
    /\ [-.3] /* Fixed Volume // */

[1.1] /\ [-.1] /* Quantity(Steam) // +,TIME */
      /\ [-.2] /* Quantity(HC) // +,TIME */

[1.2] /\ [1.1.1] /* // +,TIME */
      /\ [1.1.2] /* // +,TIME */
      /\ [-.3] /* Temperature(Steam) // + */
      /\ [-.4] /* Temperature(HC) // + */
      /\ [-.5] /* Temperature(Catalyst) // + */

[1.3] /\ [-.1] /* Closed(Vent1) // */
      /\ [-.2] /* Closed(Vent2) // */

[1.1.1] /\ [-.1] /* Open(SteamValve) // */

#include "Steam.ci"

#include "Vents.ci"
```

Figure 30: The Pressure Tank CI-Script

```
[1.1.1] /\ [-.1] /* Pressure(Tank) > N // */

[2]  /\ [-.1] /* Command(Open(Vent2)) // */

[2.1] /\ [1.1.1] /* // */
```

Figure 31: The Relief-Vent Subsystem CI-Script

```
[0] /* Quantity(Steam) // */
[1] /* Not Open(SteamValve) // */ {< A }

[1] /\ [-.1] /* Command(Close(SteamValve)) // */
    /\ [-.2] /* Operator commands Close(SteamValve) // */

