THE UNIVERSITY OF HULL

Pandora: A Logic for the Qualitative Analysis of Temporal Fault Trees

being a Thesis submitted for the Degree of Doctor of Philosophy

in the University of Hull

by

Martin David Walker

May 2009

Pandora: A Logic for the Qualitative Analysis of Temporal Fault Trees

Abstract

Fault Tree Analysis (FTA) is a valuable systems analysis technique widely used in safety analysis and reliability engineering, but it is not without its faults; in particular, it struggles to analyse systems in which time plays an important role, because fault trees use only Boolean logic and so there is no simple way of representing time or sequences of events in a fault tree. Although there have been attempts to extend FTA to enable analysis of such systems, most have focused on probabilistic analysis and there remains a need for a technique that allows logical analysis of dynamic systems.

Pandora is a technique that aims to provide a solution to this problem. It is based around three logical gates capable of representing sequences: the Priority-AND (PAND) gate, the Simultaneous-AND gate (SAND), and the Priority-OR gate (POR). These three "temporal" gates are more expressive, allowing analysts to model sequences as part of a fault tree and thus enabling fault trees to analyse more complex dynamic systems.

In addition, Pandora provides a set of logical rules that can be used to reduce fault trees incorporating the three new gates in much the same way that existing Boolean laws can be used to reduce ordinary fault trees. This makes it possible to perform logical analysis of fault trees using Pandora, the results of which provide the analyst with information about the weak points of the system by showing what combinations or sequences of event can cause the system to fail.

This thesis presents the evolution of Pandora thus far, explaining the background that led to its inception and the choices made during its development as well as detailed explanations of how Pandora is applied. Pandora has been created with possible automation in mind, so there is also a description of some preliminary algorithms that support Pandora-based FTA. Pandora is then applied to a case study to demonstrate how it can function in practice. Finally, the success of Pandora is evaluated by contrasting it with other temporal FTA approaches as well as standard non-dynamic analysis and from this conclusions about the potential benefits of using Pandora are drawn.

Acknowledgements

The long journey is complete, the road is at its end – I have arrived at last in Ithaka[1]. But I could not have done it alone.

The one who first set me upon this path deserves first acknowledgement: Mr Noel Barron, who twenty years ago showed me how to use a computer for the first time and in doing so opened up an entirely new world, one that I have been exploring ever since. Without his foresight and wisdom, I may never have started this journey, let alone come this far.

Since then, there have been others along the road who have played vital roles in getting me here. I am deeply indebted to Kate Miller, Anne Harkness, and Marcia Healey – paragons of the teaching profession – for instilling within me a love and understanding of the English language without which this thesis would not have been possible. I also owe Alan Hunter and Graham Cardwell a great debt of gratitude for giving me the opportunity to leave my cosy shell and see what this life thing was all about, and I owe a similar debt to the illustrious trio of James Mackenzie, Mike Singleton, and especially Peter Mackenzie for their invaluable friendship and guidance during that critical time – may your jelly beans forever be delicious.

I am also grateful to Guillaume Merle, Dr Qingde Li, and particularly to Dr Len Bottaci, all of whom have helped to enrich Pandora and ensure my logic was sound – no small task. Birgit van Dalen – mathematician par excellence – deserves huge thanks for putting up with my inane questions and for finding the Fubini numbers for me. I would also like to thank Helen El-Sharkawy, Colleen Nicholson, and Joan Hopper, who keep the Computer Science Department running like clockwork and who patiently help solve all of the administrative problems (often of our own making) that we idiot students so frequently encounter.

No journey is complete without meeting fellow travellers along the way, and in my case those would be Ian Wolforth, Septa Sharvia, Nidhal Mahmud, and especially David Parker. Their support and friendship through good times and bad have been appreciated more than they perhaps realise, and I will always look back fondly on those halcyon days of shared frustrations, HiP-HOPS bugs, and Triangles battles, back when our lab still had its own bathtub.

But my greatest debt is to my supervisor and mentor, Dr Yiannis Papadopoulos. I have never been able to find the right words to express my gratitude for all he has done for me – for giving me the chance to enter the world of academic research, for guiding me through that world and

[1] See *Ithaka*, by Constantinos Cavafis.

giving me the freedom to explore it in my own way, and for tolerating my many eccentricities with good humour. Without him, Pandora would not exist and I would have missed out on an incredible opportunity; no student could ever wish for a greater teacher. I will always thank him for showing me the road to Ithaka.

Finally, I would like to give my deepest thanks to my family – David, Christine, and Michael – for putting up with me for so long, and to my friends – Daniel Casslasy and Daniel Winter in particular – for keeping me sane along the way. I don't know how you all did it; if I were you, I would have driven myself mad. Thanks everyone.

And so it only remains to say to you, brave reader, that I salute your courage in opening this colossal tome and wish you well in your voyage through its pages. I only hope that your journey is as fruitful as mine was.

Contents

Figures

Tables

1 Introduction

"The only people who find what they are looking for in life are the fault finders."
- Foster's Law

1.1 Setting the scene

We live in an age of ever more sophisticated machines, machines designed to work for us, relieve us of our burdens, and thereby improve our lives. The price we pay for this convenience is an increasing dependence on those machines and, correspondingly, increasingly dire consequences should any of those machines go wrong. We frequently entrust our lives to these creations, relying upon the hope that those who built them did not make any mistakes. This dependence does not go unrecognised, however; the systems on which we depend the most – those with the worst consequences should they fail – are known as **safety critical systems**, and it is vital to ensure they do not fail catastrophically. Safety critical systems include things as far apart as propulsion systems on spacecraft to airbags in cars, but they all have one thing in common – the potential to cause great harm to people if they fail.

This imperative has led to the field of **reliability engineering**, a discipline devoted entirely to making systems as safe and as reliable as possible. The principle of reliability engineering is to build systems with a certain level of minimum reliability in their design; the more reliable or robust a system is, the less likely it is to fail and therefore cause harm. However, in order to create reliable systems, we also need to be able to understand how they work and how they can fail, and to do that we need to study them: a process known as **systems analysis**.

Systems analysis allows reliability engineers to gain an insight into how a system can fail and therefore determine what measures should be used to prevent that failure. There are many systems analysis techniques in use, but one of the foremost is **fault tree analysis** (FTA). FTA is used to discover the relationship between a system fault and its root causes. By illustrating the connections between combinations of relatively minor failures and their effects on the whole system, FTA makes it easier to design the system to withstand such failures by preventing those combinations of failures. FTA itself is not new; it dates back to the 1960s and has been put to good use in many industries, including the aerospace, automobile, nuclear, and defence industries (Ericson, 1999).

FTA is not without its own faults, however. In particular, FTA struggles with the representation and analysis of *time*, or at least system behaviour that depends on time in some way. Fault trees model the static causal relationships between individual components in a system and the effect their failure has on the system as a whole, either singly or in combination with other failures. For many systems this is sufficient, but for some this model is too simplistic to capture the full failure behaviour. For example, some systems are capable of reacting to and recovering from certain failures, perhaps by activating a standby component or some other corrective measure; in other systems, the potential failure behaviour varies according to what state the system is in. These are **dynamic systems**, and this dynamic behaviour is difficult to model accurately using FTA. As systems become increasingly dynamic and correspondingly more complex, it becomes ever more important to find ways of overcoming this deficiency in FTA.

As a simple example of a dynamic system, consider the generic standby redundant system in Figure 1 (originally used in Walker & Papadopoulos, 2006).

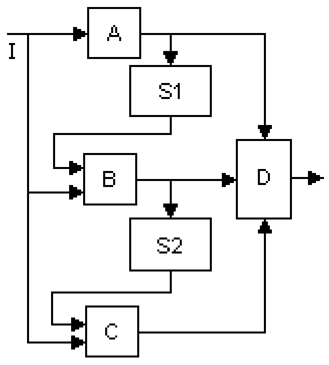


Figure 1 - Simple example of a triple-module redundant system

This system is generic in the sense that components A, B and C could be any input, control or actuating device. A is the primary component that takes input from outside the system (labelled "I"), performs some function on it, and feeds the result to the system output (D). S1 is a monitoring sensor that detects any omission of output from A and activates the first standby component, B. Similarly, S2 is another sensor that detects a failure of B and actives the second backup, C. In theory, all three components A, B, and C must fail for the whole system to fail – the failure of one or two of the three can be tolerated. Performing a classical fault tree analysis on this system seems to confirm this, showing that the system only fails if:

1. There is no input at I.
2. All of A, B, and C fail.
3. Both A and S1 fail.
4. All of A, B, and S2 fail.

Each of these four events or combinations of events is sufficient to cause the whole system to fail. This behaviour can be seen in the following state diagram, with cases 2-4 shown:

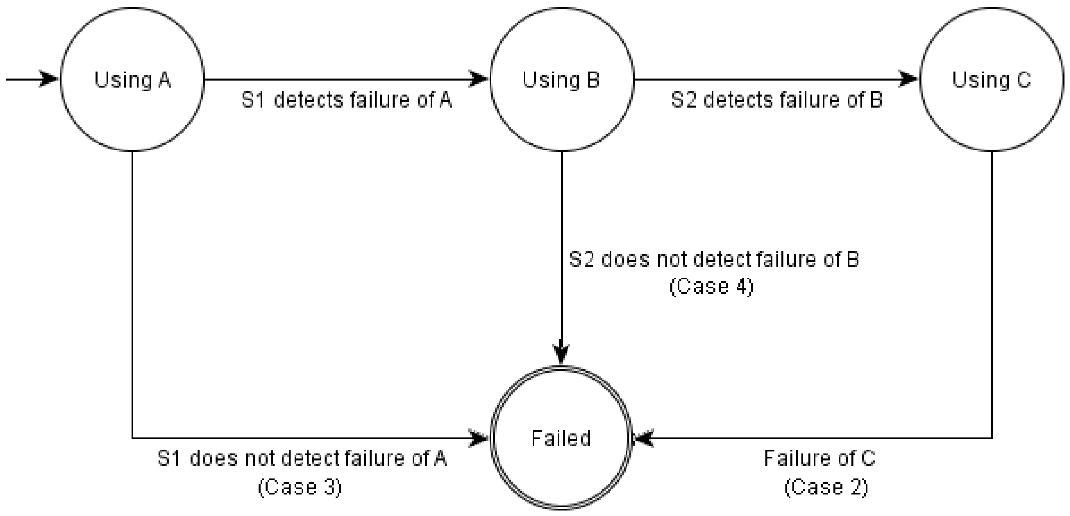


Figure 2 - State diagram showing the behaviour of the example system

However, this is a rather naïve view of things; in reality the situation is more complex. Consider cases 3 and 4: what if one of the sensors fails *after* the other components? For example, in case 3, a failure of A and S1 will cause system failure. Yet if S1 fails *after* A, then it does not lead to a system failure because S1 has already performed its function (to detect failure of A) and activated standby component B. Once B has been activated, failure of S1 is irrelevant - it can no longer have any effect on the functioning of the system. Similarly, in case 4, if S2 fails after both A and B have failed, then the second backup component C will have been activated, and there is no system failure. Again, S2 will have served its purpose, and any subsequent failure has no effect on the rest of the system, which is now operating on component C alone. Both cases 3 and 4 are therefore unnecessarily pessimistic: they suggest that a failure of the sensor will *always* cause the system to fail, when in fact this is only true in certain situations and is dependent upon the order in which the components fail.

Conversely, since the sensors work by detecting an omission of output, if the first standby component B fails before A, then S2 would never detect a cessation in output from B, because it would never be activated; C therefore remains unused. This means that the second case, failure of all of A, B, and C, is then a dangerously optimistic result, since B failing before A is sufficient to cause the system to fail, regardless of the status of component C.

This more in-depth view of the system failure behaviour is shown in Figure 3:

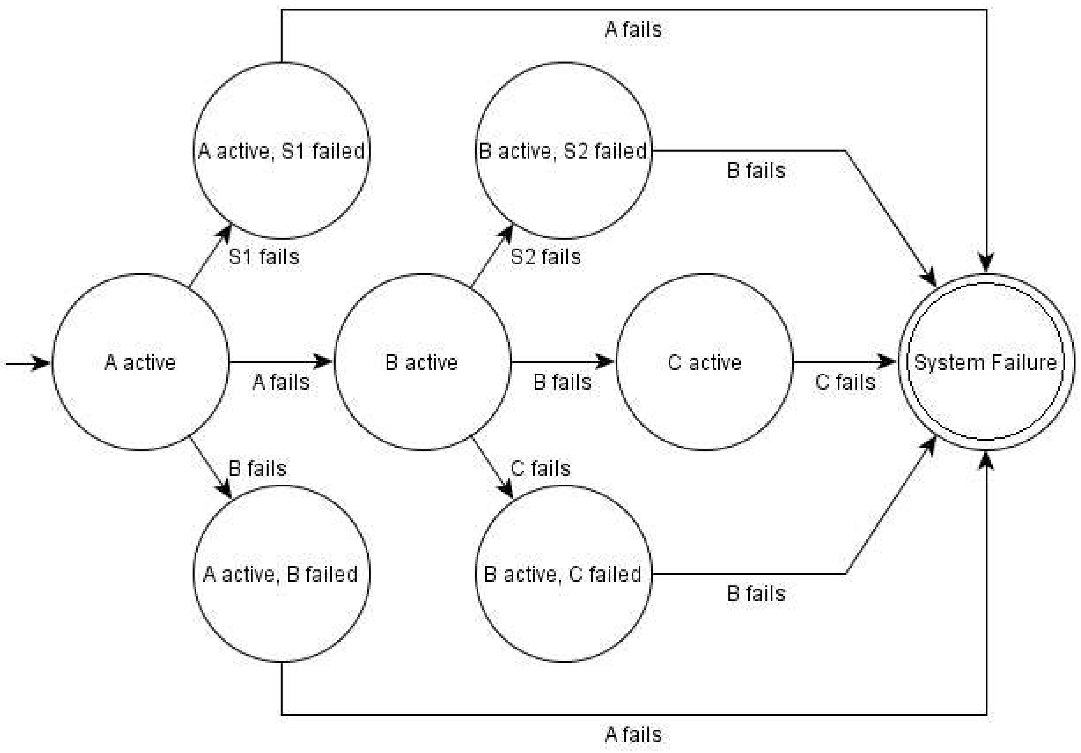


Figure 3 - A more detailed look at the failure behaviour of the system

Here it is easy to see how a premature failure of a sensor or of a backup component can lead to system failure without later backups being activated[2]. Classical fault tree analysis is unable to model this type of dynamic behaviour and would therefore fail to accurately capture the full failure behaviour of this system. FTA models only the effects of the occurrence of faults without also taking into account the effect that the *sequence* of those faults can have. Instead, the results obtained are either unnecessarily pessimistic or excessively optimistic.

This raises an important question: if a system with just five components can have such pronounced inaccuracies, how far can we trust an analysis of a system with hundreds of thousands of components, such as a nuclear power plant?

[2] There are even more possible sequences not shown in Figure 3, but these are omitted for the sake of clarity since most of them are redundant (for example, the sequence "failure of S1 followed by C followed by A" is effectively equivalent to "failure of S1 followed by A", which is shown in the diagram, since the failure of C has no effect in this case).

1.2 Pandora: a potential solution

Clearly, there is a deficiency in FTA, and the goal of this thesis is to present a potential solution called **Pandora**, which aims to solve the problem by extending FTA with new capabilities. Pandora was first introduced in *Project Pandora: Temporal Fault Tree Analysis* (Walker, 2005) and is based on the principles embodied by the Priority-AND gate, a relatively obscure and oft overlooked member of the fault tree vocabulary. The Priority-AND gate introduces the concept of *event sequences* to a fault tree, so that a failure may depend on certain contributing events occurring in a specific order. Unfortunately, as will be explained in Chapter 2, the original Priority-AND gate suffers from several problems ranging from an ambiguous definition to a lack of information on how to use it in an analysis.

Pandora aims to overcome these issues. It builds on the foundations laid by the original Priority-AND gate but takes the idea further, introducing two new temporal gates to enable a more complete expression of event sequences and introducing a set of rules and algorithms to allow for these temporally-augmented fault trees to be analysed in ways similar to ordinary fault trees. The result is a more powerful and more expressive fault tree methodology that can be applied to a greater range of systems and obtain more accurate results.

In summary:

> **Classical Fault Tree Analysis is insufficient for the analysis of complex dynamic systems in which the failure behaviour is dependent upon the order of events; Pandora solves this by extending FTA with new temporal capabilities to enable the analysis of sequences as well as combinations of failures.**

The subsequent chapters will explain why Pandora is needed, what it contains, how it works, and finally whether it manages to achieve this aim.

1.3 Objectives

"A bad ending follows a bad beginning."

- Euripides, *Melanippe the Wise*

During the original development of Pandora, a number of key objectives were identified that any solution to the problem of FTA of dynamic systems would ideally fulfil. Before these are introduced, however, it is first necessary to define the scope of the problem.

FTA covers two complementary types of analysis that are usually used in tandem but may be used separately. The two types are **qualitative analysis** and **quantitative analysis**. The former is a purely logical form of analysis that involves examining and manipulating the Boolean structure of the fault tree to obtain a set of **minimal cut sets**, the smallest possible combinations of events that can cause the system to fail. The size of each of these minimal cut sets can be used as a crude approximation of how likely it is to occur (going by the assumption that it is less common for two events to occur in conjunction than for either to occur on its own) and is known as the **order** of the cut set. The other form of analysis is a probabilistic analysis designed to elicit more information about the likelihood of failures occurring. This involves assigning additional data such as failure rates or repair rates to component failure events and then calculating the probability of system failures from the failures of their constituent events. It is usually performed after qualitative analysis, since it can take advantage of the minimal cut sets, and its output is a probability for the top event (system failure) of the fault tree.

Each type of analysis has disadvantages and advantages, which is why they are usually conducted together. Quantitative analysis has the advantage of providing mathematical data and therefore being more precise, but it typically relies on a prior qualitative analysis (though there are other methods) and requires a considerable amount of additional data. Qualitative analysis may be more abstract, but that means it is possible to conduct an analysis on a system for which no failure data is available, or even for which no failure data is possible, such as an abstract functional model of a system. Both forms of analysis can be used during a system design process. In an iterative process, for example, a qualitative analysis can be applied to early models where the choice of components has not been made and thus exact failure data is not available, and then a more detailed quantitative analysis can be conducted later, once the design has been developed further and actual components have been chosen for which failure data exists.

As will be seen in the next chapter, there has been some research directed at the problem of quantitative analysis of dynamic or temporal fault trees, including quantification of the Priority-AND gate, but much less attention has been paid to the problem of temporal qualitative analysis. This limits temporal FTA to systems for which failure data is available, which is not always the case. Since qualitative analysis is possible whenever quantitative analysis is possible, but not vice versa, it seems that the area of temporal qualitative analysis is underdeveloped and thus Pandora was created to help correct that omission.

Therefore, this thesis focuses primarily on the problem on temporal qualitative analysis in FTA and the issues and requirements associated with that form of analysis. Quantitative analysis will be discussed later as a possible avenue for further exploration.

Any temporal qualitative analysis technique must fulfil certain key requirements. Firstly, it must be possible to represent temporal or sequential information in the fault tree in some way. Secondly, there must be some logic or semantics underlying that information for it to make sense and be useful in an analysis. Finally, it must be possible to make use of that data in an analysis in order to be able to draw accurate conclusions about the system failure behaviour. To these key requirements, other, more subjective, requirements can be added. FTA is a well-established and popular technique and so any extension of it should attempt to remain in keeping with the spirit of FTA, retaining the features that make FTA successful. Additionally, qualitative analysis provides results in the form of minimal cut sets, so a temporal qualitative analysis technique should aim to remain compatible with them.

It is from these observations that the objectives of Pandora are defined:

1. **Retain, as far as possible, the simplicity of FTA by requiring only a minimum of additional, temporal data.**

This objective relates heavily to the notion of time itself and how it can be represented in a fault tree. The relevant issues will be covered in the next chapter, but this is the foundation that determines the nature of whatever methodology is built upon it. The goal is not to overcomplicate FTA by burdening the analyst with the requirement for an abundance of additional information. Fault trees are simple and flexible and any extensions to them should therefore ideally be simple and flexible also.

2. **Remain compatible with the existing fault tree structure by minimising the impact of any extensions.**

Fault trees have a clear logical structure: a tree containing intermediate Boolean gates and leaf nodes typically representing failure events in a system. The goal is to extend this structure, not replace it; any additions should remain faithful to the Boolean nature and aesthetic of fault trees as far as possible. This means that the extensions should ideally take the form of leaf nodes or intermediate gates so that they can form part of a standard fault tree, and it is therefore important that the temporal extensions can be mixed with normal Boolean gates too. Temporal extensions ought to be another tool the analyst can draw upon when necessary, not something they are forced to use all the time.

3. **Allow qualitative analysis of the extended fault trees, producing results similar to those already provided by FTA.**

As mentioned above, qualitative analysis is the more flexible form of FTA as it can be performed even in the absence of probabilistic failure data. Although existing qualitative analysis techniques are unlikely to be able to analyse a fault tree that has been augmented with temporal information, it may be possible to extend one or more techniques or alternatively produce a new temporal qualitative analysis technique. The goal of qualitative analysis is to find out which combinations of events can cause the system failure at the root of the fault tree, so in the case of temporal fault trees, it should be possible to determine what effect the sequences or timings of events also have on the occurrence of the system failure.

Furthermore, the output of normal qualitative analysis is a list of minimal cut sets. These can be ranked according to size and used to draw conclusions about the failure behaviour of the system and thus how best to improve the reliability and safety of that system. For any temporal extensions to FTA, it should be possible to produce outputs with a similar form and similar function, showing the analyst how the sequence of events in the system can lead to a system failure – because by preventing that sequence of events, it becomes possible to prevent the system from failing in that manner.

4. **Provide a temporal logic that underlies the extensions to support qualitative analysis.**

Existing qualitative analysis techniques are typically based on Boolean logic. Since Boolean logic is insufficient to represent dynamic or temporal information in fault trees, for temporal qualitative analysis to be possible, there has to be some form of temporal logic that underlies the extensions to the fault tree and enables qualitative analysis.

5. **Define a simple, unambiguous definition of the meaning of the temporal relationships between events to resolve the issue of contradictions.**

The Priority-AND gate can potentially introduce contradictions to a fault tree, e.g. (X PAND Y) AND (Y PAND X). Any temporal fault tree methodology that allows the representation of sequences in this way should therefore be able to deal with such contradictions. The precise definitions of the gates and events is also important to ensure that other ambiguities do not arise, e.g. the possibility of events occurring simultaneously.

These objectives provide a metric with which to judge the success of Pandora and also provide some form of benchmark to compare other techniques against (though it is important to note that not all temporal FTA techniques were necessarily created with these goals in mind). They also shape the development of Pandora and help explain why certain choices were made in Pandora.

1.4 Thesis Structure

This thesis is structured in a specific sequence designed to introduce the various components of Pandora in the appropriate order:

1. Introduction

This Introduction explains the *why* of the thesis by introducing the key problem to be solved: the inability of FTA to represent time or event sequences in fault trees. It also introduces the goals and objectives of Pandora and sets out the structure of this thesis, starting with a brief overview of the Introduction, which contains a self-referential section describing the structure of this thesis.

2. Background Information

The Background chapter provides a foundation for understanding the rest of the thesis. It begins with an introduction to Fault Tree Analysis itself and explains in further detail why it cannot analyse dynamic systems adequately. It then launches into a brief discussion of time before going on to describe some of the other solutions to this problem and their various advantages and disadvantages. The conclusions gained from this background study have shaped the course of the development of Pandora.

3. Pandora

This chapter is the *what* of the thesis: it introduces Pandora itself, explaining the basic concepts underlying it and describing the various components involved, including the three temporal gates and the temporal laws that connect them. This information is necessary for full understanding of the subsequent chapter.

4. Temporal Analysis

This chapter covers the *how* of the thesis: the process of temporal qualitative analysis using Pandora. It introduces the ideas of minimal cut sequences and doublets and also introduces Euripides and Archimedes, which are algorithms for the reduction of Pandora-augmented fault trees using the various temporal laws. The process is illustrated on the simple example system from Figure 1. In addition it discusses the possible automation of this process with reference to the obstacles posed by the formidable Fubini numbers.

5. Case Study

Having provided algorithms for the production of minimal cut sequences, this section illustrates how Pandora can be used in practice by applying it to an automotive brake-by-wire system. The analysis is explained step-by-step and the results discussed. These results are contrasted against the results obtained from a standard, non-temporal analysis, showing how Pandora can provide a greater degree of accuracy when analysing dynamic systems.

6. Evaluation

The penultimate chapter evaluates the achievements of Pandora. It analyses the successes and failures of Pandora with reference to the objectives set out above, as well contrasting it with some of the other alternatives mentioned in Chapter 2. It also includes a section on the potential for further research into Pandora.

7. Conclusions

This final chapter summarises what has been achieved in Pandora.

References

A list of references used in this thesis.

Appendix I: Glossary

An explanation of some of the commonly used terms and abbreviations found in this document (usually those terms which first appear in bold).

Appendix II: Temporal Laws

This appendix contains a list of all the temporal laws in Pandora.

Appendix III: The Adventures of Martin in Pandora Land

A brief glimpse of what the development process of Pandora looked like.

1.5 A Note on Publications

Some of the material in this thesis has already been published (in condensed form) in a number of conference and journal papers. This thesis is built upon the work begun in Walker (2005) and the material in Chapter 3 and to a lesser extent Chapter 4 forms the basis for Walker & Papadopoulos (2006), Walker & Papadopoulos (2007), Walker & Papadopoulos (2008) and Walker & Papadopoulos (2009). A simplified version of the automotive braking system case study from Chapter 5 is used as the basis for Walker *et al.* (2009). Formalisations of Pandora can be found in Walker, Bottaci, & Papadopoulos (2007) and Merle & Rousseau (2007).

2 Background

"If we had no faults, we should not take so much pleasure in noting those of others."
- François de La Rochefoucauld

2.1 Brief Introduction to Reliability Engineering

"Forgive, son; men are men; they needs must err."
- Euripides

Before delving into the inner depths of fault trees and temporal analysis, it is helpful to explain the process of reliability engineering and to establish definitions for some of the common terms used. First, what is '**reliability**'? Generally, something is reliable if it is unlikely to fail, and it is defined more precisely by the IEC as "the ability of an item to perform a required function under stated conditions for a stated period of time." (IEC 271, 1974). Here, 'item' is used to refer to any piece of equipment, component, system, or subsystem that can be considered as an individual unit and subjected to tests and analysis. Reliability is usually represented as the probability that an item is in its normal (i.e. not failed) state from a starting time to time *t*, assuming that the item was new or as good as new at the starting time (Sundararajan, 1991, pg. 51):

$$R(t) = P(\text{item does not fail during period } [0,t])$$

Note that reliability is defined as the ability to perform a function; therefore, if an item has multiple functions, it may also have multiple reliabilities associated with those functions. The complement of reliability is unreliability, which is therefore the probability that an item will be unable to perform its function(s) and is equal to 1 – R(*t*).

Another two widely used terms, especially in FTA, are **availability** and its complement **unavailability**. Availability is defined as "the probability that the component is in its normal state at time *t*, given that it was new or as good as new at time zero." (Sundararajan, 1991, pg. 51). Thus while reliability is the probability that an item will be functional over a period of time, availability is the probability that an item will be able to perform its functional at any given point in time. Its complement, unavailability, is usually represented by Q (to distinguish it from unreliability U) and is defined as 1 – A(*t*) where A represents availability.

Note that the precise meaning of these definitions differs for repairable and non-repairable items: if an item is non-repairable, then its availability and reliability will be the same, but if an item can be repaired, then its reliability and availability may differ, as once an item fails it can then later be repaired. This results in an improved availability (since a repaired item can still perform its function) but does not improve the reliability (since the item still fails).

The **failure rate** λ of an entity is the "rate at which failure occurs during a specified interval" (Sundararajan, 1991, pg. 51), generally expressed as a number of failures per time period (e.g. failures per hour). The **repair rate** μ is "the probability density (that is, probability per unit time) that the component is repaired at time *t* given that the component failed at time zero and had been in a failed state (that is, the component is not yet restored to service) to time *t*." A common assumption made is that failure rate and repair rate are independent of time, i.e. they are constant. Many components are assumed to have a constant failure rate once they are in the prime of their lives, i.e. a state of steady operation; this follows a 'break-in' period of higher failure rates (due to undetected design defects, manufacturing problems, or installation errors etc) and precedes an end-of-life 'wear-out' period of higher failure rates due to the deterioration of components as they age and near the end of their designed life spans. This is the so-called 'bathtub curve'. Systems are also often assumed to have a constant failure rate as if a system or subsystem contains multiple subcomponents, each with different failure rates, the overall failure rate of the system is approximately constant (Sundararajan, 1991, pg. 62). Similarly, it is valid to assume a constant repair rate if the Mean Time To Repair (see below) is much smaller than the Mean Time To Failure.

MTTF is the **Mean Time To Failure**. It represents the average lifetime of a component (assuming it is non-repairable) or the average time before first failure (if it is repairable) and is "the expected value of the time at which the component fails, given that it was new or as good as new at time zero" (Sundararajan, 1991, pg. 51). MTBF or Mean Time Between Failures is also used in the case of repairable components. MTTR is the **Mean Time To Repair**, the average duration of time between the moment a component fails and the moment it becomes operational again after repair (including time for detection and testing of the repaired component). MTBR is the Mean Time Between Repairs, which is the average time between the start of one repair and the start of the next. When failure rate and repair rate are constant, then MTTF and MTTR are inversely related to them:

$$MTTF = \frac{1}{\lambda} \qquad\qquad MTTR = \frac{1}{\mu}$$

In which case, unavailability can also be calculated like so:

$$Unavailability = \frac{\lambda}{\lambda + \mu} = \frac{MTTR}{MTTR{+}MTTF}$$

There is one other concept that deserves particular mention: **safety**. Safety is defined as "the ability of an entity not to cause, under given conditions, critical or catastrophic events" (Villemeur, 1992a). By improving the reliability of an entity, we reduce the chance of it failing; in doing so, we also reduce the chance of a critical event occurring as a result of that failure. It is however possible for a system to be safe but unreliable (if it causes no critical or catastrophic events) or unsafe but reliable (if it can cause catastrophic events, but only infrequently).

This concept is represented by **risk**, a combination of reliability and safety. Risk is the potential harm an entity may cause and is usually represented as the product of the severity of the consequences of each failure (i.e. a measure of how safe it is) and the probability of that failure occurring (a measure of reliability). Risk can be minimised by either reducing the likelihood of failure or improving the safety by lessening the severity of the potential effects of a failure.

Reliability engineering is therefore the process undertaken to try and improve the reliability of an entity, either in order to minimise the risk it poses or simply to reduce cost and improve efficiency. Through analysis, it is possible discover where the main flaws lie and therefore take steps to improve the design by amending those flaws. For example, after identifying which components are the weak points of the system (i.e. those most likely to cause system failure), they can be replicated to provide redundancy, replaced with better quality components, or simply omitted during a subsequent revision of the design. There are many different techniques available to perform these analyses, often known as **systems analysis** techniques, each of which works in different ways. However, they generally share a common goal: to discover how reliable an entity is so that it can be improved if necessary. According to the *Fault Tree Handbook*, systems analysis is:

> *"a directed process for the orderly and timely acquisition and investigation of specific system information pertinent to a given decision."* (Vesely *et al.*, 1981, p I-2)

Ultimately, systems analysis is a process by which we can discover information about a system that we can then use to make a decision. If this information we obtain happens to be an assessment of system reliability, we can use that information to decide how to improve that system. This is the basis of most reliability analysis techniques, including Fault Tree Analysis.

Before we consider this further, however, it is necessary to understand what we mean by a system in this context. A system implies some form of organised group of elements that work together in some way, and the *Fault Tree Handbook* kindly provides us with another formal definition:

> *"A system is a deterministic entity comprising an interacting collection of discrete elements."* (Vesely *et al.*, 1981, p I-4)

Therefore, the primary role of systems analysis, and by extension FTA, is to gather information on such a system. Furthermore, in the context of systems analysis, the definition of a system given above can be further qualified:

- a system should be identifiable so it can be properly analysed;
- in addition, the discrete elements in the system should be identifiable;
- those discrete elements may in themselves be systems;
- a system should also have some purpose;
- finally, a system should have a defined external boundary between the system itself and the environment in which it functions.

These extra definitions have important consequences for system analysis. For instance, the principle of an external boundary is vital in limiting the scope of the analysis; a telephone, for example, is connected to a telephone network and millions of other telephones, and an analysis of that one telephone would most likely not include an analysis of the whole network. In addition, if the system or its components are not identifiable, directed analysis of such abstract entities is largely futile. Also, the idea that a component may in itself be a system leads to the necessity of analysing in a recursive fashion and the definition of a 'limit of resolution', i.e. the level of depth sufficient for the analysis; analysing a telephone at the molecular level, whilst no doubt fascinating, is almost certainly excessive. Finally, the fact that a system should have a purpose is intrinsic to system analysis, because it provides a means of measuring the system: how well does it fulfil its purpose?

There are two generic forms of analysis: inductive, or bottom-up, analysis; and deductive, or top-down, analysis. In relation to reliability engineering, the inductive approach constitutes proposing a certain event or condition and then trying to assess the effects of that initial event on the rest of the system. There are several inductive methods of system analysis, including Preliminary Hazard Analysis (PHA), Failure Modes and Effects Analysis (FMEA), and Event Tree Analysis. Deductive analysis, by contrast, works in the opposite direction. In deductive analysis, rather than postulating an initial event, the assumption is made that a final state has already been reached and then the analyst attempts to determine the course of events that led to that final state. A murder investigation, for example, is a deductive technique, because the detective starts off with the victim and tries to deduce the culprit from the evidence; similarly, Fault Tree Analysis, a deductive technique, begins with a fault and tries to determine its cause. The *Fault Tree Handbook* has a concise summary: "Inductive methods are applied to determine what system states (usually failed states) are possible; deductive methods are applied to determine how a given state (usually a failed state) can occur." (Vesely *et al.*, 1981, p I-8)

2.2 Fault Tree Analysis

"Judge a tree from its fruit, not from its leaves."
- Euripides

2.2.1 Introduction & Definitions

Fault Tree Analysis (FTA) was first conceived in 1961. It was invented by H.A. Watson of Bell Laboratories, with the assistance of M. A. Mearns, to aid in the design of a new US Air Force weapon system, the Minuteman missile. David Haasl, of the Boeing Company, used the new technique to analyse the entire system. It was successful, and at the first System Safety Conference in 1965, several papers on FTA were presented (Ericson, 1999).

FTA is a deductive system analysis technique that uses a graphical model with a logical structure (the tree) to represent events (faults) leading to a certain undesired outcome (system failure). It works by first considering an undesirable event, such as a system failure, and placing it at the top of the tree. This is called variously the 'top event' or the 'undesired event'. FTA then works backwards to determine the causes of the top event in terms of logical combinations of basic fault events (the leaf nodes of the tree).

As the purpose of FTA is to obtain information to aid in the making of decisions, it is useful to describe briefly some of the ways in which FTA does this. Fault Tree Analysis is a versatile tool, and the information it obtains is useful for a variety of tasks, not just improving reliability. The *Fault Tree Handbook with Aerospace Applications*, the updated handbook, lists a number of uses of Fault Tree Analysis in decision making, listed below:

1. To understand the logic leading to the top event
2. To prioritise the contributors leading to the top event
3. As a proactive tool to prevent the top event
4. To monitor the performance of the system
5. To minimise and optimise resources
6. To assist the designing of the system
7. As a diagnostic tool to identify and correct causes of the top event

(Vesely *et al.*, 2002)

FTA can be used at different times during the system's lifecycle. It can be used during design, as numbers #3, #5, and #6 would indicate; it could be used to improve an existing system's reliability, by identifying weakness and improving them, as #2 and #4 show; or it can be used after a top event has actually occurred in order to try and find out why, as #1 and #7 indicate. There are a number of software packages available to help automate FTA and new ones are continually being developed. Some of the earliest FT software was created in the early 1970s, when FTA was first becoming popular; amongst the most famous are W.E. Vesely's PREP and KITT (Vesely & Narum, 1970) programs. Isograph's FaultTree+ is an example of more modern FT software, capable of drawing and analysing fault trees (Isograph, 2002).

But before it is possible to properly use fault trees, it is important to understand what a fault tree actually represents and what is meant precisely by a 'fault'. For such a simple word, the 'fault' in fault tree has a remarkable number of semantic nuances and different meanings can lead to different interpretations of the fault tree. Furthermore, there are different types of faults and failures, which are represented in different ways in a fault tree.

A **fault** is defined by Villemeur to be the:

> *"Inability of an entity to perform a required function."* (Villemeur, 1992, p27)

but it is defined by Sundararajan as follows:

> *"A* fault *is a noncompliance with specifications."* (Sundararajan, 1991, p48)

As for exactly what is meant by a **failure,** Villemeur tells us it is:

> *"The termination of the ability of an entity to perform a required function."* (Villemeur, 1992, p22)

and Sundararajan states that:

> *"*Failure *is the inability of a component to perform its intended function as specified."*
> (Sundararajan, 1991, p49)

You would be forgiven for thinking that this is not really very helpful as the differences in meaning between the various definitions are rather subtle. The *Fault Tree Handbook with Aerospace Applications* sheds some light on the matter by explaining that while a failure is always a fault, a fault is not always a failure, i.e. that there can be other causes of a fault besides the failure of the entity itself. One possibility is that of an entity being produced incorrectly, such that it is never able to perform its function, and this would appear to fit both of the above definitions of a fault.

Although the occurrence of a design flaw going unnoticed is one possibility, the *Fault Tree Handbook* provides us with another possible situation: a barrier capable of raising and lowering to block an entrance. If the barrier's motor failed, it would no longer be able to open or close. This would be both a fault and a failure, since it can no longer fulfil its purpose. However, if the mechanism raised the barrier when there was no requirement for it to allow entry, then though the barrier suffered a fault (because it was no longer blocking the entrance) it would not have suffered a failure, because the barrier is still in working order and is still capable of closing again.

The notion of time, then, is an important component of a fault. A failure is more narrowly defined than a fault and occurs when an entity is no longer able to function; it is the opposite of a success. A fault, by contrast, is more inclusive as it also includes situations in which the entity operates successfully, but at the wrong time or place:

> *"The proper definition of a fault requires the specification of not only what the undesirable component state is but also when it occurs. These 'what' and 'when' specifications should be part of the event descriptions which are entered into the fault tree."*
> (Vesely *et al.*, 2002, p26)

Crucially, the notion of a fault also allows us to consider human input into a system, and more importantly, human error. If a human operator had left the barrier up, thereby unwittingly admitting unauthorised access, it would be a fault caused by human error, but not a failure of the entity itself. This important distinction between faults and failures therefore also allows the analyst to consider the impact of other factors upon the entity.

This distinction is frequently represented by the classification of faults into one of three categories, known as primary, secondary and command:

- A primary fault occurs when a component suffers a fault during normal operation, i.e. the component is operating under conditions for which it was designed, but it still fails. This could be due to the component simply wearing out, for example.
- A secondary fault occurs when a component suffers a fault during abnormal operation, i.e. the component fails when it is operating in conditions for which

it was not designed. An example would be a computer processor operating in higher temperatures than it was designed to cope with and thus overheating.

- A command fault is when the component operates normally but it does so in the wrong circumstances, due to an incorrect command signal. An example of this kind of fault would be a bomb whose trigger causes it to explode prematurely. The bomb was always meant to explode, so it fulfilled its function correctly – just at the wrong time.

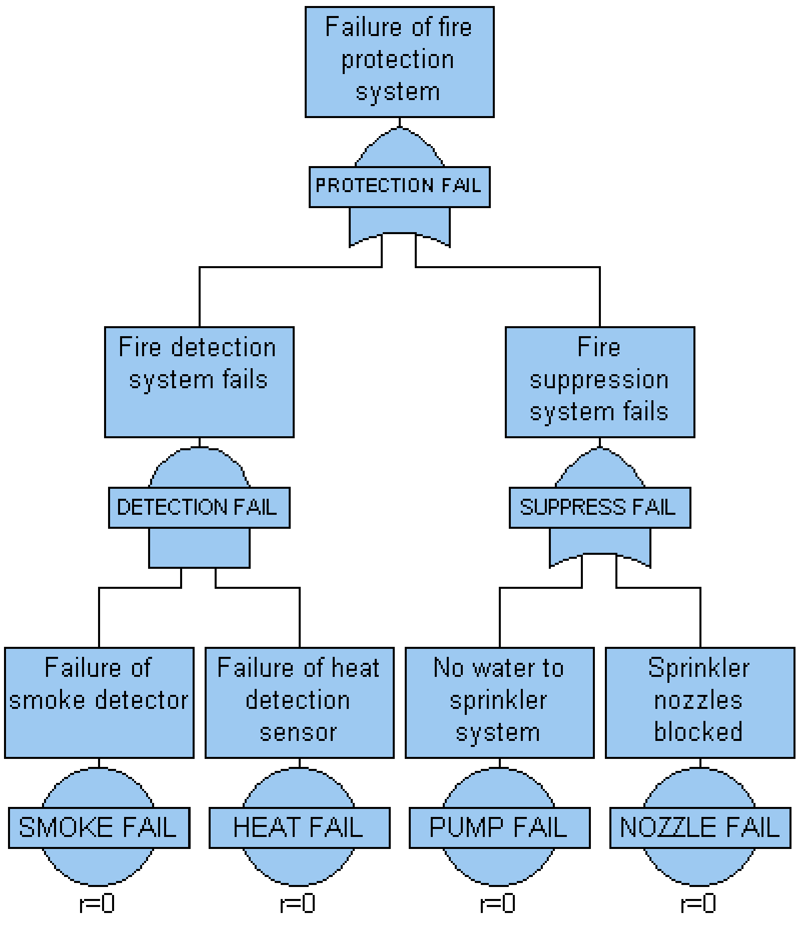
Here it can be readily seen that both primary and secondary faults will normally be failures, and are often called primary and secondary failures for that reason, but a command fault is not a failure. Both faults and failures can also be classified according to different criteria, such as how suddenly they occur (e.g. gradual vs. sudden), their degree (e.g. partial vs. complete), when they occur during the lifetime of the entity, and most importantly, according to the severity of their effects (e.g. minor, critical, or catastrophic).

This issue of time is not consistent, however. In standard fault trees, no distinction is made between the *existence* of a fault and the *occurrence* of a fault (an issue that will become important in section 2.4). Once a fault has occurred, it is considered to be 'true' by the fault tree; that is, it is presumed to exist from then on. However, in a system that can be repaired, a fault may occur, then cease to exist once it has been repaired; the fault still occurred, but it is no longer 'true'. Standard fault trees do not make this distinction. There are many efforts in progress to attempt a solution to this problem, some of which will be discussed shortly later in this chapter.

Figure 4 shows a simple example of a fault tree. The four circular leaf nodes represent contributory faults. The AND (left) and OR (right) gates above them show how these basic events can combine to produce the system failure in question (also an OR gate in this tree). In each case, the result at each stage is a simple binary result: either the event occurs, or it does not. All events in the figure have descriptions above them in rectangular boxes, but these are sometimes omitted in simplified fault tree diagrams.

The various types of node in the fault tree will be explored in greater detail in section **2.2.3** but there is a simple division in fault trees between **intermediate events** – events or faults with causes of their own and usually represented by logical gates – and **basic events**, the leaf nodes representing faults that do not need developing further[3]. The **top event** represents the system fault being investigated and is usually represented by a logical gate.

[3] Basic events may have causes of their own, but these are disregarded as being out of scope or not relevant to the system. For example, a motor failure may be caused by wear and tear which is in turn



Failure of fire protection system
PROTECTION FAIL
Fire detection system fails
Fire suppression system fails
DETECTION FAIL
SUPPRESS FAIL
Failure of smoke detector
Failure of heat detection sensor
No water to sprinkler system
Sprinkler nozzles blocked
SMOKE FAIL
HEAT FAIL
PUMP FAIL
NOZZLE FAIL
r=0
r=0
r=0
r=0

Figure 4 - Example of a fault tree (from tutorial by J.D. Andrews, www.fault-tree.net)

2.2.2 Coherency of fault trees

Fundamentally, fault trees are simply Boolean logic. Each event either occurs or does not occur and this is represented logically as either true or false respectively. The events are then combined through logical gates (most of which correspond to Boolean operators like AND and OR), which creates the logical structure of the tree. The Boolean nature of fault trees facilitates their easy evaluation, either by a computer or by hand. The simple underlying logic also means that many fault trees can readily be converted into other representations more suitable for automated analysis.

One such representation is known as the **structure function** of the fault tree. A structure function is a function representing the undesirable event in terms of the basic events. Firstly, a

caused by friction between moving parts, but this may be an unnecessary level of detail; as another example, an alarm system may fail due to lack of power which in turn is caused by a mains power cut, the causes of which could be an electrical storm affecting the substation or bringing down power lines etc, but this is out of the scope of the system being analysed.

Boolean variable (i.e. one that can have only two possible values, true or false) is assigned to each of the basic events in a fault tree. Using the example from Figure 4, for each of SMOKE FAIL, HEAT FAIL, PUMP FAIL, and NOZZLE FAIL, we can use the variables x_1, x_2, x_3, and x_4 to represent whether or not that event has occurred:

$x_i = 1$	-	the event has occurred (= true)
$x_i = 0$	-	the event has not occurred (= false)

By using z to represent the undesirable event, the structure function is:

$z = \phi(x)$ where $x = (x_1, x_2, x_3, x_4)$

For an OR gate, the structure function results in 1 if any input is true and 0 otherwise, i.e.

$$\exists i : x_i = 1 \rightarrow \phi(x) = 1$$
$$\forall i : x_i = 0 \rightarrow \phi(x) = 0$$

Similarly, for an AND gate, the structure function results in 1 if all inputs are true and 0 otherwise, i.e.

$$\forall i : x_i = 1 \rightarrow \phi(x) = 1$$
$$\exists i : x_i = 0 \rightarrow \phi(x) = 0$$

In the case of the fault tree from Figure 4, the structure function is 1 if x_3 is 1, if x_4 is 1, or if both x_1 and x_2 are 1, and 0 otherwise. This can be represented by a Boolean **truth table**:

x_1	x_2	x_3	x_4	$\phi(x)$
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**1**
0	0	1	1	**1**
0	1	0	0	**0**
0	1	0	1	**1**
0	1	1	0	**1**
0	1	1	1	**1**
1	0	0	0	**0**
1	0	0	1	**1**
1	0	1	0	**1**
1	0	1	1	**1**
1	1	0	0	**1**
1	1	1	0	**1**
1	1	0	1	**1**
1	1	1	1	**1**

Table 1 – A Boolean truth table

A truth table can be used to determine the Boolean value of a structure function (or more generally, any Boolean expression) from the Boolean values of its constituent parts. In Table 1, each row lists the values of every input (represented by the first four columns) and the corresponding output value of the structure function (the fifth column). Any Boolean fault tree containing only AND and OR gates[4] can be represented in this way by a truth table, but as the table must contain 2^n rows where n is the number of inputs, truth tables are impractical for expressions with large numbers of inputs.

Structure functions are important for another reason, however. Although the most common fault tree gates are the standard Boolean AND and OR gates, there are others, including NOT and XOR gates, both of which introduce the idea that the *non-occurrence* of an event can contribute to the system failure at the top of the fault tree (or alternatively that the occurrence of an event can *prevent* the system failure). A fault tree with only AND and OR gates will always be a **coherent** tree, whereas a tree containing NOT gates is **non-coherent**. A tree can be proved to be coherent if its structure function obeys the following restrictions:

[4] And optionally NOT and XOR gates, but these can lead to non-coherent fault trees.

- Each element in the structure function is relevant, i.e. all elements in the function must affect the output:

  $$\phi\,(1,x_i) \neq \phi\,(0,x_i)$$

- The structure function must be *non-decreasing* in each x_i:

  $$\phi\,(x) \geq \phi\,(y) \qquad \text{if } x \geq y$$

  If an element x_i becomes true (i.e. it suffers a fault) then the system either stays the same or deteriorates – in other words, if a component in a system fails, it will not lead to an improvement in the functioning of the system.

A non-coherent fault tree, by contrast, means that "components NOT failing, i.e. working, contribute to the system failing," (Andrews, 2000). Thus in a non-coherent tree, a failing component can lead to overall system success. The meaning of the second restriction for coherent trees should now be more clear: to be non-decreasing, a component failure must either cause system failure or no change in system status.

The distinction between coherent and non-coherent trees is important due to the fact that they need to be handled in slightly different ways. Thus the presence of gates other than just AND and OR may alter the logical structure of the fault tree and require a different (and normally more complex) form of **qualitative analysis**.

Qualitative analysis of fault trees relies on the Boolean properties of the fault tree to obtain simplified logical equivalents that yield more readily usable information. Generally, this means obtaining the **minimal cut sets** (MCS) of the fault tree. A **cut set** is simply a combination of basic events that can cause the top event, and a minimal cut set is a cut set with no redundant basic events, i.e. all of the basic events in a minimal cut set are required to cause the top event. The **order** of a minimal cut set is the number of basic events it contains: a 1st order MCS would contain just one event, whilst a 5th order MCS would contain five basic events, all of which need to occur to cause the top event. Minimal cut sets are useful because low order MCS, such as 1st and 2nd order, indicate areas of particular vulnerability and help the analyst to identify those 'critical' components that the operation of the system relies upon. For a 1st order minimal cut set, a single event is all that is necessary to cause the top event (known as a *single point of failure*), and the component in question would therefore be a good candidate for replication and/or replacement with a more reliable component.

However, the use of non-coherent fault trees complicates qualitative analysis, because minimal cut sets no longer apply; it is no longer sufficient to indicate only which events have occurred – it is also necessary to indicate which events have *not* occurred. Therefore **implicants** (analogous to cut sets) and more importantly **prime implicants** (analogous to minimal cut sets) are used. Whilst a cut set is a set of events that must occur to cause the top event, an implicant is a set of events or their complements necessary to cause the top event, e.g. A and B and not C. Implicants generally contain more events than cut sets because of the need to account for the non-occurrence of events too. A prime implicant is the equivalent of the minimal cut set: it is an implicant that contains no other implicants. Whatever the differences, however, they are used in the same way: to identify weak points in the system.

Although a fault tree is purely a qualitative model, and primarily conveys the logical combination of events leading to a system fault, it is also a convenient model to use for **quantitative analysis**, which allows the analyst to obtain numbers and statistics about the system. By including extra information in the modelling process, a fault tree can be quantified to calculate the probability of the top event occurring, as well as the relative importance of the contributing events. Quantitative analysis is usually performed after qualitative analysis, because it is more efficient to apply probabilistic analysis to a set of minimal cut sets or prime implicants than it is to apply it to *all* the cut sets in a fault tree (and there may be tens of thousands for larger trees). In this way, the analyst not only discovers what the weak points in the system are, but also discovers *how* weak they are. Both forms of analysis provide information that could potentially be very useful when making decisions about the development of any system, but only qualitative analysis is dealt with in this thesis.

2.2.3 Fault Tree Symbology

There have been several attempts to extend fault trees with additional gates and symbols in order to represent further types of information within the fault tree (e.g. the DFT approach in Vesely *et al.* (2002) or the TFT approach in Palshikar (2001)) and there are many minor variations in the appearance and layout of fault trees. However, the core set of symbols is common to all fault trees and these symbols, as defined in the *Fault Tree Handbook*, can be divided into three categories: event, gate and transfer symbols, as shown below. There are also NOT gates but these are not shown in the *Handbook*.

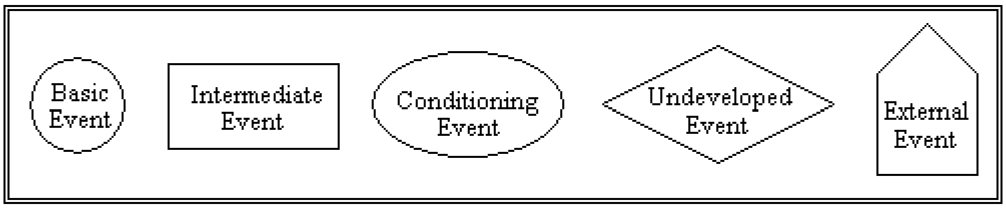


Figure 5 – Fault Tree event symbols

<u>Basic Event</u>

A basic event is a basic fault that requires no further development or expansion. Basic events form the leaf nodes of the tree and combine to cause intermediate events and (ultimately) the top event. In qualitative analysis, cut sets are composed of basic events, and in quantitative analysis, basic events will usually be assigned failure rates and repair rates so that the top event unavailability can be calculated.

<u>Intermediate Event</u>

An intermediate event is a fault that occurs because of combinations of other events occurring further down the tree; for this reason an intermediate event is almost always a type of logical gate. The top event is a special event of this type at the top of the tree.

<u>Conditioning Event</u>

A conditioning event is an event that serves as a special condition or constraint for certain types of gates (e.g. Priority-AND gates and INHIBIT gates). For example, the INHIBIT gate is only true if both all of its inputs are true and if its conditioning event is true. A conditioning event does not necessarily have to be a fault; it could be that the system is in a certain state.

<u>Undeveloped Event</u>

An undeveloped event is an intermediate event whose contributing events are not considered in the analysis. This may be because of insufficient information about this event or it may be because the event is considered inconsequential. It may, for example, have such a low probability of occurrence that to analyse it in further detail would be unnecessary.

<u>External Event</u>

An external event is an event that is not a fault, i.e. an event that could be expected to occur during the normal operation of a system. It is used to represent events that ordinarily would not cause any problems, but in combination with other events, may lead to an undesired event.

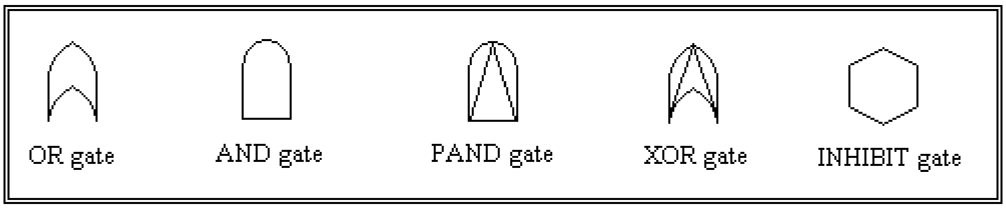


Figure 6 – Fault Tree gate symbols

OR Gate

The OR gate is true if any of its input events are true. The OR gate does not necessarily represent a causal relationship between its inputs and outputs; each of the inputs are often restatements of the output. For example, the output 'valve is failed open' could be further described as 'valve left open during maintenance' or 'valve fails open due to mechanical fault', but both descriptions refer to the same result: the valve is open, and the inputs did not cause the output. OR is represented in this thesis by a '+' symbol in text or in expressions.

AND Gate

The AND gate is true if all of its input events are true. Unlike an OR gate, an AND gate typically represents a causal relationship between its inputs and its outputs; that is to say, the combination of input faults causes the output fault. For example, 'No power to system' could be caused by both 'battery failure' and 'generator failure', but not by just one. AND is represented in this thesis by a '.' (full stop) symbol in text or expressions.

Priority AND (PAND) Gate

The PAND gate is only true if all of its input events are true and they occur in a certain order. The order can be specified by a separate conditioning event, but it is often omitted. The PAND gate will be covered in greater depth later in this chapter (see section **2.4.1**).

Exclusive OR (XOR) Gate

The XOR gate is true if one and only one of its input events is true.[5]

INHIBIT Gate

The INHIBIT gate is a special case of the AND gate in which the output of the gate is only true when the input event is true whilst a conditioning event is also true. For example, an explosive reaction may only take place if above a certain temperature or a catalyst is present, even if the

[5] Note that the *Fault Tree Handbook* also shows a strange diagram with a standard OR gate with a conditioning event attached; this will be mentioned again later in the context of the Priority-OR. Also note that the XOR gate symbol in the *Handbook* differs from XOR gate symbols found in other fields, e.g. electronics.

constituent ingredients are present. It is also known as an IF gate but in logical terms it functions as a normal AND gate.

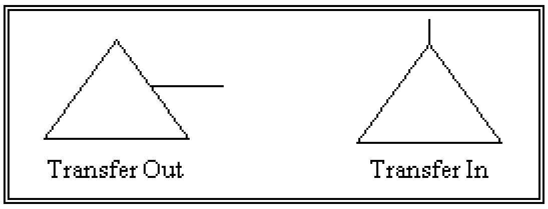


Figure 7 – Transfer gates

Transfer In

This indicates that this particular branch of the tree is displayed at the corresponding Transfer Out symbol. It is used to save space or to indicate a shared branch.

Transfer Out

This signifies that this branch connects to the rest of the tree at the corresponding Transfer In. Transfer Out symbols can be used to represent shared branches (i.e. multiple Transfer In gates linking to a single Transfer Out).

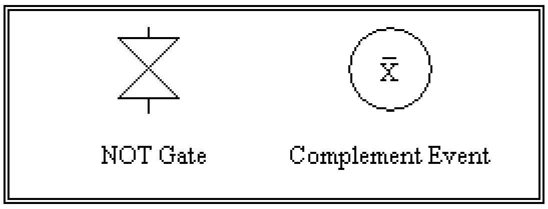


Figure 8 – NOT gate & Complement Events

Although NOT gates are not present in the *Fault Tree Handbook,* when used, they usually have the symbol shown in Figure 8, though other symbols are also used. They flip the value of an event, i.e. if we have an event X, then NOT(X) is true only if X did not occur, and false if it did occur. NOT gates can also be represented implicitly as the complement of a basic event. NOT is represented in this thesis by the '¬' symbol.

2.2.4 Relevant Fault Tree Algorithms

There are many important fault tree algorithms available with a number of different purposes (e.g. qualitative analysis, quantitative analysis, sensitivity analysis, importance measures etc).

For this thesis, the most relevant algorithms are qualitative analysis algorithms. A good summary can be found in Wolforth (2005) but a couple of important algorithms that will be referred to later are briefly described below.

MOCUS – Method of Obtaining Cut Sets

MOCUS (Fussel & Vesely, 1972) is one of the main classical fault tree reduction algorithms and one upon which many other solutions are based. It is a top-down approach (a similar bottom-up approach, MICSUP, followed three years later) that consists of recursively expanding intermediate events into basic events until there are no more basic events remaining. The basic algorithm is as follows:

1. Make a table with rows and columns. Each row represents a cut set and each column is an element in the cut set.
2. Put the top event in the first column of the first row.
3. Scan through the table, and for each gate:
   a) If the gate is an AND gate, put each of its children in a new column.
   b) If the gate is an OR gate, put each of its children in a new row.
4. Repeat Step 3 until there are no gates in the table. This means the fault tree has been fully expanded, and the MOCUS table contains only basic events.
5. Search for and remove all redundancies within the table, according to the usual Boolean laws (Absorption, Distributive, Idempotent etc; see **Appendix II: Boolean & Temporal Laws**).

The resulting table then contains a minimal cut set in each row. MOCUS is accurate and quite fast for smaller trees. However, as the table size grows much larger depending on the number and placement of OR gates in the fault tree, MOCUS struggles to store all of the necessary information for large fault trees. For example, two OR gates, each with four inputs, ANDed together, will result in 4 x 4 = 16 different cut sets. To see how the process works, consider the simple fault tree in Figure 9:

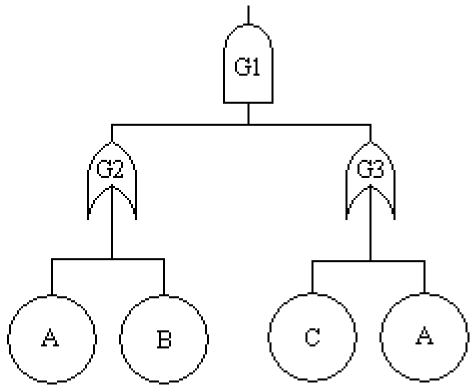


Figure 9 – Example fault tree for MOCUS

The first step is to make a table to store the top event, which in this case is the AND gate, G1:

G1

Then this is expanded, according to step 3. Because it is an AND gate, we put each of its children in a new column:

G2	G3

Next, these are expanded as well. They are OR gates, so their children are put in a new row, starting with G2:

A	G3
B	G3

And then G3:

A	C
A	A
B	C
B	A

Now that all of the gates have been expanded, Boolean laws can be applied to check for redundancies. According to the Idempotent law, two identical events in the same row can be reduced to just one, i.e. X AND X ⇔ X, thus:

A	C
A	
B	C
B	A

And finally, according to the Absorption law, if all of the events in a row also occur in another row, then the larger row is redundant, so:

A	
B	C

Which gives the result: two minimal cut sets, one containing just A and the other containing B and C.

Despite its problems, however, MOCUS is both one of the simplest and the most popular of FTA techniques, as evidenced by its 30 year life-span. Because it is accurate and easy to understand, it is a good technique to analyse smaller fault trees; furthermore, it makes an excellent foundation for further expansion – or extension – with new techniques. However, it is not the most efficient technique and newer algorithms like MICSUP tend to be faster.

<u>ELRAFT – Efficient Logic Reduction Analysis of Fault Trees</u>

ELRAFT is a mathematical reduction algorithm proposed by S. N. Semanderes in 1971 (Semanderes, 1971). Unlike the simple, logical method used in MOCUS, ELRAFT attempts to improve the performance of the reduction process by exploiting a property of prime numbers. Although it still applies the Boolean laws of Absorption and Idempotence to reduce the fault tree to its minimal cut sets, rather than checking each event against every other event to determine whether or not it can be reduced (as in MOCUS), ELRAFT can check entire cut sets at once.

Semanderes makes use of a mathematical feature to perform the analysis. It is a property of all positive integers greater than 1 that they are the exact product of only one set of prime numbers. ELRAFT exploits this fact by assigning each unique basic event a different prime number. The gates then obtain values derived from the child values. For example, using the fault tree in Figure 9, we might assign values such as A = 2, B = 3, C = 5. Expansion (in the style of MOCUS) will produce four cut sets and the value of a cut set is equal to the product of the prime values of its constituent events:

A.A	= 2 x 2 = 4
A.C	= 2 x 5 = 10
B.A	= 3 x 2 = 6
B.C	= 3 x 5 = 15

It then becomes possible to check whether or not a cut set contains a given event by performing a modulo operation on the cut set using the prime number of the event: if there is a remainder, then it does not contain that event; if there is no remainder, then it does. For example, B.C is 15 and C is 5, so B.C modulo C is 0. A is 2, so B.C modulo A is 1. Thus B.C contains C, but does not contain A.

This applies also to entire cut sets; if there was a cut set A.B.C, it would have the value 2 x 3 x 5 = 30, and so to check whether it contained B.C (which is 15), 30 modulo 15 gives us the answer 0, thus A.B.C contains B.C. Of course, before this is possible, it is first necessary to ensure that there are no duplicates in the cut set, e.g. by scanning through and checking for repeated events. Then, by dividing the product of each cut set against the product of every other cut set, we can determine whether or not it is redundant.

This method is purely mathematical and does not require any searching through pairs of cut sets to test each event against every other event; instead, a simple mathematical calculation is performed for each cut set, which is much faster. It should be noted, however, that in practice there are often problems with storing the products of prime numbers in computer programs, as they can grow very large very quickly and often exceed the ranges of most numeric representations.

Linear Time Modularisation Algorithm

The linear time modularisation algorithm (Dutuit & Rauzy, 1996) is not a traditional qualitative or quantitative analysis technique: instead, it is used to find independent modules in fault trees. An independent module is a sub-tree in which none of the basic events are repeated elsewhere in the tree. This independent sub-tree can then be analysed separately as part of qualitative or quantitative analysis, making the process more efficient by avoiding repeatedly analysing the same branches of the tree. The division of the tree helps to simplify the analysis process, and in the case of certain techniques, allows for different algorithms to be used depending on the nature of the sub-tree in question.

The modularisation algorithm itself consists of two depth-first left-most traversals through the fault tree and a set of three variables for each node. The first pass sets a counter v_1 indicating the first visit, i.e. the number of nodes visited so far. During the same pass, on the way back up the tree, moving from the right-most child to its parent, it sets a second counter v_2 with the number of its *first* return visit. Any subsequent visits to the same node set v_3, which stores the number of the most recent visit. A second traversal of the tree is then made to determine whether each node is a module. A node is then head of an independent module if none of its children have a v_1 less than its own v_1 and none have a v_3 greater than its own v_2. By contrast, if the most recent visit counter v_3 of a child is higher than the first return visit counter v_2 of the node, it contains a shared branch. The name of the algorithm comes from the fact that performance is linear because each link in the fault tree is only visited twice and so the algorithm has a performance of O(*n*).

To see how this process works, consider the simple fault tree in Figure 9, which consists of (A OR B) AND (A OR C). For each event, the 'visited' values are as follows:

Event	*v1 (1st visit)*	*v2 (1st return)*	*v3 (last visit)*
G1	0	9	9
G2	1	4	4
A	2	2	7
B	3	3	4
G3	5	8	8
C	6	6	6

Table 2 – Linear Time Modularisation Algorithm

The numbers represent the number of nodes visited so far, so G1 has the first visit counter 0 because no other nodes have been visited yet. The visitation of nodes is G1 → G2 → A → (G2) → B → G2 → (G1) → G3 → C → (G3) → A → G3 → G1. Notice that because A occurs beneath both G2 and G3, they are not independent; the maximum of G2's childrens' v_3 values is 7, which is greater than its own v_2 counter of 4; similarly for G3, the minimum of its childrens' v_1 values is 2, which is before its own first visit counter of 5. Only G1 is an independent tree, but in this case it always would be since there are no higher nodes.

Modularisation is useful for many reasons, not least because it can break the tree down into parts that can then be more readily analysed. This is very useful with Boolean analysis techniques such as MOCUS, because it means the algorithm can deal with several smaller trees of reduced complexity rather than one big tree. Because of the combinatorial explosion effect of lots of OR gates high up in a fault tree, analysing modules can result in much reduced demands on memory for MOCUS-style techniques. However, one of the more subtle benefits of modularisation is that different modules can be analysed independently with different techniques, as exploited by Dynamic Fault Trees (see **2.4.2**).

2.3 Time

"What then is time? If no one asks me, I know what it is. If I wish to explain it to him who asks, I do not know."
- Saint Augustine

"Time is an illusion. Lunchtime doubly so."
- Douglas Adams

2.3.1 Representing Time

Time is a tricky thing. We take it for granted in everyday life, but the more closely you think about it, the more intricate and complicated it becomes. That this is true is evident from the many different schools of thought on how to model 'time' in a mathematical or philosophical way (e.g. Cleugh, 1937). The first point of divergence is the distinction made between **absolute** time, where time is measured from a fixed starting point (as in a calendar), and **relative** time, where any one point in time is measured relative to any other. Relative time allows imprecision in that it is not necessary to know the exact time at which an event occurred, just whether it occurred before or after another event. Absolute time, by contrast, is necessarily much more precise: it describes exactly when an event occurred.

This is not to say that relative time cannot have precision – it is possible to say, for example, that one event occurred 3.142 seconds after another, in which case it is **quantitative** and has a metric for time (i.e. time is measurable). Absolute time always has a metric, since it measures time since a common starting point, whereas relative time just means it is not necessary to have a common frame of reference to relate all events to each other.

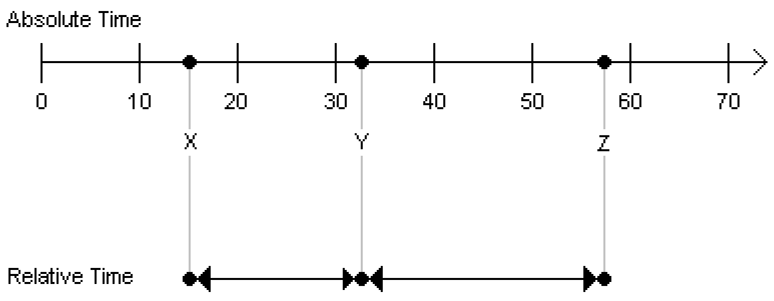


Figure 10 – Absolute versus relative time

Figure 10 shows the difference between absolute and relative time. Absolute time has a starting point and all events are measured from that point, so X may occur at time 15, Y at time 33, and

Z at time 57, for example. In relative time, it is only possible to know that Y occurred after X (or X occurred before Y) and similarly Z occurred after Y. If a metric exists, we can also say that Y occurred 18 time units after X, for example.

A second division between different models of time is whether or not time is represented as being **continuous** (also known as **dense** time) or **discrete**. In a continuous model of time, there are no gaps between moments – there is infinite precision. In a discrete model of time, time is represented by a series of moments or states that are clearly delineated. The common analogy used in this case is the difference between real numbers and natural numbers; in the former case, there is always another number between any two numbers, in the latter case, there is not.

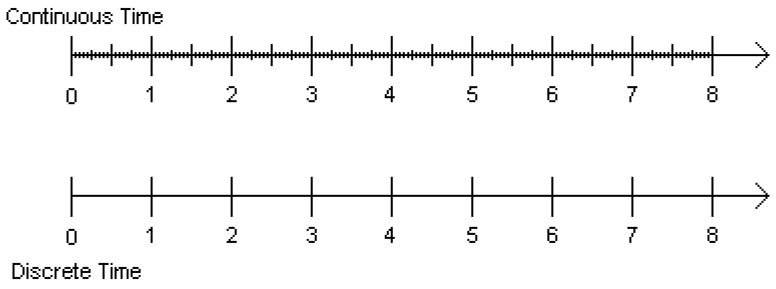


Figure 11 – Continuous versus Discrete time

Figure 11 shows the difference between continuous and discrete time. In the former case, it is possible for events to occur at any point along the scale, e.g. at 1.5, 2.222, 7.123 etc. In discrete time, events can only occur at discrete moments, not in between, e.g. only at 1, 2, or 7. Continuous time is more frequently used in absolute time models whereas discrete time is often used in relative models of time (e.g. in which each discrete time value is a separate state).

Yet another variation, more philosophical this time, is between **linear** time and **branching** (or **non-linear**) time. With linear time, time is represented by a single time line, but with branching time, there can be multiple paths in the future (and optionally in the past), and events cause different timelines to split from the original line (e.g. in one path the event was true, in the other the event was false). These two options are also known as **deterministic** and **non-deterministic** time, because branching time allows the possibility of events occurring in some future time lines but not others. One further point to note here is the notion of *bounds*. A linear time model can be bound in the past (i.e. it has a fixed starting point) and in the future (it has an end point), one or the other, or neither. Similarly, branching time can branch only in the future, only in the past, or both.

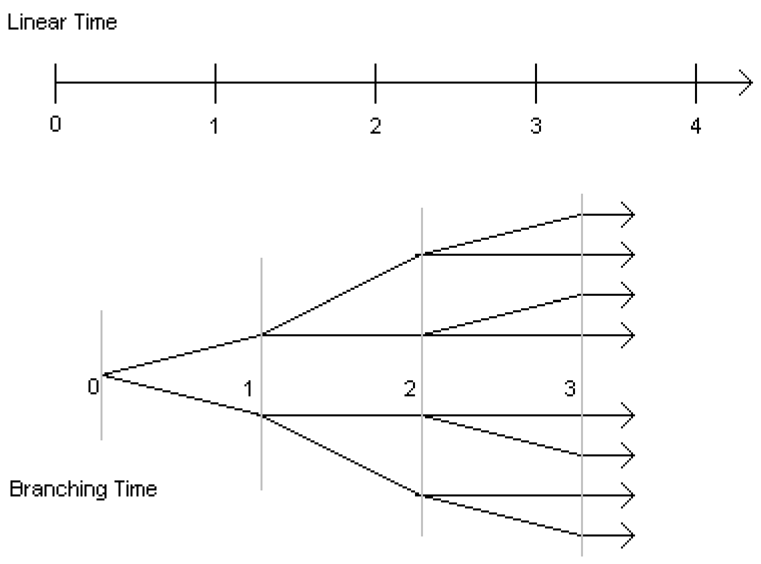


Figure 12 – Linear vs. Branching time

There is one more common distinction in time models – the choice of the unit of time itself. The first option is the notion of a **point-based** time, where essentially time is represented by a series of moments in time (either continuous or discrete). Intervals are represented as a collection of time points. For example, looking at the linear time line in Figure 12, we could define an interval as {1, 2, 3} containing three moments and therefore lasting three units of time, or we could simply define it as [1,3] by giving a start and end point. The other option is to use **interval-based** time, in which the interval is assumed to be the atomic unit of time. Again using Figure 12, the base interval unit would be the time between 0 and 1. Whichever interval is chosen is assumed to be atomic, i.e. indivisible, so this obviously has implications for the granularity of time the model can represent.

A good analogy to these two ideas is the notion of a measurement of length. We can define a distance using a standard unit of measurement, e.g. the metre. This is the interval-based method. The problem with this is that we must make sure our unit is sufficiently small to be able to achieve the level of precision we require, otherwise something may need to take a fraction of an interval, which the model does not allow. The alternative is the point-based method, which is analogous to measuring distance using a series of markers, like milestones. Every time you pass a milestone, you have travelled further. The obvious flaw with this approach is that there are mile-long "gaps" between milestones, i.e. in other words, potentially minute periods of time between the points. This can pose problems: for example, if one period of time starts when another ends, do they overlap at the same point in time (so that both are true at that point) or is there a minute amount of time between the first ending and the second starting? This problem is illustrated in Figure 13:

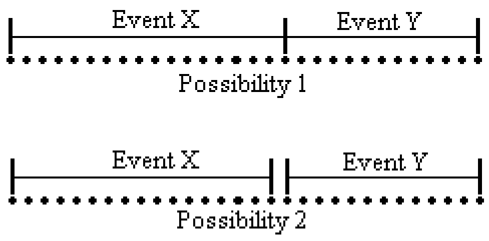


Figure 13 – The Point-Based Time Problem

In the first possibility, we define the duration of event X as [0,10] and Y as [10,15] – but in that case, is point 10 included in both events? The second possibility is to make sure the events do not overlap, e.g. X = [0,9] and Y = [10,15]. But in that case there appears to be a "gap" between points 9 and 10 that the events do not cover.

In Allen (1983), the author categorises the different approaches to representing time into four general groups:

- *State space approaches*

  State space approaches work by modelling the system as a database of information. Each state is a database representing the system at one time; when an event occurs, it causes a transition from one state to another, thereby representing a progression through time. The items of information in the database may be true for one state but false for another and so simulate the changes to the system over time. Events are persistent in that once an event causes a change in one facet of the system, the changed facet of that state persists unless another event changes it again. State-space approaches are necessarily discrete, point-based, and relative, but can be either branching or linear.

- *Date-base/Date-line systems*

  In these systems, information is stored in a database and indexed with a date. By comparing the dates of two items of information, it is possible to discern the temporal order in which they occurred. Unfortunately, it is often difficult to assign the necessary precision or imprecision to dates, making it difficult say that two events did not occur or did occur at the same time, for example. This scheme relies on the use of linear, absolute times for all items of information, but the model of time can be continuous or discrete and either point-based or interval-based.

- *Before/After chaining*

  Before and after chaining is a way of explicitly modelling relative times of events by linking them according to the order in which they occur. However, for large numbers of events there are problems of storage and overly-expensive searching, since searching is typically linear. It is also difficult to represent durations (since each 'event' is a link in the chain). Before/after chaining is usually independent of whether time is point-based or interval-based, or continuous or discrete.

- *Formal Models*

  Formal models apply to a range of disciplines, from philosophy to artificial intelligence. One such formal model is *situation calculus*, which is the principle behind state-space approaches; in situation calculus, time is represented as a series of states, each of which represents a system at a point in time. Transitions between the states are actions or events. The main drawback with situation calculus is that only one state is true at a time – there is no notion of overlap.

Allen also suggests that the formal concept of point-based time is unhelpful, because even a seemingly instantaneous event can usually be further decomposed (Allen, 1983), and if the original event occurs at one point, then it is not clear when the sub-events occur. Instead, he suggests the use of an 'interval' unit instead – the smallest relevant period of time. We use this concept all the time – rather than measuring time in seconds all the time, we measure it in minutes, hours, or days etc according to our needs. In effect, we treat a period of time (a time interval) as if it were indivisible. Allen's time intervals, unlike points, are consecutive – there are no gaps between them, however small. Furthermore, to allow conclusions to be drawn about the relative time of these time-intervals, Allen provides 13 temporal relations to cover all possible relations between two intervals in time:

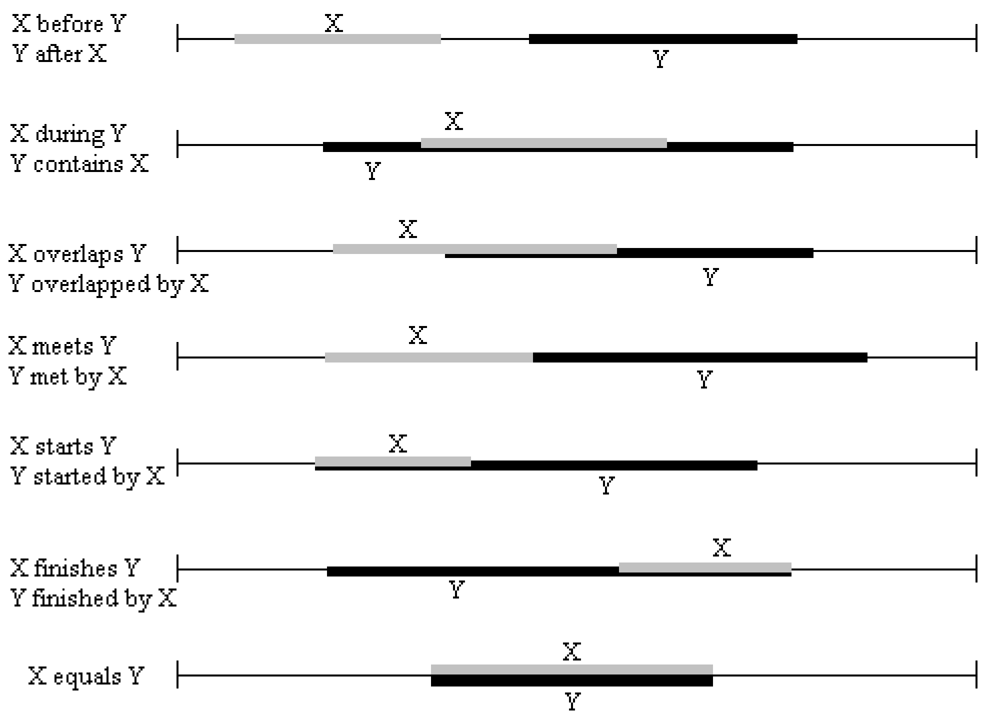


Figure 14 – Allen's 13 Temporal Relations

The choice of time model (or lack thereof) can have important repercussions when looking at different approaches to modelling time, whether in fault trees or elsewhere. In particular, the choices of relative vs. absolute and linear vs. branching time have significant impacts on how any logic underlying any such model would work. However, it is often the case that certain models of time suit certain type of applications better than others, and in such cases it is possible to see that some choices are better than others.

2.3.2 General Temporal Logics

In any model of time, there also has to be some rules to govern its structure and provide semantics. These rules are generally known as *temporal logics* and allow us to represent and reason about time in a more structured fashion.

The field of temporal logics began with the 'Tense Logic' introduced by Prior (1957, 1967, 1968; also summarised in McArthur, 1976). Tense logic was based on *modal logic* (Rescher & Urquhart, 1971) which is an extension to propositional logic that adds two additional operators – a 'necessarily' operator, indicated by □ or by L, and a 'possibly' operator, indicated by ◊ or by M. Modal logic is designed to allow reasoning about *modalities* and these operators allow us to say things like: "It is necessarily true that jelly beans taste nice," and "It is possibly true that some people don't like jelly beans." Modal logic recognises that there are many possibilities and that sometimes a proposition may be true and at other times false, while in other cases a proposition will always be true (or always false).

Tense Logic takes this idea and applies it to time. It defines four **temporal operators** as follows:

F or ◊	Sometime in the future ("It will at some time be the case that...")
G or □	Always in the future ("It will always be the case that...")
H or ■	Always in the past ("It has always been the case that...")
P or ♦	Sometime in the past ("It has at some time been the case that...")

The first two refer to the future tense and the second pair to the past tense. G and H are the 'strong tense' or 'always' operators, corresponding to the 'necessarily' in modal logic, while F and P are the 'weak tense' or 'sometimes' operators corresponding to the 'possibly' modal operator. They are also inter-definable, e.g.:

Px = ¬H¬x (If *x* has at some time been true, then *x* has not always been false.)

Fx = ¬G¬x (If *x* some time becomes true, then *x* will not always be false.)

(where ¬ represents negation)

Extensions to Tense Logic have been proposed at various times, but amongst the most popular additions are the 'Until' (*U*) and 'Since' (*S*) operators introduced by Kamp in 1968. (Galton, 2003) These are binary operators since they take two operands, e.g.:

S*xy*	"*y* has been true since a time when *x* was true"
U*xy*	"*y* will be true until a time when *x* is true"

Prior himself added another operator, the 'metric tense' operator F*np*. This allowed specification of time intervals in both future (positive *n*) and past (negative *n*). F*np* means that "*p* will be true after interval *n*". If *n* is 0, then it refers to the present moment.

Other additions include the 'next time' operator, O. O*p* means that *p* is true in the next moment (but not at the present moment); this implies the existence of a discrete time model as otherwise the definition of the next 'moment' could pose problems.

There have been other approaches; for example, Allen suggested a predicate-based temporal logic with a 'holds' operator for use in modelling temporal knowledge, e.g. Holds(Open(Shop), (8.30am, 5.30pm)). However, Tense Logic proved very influential and many of the subsequent temporal logics reuse elements of it; in particular, the future *sometime* and *always* operators usually feature in other temporal logics and the *until* and *next* operators are also quite common.

One of the foremost temporal logics is **Propositional Temporal Logic (PTL)**, which was proposed by Pnueli (1977) for reasoning about concurrent programs. It was based on Tense Logic and used the *always*, *sometimes*, *next* and *until* operators to describe the relationships between states in a system. It represented time as a series of instants corresponding with states (i.e. each state was an instant of time), meaning it was both point-based and discrete. It has also been extended with other operators, e.g. past operators like *previously*, *once*, *so-far*, *since* (Lichtenstein & Pnueli, 2000; Kesten *et al.*, 1993). There is also a variation of PTL known as 'Choppy Logic' as it adds a 'chop' operator to permit the concatenation of states.

Other variations of PTL include Branching Time Temporal Logic (BTTL), which is a variation of PTL that uses branching time rather than linear time, and Interval Temporal Logic (ITL), which, as the name suggests, uses intervals instead of points as the basic unit of time (Bellini *et al.*, 2000). BTTL is better suited to non-deterministic systems and it provides quantifiers for this purpose: the $\forall$ quantifier means that something is true in all subsequent time branches whereas $\exists$ means it is true in at least one branch. By contrast, ITL is a linear logic, making it suitable for tasks like modelling digital signals where there is only one timeline. ITL replaces PTL's *until* operator with a *chop*, as in Choppy Logic, which means it can only indicate order by concatenation. Yet more variants include CARET, which is designed to represent nested calls and returns in concurrent programming (Alur *et al.*, 2004), and another that adds a *within* operator to represent nested intervals (Alur *et al.*, 1997). Both ITL and PTL are examples of **linear temporal logics (LTLs)**; although others exist, e.g. Linear Logic (Girard, 1987), PTL (or a variation thereof) is usually considered the main LTL and is consequently sometimes described as PLTL or even just referred to as LTL.

The main alternative to the PTL family is **Computation Tree Logic (CTL)** (Emerson, 1990). As the name suggests, it is a branching-time logic, similar to BTTL. CTL offers capabilities for modelling non-deterministic systems and is frequently used in model-checking and formal verification contexts, e.g. DCCA (Ortmeier *et al.*, 2005). It also has a number of its own variations, one of the most important of which is CTL*, a superset of CTL and LTL. Like PTL, however, it has its roots in the original Tense Logic and so CTL* features familiar operators such as G (always), F (eventually), U (until), X (next), together with two quantifying operators (as in BTTL): A meaning true in all branches and E meaning true in at least one branch. Another combination of linear and branching time is ITL+ (Ortmeier *et al.*, 2008), which is a combination of ITL and CTL/CTL*. Other extensions include RTCTL (Real-Time CTL), which provides a metric for time and is better suited to modelling real-time systems (which require more precision than normal CTL can provide).

Another family of temporal logics is the **Interval Logic (IL)** family. These are interval-based logics where the intervals are bounded by events; it provides bounded operators to reason about these intervals, e.g. to suggest that something is always true inside a given interval or that it is true sometime in that interval. Extensions include EIL (Extended IL), which allows quantitative constraints, and RTIL (Real-Time IL), which provides a metric of time.

Yet another major contributor to the field of temporal logics in concurrent programming is Leslie Lamport, who developed the **Temporal Logic of Actions (TLA)** (Lamport, 1983; 1994a; 1994b). TLA uses three main operators (*always*, *eventually/sometime*, and *leads to*), which can also apply to each other (e.g. *x* will always lead to *y*, *y* sometimes leads to *z*, etc).

The majority of these temporal logics are intended for formal specification and verification, particularly in concurrent programming, but even with this one field, they all have different advantages and disadvantages. Vardi (2001) suggests that specification is easier in a LTL like PTL but verification is easier in a branching logic like CTL; this is because model checking is linear in CTL but exponential in LTL. However, CTL is more complex and less intuitive to use compared to LTL. Nevertheless, CTL forms the basis of a number of model checkers, e.g. SMV. Lamport also compared linear and branching logic (Lamport, 1980) and he observed that the ubiquitous *always* and *sometime* operators can mean different things in linear and branching logics; in linear logic, *sometime* is equivalent to *not never* (i.e. $\mathrm{F}p \Leftrightarrow \neg\mathrm{G}\neg p$ meaning "*p* will be true sometime" is equivalent to "*p* is not always false") whereas in branching logic, they are not equivalent. This is because in branching time, *sometime* refers to *every* possible future, whereas *not never* means there is at least one possible future, but not necessarily all; i.e. $\mathrm{F}p$ means "*p* is true at some point in every possible future", whereas $\neg\mathrm{G}\neg p$ means "*p* is not false in every possible future".

Another common use of temporal logics is for modelling real-time systems. However, most of the general temporal logics described thus far have no metric for time and are thus ill-equipped to model real-time systems. Instead, a number of temporal logics exist that are designed specifically for this purpose, e.g. RTTL (Real-Time Temporal Logic), TPTL (Timed Propositional Temporal Logic), RTL (Real-Time Logic), TRIO (Tempo Reale ImplicitO / Implicit Real Time), MTL (Metric Temporal Logic), and TILCO (Time Interval Logic with Compositional Operators) (Bellini *et al.*, 2000).

Several of these are also extensions of PTL; RTTL and TPTL, for example. RTTL is linear and discrete but allows quantification of time and represents time directly as a variable. TPTL makes a direct association between time and natural numbers. TPTL also provides a 'freeze quantifier' that, when used, binds a time variable to the present, making it possible to specify constraints relative to the current context, e.g. by freezing *x*, we can say that *y* must occur between *x* and *x + 5*, meaning it must occur in the next 5 time units (Alur & Henzinger, 1992).

There are typically three approaches to introducing a metric of time and creating a quantitative temporal logic. Some logics allow their operators to be bound to a time interval, e.g. sometime[2,4] means sometime in the next 2 – 4 time units; MTL (a variant of PTL) uses this approach, as does IL. MTL also has a variant known as MTL*p*, which adds past time operators (Alur & Henzinger, 1993). Other logics provide a freeze operator, like TPTL. Still other logics use an explicit clock to represent the actual value of time at the moment of any evaluation, e.g. RTTL and XCTL (Explicit Clock Temporal Logic). Although these are all linear, discrete logics, MITL (Metric Interval Temporal Logic) allows continuous time with non-negative real numbers and RTCTL (Real-Time CTL, mentioned above) and TCTL (Timed CTL) both use branching time (Alur & Henzinger, 1991).

Not all temporal logics are used for modelling real-time systems or concurrent programming, however. Some are used, or even designed for use, in temporal databases – databases which allow querying based on time (Chomicki & Toman, 1997). Temporal query languages like TQUEL and TSQL2 exist (Toman, 1996) and it is possible for queries to be expressed in point-based representations and then converted into interval-based logic. However, it can be difficult to get the precision correct, e.g. to determine whether two events occurred at the same time or not, one must define what 'at the same time' means. It might mean on the same day or on the same year; it is complicated further if the events themselves have a duration as then this must also be taken into account.

Clearly, there are many temporal logics for many different applications, as few temporal logics are flexible enough to be applied in multiple domains. The choice of temporal logic depends therefore on what is being modelled; since different temporal logics fit some models of time better than others, it is important to find a temporal logic that offers the right features, e.g. a metric of time for real-time systems or branching quantifiers for non-deterministic models. Bellini *et al.* (2000) suggest that for modelling real-time systems, the most suitable temporal logics are first-order, interval-based logics that offer a metric of relative time and that employ only a small number of basic temporal operators.