[1.1] /\ [-.1] /* On(SteamPipePressureSensor) // */ { A >}

[1.1.1] /\ [-.1] /* Pressure(SteamSupply) > M // */

[1.2] /\ [-.1] /* Operator perceives On(WarnLight) // */

[1.2.1] /\ [-.1] /* On(WarnLight) // */

[1.2.1.1] [1.1.1] /* ignored // */
```


Ruptured(Tank)
Pressure(Tank) > N
+
+
Product)
Temperature(Product)
Fixe
+,TIME
+,TIME
+,TIME
+
+
+
Quantity(HC)
Temperature(Steam)
Temperature(HC)
Temperature(Catalyst)
Clos

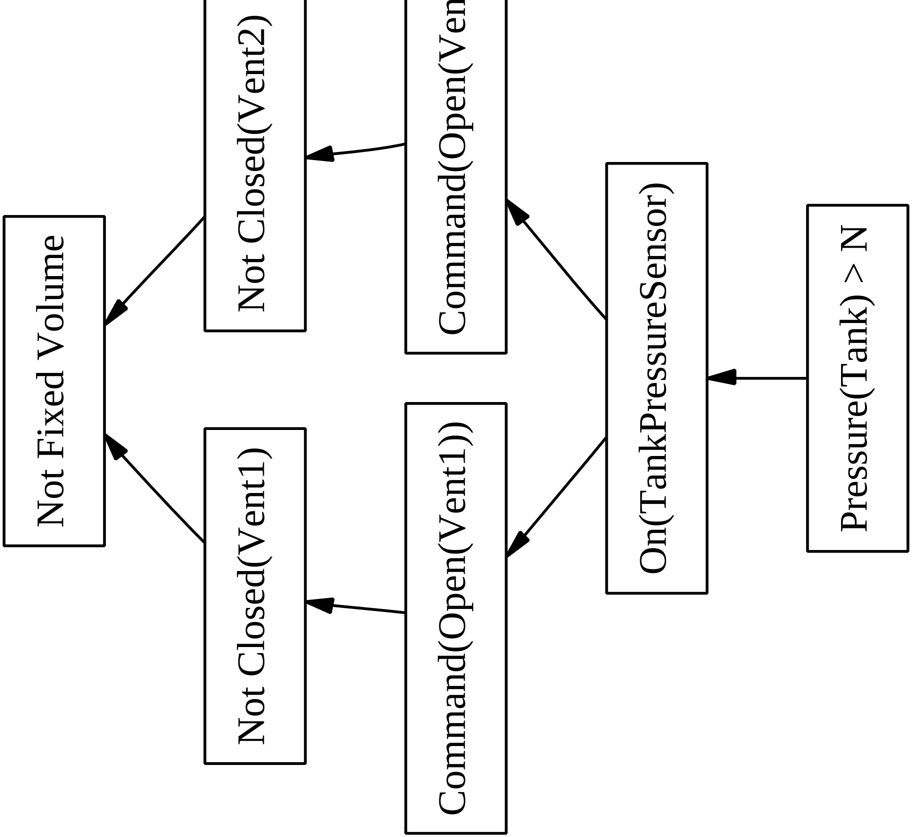


Figure 34: The Relief Vent Subsystem CID

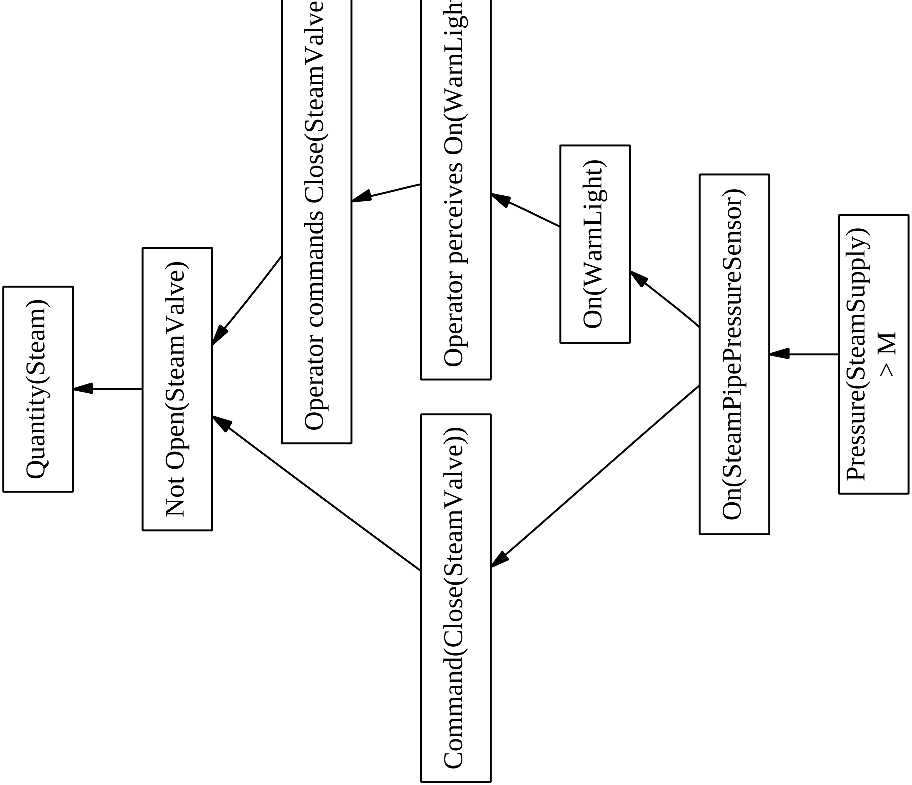
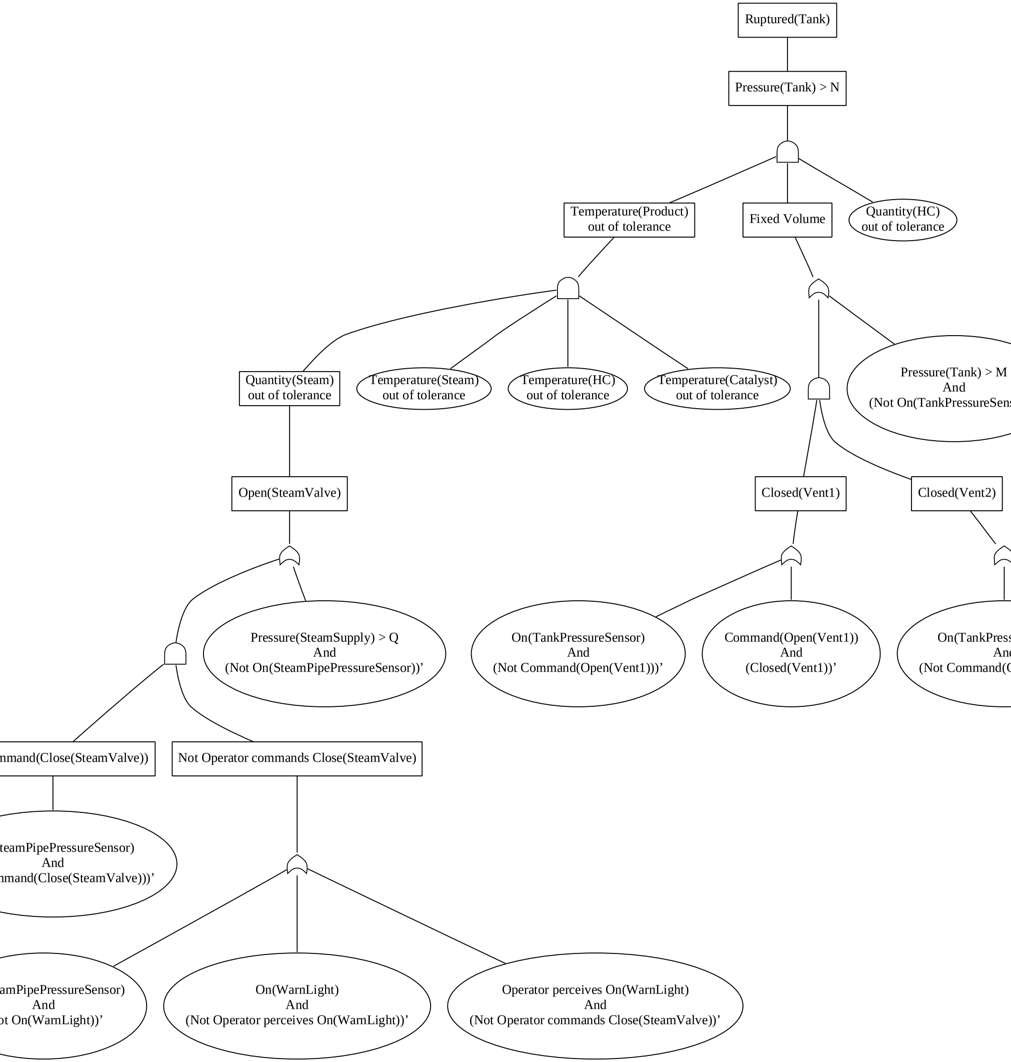


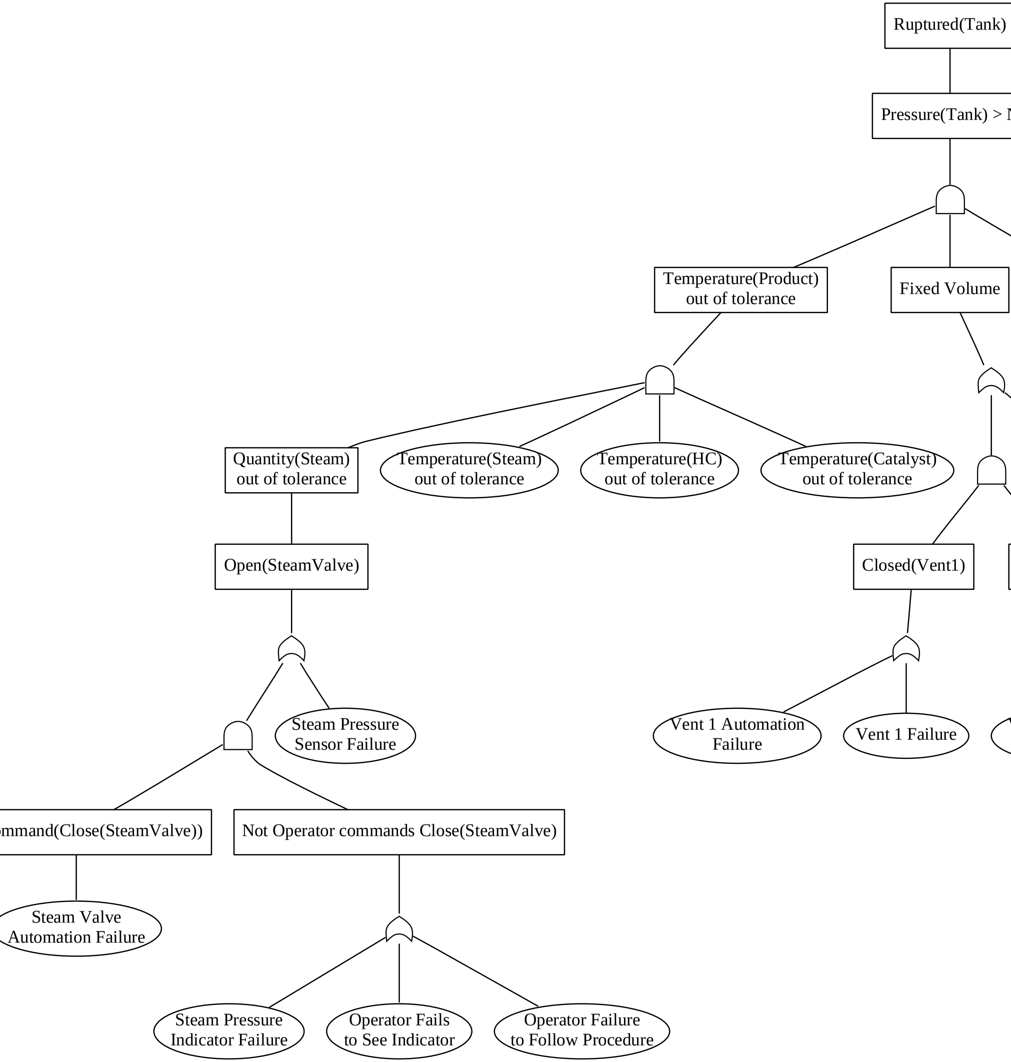
Figure 35: The Inlet Steam Pipe CID



Ruptured(Tank)
Pressure(Tank) > N
Temperature(Product) out of tolerance
Fixed Volume
Quantity(HC) out of tolerance
Quantity(Steam) out of tolerance
Temperature(Steam) out of tolerance
Temperature(HC) out of tolerance
Temperature(Catalyst) out of tolerance
Pressure(Tank) > M And (Not On(TankPressureSens
Open(SteamValve)
Closed(Vent1)
Closed(Vent2)
Pressure(SteamSupply) > Q And (Not On(SteamPipePressureSensor))’
On(TankPressureSensor) And (Not Command(Open(Vent1)))’
Command(Open(Vent1)) And (Closed(Vent1))’
On(TankPress And (Not Command(C
mmand(Close(SteamValve))
Not Operator commands Close(SteamValve)
teamPipePressureSensor) And mmand(Close(SteamValve)))’
amPipePressureSensor) And ot On(WarnLight))’
On(WarnLight) And (Not Operator perceives On(WarnLight))’
Operator perceives On(WarnLight) And (Not Operator commands Close(SteamValve))’