Some of these recommendations are specific to real-time systems (e.g. the need for a metric of time) but others are true in a wider context: relative time, for example, is generally more flexible than absolute time because it does not require a starting point or a metric for time; relative time is also more suitable in future tense logics where the exact time of an event is not known because it has not necessarily happened yet. By contrast, absolute time is potentially more valuable in a past tense logic because it can refer accurately to events that have occurred at known points in time. Furthermore, interval-based logics tend to be favoured over point-based as intervals can be seen as a generalisation of points and they do not suffer the problem of 'gaps' between points. Using a limited number of operators is potentially beneficial because it makes the logic easier to remember and simpler to use, especially if those operators can be combined to form new compound operators.

Where fault trees are concerned, both linear and branching time have their uses. Taken as a whole, a fault tree can be thought of as a collection of possible sequences of events that all lead to the same point (the top event); this can be seen as a type of branching timeline. Alternatively, it is possible to view individual sequences of events (or cut sets) separately, each as a linear timeline that leads to the top event. The choice of interval-based or point-based, or relative or absolute, depends more on the nature of the basic events and the way they interact; if a real-time system was being modelled with a fault tree, then a metric of time would be more valuable, or if the fault tree was being used for fault diagnosis in an operational system, then an absolute time model could be used, with the starting point being the start time of the system.

2.4 Temporal FTA

None of the various temporal logics mentioned thus far were designed for use with fault trees (though they may well be of use in specifying the normal behaviour of the system). However, there are some approaches that have been designed for, or adapted for use with, fault trees. These approaches tend to fall into one of two categories: **event-based** approaches, which add temporal information to the events of the fault tree, and **gate-based** approaches, which add temporal information via new logical or temporal gates. The scope and goals of these approaches also vary: some are complex, all-encompassing solutions whilst others only make a few alterations; some are intended only for quantitative analysis and others for qualitative analysis, while yet others are intended for the purposes of requirements specification instead.

2.4.1 The Original Priority-AND gate

Before going through some of the more modern solutions to the problem, it is worth taking a brief look at FTA's original solution to the issue of temporal analysis: the venerable Priority-AND (PAND) gate, as mentioned in section **2.2.3**. The PAND gate allows the analyst to introduce an order to a set of events, putting them into a sequence and thus expressing some amount of time-dependent information within the fault tree. This sequence is typically defined explicitly as a conditioning event. Unfortunately, PAND gates are generally overlooked in FTA.

The problem with the PAND gate is that it was never thoroughly defined, meaning it is difficult to use it in qualitative analysis. The *Handbook* states that:

> "The PRIORITY AND-gate is a special case of the AND-gate in which the output event occurs only if all input events occur in a specified ordered sequence. The sequence is usually shown inside an ellipsis drawn to the right of the gate." (Vesely *et al.*, 1981, p IV-11)

This definition gives no indication of what sequence to use if none is given in a conditioning event, nor does it indicate whether the gate is still true if one or more of its inputs occurs simultaneously. Furthermore, it fails to address any of the issues raised by the possibility of events occurring in sequence, e.g. whether it is possible for contradictions to arise, whether events must be consecutive or not, or what happens if the same event occurs more than once in the same sequence.

Part of the problem is the precise meaning of an event. For example, the question of what happens if the same event is used more than once as an input to the same PAND gate depends to a large extent on whether it is possible for the same event to occur more than once. The *Handbook* seems to suggest not, because it states firstly that "Under conditions of no repair, a fault that occurs will continue to exist", and shortly thereafter, "From the standpoint of constructing a fault tree we need only concern ourselves with the phenomenon of occurrence. This is tantamount to considering all systems as nonrepairable." (Vesely *et al.*, 1981, p V-1) But it says nothing about whether events occur instantaneously or whether they can have a duration (in which case, they may overlap).

Furthermore, the *Handbook* simply defines an AND gate as being true "if all the input events occur." (p IV-3) But several of the temporal FTA approaches discussed later in this chapter raise the question of whether or not these input events should have to occur simultaneously or whether it is sufficient for them to occur in any order. The *Handbook* does seem to suggest the latter interpretation, as on pages IV-7 and IV-8, it discusses the problems of dependencies:

> "When describing the event input to an AND-gate, any dependencies must be incorporated in the event definitions if the dependencies affect the system logic." (p IV-7)

The example given separates the possible sequences of inputs into two AND gates and an OR gate, such that one AND gate has the inputs "X" and "Y given that X has occurred" and the other has the inputs "Y" and "X given that Y has occurred". Clearly neither of these pairs of events can occur simultaneously as the event definitions themselves preclude that possibility. What is not clear is why a Priority-AND was not used in these situations instead.

Clearly, there are a lot of unanswered questions surrounding the *Handbook*'s definition of the PAND gate, in particular:

- What sequence of events is used if none is specified? A default left-to-right sequence?
- Do events occur instantaneously or do they have a duration? (And if they have a duration, what happens if they overlap?)
- What happens if inputs to the PAND gate occur at the same time?
- What happens if the same event is used more than once as an input to the same PAND gate?
- What happens if the logic introduces a contradiction? For example, if we have an expression like (X PAND Y) AND (Y PAND X), assuming that X PAND Y means that X occurs before Y, the situation is impossible – how can X occur before Y and Y also occur before X?

With all this confusion, it is no wonder that many cut set analysis algorithms, ranging from older tools such as SETS (Worrell & Stack, 1978) to more modern software packages like older versions[6] of FaultTree+ (Isograph Ltd, 2002), simply treat the PAND gate as a normal AND gate for the purposes of logical reduction. It is often argued that the replacement of a PAND by an AND simply leads only to a conservative prediction of the failure behaviour of the system, but this claim is not necessarily true. Consider the simple example of a standby recovery system in Figure 15.

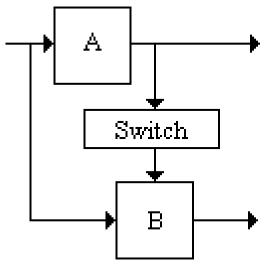


Figure 15 – An example system where a PAND might be useful

Normally the system performs its function using component A. Component A is monitored by a switch which starts standby component B when there is an omission of output from A. In a classical fault tree, the following expression relates the top event to logical combinations of causes:

```
System Fails = A.B + A.Switch
```

i.e. if both A and B fail, so will the system, or if A fails and the switch fails, the system will also fail. However, the above is not only pessimistic in quantitative terms, but it is also logically wrong: the system does not fail if the switch fails *after* A. It is necessary to be able to specify the order of events more precisely to be able to distinguish between sequences that cause failure and sequences that do not:

```
System Fails = A.B + Switch fails before
           or at the same time as A
```

The Priority-AND gate is intended to do this, but the difficulties with the gate stem from its lack of a rigorous definition. The PAND gate is true if its inputs occur in a set order, but what if two of the events occur at the same time, e.g. what if the switch fails at the same time as A? It will still cause a failure, but it might not strictly be considered part of the PAND's defined sequence. For that matter, it is not clear how to define whether one event occurs before another as they may overlap if they have durations.

It is these problems that have resulted in the Priority-AND gate being so often ignored, at least during qualitative analysis. However, the story is quite different when it comes to quantitative analysis, and over the years there has been a considerable body of work focused on finding methods of quantifying PAND gates. The quantification of sequential failures in fault trees (as modelled by PAND gates) is often termed 'Sequential Failure Logic', or SFL. SFL has been used in the quantitative analysis of a number of different types of dynamic systems, including space satellites, human-robot systems, product liability prevention, and so on (Long *et al.*, 1999).

There are several methods of solving SFL as part of quantitative FTA. One method is to use Markov chains, as in the Dynamic Fault Tree approach described in the next section. However, the Markov approach suffers from a number of drawbacks: in particular, it is computationally expensive and struggles to cope with shared input events. Another method is the Priority-AND Quantification (PAQ) method, originally proposed by Fussel *et al.* (1976). The PAQ method is an approximation but it is much simpler and applicable to a wider range of systems; it involves treating the Priority-AND as an INHIBIT gate, i.e. an AND gate with an additional conditioning event which specifies the order in which the events should occur. Fussel *et al.* also described an exact solution but stated that it "cannot readily be used in existing methodologies for quantitative system logic model evaluation, such as fault tree analysis techniques." (p 325). However, both methods are only suitable for non-repairable systems. Long & Sato (1998) conducted a comparison between PAND quantification methods and concluded that the PAQ method had the advantage in simplicity and, for a PAND gate with three inputs, the results were the same as the Markov method. Also, while the Markov approach grows increasingly difficult as more input events are added, the PAQ method can be used with an arbitrary number of inputs. Long *et al.* (1999) also provide methods of solving the analysis of Priority-AND gates with many inputs, since multiple integration for large numbers of inputs can be difficult.

Yuge and Yagani (2008) present an additional method of quantifying PAND gates without resorting exclusively to the use of Markov chains. Because Markov models have a number of disadvantages when created from fault trees, especially if the dynamic gates of the fault trees have shared events (i.e. the same event is an input to more than one gate, in which case the gates are no longer independent), Yuge and Yagani propose the use of the inclusion-exclusion method

[6] Version 10 of FaultTree+ did not detect contradictions; Version 11 (the current version at the time of writing) detects simple contradictions like (X PAND Y) AND (Y PAND X) but cannot analyse them –

to calculate the probability of the top event. This method can handle the presence of repeated/shared inputs to PAND gates, but it relies on the minimal ordered cut sets already being known; thus for a complex dynamic fault tree, without a clear method of obtaining its cut sets, this approach is not as suitable. The computation time also strongly depends on the number of cut sets.

Regardless of which approach is chosen, the problem remains that performing quantitative analysis on PAND gates without prior qualitative analysis can lead to anomalies and errors in the calculations. Most importantly, quantitative analysis has no way of detecting contradictions, which – being impossible – have a probability of 0.

Unfortunately, whilst there are several examples of techniques of quantifying PAND gates, there are far less examples of qualitative techniques. There are some approaches (described later in this chapter) which mention the generation of ordered cut sets or minimal cut sequences etc, such as the work of Güdemann *et al.* (2008) and the CSSA approach (Liu *et al.*, 2007), but usually only as a step towards quantitative analysis. The effect of PAND gates on the identification of minimal cut sets is rarely considered, and methods of reducing them in certain situations (e.g. when the output is a tautology or a contradiction) are completely absent. The implications of including a PAND gate in qualitative analysis are not taken into account at all; for example, not all of the Boolean rules that apply to an AND gate apply to a PAND gate. Most obviously (using '<' to represent PAND):

	Non-Temporal	Temporal
Commutative:	$X.Y = Y.X$	$X<Y \neq Y<X$
Idempotent:	$X.X = X$	$X<X = 0$ [7]

The Idempotent law is one of the main laws used in static qualitative analysis, since it removes redundancies from cut sets; it is doubtful that (X PAND X) could be reduced in the same way if PAND does not account for the simultaneous occurrence of its inputs.

Thorough qualitative analysis of PAND gates needs to be able to detect such situations and deal with them appropriately, particularly when those situations – like contradictions – have a bearing on any subsequent quantitative analysis. Although, for certain levels of detail, replacing PAND gates with AND or even INHIBITed AND gates is a sufficient approximation, in many cases this can lead to pronounced inaccuracies or even logical errors.

the user is forced to change the fault tree.

2.4.2 Dynamic Fault Trees

Of all the various modern approaches, perhaps the most prominent is the **Dynamic Fault Tree (DFT)**[8] methodology (Manian *et al.*, 1998), a gate-based temporal fault tree methodology designed for quantitative analysis of dynamic systems. DFTs make it possible to analyse fault tolerant computer systems using Markov chains (Norris, 1997). Markov models are capable of modelling the sequence-dependent behaviour typically found in such systems, but they are large and cumbersome, and the production of Markov chains is often tedious; by generating them automatically from fault trees, it becomes possible to avoid many of the disadvantages of manually producing Markov models while also granting fault trees the ability to analyse sequence-dependent failure behaviour in systems. DFTs therefore allow for the analysis of systems with more complex interrelationships between events, including functional dependencies, standby components, and event sequences.

DFTs are a comprehensive attempt at a solution, as evidenced by their inclusion in the newer *Fault Tree Handbook for Aerospace Applications* (Vesely *et al.*, 2002). They have been incorporated into automatic safety analysis tools or methodologies such as DIFTree (Dugan *et al.*, 1997) and its successor, Galileo (Sullivan *et al.*, 1999; Dugan *et al.*, 1999; Manian *et al.*, 1999); they have also been put to a variety of uses, such as dependability analysis (Meshkat *et al.*, 2002), formal models (Coppit *et al.*, 2000), expert systems (Assaf & Dugan, 2003), analysis of software (Dugan & Assaf, 2001), sensitivity analysis (Ou & Dugan, 2000), phased-mission systems (Xing & Dugan, 2002), common-cause analysis (Tang & Dugan, 2004), and linked with event trees (Xu & Dugan, 2004). A formal compositional semantics using Markov chains can also be found in (Boudali *et al.*, 2007a & b).

DFTs represent temporal information by defining two main special purpose 'temporal' gates, described below.

---

[7] Assumes that PAND is not inclusive of simultaneous occurrence of events. If X PAND Y includes the possibility of X and Y occurring at the same time, then X < X = X and the Idempotent law holds.
[8] The capitalised term 'Dynamic Fault Tree' and acronym 'DFT' are used exclusively in this thesis to refer to the dynamic fault tree methodology of Dugan *et al* and should not be confused with other dynamic FTA approaches or 'dynamic'/temporal fault trees in general, which are referred to only in lower case.

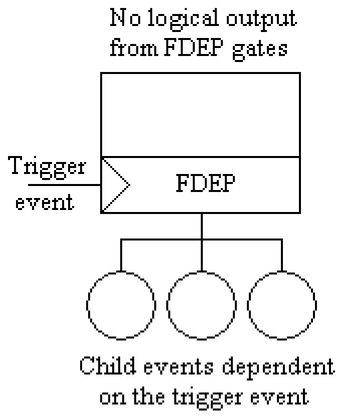


Figure 16 – Functional Dependency (FDEP) gate

Functional Dependency (FDEP) gates allow DFTs to model situations where one component A is dependent on another component B for operation. If component B fails, then component A will also fail; the failure of component B is then the *trigger* event for the failure of component A. FDEP gates have a single trigger event and multiple dependent children events; the occurrence of the trigger event will cause the occurrence of all the children. If any of the children occur by themselves, this does not affect the other children or the trigger event. This type of gate is very useful for modelling networked components or components connected by a central bus where the failure of the interconnection will cause the failure of the individual components. They also allow fault trees to model interdependencies that would otherwise cause loops in the fault tree.

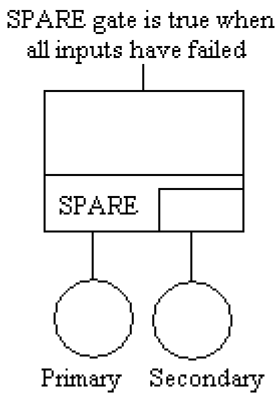


Figure 17 – Generic Spare Gate

Spare gates allow DFTs to model secondary backup components that are not activated until needed. These are difficult to model in ordinary fault trees because they do not fail until being activated, i.e. after the primary has failed. The inputs to spare gates are all basic events. The first (left-most) is the primary component, and any further inputs are secondary components, which are activated in order (so if there are three secondaries, each will be activated when the previous one fails). The SPARE gate itself is only true when the primary and all the secondaries have failed. There are usually three varieties of SPARE gates: hot spares, which are always on but only provide function when the primary fails; warm spares, which are kept in a state of reduced readiness until needed; and cold spares which are kept deactivated until required. Hot spares will have higher failure rates than cold spares because they are always active, and failure rates of warm spares will fall in between hot and cold spares. This is modelled by a *dormancy factor* which affects the failure rate; a hot spare will have a factor close to 1.0λ, a cold spare somewhere close to 0, and a warm spare somewhere in between, e.g. 0.5λ. Spare gates can also share secondary backup components, i.e. one secondary can be the backup for two primaries. In this way, spare gates can also model a common pool of backup components.

DFTs also make use of a version of the Priority-AND gate (which as mentioned earlier, is true if its inputs all occur in a specific order) and sometimes also include separate sequence or SEQ gates which impose a left-to-right sequence on input events (and can thus be viewed as a specialised version of a PAND gate). The potential ambiguities of the PAND gate are not often addressed in the DFT methodology, though a more formal definition of the DFT gates is given in Coppit *et al.* (2000) (and discussed in Section **6.2.1**).

A DFT using any of these gates can then be analysed using Markov chains; any fault tree model with exponential-like distributions can be solved quantitatively as a Markov chain, though in practice the computational complexity may be prohibitive for large models. Markov chains are constructed by considering the effects of all possible component failures in all possible operational states in turn. These child states are then considered in turn until all possible states, both failed and operational, are taken into account. From these, and the failure rates of each component, quantitative analysis is possible.

Markov chains, and by extension DFTs, have various advantages and disadvantages. It is easier to use a fault tree than to use Markov chains directly, and DFT tools like Galileo handle this automatically, converting to and from Markov chains as necessary. Obviously, DFTs also allow the analysis of dynamic system elements, like redundant components and functional dependencies. However, Markov chains are slow. For a large tree with lots of dynamic modules, this could become a significant problem. The reason for this poor performance is that Markov chains have a near exponential state space; virtually every basic event must be considered against every other basic event. Even for moderate trees, the state space could be massive. The problem becomes much worse in cases where dynamic gates share common events, i.e. the same event is an input to more than one gate; for this reason, tools like Galileo often disallow repeated/shared events. It is also difficult to perform other types of analysis (i.e. qualitative) using Markov chains. Identifying the weak points in the system, rather than just determining an overall reliability, is even more expensive with Markov chains.

To help solve this problem, DFTs use the modularisation method by Rauzy & Dutuit, explained in section **2.2.4**, to break up the fault tree into independent modules that can be analysed separately (Gulati & Dugan, 1997; another technique can be found in Huang & Chang, 2006). This has two benefits. Firstly, it reduces the complexity of the fault tree as a whole; three sub-trees each with 1000 states are more manageable than one whole tree with 1000x1000x1000 states. Secondly, it enables separate analysis of different modules using different methods. Therefore, a module containing only static elements can be analysed with the much faster BDD approach (Sinnamon & Andrews, 1996), whereas a module containing dynamic/temporal elements can be analysed with Markov chains. In this way, DFTs combine the speed of static methods with the added expressive power of dynamic methods. However, this approach is only effective in fault trees containing several independent modules; if there are none, or if dynamic gates occur near the top of the tree, then the advantages of modularisation are lost.

Another solution to this problem has been put forward by Amari *et al.* (2003), who use conditional probabilities to perform quantitative analysis of a DFT without first converting it into a Markov chain. This is particularly useful if the top node of the tree is a dynamic gate (which would render the modularisation ineffective), but it is potentially not as effective in cases where the tree is mostly static, as in that case traditional methods may be more efficient for most modules.

There has also been some work on performing qualitative analysis using DFTs (Tang and Dugan, 2004). The goal of the method is to obtain the **minimal cut sequences** from the DFT, which are ordered minimal cut sets, i.e. the events must occur in a specific order. Although the minimal cut sequences can be extracted from the resulting Markov model, this is an expensive operation. The proposed alternative uses a variation on the BDD method instead. The method consists of first separating the timing constraints – i.e. the temporal information – from the logical constraints. For example, a PAND has the logical constraint that all of its inputs must occur and the timing constraint that all of its inputs must occur in a specific sequence. Once this has been done, the dynamic gates can all be replaced with static equivalents (e.g. PANDs are replaced by ANDs). The resulting tree can then be used to produce a ZBDD (Zero-suppressed BDD – a minimised version of the BDD), which can be minimised to produce minimal cut sets in the normal way; BDDs are much quicker than Markov chains for this purpose. Once the minimal cut sets have been obtained, they are expanded to include the timing constraints removed earlier to form the minimal cut sequences (if appropriate).

The problem with this method is that it explicitly separates the temporal information from the fault tree during the reduction process. The result is that any possible redundancies or contradictions can only be identified and removed accordingly once the temporal information has been restored at the end, potentially missing the opportunity to do so earlier in the process. Furthermore, it is not clear whether or not any such reduction is done even at the final stage; in which case, possible logical reductions or contradictions could be missed, resulting in inaccuracies in the quantitative analysis.

Liu *et al.* (2007) propose another method of combined qualitative and quantitative analysis of DFTs called CSSA (Cut Sequence Set Algorithm). In CSSA, the DFT is broken down into a set of 'cut sequences' called *sequential failure expressions* (SFEs), which are ordered lists of events separated by the sequential failure symbol, $\rightarrow$. For example, $X \rightarrow Y$ is a SFE in which X fails first and then Y fails. The CSS (Cut Sequence Set) is the collection of all SFEs that represent the fault tree. AND gates are converted into SFEs by enumerating all possible sequences, of which there are *n!* for a gate with *n* inputs; thus an AND gate with three inputs, e.g. X.Y.Z, would yield 6 SFEs: $X\rightarrow Y\rightarrow Z$, $X\rightarrow Z\rightarrow Y$, $Y\rightarrow X\rightarrow Z$, $Y\rightarrow Z\rightarrow X$, $Z\rightarrow Y\rightarrow X$, and $Z\rightarrow X\rightarrow Y$. PAND gates indicate a single SFE directly, e.g. X PAND Y is the same as $X\rightarrow Y$. FDEP gates are represented as (E_1 AND E_2) OR E_3, where E_1, E_2, and E_3 are SFEs representing the trigger event, the triggered events, and any non-triggered events respectively. Finally, SPARE gates are represented by specific SFEs that link the failure of the primary to the failure of the secondary. Once the CSS has been generated, it can be quantified using conditional probability formulae. Therefore, the CSSA method avoids the use of Markov chains entirely.

2.4.3 Dynamic fault trees (again)

The DFTs created by Dugan *et al.* are not the only type. Čepin and Mavko (2001) propose a different type of dynamic fault tree of their own. Instead of the gate-based approach of Dugan's DFT methodology, this approach is an event-based one. It uses a 'house events matrix' and time-dependent probabilistic models for basic events. The matrix shows the state of all events at discrete points of time in the system lifetime and so the state of the overall system can be determined at any point in time by examining the matrix. The probabilistic models associated with each event then allow analysis and even optimisation to take place.

There are several drawbacks to this approach when compared to the better known DFT methodology discussed above. Firstly, the size of the matrix is determined by the number of events in the system and the number of discrete points in time that need to be modelled; clearly, for larger and more complex systems, this matrix will grow to a considerable size. Secondly, qualitative analysis is not included, meaning that the technique is dependent upon appropriate probabilistic failure data being assigned to each event; if this is not available or not appropriate, then the approach offers no advantage. Finally, it is necessary to know what the status of each event is at each moment in time. However, these drawbacks are a consequence of the intended use of this approach, which is aimed at the evaluation and monitoring of active, operating systems – in which case, the disadvantages are nullified as data can be obtained directly from monitoring of the system itself.

2.4.4 Temporal Fault Trees

Temporal Fault Trees (TFT) are another kind of temporal fault tree[9], taking a slightly different direction to DFTs (Palshikar, 2001). Although TFTs continue the theme of representing temporal information using new temporal gates, the aim is to improve the specification of general temporal information in fault trees to better reflect the temporal relationships between events, rather than concentrating on extending fault trees to account for different states of operation and specific dynamic situations. TFTs use a more traditional style of temporal logic, based on modal logic, to enable a more accurate and more formal specification of temporal information in fault trees.

TFTs largely retain the standard fault tree structure but add many new gates to represent the various temporal relations represented by a variation on LTL known as PLTLP (Propositional Linear Temporal Logic (Past)). The gates are as follows:

Instance-oriented past temporal operators:

• PREV	X is true at the previous instant
• PREV n	X is true at the instant t_{k-n} in the past (if $k \geq n$); false if $k < n$
• ALLPAST	X is true now and for all the previous instants in the past
• FORPAST n	X is true now and for the last n instants in the past
• SOMETIME-PAST	X is either true now or at some instant in the past
• WITHIN n	X is either true now or at some instant within the last n instants
• UNTIL-PAST	Sometime in the past, Y holds and X holds everywhere before that

Pseudo-temporal "chop" operators:

[9] As with DFTs, the use of the capitalised term Temporal Fault Tree and its acronym TFT relate only to the approach by Palshikar and not to temporal fault trees in general (which uses lower case).

• m CHOP n	X is true now and at the previous *m* instants and Y is true for *n* instants before that
• m STRONG-CHOP n	X is true now and at the previous *m* instants and Y is true for the strictly earlier *n* instants before that
• m CHOP-FAIL n	X is true now and at the previous *m* instants but Y is not true at all the earlier *n* instants before that
• m FAIL-CHOP n	X is not true either now or at some of the earlier *m* earlier and Y is always true for all the previous *n* instants before the m instants from now

The past-oriented nature of the operators is due to the goal of the TFT methodology, which is designed for fault diagnosis (i.e. determining the cause of a fault in an operational system based on sensors and a diagnostic model of the system). In this context, the deductive nature of the fault tree – which starts with an event and works backwards to determine what caused it – means that the operators must be past-oriented because they refer to events that have already happened. The operators with subscripts (i.e. with an *n* or an *m*) are *counting connectives*, which actually represent a series of non-counting connectives for each specific value of the subscript(s). PLTLP also provides additional Boolean operators such as ¬ (Not), → (Implies), and ↔ (If and Only If). It is interesting to compare an example timeline for TFTs, which is point-based, against the interval-style timelines given earlier in Figure 14:

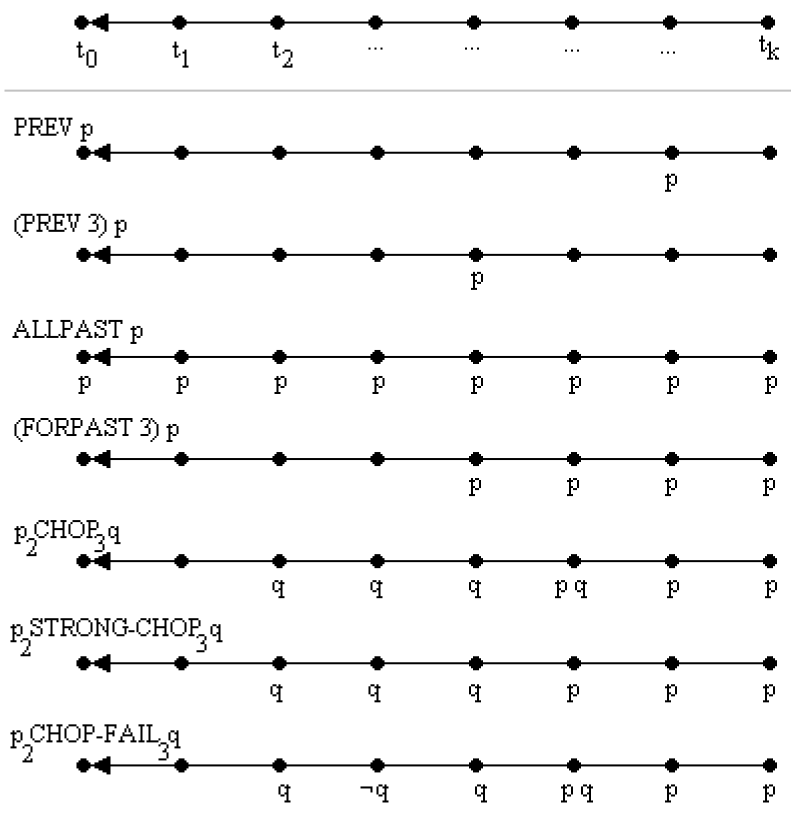


Figure 18 – TFT timelines

Notice how these timelines use specific instants (represented by dots on the line; the right-most dot, t_k, is the current instant) and how some events overlap on points (in CHOP and CHOP-FAIL). The actual instants are not defined; they can be minutes, seconds, hours, days, or whatever is appropriate to the system being modelled. These operators must be used with caution, however, as they are not designed to represent repetitions, e.g. 10 cycles of p $_2$CHOP$_3$ q. Instead, the chop operators can be used to model situations such as a train unable to brake in time, e.g. *HighSpeed* $_{30}$CHOP-FAIL$_{20}$ *Braking*, which means that the condition *HighSpeed* holds for the first 30 seconds, and then the condition *Braking* holds for the past 20 seconds.

A technique is also provided for the analysis of TFTs and PLTLP, which consists of a depth-first traversal of the TFT once the top event has occurred. At each gate, temporal or logical, the expression represented by the gate is checked against the system log to see whether or not it is true, which will locate the path that caused the top event to occur. The path to the top event essentially gives the cut set(s) that caused the top event. An alternative is to generate all cut sets first, ranking them in order of probability or testability, and then ascertaining which of them is true to determine what caused the top event (known as a *confirmed diagnosis*). This is done in one of two ways: by replacing all temporal gates with a basic event containing a temporal formula, or by replacing the temporal gates with a logical gate and the adding one or more basic events containing the temporal information (essentially, converting it to an INHIBIT gate with the temporal constraints as the conditioning event). The latter method results in much larger fault trees, but can more easily handle recursively nested temporal gates.

Although the TFT notation is very powerful and expressive, and well-suited to the task of more formally specifying the temporal constraints on events in fault trees, TFTs are ultimately intended to be used in a diagnostic fashion *after* a system has failed, relying on a log containing periodic samples of the system state to determine what caused the top event. In this respect, TFTs are better suited to maintenance and repair rather than initial design work (where fault trees are also widely used). They are meant to be relatively simple to use, and thus the extent of the temporal logic was deliberately kept limited (hence the use of linear point-based time), but even so, it is felt that the gates introduced in Temporal Fault Trees are not particularly intuitive unless the user has at least a passing familiarity with temporal logics. The author does propose a wide range of further work, including research into the possibility of TFTs being used alongside other techniques, such as Durational Calculus (see below) and Interval Temporal Logic, and also research into the possible automatic synthesis of TFTs.

2.4.5 CSDM & Durational Calculus

Both DFTs and TFTs are gate-based approaches and attempt to represent the temporal information as part of the fault tree structure by incorporating new, temporal gates with fixed meanings into the fault tree. The alternative is the event-based approach, as exemplified by the work of Gorski & Wardzinski and Hansen *et al.*, which incorporates the temporal information in the definitions of the events (whether intermediate or basic) instead. These approaches are meant to allow for a more precise and more formal specification of the temporal constraints on the events of a fault tree.

Gorski and Wardzinski accomplish this through the use of an *enabling condition,* which is an intermediate event that represents the temporal constraints. This is then included in the analysis as a kind of conditioning event and thus can be derived from the minimal cut sets at the end, in the same way TFTs can convert their temporal gates to a form of INHIBIT gate. The enabling conditions are subject to a formal specification known as CSDM (Common Safety Description Model), i.e. they are not written in natural language like normal basic events and instead use a form of predicate logic. For example, for two events *e1* and *e2* in a cut set, there may also be an enabling condition, e.g.:

$$occur(e1) \wedge occur(e2) \wedge enabling\ condition(e1,e2)$$

The basic notion of CSDM is the event, which is defined as "a distinguished state of a system which can last for some time." (Gorski & Wardzinski, 1996) An event is therefore not instantaneous and can occur more than once (i.e. it can repeat), so to distinguish between multiple occurrences of an event, the notion of an *action* is introduced – an action is an instance of an event. The definition of an event is important in determining the formal semantics of the technique; as such, the approach also takes into account the temporal dependencies of intermediate events (i.e. logic gates), e.g. an AND gate implies a causal relationship between input and output and thus there is a temporal precedence – causes must come before effects (Gorski, 1994). However, the formal semantics are not meant to be excessively restrictive and can even be added to a fault tree post-construction, once the relationships between events have already been established.

Each action has a start time and an end time, known as *transitions,* and represented for action a by $start(a)$ and $end(a)$ respectively. Thus the duration of an action can be calculated by subtracting the start time from the end time. The presence of transitions also makes it possible for actions to overlap in time. The model of time used is linear and continuous, meaning that transitions can be mapped into real numbers by *Time* functions, each of which represents one 'scenario' of system behaviour. This system makes it possible to define very precise events, e.g. checking that the duration of an event was more than a certain value, or that two events had overlapped for at least a certain amount of time. For example, an overlap between a gas leak and a naked flame – once a sufficient amount of gas has leaked out (at time t_G), the flame would ignite it:

$$explosion = occur(gas\_leak) \wedge occur(fire) \wedge overlap(gas\_leak, fire) \wedge duration(gas\_leak, fire) > t_G$$

Transitions themselves are instantaneous and actions always end, so events are not persistent: any given occurrence of an event (an action) will eventually cease. There are two types of temporal relations between actions – temporal ordering (i.e. one action occurred before or after another) or temporal equality (both occurred at the same time, e.g. because they were triggered by the same causes).

Qualitative analysis is then carried out as in a normal fault tree, but at each stage, the enabling condition of any gate or event has to be taken into account. Once the MCS have been obtained, real-time requirements are then generated from the temporal aspects of the MCS, i.e. the enabling conditions. The system can then be specified according to these requirements, e.g. that it should not be possible for a gas leak to exist for more than t_G seconds. By preventing the enabling condition from being fulfilled, it is possible to prevent a given minimal cut set from becoming true and thereby preventing it from leading to the top event and causing a hazard. The enabling condition is not necessarily present in the minimal cut sets; it can also be derived by examining the MCS and determining what conditions are necessary for those events to cause the top event.

It is important to point out that the CSDM approach simply *augments* fault trees: it is meant to be possible to add the formal specifications to an existing fault tree as well as build a formal fault tree from nothing. In effect, the formalism adds a second layer – a temporal layer – on top of the existing Boolean layer of semantics. This is the reason for the two-stage analysis where temporal relationships between events in a minimal cut set are determined after standard qualitative analysis has taken place. The results of this second stage can then be used to produce the timing requirements for the system specification. Notably, other techniques can be used to perform a second, more detailed 'temporal' phase of analysis, e.g. time Petri nets (Gorski & Wardzinski, 1997). Petri nets are a widely used tool for analysing complex systems and can be used as a visual aid similar to flow charts. In this case, the general algorithm is as follows:

1. Produce a conventional fault tree.
2. Formalise it using CSDM, removing ambiguities and establishing temporal relationships between events.
3. Calculate MCS and, for each MCS, the enabling conditions necessary for it to cause the top event.
4. Perform a time Petri net (TPN) analysis to establish hazard reachability. The fault tree is transformed into a TPN and analysed to see whether the top event is reachable given the time dependencies present in the enabling conditions.

This second, separate analysis can check the results of stage 3 and can potentially identify anomalies caused by hitherto unidentified relationships between events.

The approach taken by Hansen *et al.* (1998) is similar in that it establishes more precise semantics for events in fault trees, but instead of CSDM it uses Duration Calculus, which is based on Interval Temporal Logic (ITL) and in particular employs the 'chop' operator (written ';' in Duration Calculus). Other operators include 'somewhere' or ◊ (i.e. an event occurs somewhere within a given interval) and 'everywhere' or □ (i.e. an event is true throughout a given interval). These can then be used to produce safety requirements, e.g.

$$\mathit{safety\_commitment} = \Box\, \neg \mathit{function}(\mathit{param1}, \mathit{param2}...)$$

which means *function* (which may in turn be composed of other functions) must not be true within a given interval. Functions are system-specific. As in CSDM, these statements are treated as basic or intermediate events and can be combined as normal by using normal logical gates. Based on the results of the fault tree analysis, formal safety requirements can be derived using the formalised events and more formal semantics of the logic gates. For example, if a system failure could be caused by X AND Y, the designer may add a safety requirement to the system to prevent this combination from becoming true, e.g. $\Box\neg X \vee \Box\neg Y$. If either X or Y can never be true, then X AND Y can never be true either.

The main disadvantage to event-based approaches like these is that the temporal information is entirely encapsulated within the descriptions of the events, not the logic of the fault tree structure itself, and so cannot take part in the qualitative analysis of the system. Instead, the cut sets are derived in the usual fashion and the temporal information dealt with separately. Unfortunately, this means that the potential for simplification and reduction due to the temporal information is lost, because the temporal data is not being analysed directly.

The advantage, of course, is that it allows for a much more precise specification of the failure behaviour of the system. In both of these approaches, this is done for the purposes of defining the safety requirements for a system, particularly software-based systems, and as such the precision is necessary to ensure that the system meets those requirements. The formalised semantics of CSDM and Duration Calculus help to remove any ambiguities in the meaning of the fault tree and ensure that the fault tree represents a more accurate model of the system failure behaviour.

2.4.6 TAND temporal logical connective

The TAND, or AND-THEN gate, is a gate-based approach that follows in the footsteps of the TFT and DFT approaches by attempting to represent temporal information by using a new type of gate. Unlike those two methodologies, however, the TAND is characterised by simplicity – it is just a single gate. Wijayarathna *et al.* (1997) deemed the Priority-AND to be insufficiently defined and instead proposed the AND-THEN gate (TAND) to replace it. By more strictly defining the semantics of the TAND gate, and by redefining the AND gate, they are able to precisely represent all 13 of Allen's different temporal relations (see Figure 14). The TAND gate is meant to be a useful replacement for the traditional Priority-AND gate as it is designed to be much more specific; in particular, it allows the fault tree to distinguish between a situation where one event occurs some time after another and a situation where one event occurs immediately after another – something the original PAND gate cannot do. Furthermore, it can be used as a building block to create more complex temporal expressions, and since it can be built up to represent all 13 temporal relations, in theory it can be used to represent just about any kind of interaction between two or more events.

The TAND gate is denoted by the Π symbol and it represents the situation in Allen's temporal relations known as 'P meets Q', where one interval immediately follows another with no overlap; this is similar to the 'next' or 'chop' operators found in many temporal logics. The normal AND gate is also redefined to mean 'P equals Q', i.e. the two events begin and end at the same time. However, the creators of the TAND also recognise the requirement for a more general AND, in addition to the re-defined AND. The TAND gate is not necessarily the only way to solve the problems with the traditional PAND gate, but it overcomes one of the central problems: namely, its ambiguous definition. The creators attempt to define the semantics for the TAND gate more precisely than those of the PAND gate and try to show how it may be used in qualitative analysis, enabling a fault tree analyst to more precisely define temporal safety requirements and more accurately specify the failure behaviour of dynamic systems. Presently, the TAND gate is only capable of qualitative analysis, but as further work, the creators propose investigation into how the TAND gate may be quantified.

The TAND is designed around the idea of states: at any point in time, for two events p and q, either one is true, both are true, or both are false; the redefined AND gate (represented by '$\wedge$') facilitates this because $p \wedge q$ now means that they are both true at the same time, and $p \wedge \neg q$ means p is true at this point but q is not. Expressions like these represent the state of the system at any given point in time. The TAND gate can then be used to string these states together to introduce a kind of sequence; for example, $p \, \Pi \, q$ - or more explicitly, $p \wedge \neg q \, \Pi \, \neg p \wedge q$ – means that first p is true and q is not, and then immediately afterwards, q is true but p is not. This could represent some kind of switch from a primary p to a secondary q, for example. More complex expressions are possible by combining more states and more TAND gates. Notably, the TAND does not obey several Boolean laws; e.g. the Boolean law $X \wedge \neg X \Leftrightarrow$ *false* does not apply, as $p \, \Pi \, \neg p$ instead means that first p is true, and then it becomes false, and though the Commutative Law states that $X \wedge Y \Leftrightarrow Y \wedge X$, for the TAND, $p \, \Pi \, q \neq q \, \Pi \, p$. It does, however, obey the Associative and Distributive Laws.

The philosophy behind the AND-THEN gate is that it attempts to be usable with any particular formalisation of time, whether point-based or interval-based. This is meant to match the flexible philosophy underlying FTA itself.

> "Both point based and interval based temporal models first describe a model to represent time, and subsequently use a predefined time model to define events and activities. This is the main reason for the problems in these models.

However our approach is different. We do not use any time model; instead we use the logical truth status of two events at any given time and ascertain whether they occur simultaneously or not. If they do not occur simultaneously, an event should follow the other event." (Wijayarathna *et al.*, 1997, pg. 2)

By attempting to be compatible with any representation of time, the TAND gate aims to cater for general applications where the exact time of an event occurrence may not be known whilst retaining a precise definition and therefore the possibility of rigorous analysis. In practice, though, the TAND gate is not as independent of different time models as it is intended to be, and it is worth looking at it in more detail to see where its particular failings lie.

The foundation upon which this all rests is the definition of an event in the TAND framework:

"In our approach to represent temporal relations, we do not consider whether an event is a state transition or state occurrence. What we consider is whether an event occurs or not. In that way, we treat state transitions and state occurrences on the equal footing. In other words, we do not consider whether an event occurs in a point in time line or in a time interval." (Wijayarathna and Maekawa, 2000, pg. 2)

This is a laudable goal, but it inevitably leads to a certain amount of ambiguity. Firstly, the *Fault Tree Handbook* clearly states that:

"From the standpoint of constructing fault trees we need concern ourselves only with the phenomenon of occurrence." (Vesely *et al.*, 1981, p V-1)

In other words, only the *occurrence* of a fault is important, not the *existence* of a fault. For that reason, fault tree events are normally a kind of state transition – they change the state of the system into a degraded state and ultimately into a failed state.

However, whether or not the TAND's events are representing state transitions or state occurrences, in practice they behave very much like states: they can become true, stay true for some period of time, and then become false again, or vice versa. The example tree given in Wijayarathna & Maekawa (2000) models the scenario where a car passenger is injured when the air bags are not deployed when an accident occurs, and the sequence of events is described using a PAND gate as follows: "Accident detected" PAND "NOT Airbags released". Later in the same paper, a supposedly equivalent fault tree with a TAND is shown:

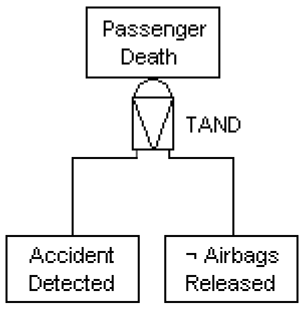


Figure 19 - Example TAND fault tree (from Wijayarathna & Maekawa, 2000)

There are two things to note here. The first is that the fault tree in Figure 19 is not necessarily equivalent to the PAND version. Although the semantics of the PAND gate are not well defined, as explained in section **2.4.1**, a PAND gate does not always mean that two events must be immediately consecutive, i.e. that in X PAND Y, Y must immediately follow X. It is also possible for two events to occur in sequence but with some interval of time between them. Although the PAND can include this possibility, the TAND cannot.

To represent this situation – 'after', in Allen's temporal relations – using the TAND, a more complex solution is needed, as shown in Figure 20:

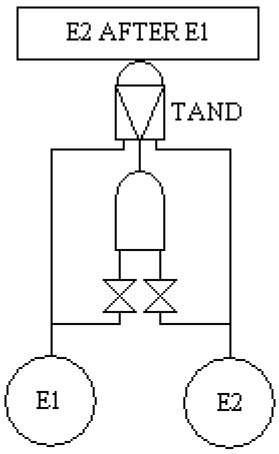


Figure 20 – The AFTER relation, using a TAND (from Wijayarathna & Maekawa, 2000)

The second thing to note is that, from their description, the child events of the TAND are meant to represent instant occurrences of events – the detection of the accident and the release of the airbags respectively. However, the semantics of the TAND gate mean that the expression $p \,\Pi\, q$ is equivalent to $p \wedge \neg q \,\Pi\, \neg p \wedge q$, and thus in this case, the fault tree represents the following:

(Accident Detected $\wedge$ *Airbags Released)* Π *(Accident Not Detected* $\wedge$ *Not Airbags Released)*

which is clearly nonsensical. Here, not only is the accident first detected and then undetected, the airbags are initially released and then not released. The reason for this confusion is that, although the event descriptions indicate that the events are meant to represent instant occurrences, the TAND gate treats them as though they were states which can go from true to false and false to true.

The event inputs to a TAND gate therefore only make sense if they represent the truth value of a state variable, such as the existence of a fault or status of a component. If TAND events represent the values of states, then expressions like $p \Pi q$ make sense (e.g. p could mean "Primary component operating" and q could mean "Secondary backup operating"; when p fails, q is activated in its place), whereas if the events can also represent the state transitions or event occurrences, then $p \Pi q$ becomes nonsensical: an event representing a state transition is then not only able to 'un-occur', i.e. go from true to false (e.g. the 'accident detected' event above), but its use with the TAND also necessarily implies that it has a finite duration, and since a state transition simply moves a state from true to false or vice versa, the value of the state *during* the transition is undetermined.

But if, as seems to be the case, TAND events can only meaningfully represent states, then they are no longer fully independent of the time model being used. A state-based system is often represented using a discrete point-based model of time, but such a model is difficult to use with this interpretation of the TAND because of the gap issue – the expression $\neg p \Pi p$ is false at one point and true the next, but as shown in Figure 13, two consecutive states must either overlap on the same point (which is impossible in this case, as it would mean both $\neg p$ and p are true at the same time) or leave a gap between the points in which the value of p and $\neg p$ are undetermined. Therefore only an interval-based time model is really compatible with the TAND gate.

Unfortunately, this is not the end of the problems. Even if TAND events represent the existence of a fault rather than the occurrence of a fault, then it is still possible for a fault to cease to exist; this implies that the fault is either repairable or temporary. However, the *Handbook* states that faults are persistent:

"Under conditions of no repair, a fault that occurs will continue to exist."
(Vesely *et al.*, 1981, p V-1)

Lastly, while the TAND gate may be useful as a basic building block to represent sequences of states (particularly system or component operating states), it is a little *too* basic for general use in fault trees, as most of the temporal relations that would be used by an analyst require them to build more complex compound gates. For example, the 'after' operator given earlier in Figure 20 is defined as follows:

$$E1 \wedge \neg E2 \; \Pi \; \neg E1 \wedge \neg E2 \; \Pi \; \neg E1 \wedge E2$$

which strictly means "E2 occurs some time after E1, but not immediately after". To generalise this further, so that E2 could either occur immediately after E1 or after some amount of time, it would be necessary to say:

$$E1 \; \Pi \; E2 + E1 \wedge \neg E2 \; \Pi \; \neg E1 \wedge \neg E2 \; \Pi \; \neg E1 \wedge E2$$

which now simply means "E2 occurs after E1". Although these sorts of compounds could be stored as ready-made combinations, or even made into specific gates, they are rather unwieldy and would require some sort of library of compound gates to be set up (e.g. to represent the 13 relations, or combinations thereof). To use them directly would result in colossal and quite unmanageable fault trees.

Generally, although the TAND gate has some laudable goals and certain advantages (e.g. it can represent recurring events, something which many other approaches cannot do, and it is well suited to describing sequences of states), in reality it suffers from a number of ambiguities relating to its definition as a kind of state-based connective, making it difficult to use with any degree of confidence as part of a temporal fault tree analysis.

2.4.7 Formalising FTA with Temporal Logics

There are a number of other approaches that seek to formalise the semantics of FTA using temporal logics and focus less on extending FTA to analyse dynamic systems. Bruns and Anderson (1993) attempt to formalise FTA to enable the verification of safety-critical systems by comparing the results of a fault tree against the behaviour of the system model. They point towards a number of ambiguous definitions in the *Fault Tree Handbook* and attempt to find a more robust definition. One issue raised is the nature of events – a recurring theme, as the discussion on the TAND gate above illustrates. Another is the nature of the AND gate and whether or not it implies temporal simultaneity. Thirdly is the issue of causality and immediacy in gates, i.e. whether the inputs of logical gates represent immediate causes or not.

Bruns and Anderson treat events as conditions having a duration rather than instantaneous occurrences, in effect modelling events as states. This is also reminiscent of the approach taken by the TAND gate. Bruns and Anderson further suggest that the AND gate should represent the simultaneous occurrence of inputs (again, like the TAND gate approach). They then use *mu calculus* (a type of modal temporal logic) to present three possible semantics of fault trees.

The first is a propositional semantics in which causality is immediate and which is described as being closest to the informal description in the *Fault Tree Handbook* (in which events provide "immediate, necessary, and sufficient causes" (Vesely *et al.*, 1981, pV-6) for the occurrence of each intermediate event or top event). In this approach, when the inputs of a gate occurs, the output of the gate is immediately true, i.e. the inputs are necessary and sufficient to cause the immediate occurrence of the gate output. This is not regarded as a temporal semantics by Bruns and Anderson.

The other two possible semantics are temporal in nature. The first defines an *even* operator to represent eventuality (i.e. equivalent to the *sometime* or *eventually* F operator in CTL and PTL). This operator is then used to indicate that, once the inputs of a gate are satisfied, the output will eventually become true, thus including a potential delay between cause and effect. However, the inputs to an AND gate are still simultaneous. The second temporal semantics instead defines a *prev* (previous) operator to indicate that, at some point in the past, the inputs to the gates were true. Thus the first approach is future-oriented and the second approach past-oriented.

This type of temporal redefinition of existing gates is designed to enable fault trees to be used for verification purposes rather than analysis purposes; Bruns and Anderson recognise that once either of the two temporal semantics are introduced, it is no longer possible to manipulate the tree using propositional laws to obtain minimal cut sets.

Schellhorn *et al.* (2002) similarly attempt to formalise the semantics of fault trees – particularly fault tree events – but use ITL rather than mu calculus. The ITL used is extended with continuous semantics based on Duration Calculus in order to allow the possibility of representing continuous changes with durations, e.g. an acceleration of 1 m/s^2 for 10s. Conditions expressed in this continuous ITL are then assigned to each gate in the fault tree and verification of these conditions confirms that the basic events will indeed cause the top event. This approach makes a clear distinction between gates that merely *decompose* intermediate events (terming them "D-gates") and gates that serve to separate causes from consequences (cause-consequence gates or "C-gates").

Schellhorn *et al.* argue that the decompositional Boolean approach (as in the *Fault Tree Handbook*) does not accurately represent the fact that causes must occur strictly before consequences and that the simple equivalence normally used (i.e. consequence = cause OR cause, consequence = cause AND cause) is inaccurate and cannot be formalised as a result. In the particular case of the AND C-gate, Schellhorn *et al.* further distinguish between a *synchronous* C-AND gate (i.e. all causes occur simultaneously) and an *asynchronous* C-AND gate (i.e. causes do not need to occur simultaneously). The result is a set of seven logical/temporal gates (D-OR, D-AND, C-OR, synchronous C-AND, asynchronous AC-AND, plus a C-INHIBIT gate and a D-INHIBIT gate) that formalise the semantics of the fault tree whilst retaining the meaning of minimal cut sets (i.e. if none of the minimal cut sets can ever be true, then the top event can never be true either). However, it is not clear how the minimal cut sets are to be calculated given the new semantics of the expanded set of fault tree gates; although the D-OR and D-AND gates would presumably still obey Boolean laws, the C-gates presumably do not.

Xiang (2005) follows a similar approach in trying to formalise fault trees. Unlike Bruns & Anderson, Xiang takes the position that fault trees are based on occurrence and creates a discrete, linear temporal logic with real time descriptions that includes both past and future operators: *next*, *previous*, *eventually*, *once*, *henceforth*, and *has-always-been*. These operators can also be augmented with real time constraints by adding a deadline, e.g. "*some time in the future before deadline* d". Time units are converted into the smallest relevant unit, such as a second, and these are the units used by the *next* and *previous* operators, e.g. the next second or the previous second. Unlike the previous approach by Schellhorn, Xiang's approach is event-based rather than gate-based; existing gates like the INHIBIT gate are given conditioning events in temporal logic and basic events are also assigned descriptions containing temporal logic.

As with the other approaches, however, it is not clear how the new temporal semantics affects analysis of fault trees, whether quantitative or qualitative. All three of these approaches are intended to formalise fault trees to better enable specification or verification of systems using fault trees. Since analysis of those systems with fault trees is not the primary aim, the details of how to conduct an analysis are omitted.

One exception to this trend is the work of Güdemann *et al.* (2008) into computing "ordered minimal critical sets" (analogous to minimal cut sets) using DCCA. This approach presents a method of automatically synthesising temporal fault tree results containing two of Pandora's temporal gates (see chapter 3), a Priority-AND (PAND) and a Simultaneous-AND (SAND). The approach uses DCCA (Deductive Cause Consequence Analysis; see Ortmeier, 2005) to obtain unordered minimal critical sets and then applies a partial ordering to them using a temporal semantics formed from CTL* and LTL. The result is a set of ordered (or partially ordered) minimal critical sets that can define either a sequence of events (using PAND) or a simultaneous occurrence of events (using SAND). This process is called *Deductive Failure Order Analysis.*

However, while this approach is capable of automatically analysing temporal failure ordering in a system, it uses model-checking to determine the temporal relations between events *after* minimal cut sets/critical sets have been generated, rather than being able to analyse an existing fault tree containing temporal gates; in this way it serves more as an alternative to a temporal fault tree analysis (although producing the same sort of results).

2.4.8 Other related approaches

There are other approaches aimed at solving the problem of modelling dynamic failure behaviour without using fault trees. Although these are not directly relevant, since the goal of this thesis is to extend FTA and not to replace it, it is worthwhile summarising some of them and how they overcome the same problems faced by FTA-based approaches.

Boolean-Driven Markov Processes (BDMPs) take a similar approach to DFTs by combining Boolean logic with Markov modelling for the representation of dynamic elements of the system (Bouissou & Bon, 2003). BDMPs are modified versions of fault trees that include additional links to represents dynamic dependencies in the model, e.g. standby components, and in fact a BDMP with no such dependencies appears almost identical to a fault tree. The BDMP approach can also make use of Petri nets for particular cases. In a BDMP, a Markov process is associated with every basic event in a fault tree or Boolean function and additional 'triggers' can be defined between gates or events, which move a Markov process from one mode to another. The fact that BDMPs are based on fault trees means it is still possible to calculate minimal cut sets; however, these do not take into account the additional dynamic dependencies modelled by the Markov side of the BDMP. The creators also attempt to assuage the problem of combinatorial explosion in Markov chains by exploiting mathematical properties of BDMPs and enabling the basic event Markov processes to be trimmed; however, although this can help minimise the problem, it does not avoid it entirely.

Another Markov-based approach is the work of Bucci *et al.* (2008), which automatically creates dynamic event trees (DETs) and dynamic fault trees (not the same as the DFTs described earlier) from Markov models combined with the cell-to-cell mapping technique (CCMT); this contrasts with DFTs, which produce Markov chains from fault trees instead. As with DFTs, it enables a separation of dynamic and static elements of the system so that non-dynamic subsystems can be analysed using traditional techniques. This helps minimise the combinatorial explosion problem by requiring Markov modelling only for a subset of the system. The dynamic fault trees are generated from dynamic event trees and are event-based in nature; the fault tree structure is purely Boolean, but the events are given a 'time stamp'. When combined with AND gates, this enables the fault tree to represent events that must occur at certain times or within certain intervals etc. In practice, this is similar to the CSDM-style event-based systems, where the temporal information is included as a separate condition.

Several other approaches use Petri nets as a solution. Petri nets are graphs that represent events; each event is a transition that leads from one 'place' to another 'place'. Places can represent preconditions for events and consequences of events and can store 'tokens', which simulate the operation of the system: the movement of a token through a Petri net represents the changing state of the system being modelled. Like fault trees, they can be used as a graphical representation of the relationships between events, allowing an analyst to see the effects of events (e.g. failures) on the system. They also include capabilities for explicitly representing concurrency and asynchronicity, by having multiple parallel transitions between the same places and allowing multiple tokens to pass through the system, enabling the modelling of parallel and concurrent systems.

Adamyan and He (2002) propose the use of Petri nets to allow the quantification of sequential failures. Instead of using fault trees to represent the sequential or dynamic failure behaviour of the system, a Petri net is used. An approximation method is used to calculate the probabilities of system failure and, by using a reachability tree derived from the Petri net, it is also possible to calculate the equivalent of minimal cut sets. Petri nets offer an advantage over Markov modelling in this scenario because they do not increase exponentially as Markov chains do.

Buckhacker (2000) also proposes the use of Petri nets in the form of an "extended Fault Tree (eFT)" notation, which combines Petri nets and fault trees. The principle is to be able to model multi-state systems. eFTs allow an analyst to build a fault tree with extra components that represent states, allowing the modelling of both failure and repair dependencies. The eFT is then automatically converted into an equivalent stochastic Petri net that can be solved to determine the probabilities of the failure events in question. As with DFTs, the eFT approach can also make use of modularisation to improve the efficiency of the conversion process; however, this can be problematic in fault trees with lots of stochastic dependencies across branches.

Another extended fault tree approach is proposed by Codetta-Raiteri (2005). Because it is more efficient to analyse a Petri net using Markov chains than a DFT using Markov chains, Codetta-Raiteri suggests a method of converting DFTs into a generalised stochastic Petri net and then analysing it with continuous-time Markov chains. Modularisation once again appears by allowing only dynamic modules of the DFT to undergo the conversion process.

State/Event Fault Trees (SEFTs) are a hybrid approach between state diagrams and fault trees, proposed by Kaiser *et al.* (2007). SEFTs make an explicit distinction between states (e.g. "safety valve is defective") and events (e.g. "pressure exceeds critical level"). They can both be combined by logical gates and can result in more events (e.g. "boiler explodes"). Gates include not just AND and OR gates, but also NOT gates, Priority-AND gates, and History-AND gates (which remembers past events), as well as non-Boolean based gates like ENTER, UNTIL, LEAVE, UPON, DELAY, DURATION, COND (i.e. condition), and FLIP-FLOP gates. Gates can also be given timed parameters to indicate that events must occur within a certain time of each other. SEFTs can then be quantitatively analysed by means of timed Petri nets, but it is not clear if qualitative analysis is possible. In addition, the SEFT approach suffers from the same state-space explosion problem experienced by other Markov and Petri net based approaches.

2.4.9 Conclusions

By looking at the different approaches taken by solutions proposed thus far, it is clear that there are a number of difficulties still to overcome and plenty of scope left for further investigation; no single solution has offered a way around all of the difficulties, and not all of them are meant for the same purpose. Very few approaches have focused particularly on qualitative analysis and none have taken into account the possibility of contradictions occurring during logical reduction of the fault tree.

Dynamic Fault Trees are probably the most comprehensive and well established of the different approaches. However, they are specifically designed to enable the quantification of dynamic models using Markov models and qualitative analysis of DFTs is a secondary priority that has received far less attention. As a result, DFTs are much less useful if failure and repair rate data is not present for the model. Neither the ZBDD qualitative analysis approach nor the CSSA approach take into account the potential for contradictions and redundancies due to temporal constraints; the ZBDD method explicitly removes any temporal information during the logical reduction of the fault tree, while the CSSA approach generates a cut sequence set but apparently does not check the set for any contradictions etc. Nor does either method seem to distinguish between simultaneity and sequence, despite the fact that an FDEP gate specifically triggers the occurrence of multiple events (presumably all at the same time). The main reason for this is the lack of a temporal logic underlying the DFT gates, which makes it difficult to interpret them in terms of a logical rather than probabilistic analysis. Although it is possible to extract qualitative data from the Markov models, this is much more complicated and considerably more computationally expensive.

On the plus side, DFTs do not require a great deal of extra information and introduce only a small number of new gates that fit relatively well into the fault tree structure; this means it is comparatively easy to learn and to use, and the use of the modularisation algorithm together with the combined Markov/BDD approach helps overcome some of the issues involved in using Markov chains. Furthermore, as already mentioned, some approaches that eschew the use of Markov chains altogether have been proposed. The fact remains, however, that DFTs were simply not designed for qualitative FTA.

Temporal Fault Trees are more complex than DFTs, but their goal is different; TFTs are meant to be used to diagnose faults in an operational system, to aid maintenance and repair, and to achieve this, it is necessary to specify the temporal behaviour of dynamic systems more precisely by using a more formal temporal logic, PLTLP. TFTs also introduce a lot of new gates with relatively complex semantics, at least compared to the PAND gate and the gates of the DFT approach, and it is not entirely clear how these relate to ordinary Boolean gates or whether it is possible for contradictions to emerge; however, TFTs can in theory be converted into normal fault trees by converting the temporal gates to INHIBITs/ANDs and representing the temporal information as a conditioning event instead. In that case, however, it becomes impossible to use the temporal information during the qualitative analysis. Although TFTs *can* be analysed both quantitatively and qualitatively, the analysis is a retrospective one, meant to be done by comparing a trace or log of the system operation against the modelled failure behaviour of the system (as represented by the TFTs).

The TAND gate is simpler than both the TFT and DFT approaches, but it suffers from a number of serious problems relating to its definition: most importantly, it treats events as states. This means that the semantics of a fault tree containing TANDs is no longer entirely clear. Furthermore, although the TAND gate is very flexible and designed to be used as a building block in larger compound expressions, such expressions can easily grow very large, which leads to more complicated fault trees which will often include NOT gates. It is not clear whether or not the presence of NOT gates under TAND gates will cause a fault tree to become non-coherent, mainly because it is not particularly clear what a TAND gate really means. Despite these difficulties, the TAND gate does offer a number of good ideas and it does at least attempt to remain independent of any one form of temporal representation. It also allows for the representation of repeating events, although so far only limited qualitative analysis is possible and there is no mention of logical contradictions etc.

The event-based approaches, e.g. CSDM, the Duration Calculus, and the matrix-style dynamic fault trees, are designed to be able to formally represent the temporal constraints on events in a fault tree. To this end, they represent the temporal information as part of an event, and not as a new type of temporal gate in the fault tree itself. The advantage of this type of approach is that existing analysis algorithms can continue to analyse the fault tree, but it also means that those algorithms cannot take advantage of the temporal information to perform additional checks for redundancies or contradictions. These approaches are designed more to improve the precision of the fault tree event semantics rather than extend the fault tree algorithms to be able to analyse more dynamic systems. In theory, this approach could be combined with one of the temporal gate approaches, and indeed the creator of the TFTs mentions this as a possibility for future investigation.

Other formal approaches, such as those of Bruns, Schellhorn, and Xiang, have similar goals – the formal specification and verification of failure behaviour by assigning temporal logic semantics to fault trees. Some use new gates for this purpose and others simply add additional semantics to events and existing gates, but none of the approaches explain how the resultant tree is meant to be analysed. In event-based approaches, presumably existing techniques can be used (as long as the semantics of the gates have not been altered), but in gate-based approaches like Schellhorn's, it is not clear how a qualitative or even quantitative analysis could be conducted on a temporal fault tree, as the logical structure of the fault tree has been radically altered.

Finally, there is the traditional Priority-AND gate. The PAND gate is included unaltered in the DFT approach and the TAND gate is an attempt to replace it; there are also a number of different ways to perform quantitative analysis on them. But nobody has hitherto suggested a better, more thorough definition for the gate, nor have they considered the potential for redundancies and contradictions it presents. Qualitative analysis on PAND gates is consistently either ignored or overlooked. Despite this, the PAND gate still remains useful as a way of specifying sequences of events and, as evidenced by its inclusion in the DFT framework, it still has an important role to play if its deficiencies can be overcome.

Table 3 provides a more concise summary of the salient features of the various temporal FTA approaches described above.

Approach	**Scope**	**Complexity**	**Logic**	**Qualitative Analysis**	**Contradiction handling etc**
Priority-AND	Very general; can be applied to any event sequence .	Very simple, but ill-defined.	Undefined.	No.	No.
TAND	General; can also be used to construct compound	Simple in theory, but ambiguous in practice.	State-based in practice. Uses only Boolean laws for	Using Boolean laws, no mention of temporal	No, but given the state-based nature, it is not clear how they

	expressions.		reduction. Uses interval-based time in practice.	reduction possibilities.	may occur.
DFT with ZBBD	Fairly general in application; can represent dependencies and standby components.	Moderate; limited number of gates, easy to understand, but analysis is potentially complex.	Sequential logic – one event causes another to occur; events represent occurrences.	Yes, but the Markov method is very expensive and the ZBDD method extracts temporal info.	No. Simultaneous occurrence is also not mentioned.
DFT with CSSA	As DFT.	Simpler than standard DFTs as it avoids use of Markov chains.	As with DFTs.	Yes.	No. Furthermore, simultaneous occurrence is apparently not handled either.
House matrix dynamic FTs	Designed for fault monitoring and diagnostics.	Relative simple but requires a great deal of information.	State-based using a point-based discrete timeline.	No.	N/A. Contradictions should not arise.
TFTs	Designed for fault monitoring & diagnostics, detecting faults after they have occurred.	Quite complex; lots of new gates and a formal temporal logic.	PLTLP – Propositional linear temporal logic; point-based time.	Yes, but only by converting TFT into normal Boolean fault tree.	N/A. Contradictions should not arise (as TFTs are used *after* failures occur).
Duration Calculus	Designed to improve specification of temporal constraints.	Moderate; expressions are contained in basic events, but expressions have a specific semantics.	Duration Calculus (a form of ITL). Continuous linear time.	No – although standard methods can be used to produce cut sets containing temporal info.	Not mentioned, but any contradictions would be contained in the event descriptions.
CSDM	Designed for improved specification	Quite complex; formal	CSDM, a form of temporal predicate	No – although standard methods can	Not mentioned, but contradictions

	of fault tolerance.	language used to describe temporal relations between events.	logic. Continuous linear time.	be used to produce cut sets containing temporal info.	would probably be accounted for when identifying enabling conditions.
Bruns / Anderson	Aimed at formal verification.	Moderate; two temporal semantics.	*Mu* calculus. Events have duration.	No.	No, but logic may prevent them.
Schellhorn *et al.*	Formal verification.	Complex; many new gates.	ITL + Duration calculus.	No.	No, but logic may prevent them.
Xiang	Formal verification.	Quite complex, event-based.	Linear, discrete logic.	No.	No, but logic may prevent them.
DCCA with DFOA	Formal verification.	Quite complex, model checking.	CTL* + LTL	Yes, in separate stages.	Not mentioned.

Table 3 – Comparison of dynamic/temporal fault tree approaches

The precise qualities of any solution will depend on the goal it is intended to fulfil. Both PAND and TAND gates are meant as simple additions to the fault tree and, in the case of the TAND at least, designed to fit in with existing Boolean reduction methods. Both are also very simple to use and consist of only one new gate. At the opposite end of the spectrum are the formal specification and verification approaches and TFTs; these use more complex, higher order temporal logics and introduce a lot of new operators, either in the form of gates or in the form of functions in temporal expressions. However, they are designed to increase the expressiveness of fault trees by more precisely specifying temporal constraints and so are not designed to improve the analysis of fault trees; instead, they are meant to make the results of any standard analysis more informative. In the middle is the DFT approach, which adds a small number of relatively simple new gates and can represent a fairly wide range of situations with them. It is also relatively simple but is designed for quantitative analysis; although qualitative analysis techniques have been proposed, they are not as detailed and do not cover all of the implications of incorporating temporal information into the fault tree.

It would seem, therefore, that there is still scope at the general, simple end of the spectrum for a solution that allows the representation of simple temporal relationships between events in fault trees without introducing complex new semantics based on higher-order temporal logics or altering the structure of the fault tree. Most crucially, *none* of the approaches mentioned above offer the ability to deal appropriately with the various contradictions and redundancies that may arise as a result of introducing temporal relations into the tree. In general, this is because the qualitative analysis elements of these techniques (if present at all) are generally weaker than their quantitative analysis algorithms; this also reduces their utility if sufficient quantitative data is not available.

From a comparison of existing solutions, any solution that intends to fulfil the objectives set out in the **Introduction** ideally need to be:

- General in application, like the DFT, TAND, and PAND approaches; this allows it to be applied to a wide range of different situations and systems while still allowing the flexibility to allow it to perform more specific tasks in the future, like verification or specification.
- Simple to use; ideally, a simple Boolean-style sequential logic, one that avoids the pitfalls of the TAND's state-like approach but that provides more thorough semantics than the vague DFT gates without resorting to full temporal logics like PLTLP.
- A full qualitative analysis method that accounts for the potential for contradictions and simultaneous occurrences that can arise as a result of the temporal information in the fault tree.

In terms of problems to avoid, any solution should:

- Avoid committing to a single representation of time if possible, e.g. a more general, relative time approach like DFTs compared to the absolute, point-based timelines used in TFTs.
- Represent events as occurrences and not as states to avoid the problems inherent in the TAND.
- Define the gates such that it is clear whether or not events can overlap or occur at the same time.
- Try to avoid a complex logic based on existing temporal logics like ITL etc, as these may complicate qualitative analysis and do not always fit well with the Boolean-based fault tree structure.
- Use a small number of temporal gates, rather than lots of complex temporal gates with specific uses or encapsulating the temporal information entirely within events.

The next chapter presents an attempt at creating a solution that takes these recommendations into account.

3 Pandora

"If all else fails, immortality can always be assured by spectacular error."
- John Kenneth Galbraith

3.1 Introduction: The Birth of Pandora

The previous chapter describes several existing techniques of incorporating temporal information in fault trees. This chapter presents a new method: **Pandora**. As explained previously, there are few general solutions to this problem, especially ones which can perform true qualitative analysis on temporal fault trees; this is significant because, as stated by the *Fault Tree Handbook*, "a fault tree is not in itself a quantitative model. It is a qualitative model that can be evaluated quantitatively and often is." (Vesely *et al.*, 1981, p IV-1). Pandora aims to correct this deficiency by allowing the creation and analysis of temporal fault trees, fulfilling the objectives set out in the **Introduction** and taking into account the lessons learnt in the previous chapter. It is therefore intended to be general and simple with an emphasis on qualitative analysis, which is so often treated as secondary by other approaches. It is also meant to remain as close as possible to the spirit of FTA by prioritising flexibility and ease-of-use.

To that end, Pandora is based on extending the Priority-AND gate[10] – FTA's original solution to the problem. The PAND gate suffers from several flaws, but in principle it is a useful and valid addition to fault tree analysis, as evidenced by its original inclusion in FTA. The PAND is also used in the DFT methodology and the TAND was in large part an attempt to redefine the PAND; taken together with the large number of possible quantitative analysis methods available for the PAND, although the PAND is often overlooked, it is far from forgotten. If the difficulties in using PAND gates can be overcome by redefining and extending it, then it may become a more useful tool that can be successfully used as part of a new form of temporal fault trees. Furthermore, by starting from this root, it is hoped that the resulting methodology will be in keeping with the spirit of FTA.

The main problem with the Priority-AND gate is that it was never thoroughly defined – it was never clear exactly how it worked or what it meant. Pandora grew out of a simple question: what happens if you have (X PAND Y) AND (Y PAND X)? Once the box was opened, many other questions emerged, and the first goal for Pandora is to answer them.

3.2 Opening the box: The Fundaments of Pandora

As can be seen from the earlier examination of some of the other alternative temporal fault tree approaches, there are many potential problems to avoid. The best way to do this is to build Pandora up from a firm foundation, defining at each stage what the various components of Pandora mean and how they behave, thus preventing the type of ambiguities that plague both the original PAND gate and some other approaches, like the TAND gate.

In fault trees, there are a number of constituent parts, as described in Chapter 2; broadly, these can be divided into two categories: *events* and *gates*. Before the gates are introduced, representing combinations of events, it is useful to look at what the events themselves actually mean in Pandora.

3.2.1 Events

There are five types of events in normal fault trees, as described earlier. The most important is the **basic event**, which represents a basic initiating fault event – one that is not developed any further, and so has no contributory causes of its own. This is usually because the limit of resolution has been reached, i.e. there is no benefit in looking at what might have caused a basic event. Basic events are similar to undeveloped events, except that undeveloped events are typically not further developed either because its root causes are unimportant or because there is not enough information to develop it further.

These two types of events both represent **faults** and therefore indicate that something has gone wrong somewhere in the system. This could be a component failure or a command fault or an environmental factor or something else, but ultimately a fault. Furthermore, we know from the *Handbook* that we should deal only with the *occurrence* of faults, not the *existence* of faults (Vesely *et al.*, 1981, p V-1). Therefore, a basic or undeveloped event represents the occurrence of a fault: it becomes true if that fault occurs and stays false if it does not occur. Because faults in a fault tree are often assumed to be non-repairable and have a permanent effect, once an event becomes true, it stays true – thus it is not possible for an event to "un-occur". Thus events are considered to be *persistent*. Persistence is the principle that if an event occurs that changes the state of the system, then the system will remain in that state until a further event changes the system state once more.

[10] Although Pandora is not an acronym and was named for the myth, the Greek etymology of the word can be interpreted as the "time of PAND": *ora* ([*ώρα*] in Greek) means hour or "time", hence ORA of PAND.

External and conditioning events are slightly different in that they do not necessarily represent faults. External events signify an event that is expected to occur as part of normal system operation, e.g. a certain input or command being given to the system, for instance. Conditioning events are used to represent conditions or constraints that apply to a gate, and can be almost anything, from environmental conditions (e.g. temperature, pressure) to temporal conditions (e.g. for a certain amount of time) to the system being in a certain state. In the case of external events, they still represent the occurrence of a certain event, but conditional events can also represent a state. For the purposes of Pandora, conditional events that represent states are assumed to become true at the state transition, e.g. if a conditioning event states that the temperature must be above 50 degrees, it will become true if the temperature rises above 50 degrees. To model the possibility of the temperature then dropping, Pandora requires the use of a second event ("temperature drops below 50 degrees") so that if the first is true and the second is false, the temperature is still above 50 degrees. This means that conditional events can be treated the same as the other types of event.

Note that in all these cases, the event is not synonymous with the fault. A fault could possibly be repaired or be intermittent, or for external events an input to the system can be processed, and then the fault or the input ceases to exist; however, it would still have occurred, and therefore the event will still be true. As a result, when we speak of an **event** occurring in Pandora, it refers only to the first occurrence of whatever the event refers to, not the existence of whatever it represents. This means that an event can only occur at most once. Modelling repeated or repairable faults would require a series of events to represent the initial occurrence and then cessation of those faults. Thus there is no concept of an *action* representing an instance of an event, like there is in CSDM, and it is not possible for events to go from true to false as in the TAND approach.

To summarise, then:

Event
An event in Pandora represents the occurrence of something, whether a fault or a state transition. If the fault/state transition has not yet occurred, the event is false. When the fault/state transition occurs, the event instantly becomes true and remains true thereafter. Events are monotonic, i.e. they cannot go from true to false, only from false to true.

Note that an event is considered to be *instantaneous*, i.e. it takes no time to go from false to true. For events that would ordinarily have durations, e.g. "brake failure due to brake fluid leakage", or that grow worse over time, e.g. "overheating", the event is considered to become true either on demand (e.g. "brake failure when pedal is pressed") or when the effect becomes significant enough to affect the rest of the system (e.g. "25% drop in performance due to overheating").

There is also one last type of event: intermediate events. However, these typically represent combinations of two or more of the other types of events subject to certain conditions indicated by a logical (or temporal) gate, e.g. an OR gate requires at least one of its inputs to be true before the intermediate event is true and an AND gate requires all of its inputs to be true. In Pandora, intermediate events are treated synonymously with gates, therefore, for the purposes of this thesis, the first four types of events are referred to simply as "events" and intermediate events referred to as "gates". Anonymous events used as examples in this and future chapters are represented by capitals for brevity, e.g. A, B, C, X, Y, Z, or E1, E2, E3 etc. Real events in fault trees can have any name.

3.2.2 Gates

Gates are the other major component of fault trees and represent intermediate events – events caused by or composed of combinations of other events. Gates can also be inputs to other gates. As with events, gates are considered in Pandora as representing occurrences: an intermediate event occurs when one or more of its input events occur, subject to the conditions imposed by the logical gate (obviously dependent on the type of gate), at which point it becomes true[11].

Gate
A gate in Pandora is an intermediate event and represents the occurrence of a combination of input events subject to conditions imposed by the type of gate. Gates begin as false and become instantly true when their conditions are fulfilled. Gates cannot go from true to false once they have occurred.

In ordinary 'static' fault trees (SFTs), there are two main gates: OR gates and AND gates.

OR Gates

OR gates represent the Boolean OR operation: they are true when at least one of their inputs is true. There can be any number of inputs to an OR gate. Strictly speaking, an OR gate does not always represent a causal relationship: the intermediate event is not necessarily *caused* by one or more input events occurring. Instead, an OR gate is often a restatement or refinement; for example, the intermediate event "valve fails shut" could be represented as an OR gate with a number of inputs representing different specific ways the valve could fail shut, e.g. through human error (e.g. someone mistakenly shut it or forgot to open it) or through a mechanical fault (e.g. corrosion, damage etc). In this case, the inputs to the OR are different restatements of the event, not different events that can cause it.

AND Gates

AND gates represent the Boolean AND operation: they are true when all of their inputs are true. Like OR gates, there can be any number of inputs to an AND gate, but unlike OR gates, AND gates *do* usually represent a causal relationship: the intermediate event is caused by a combination of all input events, as a subset of those events is not sufficient on its own to cause the intermediate event. In this way, the AND gate can be thought of as a compound event with two or more sub-events.

Other gates

There is also the rare INHIBIT gate, which in practice is a type of AND gate. INHIBIT gates only have one normal input (e.g. a basic or undeveloped event), but also have a conditioning event attached. In Pandora they can be treated in logical terms as AND gates with two events, one basic and one conditioning, because the conditioning event still represents an occurrence of the condition becoming true rather than the state of the condition. NOT gates are sometimes included in fault trees but these are dealt with separately (see section **3.4**).

Some other approaches (e.g. Schellhorn's formalisation), as described in the previous chapter, choose to divide these gates into two versions – one version explicitly describing a decompositional relationship and the other explicitly describing a causal relationship. Part of the reason for doing this is to formalise the temporal relationship between the inputs of a gate and its outputs; in a decompositional gate, the inputs and output are the same thing (e.g. "valve rusted shut" is a type of "valve stuck closed" failure), whereas in causal gates, the inputs are causes and the outputs represent their consequences or effects, and thus inputs must occur before outputs.

In Pandora, this sort of distinction is deemed unnecessary due to the fact that all events and gates represent occurrences and are instantaneous. A gate becomes true at the same instant its conditions are fulfilled, and this is true whether it represents decomposition (in which case it would be instantaneous anyway) or causality (in which case, the causes are modelled as instantly causing their effects). Thus an AND gate (and by extension, an INHIBIT) becomes true at the moment when all of its input events have become true, i.e. the moment at which the last event to occur becomes true, and an OR gate becomes true at the moment when the first of its input events occur. If a gate's conditions have not yet been met, then they are false.

[11] Note that NOT gates are an exception to this rule in that they can begin as true and then become false. However, the introduction of NOT gates complicates Pandora immensely and are thus best avoided; the

Furthermore, because events remain true once they have occurred, OR and AND gates do so too, since none of their inputs will become false once they have occurred. This also helps to ensure consistency, because gates are a type of event too and so should have the same persistent behaviour.

This approach does not necessarily preclude the possibility of having *delayed* effects. For example, the CSDM approach assigns temporal semantics to existing gates. The event "gas explosion" can be modelled in CSDM as the conjunction of two events ("naked flame" and "gas leak") and the fulfilment of the enabling condition ("duration of gas leak exceeds time need to attain explosive concentration"). Thus even if both input events of the AND have occurred (i.e. there is both a gas leak and a flame), the AND will only be true once the condition is fulfilled – necessitating a delay. This same behaviour can be represented in Pandora by explicitly representing the condition as an event, i.e. having a separate conditioning event to represent the explosive concentration. Once the condition becomes true, a certain time after the other two events occur, the AND gate will become true as well.

3.2.3 Time

As explained in section **2.3**, there are many different possible ways of modelling time, each with different advantages and disadvantages. The choice of time model can also have important repercussions on how time is represented in a fault tree. Some approaches specify a particular model of time, e.g. continuous linear time for CSDM, point-based linear time for TFTs etc. In these cases, the time model is referenced directly by the temporal logic – using predicates like *duration* in CSDM, or the PREV gate in TFTs. For a PREV (or NEXT) operator to make sense, it must be possible to determine what the previous or next 'moment' in time actually is. For continuous time models this is not possible, and in discrete time models, the answer depends on whether time is modelled with points or with intervals. Notably, virtually all temporal/dynamic fault tree approaches seem to use a linear system of time rather than branching time; although a fault tree can be interpreted as a time line branching into the past from the top event, representing the present moment, it is more common to think of the fault tree as representing lots of different possible linear time lines, each of which ends with the top event.

In any case, the more general temporal FTA approaches (DFTs, TAND, PAND) all attempt to avoid specifying a particular model of time (though all assume linear time models). The idea is that these approaches can then be used regardless of how time is modelled, because any explicit references to time – particularly quantitative references to a time metric, e.g. 4 seconds – are encapsulated in the events, not in the logic of the fault tree structure. This is a good choice in a

reasons for this are discussed in more detail at the end of this chapter, in section **3.4**.

general solution, because it is more flexible and does not force the analyst into certain modes of thinking. Unfortunately, as the experience of the TAND gate shows, this is not always a simple task.

Pandora is based on the Priority-AND gate, which simply introduces the concept of a sequence of events to the fault tree – perhaps the most elementary representation of time possible. For this purpose, it is not necessary to know exactly when an event occurred or how long it lasts, only when it occurs in relation to the other events in the group. This is relative time in its most broad sense, because there is no indication of how much time passes between events. In theory, this requirement also fits both a quantitative relative system of time (where the periods between events are of a known duration) and also an absolute system of time (in which it is always possible to know whether one event came before another by comparing the times at which they occur).

From the definitions of the events, we also know that events occur at most once and that it is the occurrence rather than the existence of a fault that matters. This means that the choice of a point-based or interval-based system, whether discrete or continuous, is largely irrelevant, because we only ever need to refer to a single instant, without a duration. In Pandora, only the moment of occurrence is important. Furthermore, because events and gates are all persistent – they remain true once they have occurred – the issue of 'gaps' between intervals in a point-based system is not important, because intervals are not needed: in a point-based system, we only use and compare the two start points, and in an interval-based system, we compare only the start of the intervals in question.

All of this is shown in Figure 21:

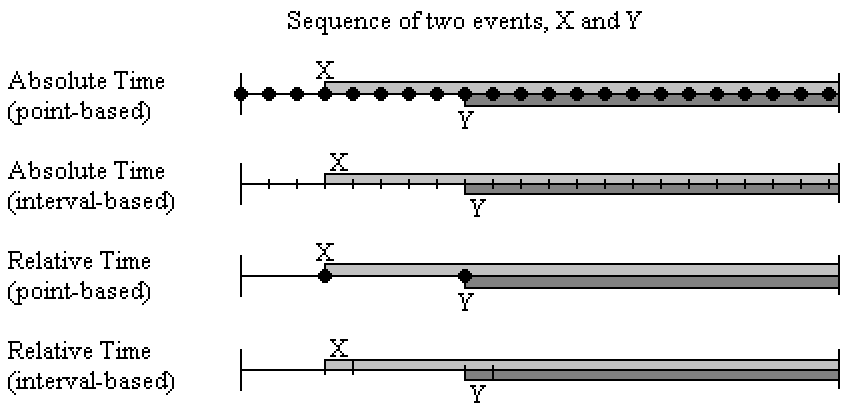

Sequence of two events, X and Y
Absolute Time (point-based)
Absolute Time (interval-based)
Relative Time (point-based)
Relative Time (interval-based)
X
Y

Figure 21 – Pandora in different systems of time

Figure 21 shows how a sequence of X and then Y can be modelled in all four of these systems of time; initially both are false, then X occurs, and then Y occurs. They remain true thereafter. It does not matter whether an event occurs at a point or in an interval, or whether there is a timeline of events or merely a relative sequence of events: what matters is the ability to determine whether one event preceded another. This is possible in all of these systems of time, because we use only the initial occurrence of the event as the moment of time in question, whether that moment is modelled as an interval or a point or anything else. This is assumed to be the 'rising edge' as seen in the figure.

The one thing Pandora does assume – in common with virtually all other temporal FTA approaches – is a linear system of time rather than a branching one. A PAND gate is deterministic in that if it is true, then its inputs always occurred in that sequence; there is no concept of events 'sometimes' occurring in some possible time lines but not all. This fits with the general concept of a fault tree, which is a deductive model working *backwards* from a system failure to determine what caused it. If a system failure has already occurred, then a deterministic sequence of events must have caused it.

There is still one outstanding issue, however. One of the ambiguities that affected the PAND gate was the question of whether or not two events occurring simultaneously could be considered to be part of a sequence. Again, this is independent of whether time is point-based or interval-based; if two events both occur at the start of the same interval, or both occur at the same point, then they occur simultaneously. This possibility was also considered by the TAND gate, which redefined the normal logical AND gate to mean that its inputs also have to occur (and end) at exactly the same time – it represented Allen's 'equals' relation.

In Pandora, there is no concept of an event 'ending', because all events and gates are persistent. Only the moment of occurrence is important. Therefore, either two events occur at the same moment or at different moments, and so there are only three basic temporal relations between two events X and Y:

- **Before** X occurs before Y.
- **After** X occurs after Y.
- **At the same time** Both X and Y occur simultaneously.

These three temporal relations are independent of any specific model of time, with the caveat that time is linear (if it was branching, it would technically be possible for an event precede another event in one timeline and precede it in another). These three temporal relations also avoid one of the problems that affected the TAND gate, namely that the TAND was true only if the second event immediately followed the *end* of the first event. In Pandora, the period of time between the occurrence of two events is simply either zero (if they are simultaneous) or non-zero (if they are not).

As a result of this, exactly one of the three relations will be true, assuming both events have occurred. It is not possible for more than one to be true at once (an event cannot occur both at the same time as and not at the same time as another event, because we assume linear time). If one or both of the events have not yet occurred, then none of the three relations will be true until they do both occur. Note that in Pandora, all three temporal relations generally assume that both events occur, but there is a possible exception: the *before* relation can also be interpreted as being true if the first event occurs regardless of whether or not the second event occurs, whereas *after* can only be true when all events have occurred (because it is impossible to state, for example, that "Y has occurred after X" unless both Y and X have occurred).

The time model in Pandora can thus be summarised as follows:

Time
Pandora assumes a linear model of time in which events are persistent but does not specify whether the model used is to be discrete, continuous, point-based, or interval-based. There are only three possible temporal relations: before, after, and simultaneous.

3.2.4 Temporal Gates – Bringing Order to the Fault Tree

Temporal gates are the means by which Pandora introduces temporal information to the fault tree, by establishing the sequence or order of events. There are three such gates, all described below.

3.2.4.1 Priority-AND Gates

Having decided on a model of time and settled on a behaviour for events, it is possible to introduce the temporal gates themselves. First and foremost is the Priority-AND or PAND gate – the gate upon which Pandora is built. The basic definition remains the same as the original – it is true if its inputs all occur in a specific sequence. Thus the PAND gate models the *before* and *after* temporal relations mentioned above and the questions that surround the original gate can now be answered:

- The sequence of events is always left-to-right – each subsequent event after the first must occur after the one to its left.
- The PAND is considered to be 'exclusive' and so inputs must occur strictly in sequence, and not at the same time. Inputs occurring simultaneously result in the gate being false.

- Because inputs to a PAND must not occur at the same time, the same event used more than once as an input to the PAND results in a contradiction (e.g. X PAND X would mean 'X occurs before X').
- Temporal relations are mutually exclusive; (X PAND Y) AND (Y PAND X) represents a conjunction of two different temporal relations (X *before* Y and X *after* Y) and so is impossible – and hence always false.

Note the exclusive nature of the PAND gate in Pandora, meaning that it is not true if any inputs occur simultaneously; the alternative is an inclusive definition, which would be true if either the inputs occurred in sequence *or* if they occurred simultaneously. An exclusive definition was chosen to ensure that the PAND gate does not represent more than one temporal relation at a time; under an inclusive interpretation, X PAND Y could mean that either X occurred before Y or that X and Y occurred at the same time, and there would be no way of knowing which was the case.

The Priority-AND gate is defined informally as follows:

Name:	Priority-AND gate
Abbreviation:	PAND
Symbol:	<
Meaning:	The PAND gate specifies a sequence. It is true if all input events occur and they occur in order from left to right. It is false if any of the events occur out of sequence or if any occur simultaneously.

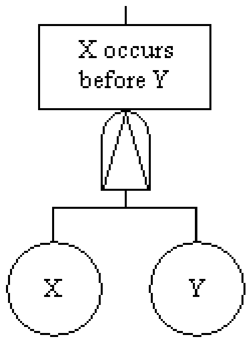


Figure 22 – Pandora's PAND gate

The '<' symbol was chosen because it indicates a relative order – X < Y intuitively means "X is less than Y", i.e. the time at which X occurred is before the time at which Y occurred. It also helps indicate the exclusive nature of the PAND (whereas an inclusive PAND would be better represented as ≤). The gate symbol given in the *Fault Tree Handbook* is retained for the PAND gate in Pandora, but notice the lack of a conditioning event; the order is always left to right for however many inputs the PAND gate has.

The PAND gate with this definition is very similar to the SEQ or SEQUENCE gate from the DFT methodology and generally similar in purpose – if not in function – to the TAND gate. It is also important to remember that the PAND gate is still fundamentally an AND gate – a PAND can never be true unless all inputs are true, just as in the normal AND gate.

3.2.4.2 Simultaneous-AND Gates

The exclusive nature of the Priority-AND gate makes the PAND less ambiguous because it only represents one temporal relation at a time – either *before* or *after*, depending on your point of view (though if read from left to right, it is more natural to read X PAND Y as "X before Y"). However, this means that there is a need for a different gate to represent the third temporal relation – *simultaneous*.