```
# Each line consists of two parts, the left one containing the automatically
# generated pre-post conditions, the right one (after the "-->" seperator)
# should be filled in manually with appropriate replacements.
#
# If the part left of the "-->" seperator is changed in any way, the
# automatic replacement will not work.
#
# Lines beginning with # are ignored.
#
# the use of '/n' for line feed is possible.
#
#
# CI-file (main input file): ../Kammen-Hassenzahl/Tank1.ci
# glossary file (this file): ../Kammen-Hassenzahl/Tank1.gls
#

On(SteamPipePressureSensor) And (Not Command(Close(SteamValve)))' -->
Pressure(SteamSupply) > Q And (Not On(SteamPipePressureSensor))' -->
On(SteamPipePressureSensor) And (Not On(WarnLight))' -->
On(WarnLight) And (Not Operator perceives On(WarnLight))' -->
Operator perceives On(WarnLight) And (Not Operator commands Close(SteamValve))' -
On(TankPressureSensor) And (Not Command(Open(Vent1)))' -->
Command(Open(Vent1)) And (Closed(Vent1))' -->
Pressure(Tank) > M And (Not On(TankPressureSensor))' -->
On(TankPressureSensor) And (Not Command(Open(Vent2)))' -->
Command(Open(Vent2)) And (Closed(Vent2))' -->
```

Figure 37: The Glossary File for the Second Example



Ruptured(Tank)
Pressure(Tank) > M
Temperature(Product)
out of tolerance
Fixed Volume
Quantity(Steam)
out of tolerance
Temperature(Steam)
out of tolerance
Temperature(HC)
out of tolerance
Temperature(Catalyst)
out of tolerance
Open(SteamValve)
Closed(Vent1)
Steam Pressure
Sensor Failure
Vent 1 Automation
Failure
Vent 1 Failure
mmand(Close(SteamValve))
Not Operator commands Close(SteamValve)
Steam Valve
Automation Failure
Steam Pressure
Indicator Failure
Operator Fails
to See Indicator
Operator Failure
to Follow Procedure

[LR] Peter Ladkin and RVS Group. RVS Group Publications. RVS Group, Technische Fakultät, Universität Bielefeld. Available through `http://www.rvs.uni-bielefeld.de`.

[VGRH81] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. Number NUREG-0492 in Report. U.S. Nuclear Regulatory Commission, Washington, D.C., January 1981.