In the TAND approach, the AND gate was redefined for this purpose: it would be true if its input events were temporally equal, i.e. they all start and end at the same time. However, the creators of the TAND gate still recognised the need for a more general AND as well. Other approaches have also redefined the AND gate in this way, e.g. in Bruns and Anderson's approach. In the work of Schellhorn *et al.*, the (causal) AND gate was split into two further variants – a synchronous AND to represent the simultaneous case and an asynchronous AND to represent the case where the inputs occur at different times. Notably, the DFT approach does not specify a simultaneous version of the AND, but its FDEP gates model a functional dependency in which a trigger can cause the (presumably instantaneous) occurrence of multiple events. However, it may be that the DFT's version of the PAND gate is inclusive rather than exclusive (though this does not appear to be firmly defined anywhere).

In Pandora, the solution is to introduce another gate to model this temporal relation rather than to redefine the existing AND gate (which already has well-understood semantics). The new gate is unimaginatively named the "Simultaneous-AND" or SAND gate. It is true if all its input events occur at the same time, while a normal AND gate is true if all of its inputs occur, regardless of the order.

The SAND gate is summarised as follows:

Name:	Simultaneous-AND gate
Abbreviation:	SAND
Symbol:	&
Meaning:	The SAND gate specifies that all of its inputs must occur and furthermore that they all occur simultaneously. It is false if any do not occur or if any occur at a different time.

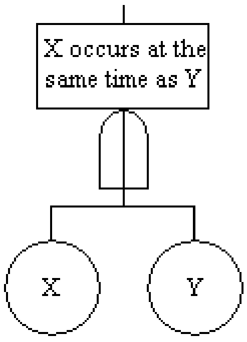


Figure 23 – Pandora's SAND gate

The SAND gate has the symbol '&', i.e. an ampersand. Not only does an ampersand itself mean "and" in general usage, which the SAND gate also does in many ways, it also literally contains the word SAND. Because the PAND is exclusive, there is no overlap in meaning between the PAND and the SAND gates: they can never both be true at the same time. This is because they each represent different temporal relations in Pandora: the PAND can represent both *before* and *after* (which are interdefinable in that X *before* Y = Y *after* X), and the SAND represents the *simultaneous* relation. As already explained, only one of these three temporal relations can be true at once, and thus the PAND and SAND are mutually exclusive.

It is also important to note that both the SAND and PAND gates fulfil the definition of gates set out in section **3.2.2**: they represent the occurrence of certain combinations of events subject to certain (temporal) conditions; if those conditions are not yet met, the gates are false, and when they are met, the gate instantly becomes true and remains true thereafter. There is no combination or sequence of events that can cause a PAND or SAND gate to become false after it has become true – they are monotonic.

Also, PAND and SAND are both types of AND gate; whenever a PAND or SAND gate is true, an AND gate with the same inputs would also be true. As explained earlier, an AND gate only becomes true when its last input to become true occurs, and this behaviour is also true of the PAND and SAND gates (which are together known as the **temporal conjunction gates**). This is easy to see of the PAND gate: because it requires its inputs to occur in sequence from left to right, it becomes true when its right-most event occurs, assuming that occurs after all the events to the left. The SAND gate requires all of its inputs to occur simultaneously, so there is no 'last' input event per se, but until they all occur, the gate is false. Because an event cannot become false after it has occurred, and because both the PAND and the SAND require all their inputs to occur before they become true, the gates too cannot become false after they occur. This is true regardless of whether their inputs are events or other gates.

It has been said that the probability of two events occurring simultaneously is vanishingly small and thus there is no need for a SAND gate, but this overlooks certain assumptions that may not always be true. Although the need for the SAND is less obvious than the need for the PAND gate, the probability of two (or more) events occurring simultaneously is only small if those events are independent. For example, the DFT's FDEP gate in practice models a simultaneous occurrence of multiple events caused by the same trigger event. If the events can have a common cause, then they may well occur simultaneously. Also, the effective meaning of 'simultaneous' depends on the system involved; it may involve a interval of time that, while short compared to the total lifetime of the system, is not strictly 'instantaneous'. For example, a system may have a safety measure that can come into effect within seconds of a particular component failure, such as a fire extinguisher that activates to put out a fire caused by an overheating component. Failure of that extinguisher within the few seconds it takes to extinguish the fire might be considered a 'simultaneous' failure because it would still result in an inability to prevent the consequences of the original component failure (i.e. the fire would still be burning).

Of course, in cases where the events are independent and the probability of the SAND occurring is thus extremely small, it may be reasonable to ignore the SAND gate, at least during any future quantitative analysis. But this does not mean that the SAND gate can *always* be ignored.

3.2.4.3 Priority-OR Gates

There is one type of possibility that has not yet been considered. So far, only conjunctions of events have been considered – but what about disjunctions of events? It is sometimes desirable to be able to specify a sequence without also specifying that all events must occur; in these cases it is enough to say that one event must occur first and then the occurrence of the other events does not matter. Pandora also provides a gate for this situation too: the Priority-OR (or POR) gate.

Name:	Priority-OR gate
Abbreviation:	POR
Symbol:	|
Meaning:	The POR gate specifies priority, and is true if its left-most input occurs before any of its other inputs or if none of the subsequent inputs occur. It is false if any of the other input events occur before or at the same time as the left-most input, or if the left-most event does not occur at all.

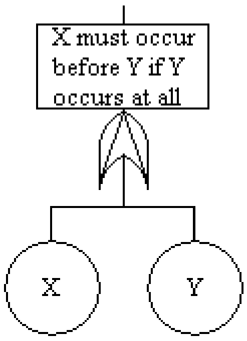


Figure 24 – Pandora's POR gate

The POR gate uses the same graphical symbol as the *Fault Tree Handbook* gives the XOR (Exclusive Or) gate. This is justified for two reasons. Firstly, the *Fault Tree Handbook* mysteriously shows the XOR gate as being equivalent to an OR gate with a conditioning event attached specifying an order, i.e. A XOR B is equal to A OR B assuming A occurs before B (Vesely *et al.*, 1981, p IV-10). This conflicts with the traditional definition of the XOR gate as an Exclusive-OR in which only one input event must be true, a definition also given in the *Handbook* on the same page. However, the traditional Exclusive-OR gate implicitly uses NOT gates to achieve this: A XOR B is traditionally equivalent to ¬A.B + ¬B.A. The *Handbook* does not include NOT gates, and using this definition would result in a non-coherent fault tree, because the occurrence of A and then B would result in the XOR first becoming true and then becoming false again. This also conflicts with the definition of a gate in Pandora, which states that a gate (and an event) can only go from being false to being true, whereas a traditional XOR gate becomes true when one event occurs and then false when the second occurs. The *Handbook*'s definition avoids this problem by specifying a sequence, i.e. it allows both events to occur as long as they occur in a certain order. This definition is more like the Priority-OR gate than the traditional XOR gate.

The second reason for the use of the symbol is that a XOR gate usually has a different symbol, an OR gate with a curved line beneath it. The two lines inside the OR symbol are more like the two lines inside the AND symbol used to signify the Priority-AND gate. Therefore, it is felt that to keep these two lines makes it easier to identify the Priority-OR as a temporal gate like the Priority-AND.

Since the POR uses the *Handbook*'s symbol for an XOR gate, one might reasonably ask the question about the possibility of a temporal XOR gate, or PXOR gate (Priority Exclusive Or gate). This would be a gate that is true when only one input is true – like the traditional XOR gate – except, as in the *Handbook*'s version, one event has priority over the other and must occur first. However, the full semantics of the PXOR are somewhat more problematic as they relate to the non-occurrence of events and the possibility of non-coherent fault trees; as a result, the possibility of a PXOR gate is discussed further in connection with NOT gates in Section **3.4** later in this chapter.

The Priority-OR by contrast is simply a temporal version of the OR gate. The OR gate becomes true when at least one of its input events becomes true. The POR is similar, but places the highest priority on its first (left-most) input. For the POR to be true, this input must also be true. Furthermore, this input must precede all of the other inputs, and thus that input has 'priority' over the others. In this it is very similar to the PAND gate. However, the PAND gate is not true until the *last* of its input events is true – and all inputs must occur. The POR becomes true as soon as its *first* input (i.e. its left-most input) becomes true, as long as none of the other inputs have occurred yet. In this respect, like the OR, it does not matter whether any of the other input events occur or not once it is true. As with the PAND, it does not include the *simultaneous* relation: a simultaneous occurrence of its priority input and any other input will cause it to stay false.

It is also important to note that the POR gate only defines a temporal relation between its first input and all of the other inputs; unlike the PAND, it does not impose any order on subsequent inputs (e.g. between its 2nd and 3rd input). In fact, PORs with more than two inputs are equivalent to POR(i_1, OR($i_2, i_3, \ldots i_n$)), e.g. X|Y|Z is equivalent to X|(Y+Z).

Because the POR is a type of OR gate, it is a **temporal disjunction gate**. In the same way that if a PAND or SAND gate is true an AND gate with the same inputs would also be true, if a POR gate is true, an OR gate with the same inputs would also be true. In Boolean logic, this can be expressed using implication, i.e.:

X PAND Y $\rightarrow$ X AND Y

X SAND Y $\rightarrow$ X AND Y

X POR Y $\rightarrow$ X OR Y

The same is not true in reverse, of course, because AND and OR do not define any temporal relationships on their inputs.

As well as being able to distinguish between the temporal conjunction gates and the temporal disjunction gates, we can also distinguish between the two **priority gates** PAND and POR, which both define sequences of events, and the simultaneous gate SAND, which says that inputs must all occur at the same time.

3.2.4.4 Behaviour of the three Temporal Gates

The normal precedence of the five gates in Pandora is as follows: OR, AND, POR, PAND, SAND, meaning OR has the lowest precedence and SAND the highest; thus X+Y&Z is equivalent to X+(Y&Z), not (X+Y)&Z. The temporal operators all have higher precedence than the non-temporal ones, and the temporal conjunction gates have a higher precedence than the POR.

Note also that all gates are left associative, e.g. X<Y<Z is evaluated as (X<Y)<Z not X<(Y<Z). This includes the POR gate; however, the semantics of the POR gate are a little different and any subsequent POR operators are equivalent to OR gates, i.e.:

`A|B|C| ... |N ⇔ A|(((B+C)...)+N)`

Thus `X|Y|Z`, which is evaluated as `(X|Y)|Z`, is equivalent to `X|(Y+Z)`. The left associativity of the five operators is important for ensuring that all temporal fault trees can be converted to binary fault trees in which each gate has at most two inputs. This can be seen in Figure 25:

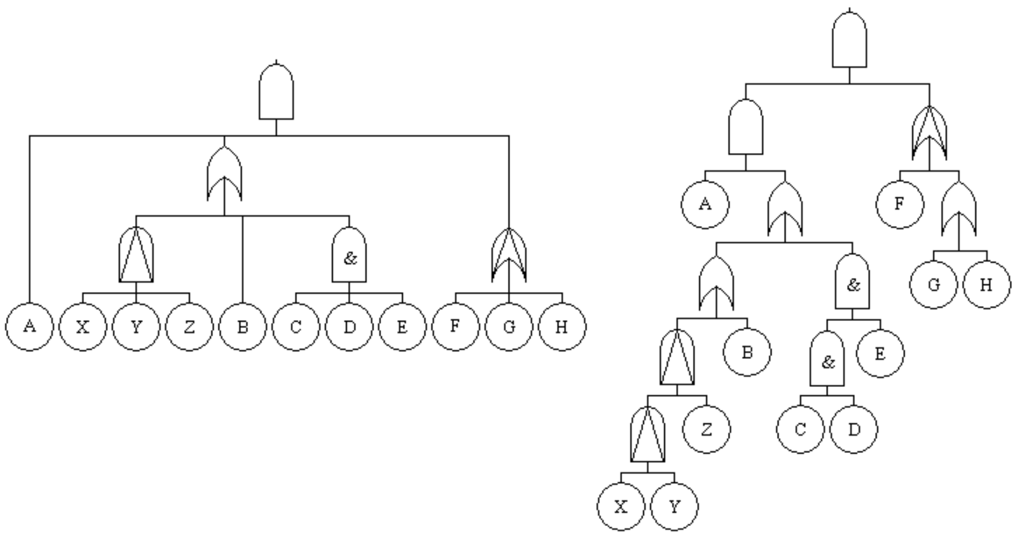


Figure 25 – A normal fault tree and its binary equivalent

The semantics of the gates are best understood when limited to only two inputs, and the behaviour of the three temporal gates in Pandora can perhaps best be expressed visually. Figure 26 shows a series of timelines illustrating how and when the temporal gates become true. Note that for the POR gates, there are two possible ways the gate can be true.

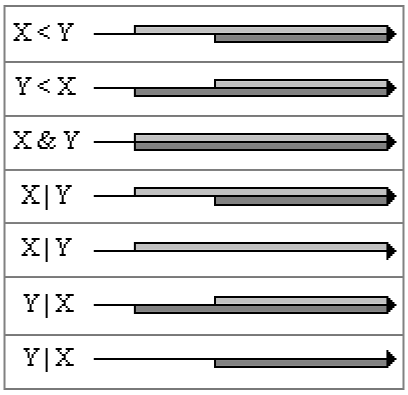


Figure 26 – Behaviour of the three temporal gates

Time is represented in the figure as going from left to right, with X above the timeline and Y below it. It can also be seen from the diagram that if X<Y is true, then so is X|Y (and similarly for Y<X and Y|X). In other words, X PAND Y → X POR Y. However, the POR gates are also true if their subsequent input events do not occur (as seen in the last and the antepenultimate cases).

Because Pandora is a gate-based temporal fault tree system, it is the gates that impose an ordering on events; basic events themselves are 'atemporal' in the sense that without a gate, they have no intrinsic temporal properties. It is only when used as an input to a gate that they take on a temporal meaning and when this happens, they become **temporally significant**. Temporal significance is conferred by the temporal gates and implies that those events have to occur in a certain sequence; this imposes certain constraints on those events (because the order in which they occur – indicated by the gate – must be preserved).

Note that all gates are monotonic – once true they can never become false again. This is important for several reasons. Firstly, it keeps the gates consistent with events (i.e. they are both persistent). Secondly, it avoids the troublesome issue of recurring events (events that become true, then false, then true again). Finally, it means that the structure function of the fault tree also remains monotonic and therefore ensures that the temporal fault trees in Pandora are coherent fault trees.

This information can also be seen in a Boolean truth table:

X	**Y**	**X+Y**	**X.Y**	**X<Y**	**X&Y**	**X|Y**
never occurs	never occurs	false	false	false	false	false
occurs	never occurs	**true**	false	false	false	**true**
never occurs	occurs	**true**	false	false	false	false
occurs first	occurs second	**true**	**true**	**true**	false	**true**
occurs second	occurs first	**true**	**true**	false	false	false
occurs sim-ultaneously	occurs sim–ultaneously	**true**	**true**	false	**true**	false

The two logical gates, OR and AND, are true regardless of the order of events, assuming their conditions are met; the three temporal gates are only true in specific temporal circumstances. It can be seen from the table, however, that the AND is true when the temporal conjunction gates are true, that the OR is true when the temporal disjunction gate is true, and that the POR is true when the PAND is true.

However, this truth table method of describing the behaviour of the gates has two significant disadvantages: firstly, it requires an explanation of when an event occurs in relation to other events; and secondly it does not indicate what happens when a gate is an input to another gate because it does not indicate when the gates become true relative to their input events.

To solve this, a new type of truth table is needed.

3.2.5 Temporal Truth Tables, Sequence Values, and Precedence Trees

Temporal truth tables are extensions of normal Boolean truth tables in the same way that Pandora is a temporal extension to normal Boolean fault trees. Just as Pandora makes it possible to specify the sequence of events as well as the occurrence of events, temporal truth tables (or **TTT**s) make it possible to specify the order in which events occur as well as whether or not they occur.

This is done using **sequence values**. In Pandora, the exact time at which an event occurs is not important – the only thing that matters is when it occurs relative to the other events, i.e. which comes first, which comes second, which comes last etc. Nor is it necessary to say exactly how long the interval between these events lasts (and thus a quantitative metric of time is not required). Sequence values are used instead as a way of representing this ordering information: they are abstractions of the time at which an event occurs. Furthermore, because Pandora models only instantaneous occurrences, sequence values can also be used for gates too.

To indicate a sequence value, the notation *S*(X) is used, where X is an event (or a gate). The sequence value is not the same as the truth value of an event, although they are closely linked. If an event or gate is false, it means it has not yet occurred. Because it has not yet occurred, it is given a sequence value of 0 (i.e. no sequence). If an event is true, it has occurred, and therefore has a sequence value greater than 0 to indicate when it occurred relative to the other events under consideration, e.g. 1 means it occurred first, 2 means second and so on. Two events can have the same sequence value, in which case it means they both occurred simultaneously. In this, sequence values can be likened to a race: the first person to cross the finishing line gets 1st place, the second gets 2nd place, and so on; if two or more people cross the finishing line, they get a joint position, e.g. joint 3rd place. Someone who did not finish the race would not get a place, i.e. 0.

This notation also allows us to distinguish easily between events that have occurred and are true, which will have non-zero sequence values, and events that have not occurred and are false, which have sequence values of zero. This is very similar to the customary notation for Boolean logic of using 0 for false and 1 for true, except now *any* non-zero positive integer represents true[12]. Because it is possible in this way to represent both the temporal and logical values of events with just the sequence values, the *S*( ) notation is often omitted in temporal truth tables (although it is possible to create truth tables in which the temporal and logical values are separate, in which case *L*( ) gives the logical value while *S*( ) gives the temporal value).

Below is the TTT equivalent to the standard Boolean one given earlier, showing the behaviour of the five gates:

X	**Y**	**X+Y**	**X.Y**	**X<Y**	**X&Y**	**X|Y**
0	0	0	0	0	0	0
0	1	1	0	0	0	0
1	0	1	0	0	0	1
1	1	1	1	0	1	0
1	2	1	2	2	0	1
2	1	1	2	0	0	0

A temporal truth table is constructed by enumerating all the possible sequences of all events in an expression, typically ordered so that rows with higher sequence values in them go further down the table. Then the expressions can be evaluated for each row by substituting those sequence values into the expression (this is explained in more detail shortly).

Notice that when one of the two events is false, the other only needs the sequence value 1 (e.g. there is no row in which X is 0 and Y is 2), and similarly there is no entry in the table where X and Y both have the sequence value 2. This is because such values are unnecessary. Going back to the race analogy, assume the winner of the race holds 1st position and the next two racers came joint 2nd. If the winner was then disqualified for cheating, then all the subsequent racers are promoted, so the two who came joint 2nd now come joint 1st. In the same way, sequence values represented in a row of a truth table must be contiguous – there must be no gaps; it is not possible to have *S*(X)=1, *S*(Y)=3 and *S*(Z)=3, for instance (they would instead be 1, 2 and 2 respectively).

This pattern of numbers is closely related to a set of numbers known as the 'Fubini numbers' or the 'ordered Bell numbers'. These represent the "number of asymmetric generalised weak orders on *n* points" or the number of ways *n* competitors can rank in a competition, allowing for ties (Online Encyclopaedia of Integer Sequences - A000670, accessed Jan 2009). In fact, the total number of rows required in a TTT for *n* events is actually twice the equivalent Fubini number for *n*, because the truth tables must also allow for the possibility of events being false (or, alternatively, the possibility that some or all of the competitors fail to reach the finishing line). The Fubini numbers grow rapidly with increasing *n*: for *n* = 3, there are 13 possible combinations (and thus 26 rows in a TTT for 3 events), but for *n* = 18, there are 3,385,534,663,256,845,323 possible combinations, and for *n* = 100, the Fubini number has 174

[12] It is not a coincidence that this is also the custom in many programming languages, e.g. in C/C++, any non-zero value is true while 0 is false.

digits. Nor is there an easy way of calculating the Fubini number for a given *n*, although there is an approximation which is exact for values of *n* less than 17: the nearest integer to:

$$\frac{n!}{2\log(2)^{n+1}}$$

Or in other words, n!/(2*log(2)^(n+1)) rounded to the nearest whole number (Online Encyclopaedia of Integer Sequences - A034172, accessed Jan 2009). Obviously this figure must be doubled to estimate the size of the equivalent TTT.

In any case, the size of these numbers means that manually creating TTT for large numbers of events is prohibitively expensive; the table for four events is about as large as is manageable, consisting of 150 rows. By comparison, the size of a normal Boolean truth table for four events is 2^4, i.e. 16 rows. However, a computer can calculate TTTs for higher numbers of events (though it still becomes very expensive for high values of *n*; indeed, beyond $n = 11$, the numbers involved exceed a 32 bit integer).

Another benefit of the use of sequence values is that they allow a more precise definition of how the temporal gates function. Because sequence values are just integer numbers, a gate becomes a function that operates on its inputs and provides a numerical output. The behaviour of the five gates in Pandora are shown below in this manner. Sequence value inputs to a gate are represented x_1 , x_2 ... x_n, where *n* is the number of inputs.

OR Gate

$\forall x_i : x_i = 0$ then $x_1 + x_2 + ... + x_n = 0$

$\exists x_i : x_i \geq 1$ then $x_1 + x_2 + ... + x_n = min(\text{all } x_i \geq 0)$

where $1 \leq i \leq n$

In other words, if all inputs to an OR gate have the value 0, so will the OR gate; otherwise, the OR gate has the minimum value of all true input events (i.e. inputs with non-zero sequence values). This can be seen in the TTT above, in which the OR gate always takes on the minimum non-zero value of X or Y if at least one of them is true.

AND Gate

$\forall x_i : x_i \geq 1$ then $x_1 . x_2 . ... . x_n = max(x_i)$

$\exists x_i : x_i = 0$ then $x_1 . x_2 . ... . x_n = 0$

where $1 \leq i \leq n$

This is the opposite behaviour to the OR gate. If any of its inputs have the value 0, then so will the AND gate; if all of them have non-zero values, then the AND gate has the maximum value. Again, this can be seen in the above TTT, where the AND always takes the maximum value of X and Y when both are true, and 0 otherwise.

PAND Gate

$\forall x_i : x_i < x_{i+1} \wedge x_i > 0$ then $x_1 < x_2 < ... < x_n = x_n$

$\exists x_i : x_i \geq x_{i+1} \vee x_i = 0$ then $x_1 < x_2 < ... < x_n = 0$

where $1 \leq i \leq n$-1

For the PAND gate, each input must be non-zero and less than the input to its right, in which case the PAND gate has the value of its right-most input. Otherwise, it has the value 0. In practice, this means its inputs must all be in ascending order and the left-most must be at least 1, e.g. for n = 3, the values must be 1, 2, and 3. The sequence values need not be contiguous, however, e.g. A < B < D will have the value 4 if A=1, B=2, D=4, assuming there is also an event with the value 3 which is not in the expression.

SAND Gate

$\forall x_i : x_i = x_{i+1} \wedge x_i > 0$ then $x_1 \,\&\, x_2 \,\&\, ... \,\&\, x_n = x_n$

$\exists x_i : x_i \neq x_{i+1} \vee x_i = 0$ then $x_1 \,\&\, x_2 \,\&\, ... \,\&\, x_n = 0$

where $1 \leq i \leq n$-1

For the SAND, all of its inputs must have the same non-zero value, in which case the SAND also has that value. Otherwise, the SAND has the value 0.

POR Gate

$x_1 > 0, \forall x_i : x_1 < x_i \vee x_i = 0$ then $x_1 \mid x_2 \mid ... \mid x_n = x_1$

$x_1 > 0, \exists x_i : x_1 \geq x_i$ then $x_1 \mid x_2 \mid ... \mid x_n = 0$

$x_1 = 0$ then $x_1 \mid x_2 \mid ... \mid x_n = 0$

where $2 \leq i \leq n$

The POR is similar to the PAND, except rather than each input being less than the one to its right, the first input must be greater than 0 and also be less than all of the other inputs that are greater than 0. If true, the POR always has the same value as its first input, otherwise it has the value 0.

These definitions allow us to mathematically obtain the value of a gate for the TTTs, given the appropriate inputs. They also allow one gate to be an input into another, e.g. (X&Y)<Z, where

the value of X&Y is first calculated and used as the input into the PAND gate. For expressions which only include AND-based gates (like X&Y<Z), it is possible to omit those rows containing false events, thereby cutting the size of the table in half:

X	**Y**	**Z**	**X&Y**	**(X&Y)<Z**
1	1	1	1	0
1	1	2	1	2
1	2	1	0	0
1	2	2	0	0
1	2	3	0	0
1	3	2	0	0
2	1	1	0	0
2	1	2	0	0
2	1	3	0	0
2	2	1	2	0
2	3	1	0	0
3	1	2	0	0
3	2	1	0	0

This is known as a **Conjunctive Temporal Truth Table** (or CTTT) and is only possible when the expressions only contain AND-based gates, because if any event is false, then all the expressions will also be false. OR-based gates, however, may still be true, and so require the full table.

Another way of representing the information contained within TTTs is to use a **precedence tree**, first introduced in Walker, Bottaci, & Papadopoulos (2007). A precedence tree is a kind of branching timeline that shows all possible sequences for a set of events. Each node represents a row from the equivalent TTT and each leaf node represents a row in the equivalent CTTT. The number of leaf nodes is a Fubini number. Consider the precedence tree equivalent to the CTTT shown above:

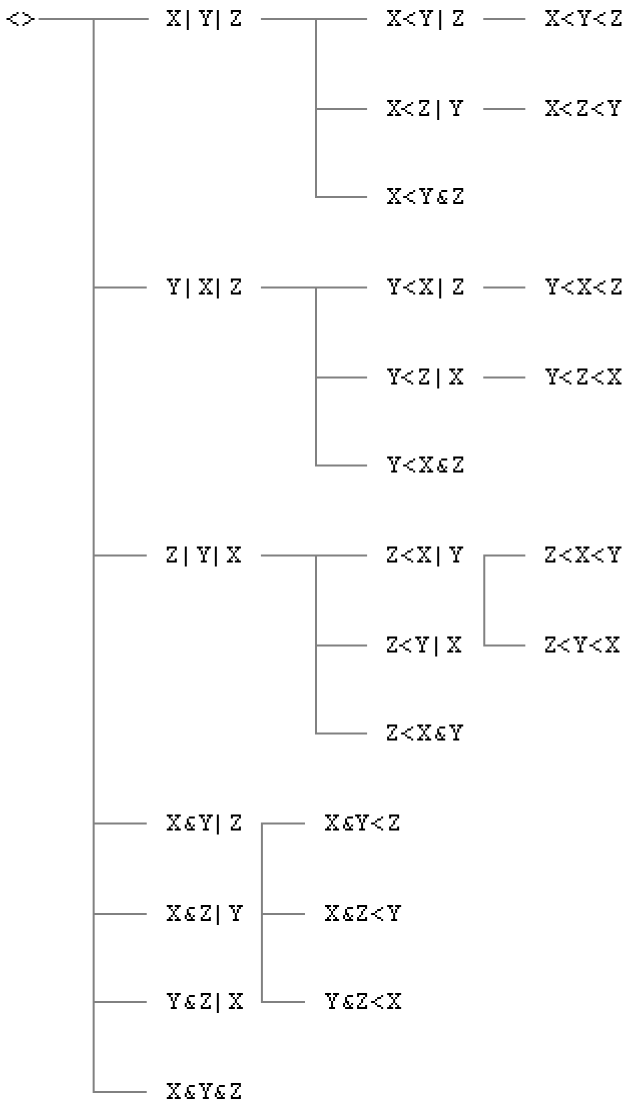


Figure 27 - Precedence Tree for X, Y, and Z

This precedence tree shows all possible sequences for three events, X, Y, and Z. It has 13 leaf nodes (the same number of rows in the CTTT above) and 26 nodes in total (the same number as the equivalent TTT would have). <> is the root and indicates that no events have occurred yet; this is the 'start' of the branching timeline. After this the tree branches to represent the seven possibilities for what may happen next: X occurs, Y occurs, Z occurs, two events occur simultaneously, or all three events occur simultaneously. The tree continues branching until all events have occurred and each time the tree branches, a new sequence value is used; thus for each of the first possibilities, the sequence value 1 is used, and for each of the second tier possibilities, the sequence value 2 is used etc. In the case of X|Y|Z, X takes the value 1 and Y and Z do not yet have known sequence values. In the next branch, X<Y|Z, X still has the value 1 while Y assumes the value 2. Finally, in the leaf node, X<Y<Z, Z assumes the value 3 and all three events now have sequence values (because all three have now occurred).

However, although precedence trees like this are useful for showing how different sequence values are assigned to each event in different sequences, the TTT is a more compact form to represent all the possible sequences for a set of events. More importantly, the main use of TTTs is to allow us to prove equivalence between expressions containing temporal gates, in the same way Boolean truth tables can be used to prove equivalence between Boolean expressions (something not possible with precedence trees). Just as with Boolean truth tables, if the values for two expressions are the same in every row, then the expressions are equivalent. The values have to be exactly the same to be **temporally equivalent**.

To demonstrate this concept, let us look again at the original three temporal relations for two events X and Y:

- X < Y
- Y < X
- X & Y

It was stated earlier that each of these imply the equivalent AND gate, X.Y, is also true. This is because each of these three statements comprise one independent part of the AND gate: if X.Y is true, then one and only one of these three expressions will also be true. This can be stated as X.Y ⇔ X<Y + X&Y + Y<X, and can be seen in the TTT below:

X	**Y**	**X.Y**	**X<Y**	**X&Y**	**Y<X**	**X<Y + X&Y + Y<X**
0	0	**0**	0	0	0	**0**
0	1	**0**	0	0	0	**0**
1	0	**0**	0	0	0	**0**
1	1	**1**	0	1	0	**1**
1	2	**2**	2	0	0	**2**
2	1	**2**	0	0	2	**2**

The two relevant columns have been highlighted in bold, and as can be seen, the values are identical. This process – of using TTTs to prove that two expressions are equivalent – is the way Pandora is able to create a set of **temporal laws** analogous to Boolean laws that can be used to simplify, reduce, or otherwise manipulate expressions containing temporal gates.

3.3 Temporal Laws: The Rules of Pandora

3.3.1 Boolean Laws

Qualitative analysis of normal fault trees consists of reducing a fault tree to its minimal cut sets. There are many methods of accomplishing this, some of which were mentioned in Chapter 2, but almost all of them rely on a set of Boolean laws to function. The key to this is that a fault tree can also be represented as a logical expression. For example, the fault tree in Figure 28 can also be represented as the Boolean expression (A+B).(C+A). The parentheses indicate the depth of the events and gates contained within them – the more brackets that enclose a sub-expression, the further down the tree it is.

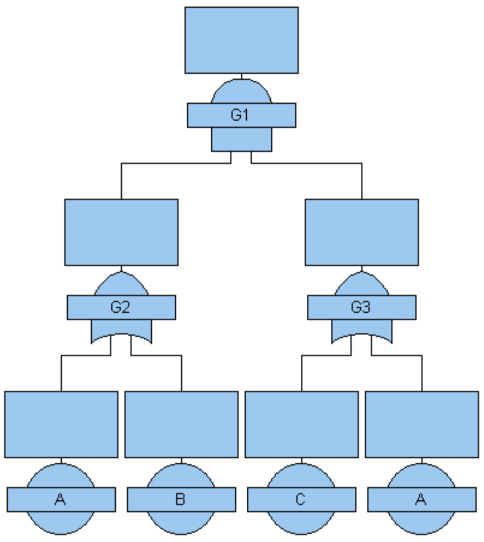


Figure 28 – A simple fault tree

Boolean laws express equivalence between two different expressions that nevertheless have the same logical values at all time. For example, the Commutative Law states that X.Y is equivalent to Y.X – in other words, although these expressions look different, they mean the same thing. Boolean laws can therefore be used to manipulate logical expressions, and more importantly, to *simplify* them. This is the principle behind qualitative analysis: the minimal cut sets are the result of a simplification of the fault tree.

The Boolean laws most relevant to qualitative FTA are explained below, but a full list can be found in **Appendix II: Boolean & Temporal Laws**.

Commutative Laws

$X.Y \Leftrightarrow Y.X$

$X+Y \Leftrightarrow Y+X$

The Commutative Law is one of the most fundamental Boolean laws and states that (in fault tree terms) the order of inputs to AND and OR gates is irrelevant. This makes it possible to reorder inputs of gates, e.g. to order events alphabetically for the sake of clarity, or more importantly to allow an expression to match another law. This would not be possible without the Commutative Law.

Associative Laws

$X.(Y.Z) \Leftrightarrow (X.Y).Z \Leftrightarrow X.Y.Z$

$X+(Y+Z) \Leftrightarrow (X+Y)+Z \Leftrightarrow X+Y+Z$

The Associative Law, like the Commutative Law, is one of the fundamental Boolean laws. In this case it enables us to reorder and remove parentheses. This is useful if there is a gate acting as an input to another gate of the same type, e.g. one OR gate being an input to another OR gate. The Associative Law means all of the second OR gate's children can simply be added to the first and the second gate removed altogether, as shown in Figure 29. This process is often known as **contraction** in fault tree terms.

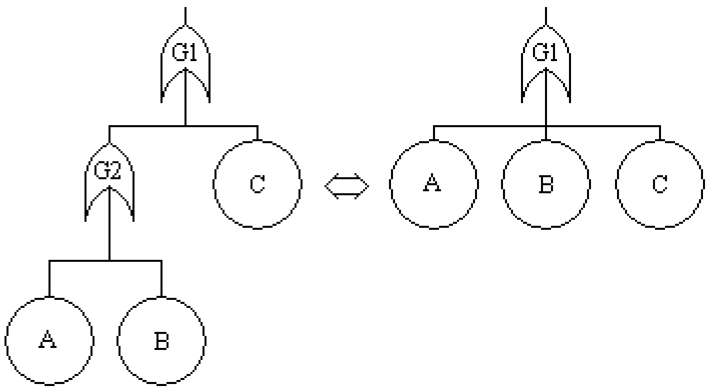


Figure 29 – Fault tree contraction

Distributive Laws

$X.(Y+Z) \Leftrightarrow (Y+Z).X \Leftrightarrow X.Y + X.Z$

$X+(Y.Z) \Leftrightarrow (Y.Z)+X \Leftrightarrow (X+Y).(X+Z)$

$(A.B)+(C.D) \Leftrightarrow (A+C).(A+D).(B+C).(B+D)$

$(A+B).(C+D) \Leftrightarrow (A.C)+(A.D)+(B.C)+(B.D)$

The Distributive Laws are the most important laws for obtaining cut sets in FTA. They make it possible to move gates up or down the fault tree without changing the meaning. Because OR gates distribute over AND gates and AND gates also distribute over OR gates, it is possible to move gates up or down as needed, but in practice the goal of standard qualitative analysis is to end up with one OR gate at the top of the tree and one or more AND gates beneath it, each containing only events and no other gates. This is the cut set format, also known as sum-of-products or disjunctive normal form. By applying the Distributive Laws and contracting, it is possible to convert a fault tree or equivalent logical expression into cut set format. For example:

```
1.    ((X.Y) + (Y.Z)) . (W + (X.Z))
2.    (Y + (X.Z)) . (W + (X.Z))
3.    Y.W + Y.(X.Z) + (X.Z).W + (X.Z).(X.Z)
4.    Y.W + Y.X.Z + X.Z.W + X.Z.X.Z
```

To get from step 1 to step 2, we apply the rule Y + X.Z ⇔ (Y.X) + (Y.Z); if the events are not in the correct order, we can apply the Commutative Law first. To get from step 2 to step 3, we use the rule (A+B) . (C+D) ⇔ (A.C) + (A.D) + (B.C) + (B.D). Then for step 4, we can contract the parentheses. The resulting expression is in cut set format; the ANDs are the cut sets, and the ORs connect them, so there are four cut sets:

```
Y.W
X.Y.Z
W.X.Z
X.Z.X.Z
```

To minimise it, however, two more laws are needed:

Idempotent Laws

```
X.X ⇔ X
X+X ⇔ X
```

The Idempotent Laws are very useful for minimisation as they make it possible to remove redundant events. If an event occurs, it only needs mentioning once. The Idempotent Laws are therefore used to simplify cut sets by removing duplicates. Continuing the previous example:

```
X.Z.X.Z ⇔ X.Z
```

Both X and Z are mentioned twice, and by applying the Idempotent Law, we can simplify it so they only occur once. An event need only appear once in a given cut set (though it may appear again in another cut set).

To minimise cut sets themselves, one last law is required.

Absorption Laws

```
X.(X+Y) ⇔ X
X+(X.Y) ⇔ X
```

This is the most useful type of law in minimisation as it allows us to remove entire cut sets if they are redundant (by using the second form of the law). If all the events in a cut set are also contained within a larger cut set (i.e. one with more events), then the larger cut set is redundant because the smaller one is sufficient to cause the top event. Returning to the example, two of the cut sets are redundant:

```
X.Y.Z + X.Z ⇔ X.Z
W.X.Z + X.Z ⇔ X.Z
```

Both of these contain the last cut set, X.Z, and so are redundant. This leaves us with just two cut sets, which are now minimal:

```
Y.W
X.Z
```

And this process, of first transforming the fault tree into an expression in cut set form (i.e. disjunctive normal form) and then simplifying redundant events using the Idempotent and Absorption laws, is the basis of qualitative FTA.

There is also one other set of Boolean laws that will prove useful shortly, even though they are not regularly used in normal qualitative analysis:

```
0.X ⇔ 0
0+X ⇔ X
1.X ⇔ X
1+X ⇔ 1
```

In these laws, 0 and 1 represent contradictions and tautologies respectively; in other words, 0 is "always false" and 1 is "always true". Unless an event also happens to be a contradiction or tautology, these should never occur in a normal, static fault tree, but they can occur in temporal fault trees.

The various laws described above are sufficient for the analysis of standard Boolean fault trees, but Pandora also has PAND, SAND, and POR gates. Therefore, we also need more logical laws that we can apply to these temporal gates: **temporal laws**.

3.3.2 Temporal laws derived from Boolean laws

Temporal laws can be divided into two categories: those laws that are based on Boolean laws and those laws that are not. In the first category are temporal versions of the Distributive, Commutative, Associative, Absorption and Idempotent laws, i.e. versions of those laws which also feature PANDs, SANDs and PORs as well as AND and OR gates. All of these laws can be proved using TTTs.

Temporal Commutative Laws

```
X<Y ≠ Y<X
X&Y ⇔ Y&X
X|Y ≠ Y|X
```

The Commutative Law, as already explained, allows us to change the order of the inputs to a gate. However, the two priority gates, PAND and POR, *depend* on the order of their inputs for their meaning: X<Y means "X occurred before Y", and changing the order also changes the meaning to "Y occurred before X." As a result, the Commutative Law does not apply to these gates, and it is not possible to reorder their inputs during qualitative analysis. The SAND gate is the exception, however – because it requires all events to occur at the same time, the order of the inputs to the gate does not matter. Therefore, the Commutative Law does apply to the SAND gate as normal.

Temporal Associative Laws

```
X<(Y<Z) ≠ (X<Y)<Z
(X<Y)<Z ⇔ X<Y<Z
X&(Y&Z) ⇔ (X&Y)&Z ⇔ X&Y&Z
X|(Y|Z) ≠ (X|Y)|Z
(X|Y)|Z ⇔ X|Y|Z
```

The Associative Laws ordinarily make it possible to remove and reorder parentheses in an expression. Once again, however, the priority gates present some difficulty while the SAND gate behaves normally. To look at why the gates behave this way, it is necessary to look at their sequence values.

Firstly, the PAND gate. When a PAND gate is evaluated, it takes on the value of its right-most event when true. If the sequence values are X=1, Y=2, and Z=3, then the innermost PAND in X<(Y<Z) means (2 < 3) which results in the PAND having a sequence value of 3. The outermost PAND is then also evaluated as (1 < 3) which results in a sequence value of 3 as well. This is what we would expect; however, there is in fact *no temporal relation* specified between the values of X and Y by this expression. Assume instead that X=2, Y=1, and Z=3. The innermost PAND now means (1 < 3) which still results in a value of 3, and the outermost now means (2 < 3), which again still results in a value of 3, but the sequence in which X and Y occur is reversed. The sequence value of the outermost PAND is therefore not affected by the relative order of X and Y, only by the orders of Y and Z and then X and Z.

By contrast, (X<Y)<Z means the same as X<Y<Z, because when the innermost PAND is evaluated, it gets the value 2 in the first case (where X=1 and Y=2) and 0 in the second case (where Y=1 and X=2). This can then be compared against Z in the usual way, and the first case would be true but the second case would be false. This can be seen in the TTT below:

X	Y	Z	(X.Y)<Z	X<Y	Y<Z	X<Y<Z	X<(Y<Z)	(X<Y)<Z
1	1	1	0	0	0	0	0	0
1	1	2	**2**	0	2	0	**2**	0
1	2	1	0	2	0	0	0	0
2	1	1	0	0	0	0	0	0
1	2	2	0	2	0	0	0	0
2	1	2	0	0	2	0	0	0
2	2	1	0	0	0	0	0	0
1	2	3	**3**	2	3	**3**	**3**	**3**
1	3	2	0	3	0	0	0	0
2	1	3	**3**	0	3	0	**3**	0
2	3	1	0	3	0	0	0	0
3	1	2	0	0	2	0	0	0
3	2	1	0	0	0	0	0	0

As can be seen, while the columns for X<Y<Z and (X<Y)<Z match, the column for X<(Y<Z) does not. In fact, X<(Y<Z) is equivalent to (X.Y)<Z instead, as shown in the table; this is also the same as saying (X<Z).(Y<Z). In other words:

$$X<(Y<Z) \Leftrightarrow (X.Y)<Z \Leftrightarrow (X<Z).(Y<Z)$$

This is not the case for the SAND gate, which behaves normally (due to its input sequence values all being the same). However, like the PAND, the POR also differs in its behaviour. X|(Y|Z) means we evaluate (Y|Z) first and then X|(Y|Z). Because of the way the POR works, the outermost POR can be true *even if the innermost is false*. In other words, Z can occur before Y, or even before X, and the outermost POR is still true as long as X occurs. In these cases, there is no direct temporal relation between X and Y and Z. Again, this can be shown by a TTT:

X	**Y**	**Z**	**X|(Y+Z)**	**X|Y**	**Y|Z**	**X|Y|Z**	**X|(Y|Z)**	**(X|Y)|Z**
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0
0	1	1	0	0	0	0	0	0
0	1	2	0	0	1	0	0	0
0	2	1	0	0	0	0	0	0
1	0	0	**1**	1	0	**1**	**1**	**1**
1	0	1	0	1	0	0	**1**	0
1	0	2	**1**	1	0	**1**	**1**	**1**
1	1	0	0	0	1	0	0	0
1	1	1	0	0	0	0	**1**	0
1	1	2	0	0	1	0	0	0
1	2	0	**1**	1	2	**1**	**1**	**1**
1	2	1	0	1	0	0	1	0
1	2	2	**1**	1	0	**1**	**1**	**1**
1	2	3	**1**	1	2	**1**	**1**	**1**
1	3	2	**1**	1	0	**1**	**1**	**1**
2	0	1	0	0	0	0	**2**	0
2	1	0	0	0	1	0	0	0
2	1	1	0	0	0	0	**2**	0
2	1	2	0	0	1	0	0	0
2	1	3	0	0	1	0	0	0
2	2	1	0	0	0	0	0	0
2	3	1	0	2	0	0	0	0
3	1	2	0	0	1	0	0	0
3	2	1	0	0	0	0	0	0

As you can see, X|(Y|Z) differs from the other two. There is also another equivalence shown in the table: X|(Y+Z) is the same as X|Y|Z. It is also the same as saying X|Y.X|Z, i.e.:

$$(X|Y)|Z \Leftrightarrow X|Y|Z \Leftrightarrow X|(Y+Z) \Leftrightarrow X|Y.X|Z$$

This is more of a distributive relationship, and will be explained next.

Temporal Distributive Laws

```
X < (Y.Z) ⇔ Y.(X<Z) + Z.(X<Y)
X < (Y+Z) ⇔ (X|Z).(X|Y).(Y+Z)
X < (Y<Z) ⇔ (X<Z).(Y<Z)
X < (Y&Z) ⇔ (X<Y).(X<Z).(Y&Z)
X < (Y|Z) ⇔ (X<Y).(Y|Z)
X < (Y|Z) ⇔ (X|Y).(Y|Z)
X & (Y+Z) ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y)
X & (Y.Z) ⇔ (X&Y).(Y&Z) + (X&Y).(Z<Y) + (X&Z).(Y<Z)
X & (Y<Z) ⇔ (Y<Z)&(Y<X)
X & (Y<Z) ⇔ (X&Z).(Y<Z).(Y<X)
X & (Y&Z) ⇔ X&Y&Z
X & (Y|Z) ⇔ (X&Y).(Y|Z).(X|Z)
X & (Y|Z) ⇔ (X|Z)&(Y|Z)
X | (Y+Z) ⇔ (X|Y).(X|Z)
X | (Y.Z) ⇔ X|Y + X|Z
X | (Y<Z) ⇔ (X|Z) + (X|Y) + X.(Z<Y) + X.(Y&Z)
X | (Y&Z) ⇔ X.(Y|Z) + X.(Z|Y) + (X|Y) + (X|Z)
X | (Y|Z) ⇔ (X|Y) + (X.Z<Y) + (X.Y&Z)

(Y+Z) < X ⇔ (Y<X) + (Z<X)
(Y.Z) < X ⇔ (Y<X).(Z<X)
(Y<Z) < X ⇔ (Y<Z).(Z<X)
(Y&Z) < X ⇔ (Z<X).(Y<X).(Y&Z)
(Y|Z) < X ⇔ (Y<X).(Y|Z)
(Y+Z) & X ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y)
(Y.Z) & X ⇔ (Y<X).(Z&X) + (Z<X).(Y&X) + X&Y&Z
(Y<Z) & X ⇔ (Y<Z).(Z&X).(Y<X)
(Y&Z) & X ⇔ Y&Z&X
(Y|Z) & X ⇔ (X&Y).(Y|Z)
(Y+Z) | X ⇔ (Y|X) + (Z|X)
(Y.Z) | X ⇔ (Y|X).(Z|X)
(Y<Z) | X ⇔ (Y|Z).(Z|X)
(Y&Z) | X ⇔ (Y|X)&(Z|X)
(Y&Z) | X ⇔ (Y|X).(Z|X).(Y&Z)
```

```
(Y|Z) | X ⇔ (Y|Z).(Y|X)

(A+B) & (C+D) ⇔ A&B&C&D + A&C|B|D + A&D|B|C + B&C|A|D
                + B&D|A|C + A&B&C|D + A&B&D|C + A&C&D|B
                + B&C&D|A
(A.B) & (C.D) ⇔ A&B&C&D + A<B&C&D + B<A&C&D + C<A&B&D
                + D<A&B&C + (A.C)<B&D + (A.D)<B&C +
                + (B.C)<A&D + (B.D)<A&C
(A<B) & (C<D) ⇔ (A<B).(C<D).(B&D)
(A|B) & (C|D) ⇔ (A&C).(A|B).(C|D)
(A+B) < (C+D) ⇔ (A|C).(A|D).(C+D) + (B|C).(B|D).(C+D)
(A.B) < (C.D) ⇔ (A<D).(B<D).(C<D) + (A<C).(B<C).(D<C)
                + (A<C).(A<D).(B<C).(B<D).(C&D)
(A&B) < (C&D) ⇔ (A<C).(B<C).(A<D).(B<D).(A&B).(C&D)
(A|B) < (C|D) ⇔ (A<C).(A|B).(C|D)
(A+B) | (C+D) ⇔ (A|C).(A|D) + (B|C).(B|D)
(A.B) | (C.D) ⇔ (A|C).(B|C) + (A|D).(B|D)
(A<B) | (C<D) ⇔ (A<B).(B|C) + (A<B).(B|D) + (A<B).(D<C)
                + (A<B).(C&D)
(A&B) | (C&D) ⇔ (A&B).(C|D) + (A&B).(D|C) + (A&B).(B|C)
                + (A&B).(B|D)
```

As you can see, there are a lot of temporal Distributive Laws, and very few of them behave exactly as the normal Boolean Distributive versions do. A full explanation of the derivation of these laws could take half a thesis in itself to cover, but for now it suffices to say that all these laws can be proved using TTTs and they enable us to transform parts of the fault tree into a more useful form (a process which will be explained in more detail in the next chapter). The important laws are the ones with three events, which can be used to construct more complicated forms (like the four event versions listed last).

Temporal Idempotent Laws

```
X<X ≠ X
X&X ⇔ X
X|X ≠ X
```

The Idempotent Law only holds for the SAND gate, not for the priority gates. This is only to be expected; both priority gates require their left-most input to occur earlier than the inputs to the right, but if those inputs are the same, then they occur at the same time. On the other hand, this is exactly what the SAND gate expects. For example, if X = 1, then X<X = 1<1 = 0, X&X = 1&1 = 1, and X|X = 1|1 = 0. In the case of the POR, if the left input happens, then so does the right (because they are the same), and so in this case it behaves exactly like a PAND gate. As a result, X<X and X|X are both contradictions, and will be covered in more detail shortly.

Temporal Absorption Laws

```
X . (X < Y) ⇔ X < Y                Y . (X < Y) ⇔ X < Y
X . (X & Y) ⇔ X & Y                Y . (X & Y) ⇔ X & Y
X . (X | Y) ⇔ X | Y                Y . (X | Y) ⇔ X < Y
X < (X . Y) ⇔ X < Y                Y < (X . Y) ⇔ Y < X
X & (X . Y) ⇔ X&Y + Y<X            Y & (X . Y) ⇔ X&Y + X<Y
X | (X . Y) ⇔ X | Y                Y | (X . Y) ⇔ Y | X
X < (X + Y) ⇔ 0                    Y < (X + Y) ⇔ 0
X & (X + Y) ⇔ X&Y + X|Y            Y & (X + Y) ⇔ X&Y + Y|X
X | (X + Y) ⇔ 0                    Y | (X + Y) ⇔ 0
(X . Y) < X ⇔ 0                    (X . Y) < Y ⇔ 0
(X . Y) & X ⇔ X&Y + Y<X            (X . Y) & Y ⇔ X&Y + X<Y
(X . Y) | X ⇔ 0                    (X . Y) | Y ⇔ 0
(X + Y) < X ⇔ Y < X                (X + Y) < Y ⇔ X < Y
(X + Y) & X ⇔ X&Y + X|Y            (X + Y) & Y ⇔ X&Y + Y|X
(X + Y) | X ⇔ Y | X                (X + Y) | Y ⇔ X | Y
X + (X < Y) ⇔ X                    Y + (X < Y) ⇔ Y
X + (X & Y) ⇔ X                    Y + (X & Y) ⇔ Y
X + (X | Y) ⇔ X                    Y + (X | Y) ⇔ X + Y
```

The temporal Absorption Laws play a large role in the reduction and simplification of temporal fault trees in Pandora. While the Distributive, Associative, and Commutative laws rearrange and transform fault tree expressions, these laws can be used to shrink them. The last six in this list are particularly important because they allow for the elimination of redundant cut sets. However, it is important to note that some of these Absorption Laws result in contradictions; this marks the single biggest difference in the qualitative analysis of temporal versus normal Boolean fault trees: in normal fault trees, the only time a contradiction might be encountered is when NOT gates are being used, but in temporal fault trees, it is a lot more common (and not necessarily indicative of any error in the modelling of the system). To deal with these contradictions, a new set of laws specific to Pandora is needed.

3.3.3 New Temporal Laws

Whereas the laws given so far have been derived from Boolean laws, the laws in this section are not. Many of them are instead drawn from the definitions of Pandora's events and gates, or provided to be able to deal with the possibility of contradictions in the temporal fault tree. Like the other laws, these can be proved using TTTs, and many of them are used in qualitative analysis to reduce and simplify the temporal fault tree. The first set of these new laws stems directly from the definitions used in Pandora.

The Completion Laws

1st Completion Law:	X.Y ⇔ X<Y + X&Y + Y<X
2nd Completion Law:	X+Y ⇔ X|Y + X&Y + Y|X
3rd Completion Law:	X ⇔ Y<X + X&Y + X|Y

There are three versions of the Completion Law. The first one is the **Conjunctive Completion Law** (or CCL) and relates the PAND and SAND gates to the Boolean AND gate. This law derives from Pandora's definitions of time and events, i.e. that there are only three possible temporal relations between two events and exactly one of them must be true if both events have occurred. The law works in both directions, so that if X.Y is true, then one of X<Y, X&Y or Y<X must be true, and if any of X<Y, X&Y, or Y<X are true, then so is X.Y. This law can also be used for reduction, e.g. if we have three cut sets X<Y, X&Y, and Y<X, we can replace all three with a single cut set X.Y.

The second is the **Disjunctive Completion Law** (or DCL), which relates POR and SAND gates to the Boolean OR gate. In the same way that the AND comprises three independent parts, the OR also comprises three independent parts. This can be best illustrated with an illustration:

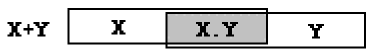


Figure 30 – AND is the overlap of X and Y

First, the relationship between AND and OR. Figure 30 shows that X.Y is true when both X and Y are true. X+Y is the area contained within the boxes and is true when both X and Y are true (i.e. when X.Y is true), when X alone is true, or when Y alone is true. The shaded area represents X.Y and can be further divided up according to the Conjunctive Completion Law:

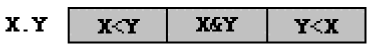


Figure 31 – Conjunctive Completion Law

Each of the three areas inside X.Y do not overlap; they are distinct, and only one of them can be true at any time. But from Figure 30, we know that X+Y consists of X alone, Y alone, and the X.Y, which in itself consists of these three parts. Therefore, X+Y consists of five independent parts, as seen in Figure 32:

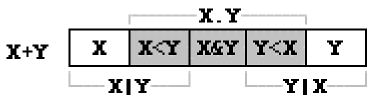


Figure 32 – Disjunctive Completion Law

The shaded area again represents X.Y, but it is possible to see the five distinct areas: X alone, X occurs before Y, X occurs at the same time as Y, Y occurs before X, and Y alone. According to the definition of the POR gate, X POR Y is true if X alone occurs or if X occurs before Y, so this is the same as saying X+Y consists of X|Y, Y|X, and X&Y, which is also illustrated in the figure.

These two laws are useful for two reasons: not only can they be used for reduction, they also allow an expression containing only logical gates to be expanded into one containing temporal gates.

The third and final version of the Completion Law is the **Reductive Completion Law**[13]. It differs from the other two because it does not relate temporal gates to a Boolean gate; instead, it demonstrates how an event can be entirely redundant. As has already been repeatedly established, Pandora's definitions state that there are three possible temporal relations: before, after, and simultaneous. These combine to make an AND, as seen in the Conjunctive Completion Law. If we also include the possibility that the second of the events may or may not occur, then there are four possibilities: X before Y, X after Y, X at the same time as Y, and X alone. Given a disjunction of these four options, Y is entirely redundant; it does not matter whether it occurs before X, after X, at the same time as X, or not at all, because the expression depends solely on the value of X – whenever X is false, all four options are also false, and whenever X is true, at least one of the four will also be true. Hence, Y<X + X&Y + X|Y ⇔ X. The second event Y is irrelevant, because its presence does not affect the result of the expression.

[13] Much credit should go to Guillaume Merle for helping to discover this third and most elusive Completion Law.

The Reductive Completion Law (RCL) can also be presented in another form, i.e.:

$$X \Leftrightarrow X.Y + X|Y$$

In this case, the other parts of the RCL are contained within the AND gate: if we expand the AND using the CCL, this becomes clear:

$$X \Leftrightarrow (X<Y + X\&Y + Y<X) + X|Y$$

The first PAND is redundant according to the Priority Laws (discussed shortly) as X|Y 'overrides' the PAND (i.e. whenever X<Y is true, X|Y is also true).

This second form of the RCL is particularly useful when multiple events are involved. For example, if there are three events X, Y, and Z, then the RCL is as follows:

$$\begin{aligned} X \Leftrightarrow\ & Y<Z<X + Z\&Y<X + Z<Y<X + X\&Y\&Z + Z<Y\&X + Y<Z\&X + \\ & X\&Z|Y + X\&Y|Z + X|Y|Z + Z<X|Y + Y<X|Z \end{aligned}$$

This is obviously quite complex, but it is much simpler in its second form:

$$X \Leftrightarrow X.Y + X.Z + X|Y + X|Z$$

This form of the RCL will prove valuable in the next chapter.

Finally, the TTT below shows how all of the Completion Laws can be proved.

X	**Y**	**X+Y**	**X.Y**	**X|Y**	**X<Y**	**X&Y**	**Y<X**	**Y|X**	**X<Y+ X&Y +Y<X**	**X|Y+ X&Y +Y|X**	**Y<X+ X&Y +X|Y**
0	0	0	0	0	0	0	0	0	0	0	0
0	1	**1**	0	0	0	0	0	1	0	**1**	0
1	0	**1**	0	1	0	0	0	0	0	**1**	**1**
1	1	**1**	**1**	0	0	1	0	0	**1**	**1**	**1**
1	2	**1**	**2**	1	2	0	0	0	**2**	**1**	**1**
2	1	**1**	**2**	0	0	0	2	1	**2**	**1**	**2**

The Completion Laws show how the temporal relations in Pandora relate to each other and to the original Boolean gates. However, one of the definitions in Pandora is that only one of the three temporal relations, as embodied by the Conjunctive Completion Law, can be true at any point. This central tenet of Pandora is embodied in its own set of laws:

Laws of Mutual Exclusion

$X<Y \,.\, Y<X \Leftrightarrow 0$

$X<Y \,.\, X\&Y \Leftrightarrow 0$

$Y<X \,.\, X\&Y \Leftrightarrow 0$

$X|Y \,.\, Y<X \Leftrightarrow 0$

$Y|X \,.\, X<Y \Leftrightarrow 0$

$X|Y \,.\, X\&Y \Leftrightarrow 0$

$Y|X \,.\, X\&Y \Leftrightarrow 0$

The Laws of Mutual Exclusion prevent two different temporal relations from being true at the same time. Taken together with the Completion Laws, they enforce the relative ordering of events in Pandora fault trees. If any of the Mutual Exclusion laws are violated, they result in a contradiction, which must be dealt with appropriately.

There is also another way to cause contradictions:

Laws of Simultaneity

$X<X \Leftrightarrow 0$

$X|X \Leftrightarrow 0$

$X\&X \Leftrightarrow X$

As mentioned earlier, the Idempotent laws do not apply to the two priority gates. Instead, they are subject to the Laws of Simultaneity – if the same event is used as an input to the same priority gate more than once, it causes a contradiction. This applies even if the events are separated, e.g. X<Y<Z<X will still result in a contradiction. The SAND gate, however, behaves according to the Idempotent Law.

Using the Mutual Exclusion and Simultaneity Laws, it is possible to detect most contradictions straight away. Unfortunately, there is one type of contradiction that is harder to detect: **cyclic contradictions**. These are caused by a series of temporal gates all connected by ANDs which do not immediately violate Simultaneity or Mutual Exclusion, but when followed to their logical conclusions, result in contradictions. For example: (X<Y) . (Y<Z) . (Z<X). None of these immediately violate Mutual Exclusion, because there is no pair of PAND gates in which the same events occur in different orders, but they are clearly impossible: X has to occur before Y, which occurs before Z, which occurs before X, which occurs before Y... and so forth. Similarly, (X<Y) . (Y<Z) . (Z&X) is equally impossible, but still does not violate Mutual Exclusion.

The solution is to first apply the Law of Extension:

Law of Extension

$$X<Y \;.\; Y<Z \Leftrightarrow X<Y \;.\; Y<Z \;.\; X<Z \Leftrightarrow X<Y<Z$$

$$X\&Y \;.\; Y\&Z \Leftrightarrow X\&Y \;.\; Y\&Z \;.\; X\&Z \Leftrightarrow X\&Y\&Z$$

$$X|Y \;.\; Y|Z \Leftrightarrow X|Y \;.\; Y|Z \;.\; X|Z$$

The Law of Extension is used to explicitly reveal any temporal relationships which are implied by the temporal gates. If X occurs before Y and Y occurs before Z, then *by extension*, X must also occur before Z. In this case, it is equivalent to saying X<Y<Z. The law also applies to the other two gates, though in the case of the POR gate, it does not result in an expression equivalent to X|Y|Z, because Y must occur (instead it is equivalent to X<Y|Z). Generally speaking, if a temporal relation holds between X and Y, and the same temporal relation also holds between Y and Z, then that relation will also hold between X and Z. Applying this law to an expression containing a cyclic contradiction will reveal the violations of Mutual Exclusion:

$$X<Y \;.\; Y<Z \;.\; Z<X \Leftrightarrow X<Y \;.\; Y<Z \;.\; Z<X \;\mathbf{.}\; \mathbf{X<Z} \;\mathbf{.}\; \mathbf{Y<X} \;\mathbf{.}\; \mathbf{Z<Y}$$

The additions are in bold. As can be seen, we now have many contradictions, e.g. a conjunction of both Z<X and X<Z, which violates Mutual Exclusion. Because they are part of a conjunction,

the entire expression is a contradiction, i.e. it is impossible and can never occur. This can also be seen from a TTT:

X	**Y**	**Z**	**X<Y**	**Y<Z**	**Z<X**	**X<Y . Y<Z . Z<X**
1	1	1	0	0	0	0
1	1	2	0	2	0	0
1	2	1	2	0	0	0
1	2	2	2	0	0	0
1	2	3	2	3	0	0
1	3	2	3	0	0	0
2	1	1	0	0	2	0
2	1	2	0	2	0	0
2	1	3	0	3	0	0
2	2	1	0	0	2	0
2	3	1	3	0	2	0
3	1	2	0	2	3	0
3	2	1	0	0	3	0

As can be seen, no more than two of X<Y, Y<Z and Z<X are ever true at once, and so the AND is never true. There is also an extended set of Extension Laws that apply when the temporal gates involved are not homogeneous or when the shared events are not adjacent (the additions are again highlighted in bold):

Extended Laws of Extension

X&Z . Y&Z ⇔ X&Z . Y&Z . **X&Y**

Y&X . Y&Z ⇔ Y&X . Y&Z . **X&Z**

Z<X . Y<Z ⇔ Z<X . Y<Z . **Y<X**

Z&X . Y&Z ⇔ Z&X . Y&Z . **X&Y**

Z|X . Y|Z ⇔ Z|X . Y|Z . **Y|X**

XpY . YqZ

X&Y . Y<Z ⇔ X&Y . Y<Z . **X<Z**

X&Y . Y|Z ⇔ X&Y . Y|Z . **X|Z**

X<Y . Y&Z ⇔ X<Y . Y&Z . **X<Z**

X<Y . Y|Z ⇔ X<Y . Y|Z . **X|Z**

X|Y . Y&Z ⇔ X|Y . Y&Z . **X|Z**

X|Y . Y<Z ⇔ X|Y . Y<Z . **X<Z**

XpZ . YqZ

X&Z . Y<Z ⇔ X&Z . Y<Z . **Y<X**

X&Z . Y|Z ⇔ X&Z . Y|Z . **Y|X**

X<Z . Y&Z ⇔ X<Z . Y&Z . **X<Y**

X|Z . Y&Z ⇔ X|Z . Y&Z . **X|Y**

YpX . YqZ

Y&X . Y<Z ⇔ Y&X . Y<Z . **X<Z**

Y&X . Y|Z ⇔ Y&X . Y|Z . **X|Z**

Y<X . Y&Z ⇔ Y<X . Y&Z . **Z<X**

Y|X . Y&Z ⇔ Y|X . Y&Z . **Z|X**

ZpX . YqZ

Z&X . Y<Z ⇔ Z&X . Y<Z . **Y<X**

Z&X . Y|Z ⇔ Z&X . Y|Z . **Y|X**

Z<X . Y&Z ⇔ Z<X . Y&Z . **Y<X**

Z<X . Y|Z ⇔ Z<X . Y|Z . **Y<X**

Z|X . Y&Z ⇔ Z|X . Y&Z . **Y|X**

Z|X . Y<Z ⇔ Z|X . Y<Z . **Y|X**

These Extended versions also help to locate implicit contradictions in complex expressions. As a general rule of thumb, if a temporal relation applies to one input of a SAND gate, it will apply to the other input too; e.g. if *p* is some temporal operator, X*p*Y . Y&Z will yield X*p*Z. This applies regardless of the order of events involved. With the priority gates, it is more complicated, as the rules above demonstrate. There is also another, related law:

The Law of SAND Substitution

X&Y . X<Z ⇔ X&Y . Y<Z

X&Y . X|Z ⇔ X&Y . Y|Z

X&Y . Z<X ⇔ X&Y . Z<Y

X&Y . Z|X ⇔ X&Y . Z|Y

or generally, where ? is another temporal operator (i.e. either PAND or POR):

```
X&Y . X?Z ⇔ X&Y . Y?Z
```

This law embodies the principle just stated, i.e. if a temporal operator applies to one member of a SAND gate, it will also apply to the others; it is a special form of the Law of Extension.

There is also another interesting type of situation that arises in the Extended Extension Laws: when the right-most input of a POR also appears in a PAND or a SAND, or on the left of another POR, the POR is equivalent to a PAND gate. This is another law:

The Laws of POR Transformation

```
X|Y . Y ⇔ X<Y
X|Y + Y ⇔ X + Y
```

The Laws of POR Transformation are technically special cases of the Absorption Law that apply only to the POR gates. As shown earlier by the Absorption Law, for both PAND and SAND, if an event occurs separately connected by an AND, the PAND and SAND remain, whereas if connected by an OR, the PAND or SAND is redundant. This applies regardless of which input is chosen, i.e.:

```
X<Y . X ⇔ X<Y
X<Y . Y ⇔ X<Y
X&Y . X ⇔ X&Y
X&Y . Y ⇔ X&Y
X<Y + X ⇔ X
X<Y + Y ⇔ Y
X&Y + X ⇔ X
X&Y + Y ⇔ Y
```

However, the POR gate is different because only its left-most input *has* to occur; the gate is still true if none of the other events occur. This leads to different behaviour; if we know that one of the right-most events also occurs, e.g. if it is connected by an AND somehow, then the POR is equivalent to a PAND, because the other event definitely occurs and so the left-most must occur first. This also applies if the other event is further down the tree, e.g.

```
X|Y . (W&(Y|Z)) ⇔ X<Y . Y&W . Y|Z
```

When dealing with cut sets, this means that if one of the other input events to a POR also occurs elsewhere in the cut set (other than as another right-most input to a POR), then there is a *before* relationship between those events, i.e. a PAND rather than a POR applies.

When the other event is connected by an OR gate, the behaviour is slightly different; because the right-most event on its own is sufficient to cause the top event, the temporal relationship between it and the left-most event becomes irrelevant. Instead, the occurrence of either (or both, in any order) will cause the top event. Hence, X|Y + Y ⇔ X + Y. When the POR in question appears as part of a larger cut set, the POR is removed and only its left-most event remains, e.g. X|Y.Z + Y ⇔ X.Z + Y. Obviously, if the POR has other events, only the one that occurs elsewhere is removed, and the POR remains, e.g. X|Y|Z + Y ⇔ X|Z + Y.

PANDs and PORs are also linked by another set of temporal laws:

The Laws of Priority

```
X<Y + X|Y ⇔ X|Y
X<Y . X|Y ⇔ X<Y

X<Y + X.Y ⇔ X.Y
X&Y + X.Y ⇔ X.Y
X|Y + X.Y ⇔ X
```

In the first form, `X<Y + X|Y ⇔ X|Y`, the PAND gate is overridden by the POR gate, which has 'priority' over it. The PAND gate is a subset of the POR gate, as shown in Figure 32; therefore, whenever the PAND is true, the POR will also be true, although the converse is not necessarily the case. Thus, when these two are connected by an OR (e.g. by occurring in separate cut sets), the PAND is overridden by the POR and is redundant. Note that this still applies even if the PAND is part of a larger cut set, e.g.

```
A<B.C&D + A|B ⇔ A|B
```

but not if the POR is part of a larger cut set, unless the other parts of that cut set also appear in the PAND's cut set, e.g.

```
A<B.C&D + A|B.E ⇔ A<B.C&D + A|B.E
A<B.C&D + A|B.C ⇔ A|B.C
```

The second version listed here, `X<Y . X|Y ⇔ X<Y`, is effectively a special case of the Law of POR Transformation, because the second input to the POR (i.e. Y) occurs in the PAND gate and must occur, meaning the POR is equal to a PAND. In other words:

```
X<Y . X|Y
X<Y . X<Y  (from POR Transformation)
X<Y        (from Idempotent: X.X ⇔ X)
```

The last three forms of this law apply to the AND gate when in a disjunction. Just as the POR has priority over the PAND, so the AND has priority over the PAND and SAND gates; if the AND is true, then the PAND and SAND are redundant, because the AND states that its input events can occur in any order, only one of which is represented by the PAND or SAND. With the POR, this is slightly different: the AND states that the events can occur in any order, but the POR states that only the left-most input *must* occur; the end result is that the other events are redundant and only the left-most remains. This particular case is due to the Reductive Completion Law:

```
X|Y + X.Y
X|Y + X<Y + X&Y + Y<X (apply Conjunctive Completion Law)
X + X<Y               (apply Reductive Completion Law)
X                     (apply Absorption Law)
```

One thing that has not yet been discussed is how to deal with contradictions once they arise. The Extension Laws, the Laws of Simultaneity, and the Mutual Exclusion help to manipulate the expressions into a simpler form that is easier to handle, where the contradictions are more readily apparent, but what then? Back in section **3.3.1**, a set of Boolean laws that apply to tautologies and contradictions were given. These are repeated here, and extended to account for the temporal gates:

The Laws of Tautologies and Contradictions

```
X.0 ⇔ 0
X+0 ⇔ X
X.1 ⇔ X
X+1 ⇔ 1
```

Temporal versions

```
X<0 ⇔ 0
```

```
0<X ⇔ 0
X<1 ⇔ 0
1<X ⇔ X
X&0 ⇔ 0
X&1 ⇔ 0[14]
X|0 ⇔ X
0|X ⇔ 0
X|1 ⇔ 0
1|X ⇔ 1
```

The first four (or more specifically, the first two) are used to eliminate contradictions when they arise during qualitative analysis. If a contradiction occurs in a cut set, then by the law X.0 ⇔ 0, the whole cut set is a contradiction too. Then, by the law X+0 ⇔ X, that entire cut set can be removed. This provides the primary means of removing contradictions from temporal fault trees during qualitative analysis and will be explained in more detail in the following chapter.

The other versions show the behaviour of the temporal gates when confronted with contradictions and tautologies as inputs. For the conjunctive temporal gates, a contradiction always means they are false; as subsets of the AND, this behaviour is effectively inherited. Conjunctive gates always require all of their inputs to be true, and if one is a contradiction, that will never be possible. This is also true of the POR if its left-most input is a contradiction, otherwise the contradiction is irrelevant, since the POR allows for the possibility that one of its other inputs might not occur at all.

All of this behaviour is best illustrated by a TTT:

0	**1**	**X**	**X<0**	**0<X**	**X<1**	**1<X**	**X&0**	**X&1**	**X|0**	**0|X**	**X|1**	**1|X**
0	1	0	0	0	0	0	0	0	0	0	0	1
0	1	2	0	0	0	2	0	0	2	0	0	1

The cases where tautologies are involved (i.e., when one of the inputs is a 1) are interesting because they have a unique effect on all events. A tautology, by definition, is always true; it is therefore true before *any* event. This means that a tautology always has the value 1, and any other events will receive sequence values from **2** onwards, not from 1 onwards as usual; this is because even the first event to occur is actually second after the tautology. Therefore, no event can ever occur before a tautology; to attempt to do so is to cause a contradiction, hence X<1 ⇔ 0 and X|1 ⇔ 0. The other way around, however, *is* possible; the tautology will *always* occur first, and so all other events will occur later, assuming they occur. Thus 1<X ⇔ X (since it relies on X occurring) and 1|X ⇔ 1 (since the POR is true when its left-most input is true, which in this case it always is).

Having tautologies as inputs to a temporal gate is only possible if a basic event is a tautology, though it is occasionally possible to see contradictions as inputs to temporal gates without a basic event being a contradiction (for instance, (X<Y.Y<Z) & Z).

3.3.4 Conclusion

The temporal laws, taken together with the definitions used in Pandora, allow for a great deal of manipulation, simplification, and reduction to take place on temporal fault trees, including the facility to detect and remove contradictions when they arise. The temporal laws can be proven using temporal truth tables, which demonstrate equivalence between two (or more) temporal expressions. If the number of events involved renders the use of TTTs impractical, it should be possible to use the temporal laws to prove other temporal expressions to be true or false; if it is possible to reach one expression from the other by applying a succession of temporal laws that *have* been proved using TTTs, then the expressions are equivalent.

The temporal laws will be discussed further in Chapter 4, where their use in qualitative analysis of temporal fault trees will be explained in greater depth.

[14] Technically, if X is also a tautology, then the result is a tautology; i.e. if X = 1, then X&1 = 1. In practice this can only occur if a tautology is a direct input to a gate somewhere, because no gate can result in a tautology as a result of any sequence or combination of normal (non-tautological) inputs.

3.4 A Note on NOTs: The Scourge of Pandora

"The language of truth is simple."
- Euripides

3.4.1 The Question

There is one type of gate which has not yet been described in connection with Pandora – the NOT gate. While not found in the *Fault Tree Handbook*, the NOT is not wholly uncommon in fault trees and it is not unreasonable to briefly explain the reasons why NOT is not currently included within the Pandora framework.

It has already been mentioned in Chapter 2 that the inclusion of NOT gates in fault trees will normally lead to non-coherent fault trees, i.e. non-monotonic fault trees in which a failure can prevent or 'correct' a system failure. This necessitates a more complicated form of analysis (both quantitatively and qualitatively), typically involving the use of **implicants**, which are analogous to cut sets except that they also include the complements of events (e.g. where a cut set may contain "X.Y", the equivalent implicant may contain "X.Y.¬Z"). Implicants are then reduced in a similar way to cut sets to obtain **prime implicants** (analogous to minimal cut sets); this involves applying extra laws, such as the Consensus Law (X.Y + ¬X.Z ⇔ X.Y + ¬X.Z + Y.Z), to ensure that all necessary prime implicants are present (Sharvia & Papadopoulos, 2008). The analysis must therefore take into account events that are false as well as events that are true and this results in a larger state space – meaning additional complexity and decreased performance.

However, despite the additional complexity they bring, NOT gates are useful in fault trees for describing situations with mutually exclusive conditions, e.g. where one failure prevents another from occurring, or phased mission systems in which certain failures can only occur in certain modes of operation (Sharvia & Papadopoulos, 2008). Andrews (2000) gives an example of a situation requiring NOT gates to model correctly involving traffic lights at a crossroads and three cars.

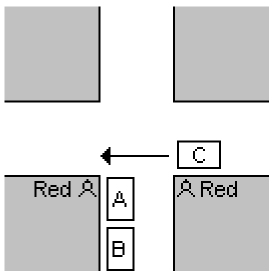


Figure 33 - NOT gate example from Andrews (2000)

Cars A and B are at a red light. Failures 'A' and 'B' are the failure of those cars to stop. Car C has right of way and thus failure 'C' is the failure of C to continue moving forward (since C continuing to move is normal behaviour). Collision between two cars therefore takes place if:

- A fails to stop and C continues (A.¬C)
- A stops but B fails to stop (¬A.B)
- B fails to stop and C continues (B.¬C)

This situation is difficult to model without a NOT gate. Another example, from Sharvia & Papadopoulos (2008), involves a leaking oil pipe where oil leaks (OilLeak) only if the pipe is ruptured (PipeRupture) and there is no omission of oil supply (¬OmissionOfOil), i.e. OilLeak = PipeRupture . ¬OmissionOfOil. Since an earlier omission of oil (a failure in itself, e.g. caused by a jammed valve elsewhere in the system) would prevent this failure, the NOT is required to ensure mutual exclusion.

Thus although the use of NOT results in a non-coherent fault tree and complicates analysis, in some situations it is necessary to fully capture the correct failure behaviour of the system. This is not unlike the case for temporal gates: they result in more complex semantics, but are necessary to correctly model sequences of failures. It is to be expected that any inclusion of NOT gates in the Pandora framework would similarly require more complicated algorithms that can deal with both the complements of events as well as the sequences of events.

However, there are a number of fundamental obstacles to be overcome before the mixing of NOT gates and temporal gates is possible. The first and foremost is a semantic problem: what does a NOT actually mean in a temporal setting? In a purely Boolean system, the NOT gate simply negates a Boolean value, so NOT(X) means that X did not happen. By this definition, the occurrence of a NOT gate represents the *non-occurrence* of a fault or other gate. In Pandora, however, although an event or gate still represents the occurrence of something, whether it be a fault or a combination of other events subject to certain conditions, it also has a *temporal* value as well as a Boolean one – if an event occurs, it must have occurred at some point in time. This also applies to compound events, i.e. gates: they must become true at a certain point in time. The NOT gate thus poses a problem: it represents the non-occurrence of some input event, and it must have a temporal value. But at what moment in time does an event "non-occur"? How can there be sequences of non-occurring events?

It is hard not to understate the effect a NOT gate has on Pandora's definition of events. In Pandora, events and gates are defined such that they are initially false until they occur and then they remain true thereafter, i.e. they are monotonic; the order in which they move from false to true is then represented by their sequence values. By contrast, the NOT gate is not monotonic; when introduced to static fault trees, it leads to non-coherent fault trees, and when introduced to Pandora's temporal fault trees, it leads to virtually incoherent fault trees. The reason for this is simple: a NOT gate makes it possible to go from true to false, as well as from false to true. In normal circumstances, a NOT gate is true when its input is false and false when its input is true; from a temporal point of view, before the input event occurs it would initially *true*, and if its input event occurs, it would later become *false*. But if the sequence value of a gate or event indicates the order in which it became true, how should we represent the order in which an event or gate becomes false?

3.4.2 Some Specious Solutions

There are several options, but all are highly problematic. One option is to assume that the NOT gate has the same sequence value as its input event, since the logical value of the gate changes when the input event occurs. This is not an unreasonable assumption and is also consistent with the other gates (e.g. an OR gate has the same sequence value as the first of its inputs to occur). The problem with this approach is that the NOT gate would then have a non-zero sequence value when it is logically false (i.e. after the input occurs) and a zero sequence value when it is logically true (i.e. before the input event occurs). As explained earlier, a non-zero sequence value represents true, so this results in an unresolvable anomaly.

Another option is to give the NOT gate a sequence value of 0 when it becomes false and a non-zero sequence value before then. This makes the NOT gate logically consistent with the meaning of sequence values. However, the problem then is one of exactly what sequence value to give the NOT gate before its input occurs – or if the input never occurs at all. It cannot be 0, because 0 represents false, and the NOT gate in this case is true. Indeed, the NOT gate is *always* true if its input does not occur, and so therefore behaves like a tautology. Therefore we could give it the sequence value of 1 in this case, and force all other events to start from 2, as with tautologies. Then the NOT gate starts off with the sequence value of 1 and changes this to a sequence value of 0 when its input becomes true. However, in this case, the NOT gate removes any temporal significance from its input event – regardless of the time the input event occurs (or if it does not occur at all), the output sequence value will either be 1 or 0.

A third option is to keep the 1 sequence value before the NOT gate has occurred but then use a higher sequence value after it occurs, matching the sequence value of the input. This is more consistent from a temporal point of view, but totally inconsistent from a logical point of view, because the NOT gate would never have the sequence value 0, which ordinarily represents false.

These three options can be summarised as follows:

Option 1

Before occurrence:	NOT gate has the value 0.
After occurrence:	NOT gate has the sequence value of its input.

Option 2

Before occurrence:	NOT gate has the value 1.
After occurrence:	NOT gate has the value 0.

Option 3

Before occurrence:	NOT gate has the value 1.
After occurrence:	NOT gate has the sequence value of its input.

Unfortunately, all of these solutions are equally unpalatable. In the first case, the NOT gate performs no negation; in the second case, where only the values of 0 and 1 are used, all NOT gates would then seem to 'occur' at the same time, making it impossible to distinguish between a NOT gate that becomes false early on and a NOT gate that becomes false later. And in the final case, the NOT gate never appears to be 'false' at all, because it always has a non-zero sequence value.

These problems are compounded by the possibility of nested NOTs, e.g. an expression such as NOT(NOT(X)). From a purely Boolean standpoint, NOT(NOT(X)) is equivalent to just X. Because Pandora is intended to extend, not override, Boolean logic, NOT(NOT(X)) should also be equivalent to X in Pandora. However, none of the three options considered so far achieve this.

Option 1

X	**NOT(X)**	**NOT(NOT(X))**
0	0	0
1	1	1

In the first option, both NOTs will have the same sequence value as X and thus they will appear to be both temporally and logically equivalent. However, *any* sequence of nested NOTs will give the same result, because no negation is taking place – NOT(X) is itself equivalent to X. The NOT has no effect.

Option 2

X	**NOT(X)**	**NOT(NOT(X))**
0	1	0
2	0	1

The second option is more subtle but no less incorrect. The innermost NOT would have the value 1 before X occurs and the value 0 afterwards. However, the behaviour of the outermost NOT is not so clear. If a NOT has the value 1 before its input occurs, i.e. when its input is *false*, then it is logically true and can be said to have 'occurred'. In which case, the outermost NOT would have the value 0 at first, because its input has occurred, and then when X occurs, the inner NOT becomes 0 and the outermost NOT becomes 1. This interpretation is somewhat counter-intuitive, because it seems as though the inner NOT is 'un-occurring', i.e. first it occurred and then later it did not. Either way, the sequence value of the outer NOT is not temporally equivalent with the value of X, and in fact the sequence values themselves suggest that the outer NOT becomes true *before* X occurs – which is clearly nonsensical.

Option 3

X	**NOT(X)**	**NOT(NOT(X))**
0	1	1
2	2	2

In the third option, the innermost and outermost NOT gates both have the same values, but neither matches the values of X. Before occurrence of X, the inner NOT is 1, i.e. it is true from the beginning of time. But under this interpretation, if the input to a NOT gate is true (i.e. it has occurred), then the NOT gate has the same sequence value as the input – thus the outermost NOT gate has the value 1 as well. Thus it appears as though neither NOT gate is ever false at all.

3.4.3 A Better Solution

The problem ultimately reduces to one simple issue: in coherent fault trees, it is only necessary to look at the sequence in which events occur, i.e. the sequence in which they become true. Once NOT gates are introduced and fault trees become non-coherent, then it is also necessary to take into account the sequence in which events become *false*. Consider the timelines in Figure 34:

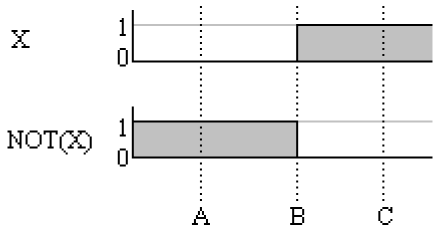


Figure 34 – NOT Timelines

Here, the shaded part indicates 'truth', so X is initially false and then becomes true at time 'B', while NOT(X) is the opposite. If an event Y occurs at point 'A', then we can safety say that Y occurs before X, i.e. Y<X would be true. But it is not so clear whether we can say that Y<NOT(X) is true. Similarly, if an event Y occurs at point 'C', then X<Y would be true, but it is not clear that NOT(X) < Y would also be true. The same problem arises at point 'B' with the SAND gate.

One way of clarifying this is to change the definition of the sequence value of an event to separate it from the logical value. Thus *S*(X) would only indicate the order in which X occurs and not whether X is true or not. The temporal gates would then operate only on the order in which an event changes its logical value, not necessarily only the order in which it changes from false to true. In which case, it would now seem valid to say that if Y occurs at 'A', then Y<NOT(X) would be true, since Y's logical value changes (from false to true) before the logical value of NOT(X) changes (from true to false). It is important to note that under this interpretation, an event can still only occur at most once, i.e. the logical value of an event or gate can change from false to true or true to false, but not both.

In this re-interpretation, NOT(NOT(X)) gives a more reasonable result. Using *S*(X) to represent sequence values, *L*(X) to represent logical values, and ¬ to represent NOT:

***L*(X)**	***S*(X)**	***L*(¬X)**	***S*(¬X)**	***L*(¬¬X)**	***S*(¬¬X)**
0	0	1	0	0	0
1	1	0	1	1	1

Note that if X has the logical value 0 (i.e. false), then it has the sequence value 0 (i.e. never occurred); if it has the logical value 1 (i.e. true), then it has a sequence value indicating the order in which it changed value, in this case 1 (since there is only one event, X). ¬X has the opposite logical values to X, and ¬¬X has the opposite logical values to ¬X, as we would expect from Boolean logic; however, both ¬X and ¬¬X have the same sequence values as X, which we would expect from Pandora's temporal logic (since a NOT gate changes value when its input changes value).

The behaviour of a NOT gate can therefore be described as follows:

NOT Gate

Exactly one input, *x*

Logical value:

If L(*x*) = true: L(¬*x*) = false

If L(*x*) = false: L(¬*x*) = true

Thus L(¬¬*x*) = L(*x*)

Sequence value:

S(¬*x*) = *S*(*x*)

Thus *S*(¬*x*) = *S*(*x*) and *S*(¬¬*x*) = *S*(*x*)

Although this re-interpretation of sequence values appears to result in the correct logical and temporal semantics for a NOT gate under Pandora, the separated logical and sequence values results in a more cumbersome representation, particularly in TTTs. A simpler way of representing the combined logical and sequence values of an event is to use signed integers: the absolute value of the integer indicates the order in which the value changed, while the sign indicates the direction of the change. Thus –1 indicates an event went from true to false first, and +2 indicates an event went from false to true second, and so forth. -0 indicates that an event never occurs at all (and is always false) while +0 likewise indicates than an event never occurs, but that it is always true. Thus +0 and –0 represent tautologies and contradictions respectively[15].

The behaviour of all five Pandora gates and the NOT gate can then be described in this system using a normal Temporal Truth Table:

[15] –1 and +1 could also be used for the same purpose if the thought of having a positive and negative 0 seems too strange. However, 0 better represents the 'start' of the period of observation, as using +/-1 implies that something that is – by definition – always the case (like a tautology or contradiction) had to 'occur' at some point.

X	Y	X+Y	X.Y	X<Y	X&Y	X|Y	¬X
-0	-0	-0	-0	-0	-0	-0	+0
+1	-0	+1	-0	-0	-0	+1	-1
-0	+1	+1	-0	-0	-0	-0	+0
+1	+1	+1	+1	-0	+1	-0	-1
+1	+2	+1	+2	+2	-0	+1	-1
+2	+1	+1	+2	-0	-0	-0	-2

Although this re-interpretation solves many problems, it involves radically changing the meaning of sequence values and would necessitate a review of all the definitions and temporal laws of Pandora.

3.4.4 Alternatives to NOT gates

Since it has been established that introducing NOT gates to Pandora is not a trivial exercise, it is worth questioning whether or not there is really a need for NOT gates; can situations normally modelled using NOT gates be modelled in other ways?

One option is the Priority Exclusive Or (PXOR) gate, mentioned earlier in this chapter. The PXOR would be true only if its first input is true and its other inputs are not. It can therefore model similar situations to the NOT gate, e.g. OilLeak = PipeRuptured PXOR OmissionOfOil. Here the oil leak only occurs if the pipe is ruptured and there is no omission of oil.

The following TTT compares the behaviour of the PXOR with the Pandora temporal gates, with ϕ representing the PXOR:

X	Y	X<Y	X&Y	X|Y	XϕY
0	0	0	0	0	0
1	0	0	0	1	1
0	1	0	0	0	0
1	1	0	1	0	0
1	2	2	0	1	0
2	1	0	0	0	0

In some respects the PXOR fills a niche in Pandora in that it represents the other part of the POR gate, i.e. the POR gate is true if its inputs occur in sequence (PAND) or if only its left-most input occurs (PXOR). However, it is not valid to say that X|Y is equivalent to X<Y + XϕY, since – as can be seen from the TTT – the sequence values are not equivalent. This also

means that a potential new Completion Law that represents all five areas of X+Y individually is invalid, i.e.:

$$X\phi Y + X<Y + X\&Y + Y<X + Y\phi X \Leftrightarrow X+Y$$

is not true because X<Y and Y<X have different sequence values when compared to the corresponding rows for X+Y.

Thus the PXOR – although at first glance a potentially close fit with Pandora – is not as compatible as it appears. Furthermore, the most fundamental argument against the PXOR is that – like the NOT gate – it introduces non-coherence to the fault tree. The NOT gate, as explained above, means that an event can also go from true to false as well as from false to true. The PXOR gate in fact goes a step further: a single PXOR gate can go from false to true *and* true to false – it can change value twice, not just once.

This can be seen by looking more closely at the sequence value of the PXOR gate. The PAND gate is true only once all input events have occurred because this is the only time at which we can be certain that all events have occurred in the correct order. The POR gate is true whenever its first event occurs, because we can safely say at that point that it has occurred before any other input events. However, is it safe to say that a PXOR gate is true when its first input occurs? What happens if its second event subsequently occurs? In order to maintain the correct semantics of the PXOR gate, it must become false at that point, meaning that it was initially false, became true, and then became false once again:

- Assume S(X) = 1 and S(Y) = 2.
- At moment 0, X PXOR Y is initially false (neither X nor Y are true).
- At moment 1, X occurs. X PXOR Y becomes true (since X is true but Y is not).
- At moment 2, Y occurs. X PXOR Y becomes false again (since X is true and Y is true).

This is completely inconsistent with the semantics of Pandora's events and may require even more re-interpretation than the NOT gate before it could be used with Pandora.

Fortunately, many of the situations in which a NOT gate or PXOR gate might be used can instead be modelled using the existing POR gate. The NOT gate is often used in fault trees to indicate a case of mutual exclusion, e.g. "X causes a system failure as long as Y does not happen." With a slight modification, this situation can also be modelled by a POR, i.e. "X causes a system failure as long as Y *has not happened yet*." For example, OilLeak = PipeRupture POR OmissionOfOil. This would mean that oil would leak as long as the pipe ruptures before the supply of oil is lost. The semantics are slightly different (in that an omission of oil in the future is not precluded) but the effect is much the same: oil will leak from the pipe.

In this respect, X.¬Y (and similarly XϕY) can be modelled as X|Y. This also means that monotonicity is retained and therefore the fault tree would remain coherent. If, however, the model of the system required that Y *never* occur, then only the NOT gate would be suitable, and in such cases a non-coherent fault tree is inevitable.

In summary then, although the incorporation of NOT gates into the Pandora framework is possible, it might not be necessary and is by no means a simple task.

3.5 Example

To show how the gates and logic of Pandora are used in practice, consider the simple example system first introduced in Figure 1 and reproduced here:

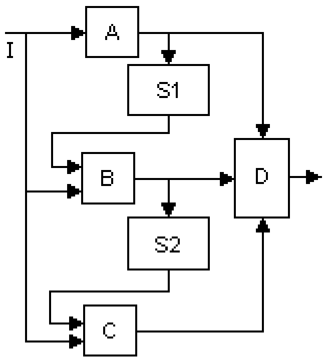


Figure 35 – The trusty triple redundant example

To describe the failure behaviour of the system, we need to be able to explain that B is only activated after a failure of A – assuming S1 has not yet failed – and that C is only activated after a failure of B – again, assuming that the sensor (S2) has not yet failed. To do this properly we need to represent the sequence of events. In other words, we need to use Pandora.

Each component can be annotated with Pandora logic to describe how it fails and how it reacts to failure at its inputs. A can fail either as a result of internal failure (`failureA`) or due to a lack of input from I (Omission of input, or `O-I`)[16]. Either failure will result in an omission of output. Thus the failure behaviour of A can be described as:

```
O-A = failureA + O-I
```

The failure behaviour of B is somewhat more complicated because it is dependent on A. We can use a special event, `startB`, to represent the trigger for B to be activated. `startB` occurs when O-A occurs, i.e. when A fails. The `startB` event is generated by sensor S1, which detects O-A. S1 can itself fail due to an internal failure, `failureS1`. Thus the behaviour of S1 would seem to be:

```
O-S1 = failureS1
startB-S1 = O-A
```

[16] Omission of input is a common cause failure for the entire system.

StartB-S1 and O-S1 are mutually exclusive in that O-S1 represents an omission of the start signal when it is required. However, this is a simplistic view, because as already explained, if the sensor fails *after* an omission of output from A has been detected, then it has no effect. Thus:

```
O-S1 = failureS1 < O-A + failureS1 & O-A
startB-S1 = O-A
```

Here the temporal gates have been used for the first time. An omission of S1's signal is caused by its failure before or at the same time as an omission of output from A. In these cases, no start signal will be sent (and B will not be activated).

Now we can also represent the failure behaviour of B. B will omit its output if it does not receive the start signal when A fails (e.g. due to a sensor failure) or if loses input or it fails internally once it has been activated. Thus:

```
O-B = O-S1 + startB-S1 < (failureB + O-I)
```

However, again the situation is not so simple. This is because if B does not receive its start signal, or fails dormant before the signal is received, then it will never activate. This actually causes an *undetectable* omission, whereas the failure of B once it has started causes a *detectable* omission, i.e. because the output of B starts and then stops, its cessation can be detected. A revised version would be:

```
Odet-B = startB-S1 < (failureB + O-I)
Ound-B = O-S1
      + (failureB + O-I) < startB-S1
      + (failureB + O-I) & startB-S1
```

This is now a more accurate reflection of the failure behaviour. A detectable omission happens once the component has been activated whereas an undetectable omission arises if the component fails before activation or never receives its activation signal.

Finally we can look at C and its activation sensor, S2. As with S1, S2 detects an omission of output from B (i.e. `Odet-B`) and activates C. It can only do this if it has not yet failed, and a failure of S2 after it has done this has no effect on the system. Thus:

```
        O-S2 = failureS2 < Odet-B + failureS2 & Odet-B
        startC-S2 = Odet-B
```

The failure behaviour of S2 therefore closely echoes that of S1. Next, C. C will omit output if it is never activated (due to O-S2) or if it fails before, during, or after activation:

```
        O-C = O-S2 + (failureC + O-I).startC-S2
```

In this case, a dormant failure of C and an active failure of C are treated the same and the order of events does not matter. If C fails at any time (or it loses input), then it will result in an omission of output – assuming C has received its start signal.

Component D is the system output and simply takes input from A, B, or C:

```
        O-D = Ound-B + O-C
```

Note that it uses `Ound-B` because `Odet-B` will lead to either an activation of C or `O-C`. We also assume here that D does not fail (if it did, it would also be a single point of failure).

These expressions representing the failure behaviour of the system can be combined into a fault tree by means of substitution, e.g. substituting `(O-I + failureA)` for all instances of `O-A`. The resultant expression is:

```
O-D = failureS1 < (failureA + O-I)
+ failureS1 & (failureA + O-I)
+ (failureB + O-I) < (failureA + O-I)
+ (failureB + O-I) & (failureA + O-I)
+ (failureS2 < ((failureA + O-I) < (failureB + O-I)))
+ (failureS2 & ((failureA + O-I) < (failureB + O-I)))
+ (failureC + O-I).((failureA + O-I)<(failureB + O-I)))
```

The fault tree for this expression is shown in Figure 36 overleaf:

To obtain the results for this fault tree, however, we must first find a way of performing a temporal qualitative analysis on it.

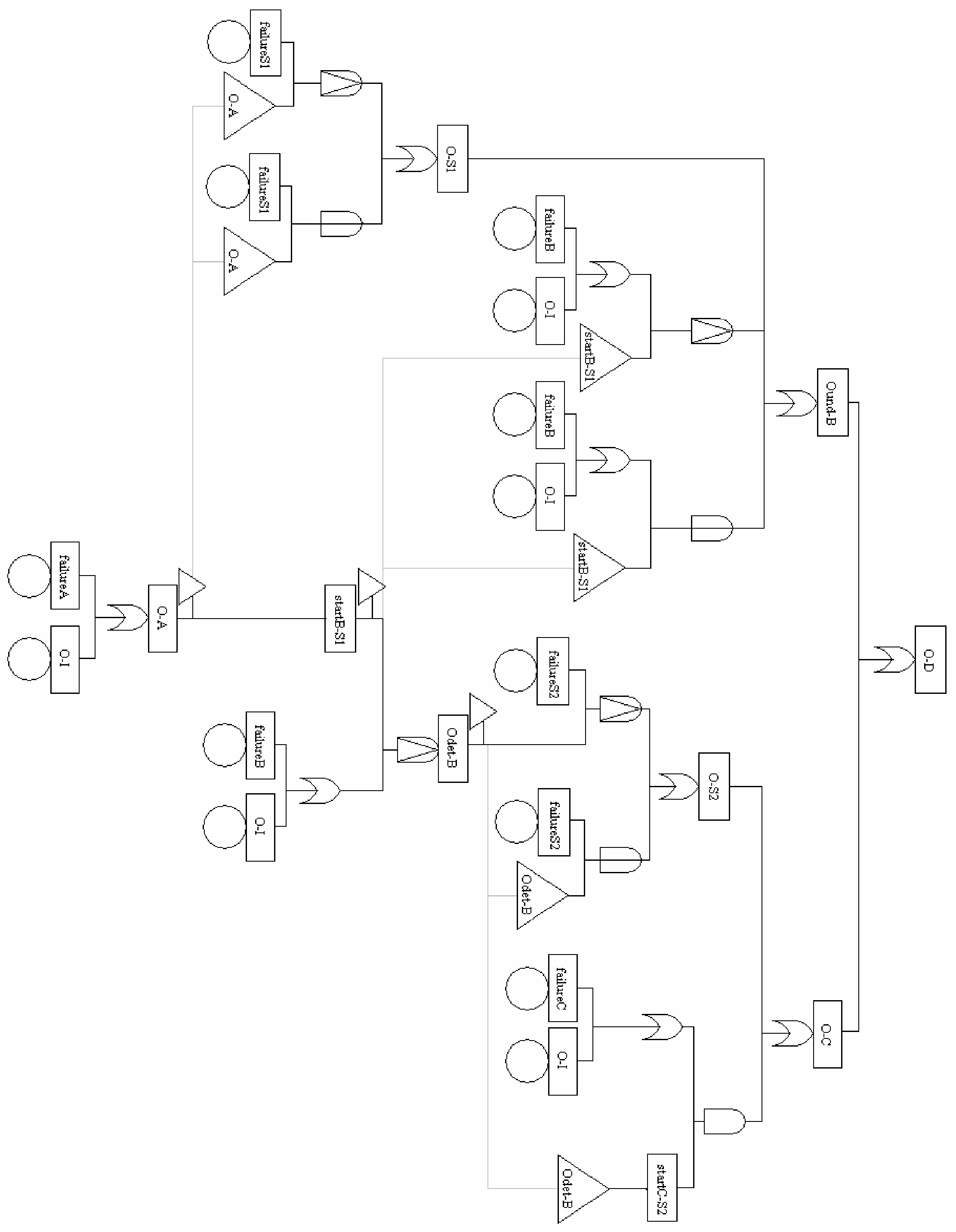


Figure 36 – Fault tree for the example system

4 Temporal Qualitative Analysis

"I have not failed. I've just found 10,000 ways that won't work."
- Thomas Edison

4.1 Basics of Qualitative Analysis

Qualitative fault tree analysis, as explained in Chapter 2, allows an analyst to draw conclusions about a system by looking at which combinations of events can cause the top event. Because the fault tree itself can contain many gates and many branches, often too many to be able to draw any conclusions from, qualitative analysis first simplifies and transforms the fault tree into a set of **cut sets**. Each cut set is a conjunction of a number of events, and the occurrence of all the events in a cut set will cause the top event to occur. Some cut sets will be redundant, and these will be removed to leave only the **minimal cut sets** (MCS); minimising the number of cut sets makes it easier to see which of the remainder are the most important. Qualitative analysis therefore consists of two broad phases. First, the fault tree has to be transformed into cut set format, i.e. disjunctive normal form. Secondly, the cut sets have to be minimised.

Once this is done, it is possible to look at the minimal cut sets and draw conclusions about the system being analysed. In general, the smaller the cut set, the more important it is, because it means that fewer events must occur in conjunction to cause the top event. The **order** of a cut set is how many events it contains, and so lower order cut sets tend to be more critical than higher order ones. Minimal cut sets of order 1 mean that only a single event is needed to cause the top event; these are **single points of failure** and generally indicate vulnerable points in the design of the system.

Chapter 2 mentions some methods of performing qualitative analysis (e.g. MOCUS), but all of them apply to normal Boolean fault trees. There are some methods of performing qualitative analysis on more complex trees, e.g. non-coherent trees containing NOT gates or Dynamic Fault Trees, but each of these methods has to be tailored to the type of fault trees they are intended to work on. Pandora is no exception: it requires a new qualitative analysis approach that can deal not only with the presence of AND and OR gates in the fault tree, but also PAND, SAND and POR gates too. Because there is no existing algorithm for performing qualitative analysis on PANDs[17], a new one must be created from the beginning.

[17] Although the DFT approach includes a qualitative analysis algorithm, as described in Chapter 2, it avoids dealing with PANDs directly, treating them instead as ANDs and then re-introducing the sequential information at the end.

4.2 Cut Sequences

4.2.1 Introducing Cut Sequences

The essential difference between Pandora and normal fault trees is that Pandora fault trees contain temporal gates that impose a sequence on a set of events: the order in which the events occur is important in Pandora. In the cut sets of ordinary static FTs, this is not the case; the events can occur in any order and still cause the top event. To distinguish between ordinary cut sets and cut sets in which the order of events may be significant, the term **cut sequence** (CSQ) is used; thus a cut sequence is a cut set in which some events occur in a certain order (i.e. some events have temporal significance). Similarly, a cut sequence in which every event must occur – and occur in the given order – to cause the top event to occur is termed a **minimal cut sequence** or MCSQ, analogous to a minimal cut set. A cut sequence takes the form of a conjunction of basic events or temporal gates (each containing only either further temporal gates or basic events).

Note that for a normal cut set to be minimal, it must contain no redundant basic events, i.e. if the occurrence of a subset of the events is sufficient to cause system failure, then that cut set is not minimal. With cut sequences, the meaning of "minimal" is not so straightforward. A cut sequence can be minimal if it contains no redundant events and no unnecessary sequences, i.e. if the occurrence of all events in any order is sufficient to cause system failure, then the cut sequence is redundant (since the ordering is not important). Thus, as will be seen shortly, a cut sequence like X<Y.Z is redundant if X.Y.Z is also a cut sequence/cut set. However, the issue of minimality is complicated further by the issue of Completion, i.e. that some operators are subsets of others (particularly PAND and SAND being subsets of AND). Thus an expression such as X.Y + (X<Y).Z is non-minimal because X.Y includes (X<Y).Z. This type of problem – where a temporal redundancy is hidden within a conjunction (or disjunction) – is known as a Completion Problem, because the best way to detect it is to first apply the Completion Law, e.g. expand X.Y into X<Y + X&Y + Y<X. In this case, the redundancy between X<Y and (X<Y).Z then becomes immediately apparent.

The definition of minimality is further complicated by the fact that there is often more than one way of representing a cut sequence – typically thanks to the Completion Laws. For example, is "X.Y" more minimal than "X<Y + X&Y +Y<X"? The latter is perhaps more explicit and more detailed, whilst the former is more concise, but the two representations are equivalent and both are 'minimal' in the sense that they contain no redundancies or unnecessary sequences. This point will be discussed again later but the reason for producing minimal cut sets or sequences in the first place is to allow the analyst to draw conclusions about the behaviour of the system; in

most cases, the fewer cut sequences there are, the easier it is for the analyst to understand the results, so the more concise form is generally preferred over the expanded form.

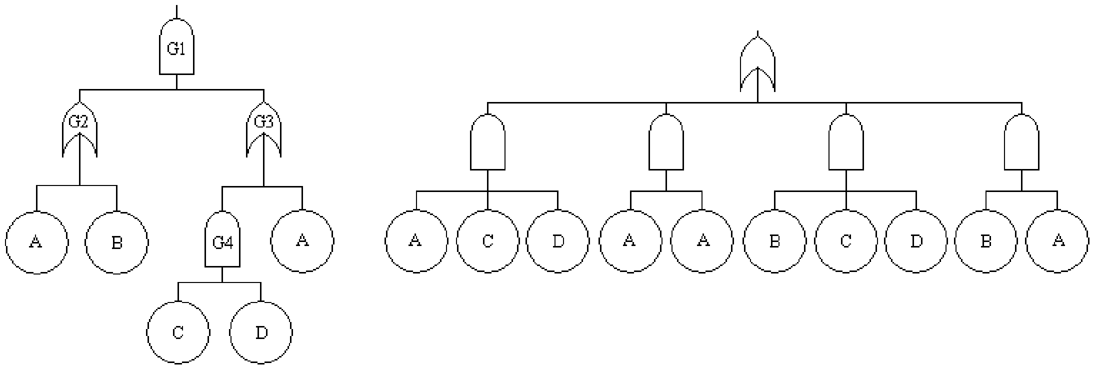


Figure 37 – Transformation of a fault tree (left) into its cut sets (right)

Regardless of their form, the goal of qualitative analysis in Pandora is to obtain the MCSQs for a temporal fault tree. Unfortunately, the process is not as simple as it is for non-temporal fault trees because there are now *five* gates to deal with, instead of just two. The first problem to be overcome is how to prioritise these gates in cut sequences – what is the equivalent of disjunctive normal form in Pandora? In normal cut set form, AND has a higher precedence than OR, so that groups of events are connected by AND gates which in turn are connected by a single OR gate. This can be seen in Figure 37; note that the cut sets are not minimised here (they minimise to just A + B.C.D). But in a cut sequence, we may also have PANDs, SANDs and PORs – so where should they appear?

One of the objectives of Pandora is to produce results as similar to existing qualitative FTA as possible. Therefore, in Pandora, CSQs are constructed such that OR and AND gates still appear at the top, in that order (i.e. one OR gate with one or more AND gates beneath it). This is still a disjunctive normal form. However, CSQs contain temporal gates, which indicate that part or all of the CSQ has to occur in a certain sequence (or, in the case of the SAND, must occur all at the same time). In cut sequence form, therefore, these must then appear beneath the AND gate, i.e. as part of the conjunction.

This is achieved by using the precedence of the operators (OR < AND < POR < PAND < SAND) to construct a hierarchy amongst the temporal gates such that SAND gates contain only basic events, PAND gates contain SANDs, and events, while PORs contain any event, PAND, or SAND, but not ANDs/ORs. For example, A.(B<(C&D)) is in the correct hierarchical order, but A.(B&(C|D)) is not. The resulting cut sequence form is called **hierarchical temporal form** or HTF; it can be thought of as a temporal disjunctive normal form in which each AND gate represents a different cut sequence.

Note that not all of a cut sequence has to be ordered; it is entirely possible for some events to be temporally significant and others not to be, i.e. only the temporally significant events need to come in a certain order, though all the events in a cut sequence still need to occur to cause the top event. The only potential exception to this is the POR gate, because its non-priority inputs (all except its left-most) do not necessarily need to occur. In this chapter, the term cut sequence (or minimal cut sequence) is often used inclusively, referring both to cut sequences proper and non-temporal cut sets collectively.

4.2.2 Obtaining Cut Sequences

The first phase of qualitative analysis, as already mentioned, is to obtain the cut sets from the fault tree – or in Pandora, to obtain the cut sequences. In traditional fault trees, this is done by manipulating the fault tree in some way, either according to Boolean laws (in the style of MOCUS and its derivatives) or by representing it in a different format (e.g. Binary Decision Diagrams). Pandora uses the first method – the application of laws to manipulate the fault tree.

In Chapter 3, several Boolean laws were described, three of which play a major role in obtaining cut sets: the Distributive Law, the Associative Law, and the Commutative Law. These three laws, and the temporal versions of them, likewise play a major role in obtaining cut sequences. The key is the Distributive Law, because this law can be used to alter the depth of a gate in the fault tree. By pushing all the OR gates to the top and all the temporal gates to the bottom, it is possible to transform the fault tree into HTF.

This is where the temporal laws provided in the last chapter come in useful, because they make it possible to apply the Distributive, Associative, and (where possible) Commutative laws to the three temporal gates as well as to AND and OR. However, the temporal versions of these laws are, on the whole, more complicated than their normal Boolean equivalents; even with just three events, the laws can result in quite large expressions. The key to managing this additional complexity is to keep things simple wherever possible; therefore, the first phase of qualitative analysis using Pandora is to transform the fault tree into a **binary fault tree**. The principle behind this is that no matter how complex the tree is or how many children a gate has, once it has been transformed into a binary tree, we only need a finite set of laws to be able to manipulate it: those involving only three events.

First we need to convert a normal Pandora fault tree into a binary fault tree. This is a relatively simple process: any gate with more than two children is converted into an equivalent gate or set of gates which only have two children each. As all operators are left-associative, this should not be problematic. Only the POR gate requires special handling, as any subsequent inputs to a POR gate are equivalent to a simple disjunction, e.g. X|Y|Z is equivalent to X|(Y+Z). This is because the POR gate is true if its left-most input occurs before any of its other inputs or if none of those other inputs occur, and so there is no temporal relation specified amongst the other inputs, only between the left-most and the rest.

4.2.3 Transforming Binary Fault Trees into Cut Sequences

Once the fault tree is in binary form, we can begin to convert it into HTF. This is usually performed in a bottom-up fashion, i.e. by starting at the leaf nodes and working upwards through the tree towards the top node. The general rule is to move OR and AND gates upwards while pushing temporal gates downwards. This is primarily achieved by applying the following Distributive Laws:

```
X < (Y.Z) ⇔ Y.(X<Z) + Z.(X<Y)
X < (Y+Z) ⇔ (X|Z).(X|Y).(Y+Z)
X < (Y<Z) ⇔ (X<Z).(Y<Z)
X < (Y&Z) ⇔ (X<Y).(X<Z).(Y&Z)
X < (Y|Z) ⇔ (X<Y).(Y|Z)
X < (Y|Z) ⇔ (X|Y).(Y|Z)
X & (Y+Z) ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y)
X & (Y.Z) ⇔ (X&Y).(Y&Z) + (X&Y).(Z<Y) + (X&Z).(Y<Z)
X & (Y<Z) ⇔ (Y<Z)&(Y<X)
X & (Y<Z) ⇔ (X&Z).(Y<Z).(Y<X)
X & (Y&Z) ⇔ X&Y&Z
X & (Y|Z) ⇔ (X&Y).(Y|Z).(X|Z)
X & (Y|Z) ⇔ (X|Z)&(Y|Z)
X | (Y+Z) ⇔ (X|Y).(X|Z)
X | (Y.Z) ⇔ X|Y + X|Z
X | (Y<Z) ⇔ (X|Z) + (X|Y) + X.(Z<Y) + X.(Y&Z)
X | (Y&Z) ⇔ X.(Y|Z) + X.(Z|Y) + (X|Y) + (X|Z)
X | (Y|Z) ⇔ (X|Y) + (X.Z<Y) + (X.Y&Z)
(Y+Z) < X ⇔ (Y<X) + (Z<X)
```

```
(Y.Z) < X ⇔ (Y<X).(Z<X)
(Y<Z) < X ⇔ (Y<Z).(Z<X)
(Y&Z) < X ⇔ (Z<X).(Y<X).(Y&Z)
(Y|Z) < X ⇔ (Y<X).(Y|Z)
(Y+Z) & X ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y)
(Y.Z) & X ⇔ (Y<X).(Z&X) + (Z<X).(Y&X) + X&Y&Z
(Y<Z) & X ⇔ (Y<Z).(Z&X).(Y<X)
(Y&Z) & X ⇔ Y&Z&X
(Y|Z) & X ⇔ (X&Y).(Y|Z)
(Y+Z) | X ⇔ (Y|X) + (Z|X)
(Y.Z) | X ⇔ (Y|X).(Z|X)
(Y<Z) | X ⇔ (Y|Z).(Z|X)
(Y&Z) | X ⇔ (Y|X)&(Z|X)
(Y&Z) | X ⇔ (Y|X).(Z|X).(Y&Z)
(Y|Z) | X ⇔ (Y|Z).(Y|X)
X . (Y+Z) ⇔ X.Y + X.Z              (traditional Boolean Distributive Law)
X + (Y.Z) ⇔ (X+Y).(X+Z)            (traditional Boolean Distributive Law)
```

Notice how, in each of these laws, there are two events on one side of the outermost operator and one event on the other – this is perfectly suited to the binary tree. If a gate has two gates as its inputs, then one can be treated as it if is a single event. For example, an expression such as (A.B)<(C+D) is in binary form, but the top level PAND contains two gates, rather than one gate and one event. Nevertheless, we can still apply the Distributive laws; we simply need to use two or three of them instead:

```
(A.B)<(C+D)
A<(C+D).B<(C+D)                  from (Y.Z) < X ⇔ (Y<X).(Z<X)
A|C.A|D.(C+D) . B<(C+D)          from X<(Y+Z) ⇔ X|Z.X|Y.(Y+Z)
A|C.A|D.(C+D) . B|C.B|D.(C+D)    from X<(Y+Z) ⇔ X|Z.X|Y.(Y+Z)
```

The last step in this example is to apply the normal Boolean Distributive Law, X.(Y+Z) ⇔ (X.Y)+(X.Z):

```
A|C.A|D.(C+D).B|C.B|D.(C+D) ⇔  A|C.A|D.C.B|C.B|D.(C+D) +
                               A|C.A|D.D.B|C.B|D.(C+D)
                            ⇔  A|C.A|D.C.B|C.B|D.C +
                               A|C.A|D.C.B|C.B|D.D +
                               A|C.A|D.D.B|C.B|D.C +
                               A|C.A|D.D.B|C.B|D.D
```

which is now in HTF (i.e. cut sequence format), giving us four cut sequences. This particular set of cut sequences is far from minimised, however. In traditional Boolean fault trees, there are two types of minimisation possible: Idempotence and Absorption. Idempotence eliminates redundant duplicate events within a cut set and Absorption eliminates redundant cut sets that contain other, smaller cut sets. In this case, we can apply Idempotence twice:

A|C.A|D.~~C.~~B|C.B|D.C ⇔ A|C . A|D . B|C . B|D . C
A|C.A|D.C.B|C.B|D.D ⇔ A|C . A|D . B|C . B|D . C . D
A|C.A|D.D.B|C.B|D.C ⇔ A|C . A|D . B|C . B|D . D . C
A|C.A|D.~~D.~~B|C.B|D.D ⇔ A|C . A|D . B|C . B|D . D

In two of the cut sets, an event is repeated: C in the first one and D in the last one. The Idempotent law says that X.X ⇔ X, so these reduce to a single event.

Next, we can apply Absorption. The Absorption law says that X + X.Y is equivalent to just X; in other words, if a subset of a cut set or sequence is sufficient to cause the top event, then that cut set or cut sequence is redundant and can be removed. In this case, we have two redundant cut sequences:

A|C . A|D . B|C . B|D . C
~~A|C . A|D . B|C . B|D . C . D~~
~~A|C . A|D . B|C . B|D . D . C~~
A|C . A|D . B|C . B|D . D

This leaves us with just two. However, because these are cut *sequences*, not cut sets, there is an additional stage of minimisation that we can do, by looking at the temporal gates. In this case, the Law of POR Transformation, X|Y . Y ⇔ X<Y, can be applied:

```
A|C . A|D . B|C . B|D . C ⇔ A<C . A|D . B<C . B|D
A|C . A|D . B|C . B|D . D ⇔ A|C . A<D . B|C . B<D
```

And now, finally, we have the two resulting minimal cut sequences. The POR Transformation law is only one of many possible methods of reducing the temporal gates, however; there are many possible ways in which temporal gates can be redundant or otherwise superfluous. The process of minimisation is much more complicated than the relatively simple process of obtaining the initial cut sequences.

4.3 Minimising Cut Sequences

There are many different ways that cut sequences can be minimised, besides the normal Boolean Idempotence and Absorption already discussed, but generally they fall into one of three categories: redundancy, contradiction, and completion. These three categories will be described next. Normally, the possibility for minimisation may exist both within cut sequences and also between cut sequences.

4.3.1 Redundancy and Cut Sequences

Redundancy is the most obvious form of minimisation as it is closest to the way normal Boolean trees are minimised. We already know that redundancy may occur when the same event (or gate) occurs more than once in a cut sequence (Idempotence) or when one cut sequence contains another (Absorption). However, when dealing with temporal gates, further forms of redundancy are possible too.

Firstly, events can be redundant if they occur individually in a cut sequence as well as inside a temporal gate in the cut sequence; for example, X.(Y<X). In these cases, the temporal versions of the Absorption Laws can be applied, which in this case yields just (Y<X). In general, if an event outside a temporal gate also occurs within one, then the event outside is redundant and can be removed from the cut sequence, i.e. the temporally significant event has priority. The relevant Absorption Laws are listed below; we only need five due to the way the cut sequences are constructed with AND gates above temporal gates (i.e. in HTF).

$$X . (X < Y) \Leftrightarrow X < Y$$
$$Y . (X < Y) \Leftrightarrow X < Y$$
$$X . (X \& Y) \Leftrightarrow X \& Y$$
$$Y . (X \& Y) \Leftrightarrow X \& Y$$
$$X . (X \mid Y) \Leftrightarrow X \mid Y$$

The exception to the pattern is the POR gate; if the event occurs as one of the POR's right-most children, then a different rule applies:

$$Y . (X \mid Y) \Leftrightarrow X < Y$$

This, as mentioned previously, is one of the POR Transformation Laws: if Y must occur, then the POR is equivalent to a PAND instead. PORs can also be absorbed into PANDs by the Law of Priority, for much the same reason:

$$X<Y \;.\; X|Y \Leftrightarrow X<Y$$

This is essentially a form of Idempotence. X<Y means that both X and Y must occur (and that they must occur in that order). If Y must occur, then X|Y is equivalent to X<Y, as we have just seen. Thus, X<Y . X|Y is equivalent to X<Y . X<Y which, due to Idempotence, reduces to just X<Y.

This form of Idempotence, the removal of duplicated gates, applies to all gates in a cut sequence, whether PAND, POR or SAND: if two gates are equivalent, then one can be removed. However, a more subtle form of this appears when gates are equivalent but do not *look* equivalent, and this can be caused by the SAND Substitution rule. If events occur together in a SAND gate inside a cut sequence, immediately below the AND, then those events are interchangeable throughout the rest of the cut sequence. Therefore, a CSQ like this:

$$X\&Y \;.\; X<Z \;.\; Y<Z$$

can be reduced to:

$$X\&Y \;.\; X<Z$$

because the Y and the X are interchangeable (i.e. they have the same sequence values), and so X<Z and Y<Z are equivalent in this case. This is the opposite of the Law of Extension, which, when applied, would add Y<Z to the cut sequence if not already present. As a result, although the SAND Substitution can shorten the cut sequence, it is generally not applied at this stage since any gates removed would be added by the Law of Extension again when checking for contradictions (see below). Idempotence also applies directly to the SAND, so that X&X ⇔ X. In this way, duplicate events in a SAND can be removed just as they can in AND gates.

The other type of redundancy applies across cut sequences. This is, generally speaking, one of the hardest forms of minimisation to perform, because it requires a great deal of checking. In Boolean fault trees, if a cut set contains all the events in another cut set, then it is redundant. This is not necessarily the case in Pandora if cut sequences are involved, because the events in the cut sequences may not necessarily occur in the same order. For example:

$$X.Y.Z + Z.Y.X \Leftrightarrow X.Y.Z$$
$$X<Y<Z + X<Y<Z \Leftrightarrow X<Y<Z$$
$$Z<Y<X + X<Y<Z \neq X<Y<Z$$

In the latter case, the events in the two cut sequences are in a different order, and so there is no redundancy. Only if all the events occur in another cut sequence *and* they occur in the same order is a cut sequence redundant. This is not always immediately apparent, e.g.

$$X\&Y.X<Z + X\&Y.Y<Z \Leftrightarrow X\&Y.X<Z$$

In this case, the X and Y are interchangeable due to the SAND Substitution rule. This can be seen explicitly by first applying the Law of Extension:

$$X\&Y \;.\; X<Z \Leftrightarrow X\&Y \;.\; X<Z \;.\; Y<Z$$
$$X\&Y \;.\; Y<Z \Leftrightarrow X\&Y \;.\; Y<Z \;.\; X<Z$$

Then by rearranging according to the Commutative Law (for AND), the two are clearly identical and thus one is redundant.

Cut sequences can also be subject to Absorption as well, though again the order has to be the same:

$$X<Y<Z + X<Y \Leftrightarrow X<Y$$
$$Z<Y<X + X<Y \neq X<Y$$

There are other, more subtle forms of Absorption, such as when the POR is involved. One of the Priority Laws states that $X<Y + X|Y \Leftrightarrow X|Y$, because POR has priority over PAND in a disjunction. Therefore, if we have two cut sets which are otherwise equivalent except one contains a POR and one contains a PAND (with the same children), then the one containing the PAND is redundant. For instance:

$$X<Y.W\&Z.Z<X + X|Y.W\&Z<X \Leftrightarrow X|Y.W\&Z<X$$

In this case, the two cut sequences are otherwise identical (W&Z.Z<X is equivalent to W&Z<X) and so, because the POR version overrides the PAND version, the PAND version can be removed. This goes back to the basic principle of Absorption and redundancy in cut sets: if

one cut set is always true when another cut set is true, then the latter cut set is redundant. This obviously only applies if the events are in the same order.

In a wider sense, the Law of Priority also applies to all temporal gates; the AND gate has priority over the two conjunctive temporal gates in a disjunction, and so cut sequences like:

$$X<Y.Z + X.Y.Z \Leftrightarrow X.Y.Z$$
$$X\&Y.Z + X.Y.Z \Leftrightarrow X.Y.Z$$

can be minimised accordingly. The POR, once again, is slightly different:

$$X|Y.Z + X.Y.Z \Leftrightarrow X.Z$$

This is a more sophisticated form of reduction: due to the Law of Priority, Y is irrelevant, because no matter what order it occurs in or whether it occurs at all, the cut sequences will still be true, and so it reduces to just the first event of the POR (and whatever else was in the cut sequence). A similar phenomenon occurs when the right-most events of a POR are found in other cut sequences, e.g.

$$X|Y.Z + Y.Z \Leftrightarrow X.Z + Y.Z$$

This is due to the other POR Transformation law, which states that X|Y + Y ⇔ X + Y. Essentially, the other possibility of the POR – that Y occurs after X – is overridden by the second cut sequence, leaving only the first possibility of the POR, that X occurs and Y does not.

4.3.2 Contradiction and Cut Sequences

One of the biggest changes in Pandora is that minimisation is now possible due to *contradiction*, as well as due to redundancy. Contradiction can arise due to a number of causes, but notably it only ever occurs *within* a cut sequence, never outside of it – an OR gate does not cause a contradiction.

The main cause of contradictions in cut sequences is the Law of Mutual Exclusion, which states that only one temporal relation can be true between a given pair of events; in other words, one must occur before the other, or they both occur at the same time. A cut sequence containing more than one temporal relation between the same two events will be a contradiction in itself. The classic example is:

$$X<Y.Y<X \Leftrightarrow 0$$

A cut sequence containing a contradiction is itself impossible, according to the law "X.0 ⇔ 0". However, as explained in the last chapter, detecting contradictions is not always trivial; sometimes the Law of Extension must first be applied. For example:

$$X<Y.X\&Z.Z\&Y$$

This cut sequence looks quite harmless at first glance, but a closer examination after application of the Law of Extension reveals otherwise:

$$X<Y.X\&Z.Z\&Y\mathbf{.Z<Y.X\&Y.X<Z}$$

Now we have three violations of the Law of Mutual Exclusion: X<Y and X&Y, X&Z and X<Z, and finally Z&Y and Z<Y. It is usual to apply the Law of Extension to a cut sequence first, before any checking for redundancy or contradiction, because otherwise more subtle contradictions like this could be missed.

The other way a contradiction can arise is due to Simultaneity, and this only applies to the two priority gates. If the same event occurs more than once as an input to a priority gate, then it is a contradiction, no matter whether it is a POR or a PAND gate. There is only one situation in which Simultaneity does not result in a contradictory cut sequence, and that is when the contradiction is one of the right-most children in an otherwise valid POR, i.e.

$$X|(Y<Y) \Leftrightarrow X|0 \Leftrightarrow X$$

In this case, that contradictory child of the POR is removed, because the right-most children of a POR are 'optional' as long as they occur after the left-most input. If the contradiction is the only other child of the POR, then the left-most input of the POR is all that remains.

In any case, once a cut sequence is found to contain a contradiction, it is removed. Due to the Law X+0 ⇔ X, any contradictory cut sequence is removed from the set of cut sequences entirely, because it is impossible. In practice, this can mean removing entire branches from the fault tree. If *all* cut sequences are found to be impossible, then it means that the top event is also impossible – in other words, *no* combination of basic events is sufficient to cause the top event. This is, however, rather unlikely unless the fault tree contains a mistake or the system is simply incapable of failing.

To aid the user in this type of scenario, where entire cut sequences (and possibly *all* cut sequences are removed), it is possible to provide the user with information about which cut sequences are contradictory and why they were removed (e.g. mutual exclusion, simultaneity), so that they are able to see the contradictions for themselves and ideally determine the causes of those contradictions in the fault tree (and presumably the system model on which the fault tree is based). This is potentially valuable if the contradictions arose from some modelling error, as the user can then try to fix the error.

4.3.3 Completion and Cut Sequences

Completion is the final type of minimisation possible in cut sequences. However, completion is also by far the hardest to detect and to deal with. Completion only ever occurs across cut sequences, never within them, and stems from cases where it is possible to apply one of the three Completion laws. The simplest example of the Completion Law is:

```
A<B + B&A + B<A
```

which reduces to just A.B according to the Conjunctive Completion Law. As a slightly more difficult example:

```
C.B|A + B&A.C + A|B.C
```

This is equivalent to C.A + C.B, according to the Disjunctive Completion Law. The law applies in this case because the three cut sequences are the same apart from the common term containing A and B. The law does *not* apply if one of the cut sequences contains a different event, for example:

```
C.B|A + B&A.C.D + A|B.C
```

does not reduce any further. Lastly, the Reductive Completion Law can also be used to great effect:

```
Z.Y<X + Z.X&Y + Z.X|Y
```

reduces to Z.X. These are merely the simplest cases of Completion, however – direct applications of the three laws. In theory, these can be identified by checking for the three relevant components in otherwise identical cut sequences, though such a check would not be simple.

Another type of reduction that is possible due to the Completion Laws involves situations like this:

X.Y.Z + X<Y + X&Y + Y<X

At first glance, these four cut sequences look quite harmless. On closer inspection, we find that we can apply the Conjunctive Completion Law to three of them, leaving us with:

X.Y.Z + X.Y ⇔ X.Y

The converse of this situation – where the conjunction renders the temporal sequence redundant can also be found, e.g.:

X.Y + X<Y.Z

reduces to just

X.Y

because X.Y includes the sequence X<Y, as can readily be seen if the Conjunctive Completion Law is applied first, i.e.:

X<Y + X&Y + Y<X ~~+ X<Y.Z~~ (X<Y.Z is made redundant by X<Y)

Now we can see that the Absorption Law comes into play as well, because the cut sequence (X<Y.Z) is redundant because it contains X<Y. In these cases, applying the Completion Law actually allows for further reduction to take place according to the other laws. Unfortunately, this sort of situation is not easy to detect. It is potentially a lot more complicated than just spotting the three components of a Completion Law (which can in itself be quite difficult). The difficulties of solving Completion Problems are discussed further in section **4.4.4**.

4.4 Doublets

" 'When I use a word,' Humpty Dumpty said in rather a scornful tone, 'it means just what I choose it to mean – neither more nor less.' "
- Lewis Carroll, *Through the Looking-Glass*

4.4.1 What are doublets?

The last section explained how it is possible to minimise and reduce a temporal fault tree by using the temporal laws described in Chapter 3. While this is quite possible (if often time consuming) to do manually, it is not always as easy as it looks, and it can be significantly more difficult to achieve automatically. Some problems have already been mentioned, such as the Law of Extension – if it is not applied first, some temporal relations can be missed, e.g. because the Law of SAND Substitution is not being taken advantage of or because it is not immediately clear that two events are contradictory. The sheer number of temporal laws also makes it more difficult to automatically reduce a temporal expression (i.e. an expression containing temporal operators) than a purely Boolean expression, where only a handful of laws are needed.

Fortunately, there is a way of making this process easier: **doublets** [18]. Doublets are very useful for two main reasons. Firstly, one of the difficulties with cut sequences is that they can contain several different types of gates: as well as AND, there may also be PANDs, PORs, and SANDs, and even when in HTF, these may in turn contain other gates (e.g. PANDs can contain SANDs, and PORs can contain PANDs and SANDs). So some events in a cut sequence can be as many as three gates deep. Doublets help to overcome this problem by reducing the maximum possible depth to just one gate, so that at most there will be one gate between the AND containing the cut sequence and an event. Secondly, as mentioned, there can be problems in ascertaining all the temporal relations between all the events in a cut sequence, mainly because they are often contained within several different gates. Doublets can also help solve this problem by making it much clearer what *all* the temporal relations are.

In essence, a doublet is a single temporal relation. Doublets are simply a way of clarifying the temporal relations between all events in a cut set because they show each temporal relation separately. A doublet contains *exactly two events*, packaged together with a single temporal operator to describe the relationship between those two events. Once a temporal expression has

[18] The name "Doublets" comes from a word puzzle invented by Lewis Carroll, which first appeared in 1879 in *Vanity*. The aim is to transform one word into another word in as few steps as possible, changing one letter at a time, forming a sequence of new words that links the two. For example: "Show that

been converted to use doublets, the doublets can be reordered, removed, and checked against other doublets with ease, because doublets are atomic and can be treated just like any other event. The idea is that doublets abstract the temporal information away from the cut sequence by encapsulating it inside the doublet itself, so that on the cut sequence level, there are only doublets and non-temporal events. A doublet is represented by square brackets; for example, [X<Y] is a doublet containing one PAND and the events X and Y. Similarly, [Y&X] is a doublet and [Z|W] is a doublet. The difference between X<Y and [X<Y] is that X<Y is a PAND gate that can have other events in it too, or even other gates; a doublet always has exactly two basic events, no more, no less:

- A doublet is an encapsulation of the temporal relation between exactly two events.
- It is represented using square brackets, e.g. [X<Y].
- It always contains two basic events and one temporal operator.
- A doublet is atomic and can be moved around and treated like a normal basic event.

Doublets are designed to be able to show the temporal relations between any number of events by explicitly representing the temporal relationship between each pair of those events. So, an expression like (X&Y)|Z results in three doublets – [X&Y].[X|Z].[Y|Z] – once the Law of Extension has been applied. These three doublets show *all* the temporal relations between the events in this cut sequence, and that is the most vital feature of doublets: once they have been created, all possible temporal relations are exposed and can thus be checked more easily. A cut sequence in which all temporal operators are encapsulated in doublets is in **base temporal form** (BTF) and is much easier to manipulate, because all temporal relations in the cut sequence are visible.

Doublets also have substantial benefits in any automated process, as will be explained in more detail in section **4.5**; primarily they make it much easier to identify contradictions and redundancies because now it is only necessary to compare two doublets at a time rather than two gates which may contain any number of inputs (some of which may be other gates). For example, checking for Simultaneity in an expression in BTF is just a matter of searching for any doublets which contain two identical events and a priority gate e.g. [X<X] or [Y|Y]. Mutual Exclusion is also much simpler: if any two doublets contain the same events in a different order (e.g. [X<Y].[Y<X]), or the same events in the same order but with different operators (e.g. [X<Y].[X&Y]), there is a strong likelihood that either they violate Mutual Exclusion or one is redundant in some other way:

GRASS is GREEN in 8 steps" would result in Grass → Crass → Cress → Tress → Trees → Frees → Freed → Greed → Green.

.	[X|Y]	[Y|X]	[X<Y]	[Y<X]	[X&Y]	[Y&X]
[X|Y]	Idempotent: **[X|Y]**	Mut. Excl.: **0**	Priority: **[X<Y]**	Mut. Excl.: **0**	Mut. Excl.: **0**	Mut. Excl.: **0**
[Y|X]	Mut. Excl.: **0**	Idempotent: **[Y|X]**	Mut. Excl.: **0**	Priority: **[Y<X]**	Mut. Excl.: **0**	Mut. Excl.: **0**
[X<Y]	Priority: **[X<Y]**	Mut. Excl.: **0**	Idempotent: **[X<Y]**	Mut. Excl.: **0**	Mut. Excl.: **0**	Mut. Excl.: **0**
[Y<X]	Mut. Excl.: **0**	Priority: **[Y<X]**	Mut. Excl.: **0**	Idempotent: **[Y<X]**	Mut. Excl.: **0**	Mut. Excl.: **0**
[X&Y]	Mut. Excl.: **0**	Mut. Excl.: **0**	Mut. Excl.: **0**	Mut. Excl.: **0**	Idempotent: **[X&Y]**	Idempotent: **[X&Y]**
[Y&X]	Mut. Excl.: **0**	Mut. Excl.: **0**	Mut. Excl.: **0**	Mut. Excl.: **0**	Idempotent: **[X&Y]**	Idempotent: **[Y&X]**

Table 4 – Detecting contradictions using doublets

As this table shows, whenever two doublets contain the same events, some reduction is possible, whether it is a simple application of the Idempotent Law, an application of the Priority Law, or Mutual Exclusion (which results in a contradictory cut sequence that can then be removed). Testing every doublet against every other doublet will help to remove most temporal redundancies from a cut sequence and identify any contradictions arising from mutual exclusion or simultaneity (at which point checking can stop, because the entire cut sequence is redundant). By then comparing the doublets against any remaining **unaffiliated events**, i.e. events that are not members of doublets, further redundancies can also be removed; the Absorption Law and the POR Transformation Law apply here, i.e. [X<Y].X ⇔ X and [X|Y].Y ⇔ [X<Y].

So, the next question is: how does one obtain some of these wonderful doublets?

4.4.2 Encapsulation

Encapsulation (or "doubletisation") is the process of converting a cut sequence into doublets by identifying and encapsulating each temporal relation. Much of the process is similar to converting an expression to HTF, since encapsulation consists mostly of applying Distributive laws to an expression. However, some of the laws need to be modified because the goal is to obtain one OR containing multiple ANDs each of which contain only doublets and unaffiliated events. The initial fault tree still needs to be in binary form first.

Here is the list of modified Distributive Laws, which produce doublets and unaffiliated events in cut sequence format:

```
X < (Y.Z) ⇔ Y.[X<Z] + Z.[X<Y]
X < (Y+Z) ⇔ [X|Z].[X<Y] + [X<Z].[X|Y]
X < (Y<Z) ⇔ [X<Z].[Y<Z]
X < (Y&Z) ⇔ [X<Y].[X<Z].[Y&Z]
X < (Y|Z) ⇔ [X<Y].[Y|Z].[X|Z]
X & (Y+Z) ⇔ [X&Y].[Y&Z].[X&Z] + [X&Y].[Y|Z].[X|Z]
           + [X&Z].[Z|Y].[X|Y]
X & (Y.Z) ⇔ [X&Y].[Y&Z].[X&Z] + [X&Y].[Z<Y].[Z<X]
            + [X&Z].[Y<Z].[Y<X]
X & (Y<Z) ⇔ [X&Z].[Y<Z].[Y<X]
X & (Y&Z) ⇔ [X&Y].[Y&Z].[X&Z]
X & (Y|Z) ⇔ [X&Y].[Y|Z].[X|Z]
X & (Y|Z) ⇔ [X|Z].[Y|Z].[X&Y]
X | (Y+Z) ⇔ [X|Y].[X|Z]
X | (Y.Z) ⇔ [X|Y] + [X|Z]
X | (Y<Z) ⇔ [X|Z] + [X|Y] + X.[Z<Y] + X.[Y&Z]
X | (Y&Z) ⇔ X.[Y|Z] + X.[Z|Y] + [X|Y] + [X|Z]
X | (Y|Z) ⇔ [X|Y] + X.[Z<Y] + X.[Y&Z]
(Y+Z) < X ⇔ [Y<X] + [Z<X]
(Y.Z) < X ⇔ [Y<X].[Z<X]
(Y<Z) < X ⇔ [Y<Z].[Z<X].[Y<X]
(Y&Z) < X ⇔ [Z<X].[Y<X].[Y&Z]
(Y|Z) < X ⇔ [Y<X].[Y|Z]
(Y+Z) & X ⇔ [X&Y].[Y&Z].[X&Z] + [X&Y].[Y|Z].[X|Z]
          + [X&Z].[Z|Y].[X|Y]
(Y.Z) & X ⇔ [Y<X].[Z&X].[Y<Z] + [Z<X].[Y&X].[Z<Y]
          + [X&Y].[Y&Z].[X&Z]
(Y<Z) & X ⇔ [Y<Z].[Z&X].[Y<X]
(Y&Z) & X ⇔ [X&Y].[Y&Z].[X&Z]
(Y|Z) & X ⇔ [X&Y].[Y|Z].[X|Z]
(Y+Z) | X ⇔ [Y|X] + [Z|X]
(Y.Z) | X ⇔ [Y|X].[Z|X]
(Y<Z) | X ⇔ [Y<Z].[Z|X].[Y|X]
(Y&Z) | X ⇔ [Y|X].[Z|X].[Y&Z]
(Y|Z) | X ⇔ [Y|Z].[Y|X]
```

This list of laws produces doublets that have already had the Laws of Extension and Priority applied, thus reducing the amount of minimisation necessary by doing some early. These laws are normally applied to the temporal gates found in cut sequences once the cut sequences have already been produced, but equally these laws can be applied to an expression not already in HTF, although technically this means that some of the 'doublets' produced are not real doublets since they may contain other gates and therefore need further encapsulation themselves. As a result, encapsulation is best performed only on expressions in binary HTF, because then the number of laws needed are reduced to just these:

```
X < (Y<Z) ⇔ [X<Z].[Y<Z]
X < (Y&Z) ⇔ [X<Y].[X<Z].[Y&Z]
X | (Y<Z) ⇔ [X|Z] + [X|Y] + X.[Z<Y] + X.[Y&Z]
X | (Y&Z) ⇔ X.[Y|Z] + X.[Z|Y] + [X|Y] + [X|Z]
X | (Y|Z) ⇔ [X|Y] + X.[Z<Y] + X.[Y&Z]
(Y<Z) < X ⇔ [Y<Z].[Z<X].[Y<X]
(Y&Z) < X ⇔ [Z<X].[Y<X].[Y&Z]
(Y&Z) & X ⇔ [X&Y].[Y&Z].[X&Z]
(Y<Z) | X ⇔ [Y<Z].[Z|X].[Y|X]
(Y&Z) | X ⇔ [Y|X].[Z|X].[Y&Z]
(Y|Z) | X ⇔ [Y|Z].[Y|X]
```

As an example, consider the fault tree from Figure 25. Figure 38 shows what that tree looks like when converted into doublets:

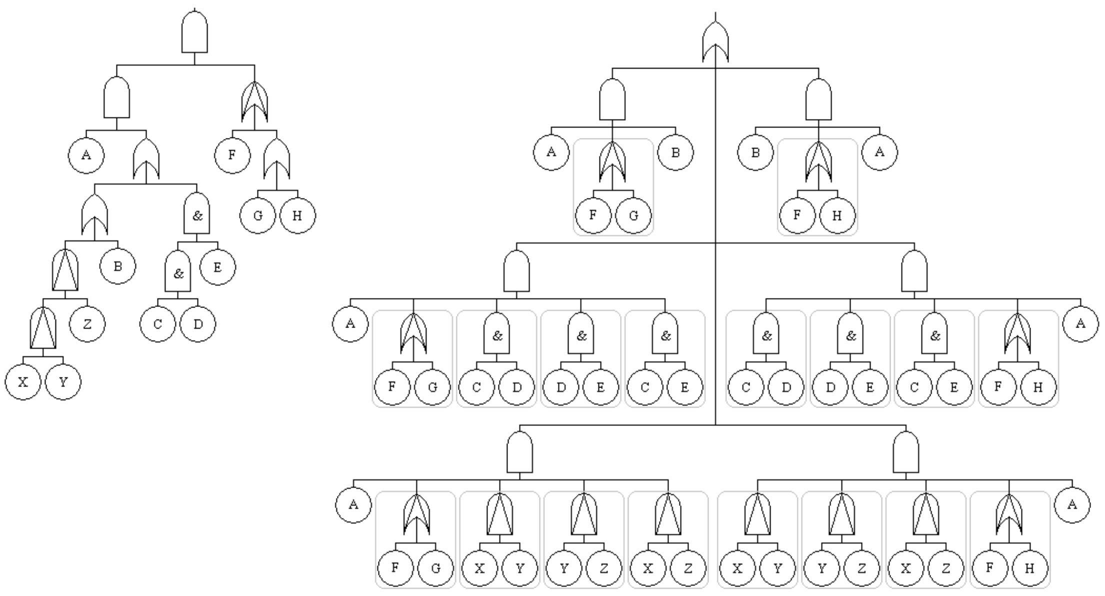


Figure 38 – Conversion of a fault tree into doublets

The tree on the left is the binary fault tree and the rather larger tree on the right is the same tree converted into doubletised cut sequences (there are six cut sequences, two on either side on three levels). The doublets are drawn with a box around them for clarity, but this is just for illustrative purposes and there is no requirement for a doublet to be shown with a box around it. Nevertheless, Figure 38 illustrates base temporal form very clearly: the maximum depth of such a tree is 3 gates: an OR, an AND, and a temporal gate encapsulated within a doublet. In this particular tree, there is no further reduction to be performed – it is already minimised, due to the fact that none of the events occur more than once in the original tree.

4.4.3 Comparison of Doublets using the Temporal Product

Frequently, the main use of doublets is to detect redundancies and contradictions with other doublets, and for this we need to be able to compare them efficiently. One method for doing this is a numerical approach inspired by Semanderes' ELRAFT algorithm (which was briefly described in Chapter 2). The principle is that each basic event is assigned a unique prime number as its identifier. In ELRAFT, these are then multiplied together to find a number that represents a whole cut set. Similarly, in Pandora, a doublet multiplies both its constituent events' prime numbers together to get a number that represents the events in the doublet. For example, if X = 2 and Y = 3, then [X<Y] would have the identifier 6. Any other doublet that contains the same events will then have this same identifying product.

However, ELRAFT is a Boolean approach without the concept of sequence; Pandora is not. Therefore, each doublet also has a sign to represent the order of its events. This is produced by subtracting the second event's prime from the first event's prime, which for [X<Y] would give -1. The sign of this value is then applied to the identifier, e.g. in this case, [X<Y] has the value -6. [Y<X], when subtracted, would give +1 instead, so [Y<X] has the value 6. This value is known as the **temporal product** of the doublet[19]. This method of calculating the temporal product makes it very easy to detect Simultaneity/Idempotence because the subtraction would yield a value of 0, e.g. [X<X] = 2 – 2 = 0. In this case the doublet is either a contradiction (for PAND and POR) or reduces to just a single event (for SAND).

It is then a simple matter to compare two doublets because we only need to compare their temporal products, instead of checking each event against every other event. Dividing one doublet's temporal product by another will show whether or not they each contain the same events and, if so, whether they are in the same order: a value of 1 means the doublets contain the same events in the same order, while a value of –1 means the doublets contain the same events in different orders. To determine the precise nature of the minimisation possible, however, the operators must still be compared directly (because the temporal product does not differentiate between a PAND and a POR, or a SAND and a PAND, for example). The temporal product of a doublet also makes it easy to check whether it contains an unaffiliated event: simply divide the temporal product by the event's prime ID, and if there is no remainder, the doublet contains the event. For example, [X<Y].X: -6 MOD 2 = 0, so [X<Y] contains X.

The temporal product is also useful for comparing entire cut sequences. Just as cut sets can have prime products in ELRAFT, cut sequences can have their own products too, composed of the product of all unaffiliated events or doublets in the cut sequence. For example, if A=2, B=3, C=5, and D=7:

```
A.[B<C].[C&D].[B<D]
     A     = 2
     [B<C] = 3x5 = 15
     [C&D] = 5x7 = 35
     [B<D] = 3x7 = 21
Product    = 2 x 15 x 35 x 21 = 22050
```

This number is very useful when comparing cut sequences against each other. If we divide one cut sequence by another, and have no remainder, then one may contain the other. For example:

```
[B<C].[B<D] = 3x5 x 3x7 = 15 x 21 = 315
22050 MOD 315 = 0
```
so first CSQ may contain second

[19] So called because it consists of two parts: the product of the prime numbers and a sign indicating the temporal

However, because it is not possible to use the signs of the temporal products at the cut sequence level (because a cut sequence may contain multiple doublets), to be sure that one CSQ contains another, it is necessary to go through and make sure that all doublets that appear in both also occur in the same order by comparing every doublet in each cut sequence.

The process of automatic minimisation using doublets is described further in section **4.5**.

4.4.4 Detecting Completion-based reductions using Doublets

As mentioned earlier in this chapter, there are three broad categories of reduction possible when performing minimisation in Pandora: redundancy (especially absorption), contradiction, and completion. The first is relatively easy to detect with or without doublets, though doublets do make this process easier by breaking up the expressions into smaller units. Contradiction is a lot easier with doublets, because detecting them becomes a simple numerical process of comparing temporal products. Completion, in all of its problematic diversity, is the hardest type of possible reduction to detect. Fortunately, doublets can help solve this problem, though the Completion reduction problem in general is better solved in other ways (see Section **4.6**).

The principle behind the introduction of doublets is that by encapsulating the temporal information inside the doublets, and therefore reducing the contents of a cut sequence to just events and doublets connected by an AND gate, we simplify the situation and make it easier to identify potential areas for reduction and minimisation. We can take this principle one step further: what if a cut sequence contained *only* doublets?

By representing *all* temporal relations, even those implicitly contained within the AND gates, we can also detect some of the elusive Completion reductions that would otherwise evade us. The key to this is the Completion Law itself. Situations like:

$$X.Y.Z + X<Y + X\&Y + Y<X \Leftrightarrow X.Y$$

are hard to detect because they require the Completion Law to be applied first; it would have to be a very sophisticated algorithm to be able to recognise the latter three cut sequences as X.Y without this intermediate step. But rather than attempting to apply the Completion law to every combination of three (or even more) cut sequences, which would obviously be time consuming and grossly inefficient, it is possible to apply the Completion Law *before* minimisation, even before the doublets have been constructed.

order the events occur in (positive means the larger numbered event occurs first, minus means it occurs second).

The Conjunctive Completion Law can be applied to every AND gate in an expression in HTF (or binary HTF). The result will be the elimination of all AND gates and the production of a larger number of cut sequences (albeit no longer in true HTF). Each cut sequence will contain only temporal gates. For example:

```
X < (Y.Z)  ⇔   Y.X<Z + Z.X<Y                       (Now in HTF)
           ⇔   Y<(X<Z) + Y&(X<Z) + (X<Z)<Y +       (Completion)
               Z<(X<Y) + Z&(X<Y) + (X<Y)<Z
```

The resulting list of AND-less cut sequences can then be converted into doublets, e.g.:

```
Y<(X<Z)    ➔   [Y<Z].[X<Z]
Y&(X<Z)    ➔   [Y&Z].[X<Z].[X<Y]
```

Although this will introduce more AND gates, the result will be a collection of cut sequences in BTF containing only either doublets or single basic events – there will be no 'mixed' cut sequences containing both, e.g. X.[Y<Z]. <u>All</u> temporal relations in the original expression will now be represented by doublets, even those temporal relations originally represented only implicitly by AND gates. This means that the minimisation algorithm can also be used to detect many possible Completion reductions by simply using the normal Absorption and Priority laws.

Take the example Completion problem given above: `X.Y.Z + X<Y + X&Y + Y<X`. We know that this reduces to just X.Y, and we can achieve this using doublets as follows:

1. First, the original expression is converted into HTF:

```
X.Y.Z + X<Y + X&Y + Y<X
```

2. Next, any AND gates are removed by replacing them (depth-first) with the Conjunctive Completion Law:

```
⇔ (X<Y + X&Y + Y<X).Z
+ X<Y + X&Y + Y<X
⇔ (X<Y + X&Y + Y<X)<Z
+ (X<Y + X&Y + Y<X)&Z
+ Z<(X<Y + X&Y + Y<X)
+ X<Y + X&Y + Y<X
```

3. This is then converted into HTF again:

```
((X<Y + X&Y) + Y<X)<Z ⇔ (X<Y)<Z + (X&Y)<Z + (Y<X)<Z
((X<Y + X&Y) + Y<X)&Z ⇔ Z&(X<Y + X&Y) . Z&(Y<X) +
                          Z&(Y<X) . Z|(X<Y + X&Y) +
                          Z&(X<Y + X&Y) . Z|(Y<X)
                        ⇔ Z&(X<Y) . Z&(X&Y) . Z&(Y<X)+
                          Z&(X<Y)|(X&Y) . Z&(Y<X)+
                          Z&(X&Y)|(X<Y) . Z&(Y<X)+
                          Z&(Y<X) . Z|(X<Y + X&Y)+
                          Z&(X<Y) . Z&(X&Y) . Z|(Y<X)+
                          Z&(X<Y)|(X&Y) . Z|(Y<X)+
                          Z&(X&Y)|(X<Y) . Z|(Y<X)
                        ⇔ Z&(X<Y) . Z&(Y<X)+
                          Z&(X&Y) . Z&(Y<X)+
                          Z&(Y<X) . Z|(X<Y + X&Y)+
                          Z&(X<Y) . Z|(Y<X)+
                          Z&(X&Y) . Z|(Y<X)
                        ⇔ Z&(Y<X) . Z +
                          Z&(X<Y) . Z +
                          Z&(X&Y) . Z
                        ⇔ Z&(Y<X) +
                          Z&(X<Y) +
                          Z&(X&Y)
                        ⇔ Z&(Y<X) +
                          Z&(X<Y) +
                          Z&(X&Y)
Z<((X<Y + X&Y) + Y<X) ⇔ Z|(X<Y + X&Y) . Z|(Y<X) .
                          (X<Y + X&Y + Y<X)
                       ⇔ Z|(X<Y + X&Y) . Z|(Y<X) . (X<Y) +
                          Z|(X<Y + X&Y) . Z|(Y<X) . (X&Y) +
                          Z|(X<Y + X&Y) . Z|(Y<X) . (Y<X)
                       ⇔ Z|(X<Y + X&Y) . Z . (X<Y) +
                          Z|(X<Y + X&Y) . Z . (X&Y) +
                          Z|(X<Y + X&Y) . Z<(Y<X)
                       ⇔ Z<(X<Y) +
                          Z<(X&Y) +
                          Z<(Y<X)
                       ⇔ Z<X<Y + X<Z<Y + Z&X<Y +
                          Z<(X&Y) +
                          Z<Y<X + Y<Z<X + Z&Y<X
```

4. And then this is converted into BTF:

(X<Y)<Z	➔	[X<Y] . [Y<Z] . [X<Z]
(X&Y)<Z	➔	[X&Y] . [Y<Z] . [X<Z]
(Y<X)<Z	➔	[Y<X] . [X<Z] . [Y<Z]
Z&(Y<X)	➔	[Z&X] . [Y<X] . [Y<Z]
Z&(X<Y)	➔	[Z&Y] . [X<Y] . [X<Z]
Z&(X&Y)	➔	[Z&Y] . [X&Y] . [X&Z]
Z<X<Y	➔	[Z<X] . [X<Y] . [Z<Y]
X<Z<Y	➔	[X<Z] . [X<Y] . [Z<Y]
Z&X<Y	➔	[Z&X] . [X<Y] . [Z<Y]
Z<(X&Y)	➔	[Z<X] . [X&Y] . [Z<Y]
Z<Y<X	➔	[Z<Y] . [Y<X] . [Z<X]
Y<Z<X	➔	[Y<Z] . [Y<X] . [Z<X]
Z&Y<X	➔	[Z&Y] . [Y<X] . [Z<X]
X<Y	➔	[X<Y]
X&Y	➔	[X&Y]
Y<X	➔	[Y<X]

5. Finally, we scan through the cut sequences, checking for any cases of Absorption. In this example, each of the longer order 2 or order 3 cut sequences contain one of the three order 1 cut sets at the end, leaving us with just those three:

```
[X<Y]
[X&Y]
[Y<X]
```

At this stage, we may choose to reapply the Completion Laws to simplify the doublets and obtain X.Y, though strictly speaking the cut sequences are already minimal now. Presenting the MCSQs in this form ensures that all temporal relations are immediately apparent, but does involve more cut sequences.

Note that for AND gates which contain only basic events (or that reduce to a form containing only basic events), particularly for AND gates with more than two events, it is possible to generate the cut sequences directly without applying the Completion Law by using the leaf nodes of a precedence tree instead. The AND provides us with a set of unordered events (in this case, X, Y, and Z). We also know that there will be thirteen different sequences of these three events (from the Fubini number for $n = 3$). These sequences can therefore be generated step by step by following the branches of the precedence tree for {X, Y, Z}. These are the 13 cut sequences given above for X.Y.Z.

To ensure that all of these sequences are represented, more comprehensive versions of the encapsulation laws are necessary; in this case:

```
X<(Y<Z) ⇔ [X<Z].[Y<Z]
```

becomes:

```
X<(Y<Z) ⇔ [X<Y].[Y<Z].[X<Z] +
          [Y<X].[Y<Z].[X<Z] +
          [X&Y].[Y<Z].[X<Z]
```

A list of these complete Encapsulation Laws can be found in **Appendix II: Boolean & Temporal Laws**.

However, the fundamental problem with the doublet-based approach to Completion reduction is that it is unable to 'reassemble' the resultant MCSQs. Although it is a simple matter to apply the Completion Law in reverse to X<Y + X&Y + Y<X, it is not so simple when more than two events are present.

4.4.5 Disadvantage of Doublets

The primary advantage of doublets – that they show *all* temporal relations – is also their primary disadvantage, because for any non-trivial temporal fault tree, a large number of doublets will be required. A temporal gate with n inputs (assuming the inputs are basic events) will result in n-1 doublets without the Law of Extension and $\frac{1}{2}n(n-1)$ doublets once Extension has been applied. Expanding AND gates to detect Completion reductions as described above means that even more doublets are required to represent all the possible cut sequences: the number of cut sequences produced is determined by the Fubini numbers and each cut sequence would require multiple doublets (for $n > 2$).

Furthermore, although the numerical ELRAFT-style approach described above makes comparing doublets very simple, it is still necessary to compare every doublet in a cut sequence against every other doublet. The result can be a very large number of comparisons that increases quickly with the number of doublets involved (for n doublets, $\frac{1}{2}n(n-1)$ comparisons may be needed in the worst case). This problem only grows worse when comparing entire cut sequences against each other; once again, in the worst case scenario, every cut sequence must be compared against every other cut sequence, which can mean checking every doublet and event in each pair of cut sequences.

However, the benefits of doublets outweigh this disadvantage: the comparisons would be necessary in any case, and the use of doublets helps to minimise the cost of performing these comparisons. Without doublets and the conversion of temporal expressions into BTF, to minimise the cut sequences would require many more laws that can be applied to larger numbers of events (not just the limited number of three event laws used to minimise doublets) – something which is simply not practical.

4.5 Temporal Qualitative Analysis: Euripides

"Whom the gods would destroy, they first make mad."
- Euripides

4.5.1 The Four Corners of Euripides

The first of the two temporal qualitative analysis algorithms in Pandora is called the Euripides Algorithm[20] and consists of four stages. The first stage is the Binarboreal stage, which converts any fault tree into a binary fault tree. This is the simplest of the four components and can be employed more than once during the overall process. It is described in section **4.5.2**. The second component is the Flattening stage, which is responsible for converting the tree into HTF. It is described in **4.5.3**. The third component is the Encapsulation stage, which is responsible for converting the HTF fault tree into doublets; it is covered in **4.5.4**. The fourth and most complex component is the Minimisation stage, which (as you might expect) is responsible for the minimisation of the doublets. It is explained more fully in section **4.5.5**. Finally, in section **4.5.6**, the current limitations of the Euripides algorithm are discussed.

4.5.2 The Binarboreal Stage

The Binarboreal stage is easily the simplest of the four stages of Euripides and was mostly covered earlier in the chapter. It consists of five laws, one for each of the gates, which are applied in a depth-first fashion:

The Binary Laws

```
A.B.C. ... .N ⇔ ((((A.B).C)...).N)
A+B+C+ ... +N ⇔ ((((A+B)+C)...)+N)
A<B<C< ... <N ⇔ ((((A<B)<C)...)<N)
A&B&C& ... &N ⇔ ((((A&B)&C)...)&N)
A|B|C| ... |N ⇔ A|(((B+C)...)+N)
```

These laws convert a gate with an arbitrary number of inputs greater than two into a nested series of the same type of gate, each with only two inputs. If a gate has only one or two inputs, it is not changed (a gate with only one input is treated as a simple intermediate event, i.e. one that imposes no constraints on its inputs).

[20] So named for the way Euripides's quotation accurately reflects the algorithm's effect on the mind of the author.

The process is very easy to follow, and consists of making a new gate of the same type, adding all but one child of the gate to it, and then assigning the new gate as a new child:

1. The process is depth-first, so first recurse for all children (this has no effect on basic events).
2. If the current gate has more than two children, first create a new gate of the same type (unless the gate is a POR, in which case an OR is created), otherwise finish.
3. If the current gate is not a POR: remove all but the last child from the current gate and add – in the same order – to the new gate. It is important that the *last* child, and not the *first* child, remains in the current gate, otherwise any PAND gates will be incorrect. Add the new gate as the *first* child of the current gate.
4. If the current gate *is* a POR: remove all but the *first* child from the current gate and add – in the same order – to a new OR gate. Add this new OR gate as the *second* child of the POR gate.
5. Repeat the process for the new gate.

Note the special handling for the POR gate; this is because it has a different Binary Law, can be seen from the list above. The Binarboreal stage is carried out first, before the Distributive Laws are to be applied, because this limits the number of Distributive Laws needed to just those with three events; however, it can also be carried out as part of later stages if needed.

4.5.3 The Flattening Stage

The Flattening stage (or 'Flattener') is more complicated than the Binarboreal stage, but still relatively simple to understand. This second stage consists of a repeated, depth-first application of Distributive and/or Associative Laws to flatten the fault tree into Hierarchical Temporal Form, i.e. one in which OR gates are highest, then AND gates, then POR, PAND and finally SAND gates. This algorithm sets a flag for each gate to indicate that it has already been flattened; this prevents it from being re-flattened later if the branch is shared or if the algorithm causes one of its parent gates to be flattened again.

It consists of two phases. In the first phase, the Distributive Laws are applied to rearrange the gates. The precise action taken depends on the gate in question, but in all cases begins the gate must be binary (and is converted if this is not the case). At most one law will be applied, as this will create a replacement gate which undergoes its own flattening.

OR Gates

Only the Associative Law is applied to the OR gate, because they have the lowest precedence (i.e. they come highest in the tree – they are "lightest" and rise to the top). If any of the OR gate's children are also OR gates, then their children are added to the current gate and the child OR gate is removed.

AND Gates

AND gates have second-lowest precedence; the temporal gates all have higher precedence. This means that AND gates will tend to "rise" above all temporal gates but not above OR gates. The only Distributive Laws applied to AND gates are the basic Boolean ones:

```
X.(Y+Z) ⇔ X.Y + X.Z
(Y+Z).X ⇔ X.Y + X.Z
```

The end result is that the AND gate is replaced by an OR gate which has two children, both AND gates. Both these new children undergo the Binarboreal process (if necessary) and are flattened once created.

POR Gates

POR gates have middle precedence; higher than the logical gates, but lower than the two conjunctive temporal gates. Ten Distributive Laws are applied to POR gates, four to let the logical gates rise above it, two to deal with POR gates having more POR gates as children, and four to let the other temporal gates sink below it.

```
X|(Y+Z) ⇔ X|Y . X|Z
(Y+Z)|X ⇔ Y|X + Z|X
X|(Y.Z) ⇔ X|Y + X|Z
(Y.Z)|X ⇔ Y|X . Z|X
X|(Y|Z) ⇔ X|Y + X.(Z<Y) + X.(Z&Y)
(Y|Z)|X ⇔ Y|Z . Y|X
X|(Y&Z) ⇔ X.(Y|Z) + X.(Z|Y) + X|Y + X|Z
(Y&Z)|X ⇔ Y|X . Z|X . Y&Z
X|(Y<Z) ⇔ X|Z + X|Y + X.(Z<Y) + X.(Y&Z)
(Y<Z)|X ⇔ Y<Z . Z|X
```

Again, all children are made binary and flattened once created.

PAND Gates

As with the POR gate, ten laws are applied to the PAND gate, depending on what its children are; four for the logical gates, two more for other PANDs, and four for the other temporal gates:

```
X<(Y+Z) ⇔ X|Z . X<Y + X|Y . X<Z
(Y+Z)<X ⇔ Y<X + Z<X
X<(Y.Z) ⇔ Y.(X<Z) + Z.(X<Y)
(Y.Z)<X ⇔ Y<X . Z<X
X<(Y|Z) ⇔ X|Y . Y|Z
(Y|Z)<X ⇔ Y<X . Y|Z
X<(Y<Z) ⇔ X<Z . Y<Z
(Y<Z)<X ⇔ Y<Z . Z<X
X<(Y&Z) ⇔ X<Y . X<Z . Y&Z
(Y&Z)<X ⇔ Z<X . Y<X . Y&Z
```

As before, all new children are re-flattened and if necessary converted to binary gates. This means that the newly created children can use their own rules, rather than having to do the work here in the PAND.

SAND Gates

Like the other two temporal gates, SAND gates have ten laws applied to them:

```
X & (Y+Z) ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y)
(Y+Z) & X ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y)
X & (Y.Z) ⇔ (X&Y).(Y&Z) + (X&Y).(Z<Y) + (X&Z).(Y<Z)
(Y.Z) & X ⇔ (Y<X).(Z&X) + (Z<X).(Y&X) + ((X&Y).(X&Z)).(Y&Z)
X & (Y|Z) ⇔ ((X&Y) . (Y|Z)) . (X|Z)
(Y|Z) & X ⇔ X&Y . (Y|Z)
X & (Y<Z) ⇔ (X&Z . Y<Z) . Y<X
(Y<Z) & X ⇔ (Y<Z . Z&X) . Y<X
X & (Y&Z) ⇔ ((X&Y) . (X&Z)) . (Y&Z)
(Y&Z) & X ⇔ ((X&Y) . (X&Z)) . (Y&Z)
```

In this case, there are some laws with more than one variation that can be used; the versions shown here are the ones chosen, because they result in children with the right precedence. For example, for X&(Y<Z):

```
X & (Y<Z) ⇔ X&Z . Y<Z . Y<X
  X & (Y<Z) ⇔ (Y<Z)&(Y<X)
```

The second version is not in precedence order (i.e. HTF), because the SAND is higher than the PANDs; therefore, the first is chosen, even though it results in three children rather than two (and will therefore need to be made binary).

Once the first phase of the Flattening stage is complete, and all gates have been shifted to their correct positions, the second phase begins: the Final Flattening. This phase, despite its magnificent-sounding name, is a depth-first traversal that applies only to AND and OR gates, and consists of an application of the Associative Law, so that any ANDs that are input to ANDs are merged and similarly any ORs that are inputs to ORs are merged, e.g. X + (Y+Z) ⇔ X + Y+ Z.

After the Flattener has completed its work, the fault tree should be in binary HTF and is ripe for encapsulation.

4.5.4 The Encapsulation Stage

The Encapsulation stage (or 'Doubletiser') is normally very simple, because most of the hard work has already been done during the Flattening stage. Although it is often possible to create doublets during the previous stage, sometimes a sub-tree needs flattening without being encapsulated, and in the case of temporal hierarchies (e.g. (X<Y)|Z), encapsulation must be done separately.

The laws necessary for encapsulation were presented earlier and are as follows:

```
X < (Y<Z) ⇔ [X<Z].[Y<Z]
X < (Y&Z) ⇔ [X<Y].[X<Z].[Y&Z]
X | (Y<Z) ⇔ [X|Z] + [X|Y] + X.[Z<Y] + X.[Y&Z]
X | (Y&Z) ⇔ X.[Y|Z] + X.[Z|Y] + [X|Y] + [X|Z]
X | (Y|Z) ⇔ [X|Y] + X.[Z<Y] + X.[Y&Z]
(Y<Z) < X ⇔ [Y<Z].[Z<X].[Y<X]
```

```
(Y&Z) < X ⇔ [Z<X].[Y<X].[Y&Z]
(Y&Z) & X ⇔ [X&Y].[Y&Z].[X&Z]
(Y<Z) | X ⇔ [Y<Z].[Z|X].[Y|X]
(Y&Z) | X ⇔ [Y|X].[Z|X].[Y&Z]
(Y|Z) | X ⇔ [Y|Z].[Y|X]
```

Because the tree should already be in binary HTF, these are the only laws required. If the top node is an OR, then we search its children; if a child is an AND gate, then we search the AND's children; either way, when a temporal gate is discovered, the appropriate laws are applied to convert it into doublets. The most complex case is where all three temporal gates are involved, e.g. (A<(B&C))|D. In this case, the deepest gate is dealt with first:

```
A<(B&C)     ⇔    [A<B].[A<C].[B&C]
```

and then each of these doublets is applied separately to the POR:

```
[A<B]|D     ⇔    [A<B].[B|D].[A|D]
[A<C]|D     ⇔    [A<C].[C|D].[A|D]
[B&C]|D     ⇔    [B&C].[B|D].[C|D]
```

yielding nine doublets (in this case, including a few duplicates):

```
                         (A<(B&C))|D ⇔
[A<B].[B|D].[A|D].[A<C].[C|D].[A|D].[B&C].[B|D].[C|D]
```

In cases where two temporal gates are involved, the laws can be used directly, and single temporal gates are converted directly to doublets.

4.5.5 The Minimisation Stage

The Minimiser is the most difficult stage. Like the Flattener, it consists of more than one phase. The first part is the intra-CSQ minimisation, which looks for possible reductions within a cut sequence and minimises it as far as possible. The second part, which is iterative, is the inter-CSQ minimisation part, which looks for possible reductions by comparing whole cut sequences. It repeats until it can find no more minimisations. The inter-CSQ part can change the contents of the CSQ so intra-CSQ minimisation is normally done both before and after; changes can occur as a result of using Priority or POR Transformation Laws or – ultimately – even Completion, which creates entirely new CSQs. As an example, while checking [X&Z].[X|Y].[Z|Y] + Y, intra-CSQ minimisation would initially take no action, but then inter-CSQ minimisation would convert the first CSQ to [X&Z].X.Z as a result of the POR Transformation Law, X|Y + Y ⇔ X+Y. This CSQ would then be reduced to just [X&Z] as a result of Absorption during the second application of intra-CSQ minimisation.

Intra-CSQ Minimisation

This consists of first applying the Laws of Extension where appropriate, then checking for violations of Simultaneity or Idempotence in SANDs, and then a doublet vs. doublet comparison. The application of the Laws of Extension is fairly straightforward: every doublet is checked against every other doublet in the cut sequence. If there is a common event in both doublets, then it is likely that a Law of Extension can be applied. The exact behaviour depends on the location of the events and the operators involved, as can be seen from the table below. The highlighted events are the common events in both doublets, and the new doublet produced is shown after the arrow. There are less entries for mixed POR/PANDs as these do not lead to additional temporal relations, e.g. [X<Y].[Z<Y] gives no relation for X and Z.

compare	**POR**	**PAND**	**SAND**
POR	[X|**Y**].[**Y**|Z]→[X|Z] [**X**|Y].[Z|**X**]→[Z|Y]	[X<**Y**].[**Y**|Z]→[X|Z] [**X**<Y].[Z|**X**]→[Z<Y]	[**X**&Y].[**X**|Z]→[Y|Z] [X&**Y**].[**Y**|**Z**]→[X|Z] [**X**&Y].[Z|**X**]→[Y|Z] [X&**Y**].[Z|**Y**]→[X|Z]
PAND	[X|**Y**].[**Y**<Z]→[X<Z] [**X**|Y].[Z<**X**]→[Z|Y]	[X<**Y**].[**Y**<Z]→[X<Z] [**X**<Y].[Z<**X**]→[Z<Y]	[**X**&Y].[**X**<Z]→[Y<Z] [X&**Y**].[**Y**<Z]→[X<Z] [**X**&Y].[Z<**X**]→[Z<Y] [X&**Y**].[Z<**Y**]→[Z<X]
SAND	[**X**|Y].[**X**&Z]→[Z|Y] [X|**Y**].[**Y**&Z]→[X<Z] [**X**|Y].[Z&**X**]→[Z|Y] [X|**Y**].[Z&**Y**]→[X<Z]	[**X**<Y].[**X**&Z]→[Z<Y] [X<**Y**].[**Y**&Z]→[X<Z] [**X**<Y].[Z&**X**]→[Z<Y] [X<**Y**].[Z&**Y**]→[X<Z]	[**X**&Y].[**X**&Z]→[Y&Z] [X&**Y**].[**Y**&Z]→[X&Z] [**X**&Y].[Z&**X**]→[Y&Z] [X&**Y**].[Z&**Y**]→[X&Z]

Table 5 – Application of the Extension Law on Doublets

Once the Law of Extension has been applied, the next step is to check for any violations of the Law of Simultaneity, or alternatively any places where Idempotence can be applied to a SAND. This is just a simple matter of checking each doublet in the cut sequence to see if it has the same event on both sides; if it does, and the doublet is a PAND or a POR, then we have a contradiction and can stop the minimisation since this cut sequence is impossible; if it is a SAND, then we can just replace it with the event on its own. In other words:

```
[X<X] ⇔ 0
[Y|Y] ⇔ 0
[Z&Z] ⇔ Z
```

As described in the last section, this is easy to detect because these doublets will all have temporal products of 0, since the prime numbers of each event in the doublet will be the same.

The final step in intra-CSQ minimisation is also the most complicated, but again consists of a number of different parts. This last step compares every event or doublet against every other event or doublet in the cut sequence. There are therefore three different possibilities:

Event vs. Event
Event vs. Doublet
Doublet vs. Doublet

Event vs. Event is easy to handle – if they are the same, then one is redundant and can be removed from the cut sequence according to the Idempotent law, i.e. X.X = X. Event vs. Doublet is easy too: if the doublet contains the event, remove the event – unless the doublet is a POR and the event occurs on the right hand side, in which case the POR Transformation Law applies.

Doublet vs. Doublet, however, is a little more complex. There are several forms of reduction possible here. First, the POR Transformation law is checked for; if the right-most event of a POR occurs in another doublet in the cut sequence (except as another right-most event of a POR), then it is replaced with an equivalent PAND doublet instead. Next, we look for redundant doublets and contradictions. If the temporal product of the two doublets is the same, then more than likely there is a contradiction or a redundancy; to be sure, we must check the operators:

Same temporal product	**POR**	**PAND**	**SAND**
POR	Idempotence: `[X|Y].[X|Y]` ⇔ `[X|Y]`	Priority: `[X<Y].[X|Y]` ⇔ `[X<Y]`	Contradiction: `[X&Y].[X|Y]` ⇔ `0`
PAND	Priority: `[X|Y].[X<Y]` ⇔ `[X<Y]`	Idempotence: `[X<Y].[X<Y]` ⇔ `[X<Y]`	Contradiction: `[X&Y].[X<Y]` ⇔ `0`
SAND	Contradiction: `[X|Y].[X&Y]` ⇔ `0`	Contradiction: `[X<Y].[X&Y]` ⇔ `0`	Idempotence: `[X&Y].[X&Y]` ⇔ `[X&Y]`

If the temporal product differs only in sign, then it is almost certainly a contradiction; the only case where it is not is if two SAND gates are involved. Any other combination of gates in which the same events are in different orders results in a contradiction.

Absolute temporal product is the same but the signs differ	**POR**	**PAND**	**SAND**
POR	Mutual Exclusion: `[X|Y].[Y|X]` ⇔ `0`	Mutual Exclusion: `[X<Y].[Y|X]` ⇔ `0`	Mutual Exclusion: `[X&Y].[Y|X]` ⇔ `0`
PAND	Mutual Exclusion: `[X|Y].[Y<X]` ⇔ `0`	Mutual Exclusion: `[X<Y].[Y<X]` ⇔ `0`	Mutual Exclusion: `[X&Y].[Y<X]` ⇔ `0`
SAND	Mutual Exclusion: `[X|Y].[Y&X]` ⇔ `0`	Mutual Exclusion: `[X<Y].[Y&X]` ⇔ `0`	Idempotence: `[X&Y].[Y&X]` ⇔ `[X&Y]`

If there is a contradiction in the cut sequence, then the whole cut sequence can be discarded.

Inter-CSQ Minimisation

Inter-CSQ Minimisation is much more complex than Intra-CSQ, and requires a great deal more checking before we can know for sure whether or not a CSQ is redundant or not. In this phase, every CSQ is compared with every other CSQ to see whether or not any reduction is possible.

As with Intra-CSQ minimisation, however, the exact behaviour depends on the cut sequences in question.

If neither cut sequence contains any doublets, then they are really only cut sets and can be reduced as such. This can be done in the ELRAFT style, e.g., if there are two cut sets with prime products CS1 and CS2 respectively:

If CS1 MOD CS2 = 0, CS1 is redundant.
If CS2 MOD CS1 = 0, CS2 is redundant.
If CS1 = CS2, they are identical and one can be removed.

The prime product is calculated simply by multiplying the prime identifiers of all constituent basic events together, e.g.:

```
X=2, Y=3, Z=5
X.Y.Z = 2 x 3 x 5 = 30
X.Y   = 2 x 3     = 6
```

30 MOD 6 = 0 so X.Y.Z contains X.Y

However, this is the simplest case. If one cut sequence contains doublets, we need some additional information. For a cut sequence containing no POR doublets, the prime products can be compared as normal, but if POR doublets are involved, then two numbers are necessary: a prime product with right-hand POR events (the 'full prime product') and a prime product without them (the 'true prime product'). For example, [X|Y].Z (using the above values) would have a normal prime product of 2 x 3 x 5, i.e. 30. However, the second prime product would be just 2 x 5 as the right hand side of the POR is omitted.

These numbers are both compared against the other cut sequence. Take the following two examples:

	#1 - Y	#2 - [X|Y].Z	#2 MOD #1
Full Prime Product	3	30	0
True Prime Product	3	10	1

In this case, the doublet CSQ contains Y on the right-hand side of the POR, as evidenced by the fact that the second product yields a remainder. In this case the POR Transformation is applicable (since the result of the first modulo is 0, the doublet CSQ contains all events in the non-temporal CSQ) and the result would be Y + X.Z.

	#1 - X	#2 - [X|Y].Z	#2 MOD #1
Full Prime Product	2	30	0
True Prime Product	2	10	0

In this case, the modulo comparison succeeds both times, meaning that the doublet CSQ contains X properly, i.e. not on the right-hand side of a POR. In this case, the doublet CSQ is redundant and can be removed.

It is also possible for the non-doublet CSQ to be redundant if its prime product is larger and the modulo results in 0:

	#1 –X.Y.Z.A.B	#2 - [X|Y].Z	#1 MOD #2
Full Prime Product	2x3x5x7x11 = 2310	30	0
True Prime Product	2x3x5x7x11 = 2310	10	0

In this case, the non-temporal CSQ is removed. However, if the full and true modulos differ:

	#1 –X.Z.A.B	#2 - [X|Y].Z	#1 MOD #2
Full Prime Product	2x5x7x11 = 770	30	20
True Prime Product	2x5x7x11 = 770	10	0

Here neither CSQ is redundant because the modulo of the full primes is not 0.

In cases where the primes are the *same*:

	#1 –X.Y.Z	#2 - [X<Y].Z	#2 MOD #1
Full Prime Product	30	30	0
True Prime Product	30	30	0

If both full and true primes are the same as the non-temporal prime product, then the temporal CSQ is redundant according to the Law of Priority (X.Y + X<Y = X.Y).

	#1 –X.Y.Z	#2 - [X|Y].Z	#2 MOD #1
Full Prime Product	30	30	0
True Prime Product	30	10	0

However, if a POR is involved, the Reductive Completion Law may apply, and in this case the result is simply X.Z.

The most complicated case, however, is comparing temporal CSQs against other temporal CSQs. The table below illustrates what type of reduction is indicated by the comparison of prime products. TPP means true prime product and FPP means full prime product.

Comparison	**Reduction?**
FPP1 % FPP2 = 0, TPP1 %TPP2 = 0	CSQ1 may contain CSQ2, further checking required.
FPP1 % FPP2 = 0, TPP1 %TPP2 ≠ 0	CSQ1 may contain CSQ2, further checking required.
FPP2 % FPP1 = 0, TPP2 %TPP1 = 0	CSQ2 may contain CSQ1, further checking required.
FPP2 % FPP1 = 0, TPP2 %TPP1 ≠ 0	CSQ2 may contain CSQ1, further checking required.
FPP1 = FPP2, TPP1 = TPP2	Both CSQs contain same events – further checking required.
FPP1 = FPP2, TPP1 ≠ TPP2	Both CSQs contain the same events, but some are in PORs; further checking required.
FPP1 % FPP2 ≠ 0, FPP2 % FPP1 ≠ 0	No further checking necessary.

Table 6 – Results of comparisons between temporal CSQs

In this scenario, the modulo check can only indicate whether further checking is necessary or not. If the comparison of the full primes is negative, then no further checking is necessary, e.g. [X|Y] and [X|Z] have FPPs 6 and 10 respectively. In other cases, more detailed checks are required to ensure that doublets are in the same order. To be sure, we need to check every doublet or event in CSQ1 against every doublet or event in CSQ2. For each doublet comparison, we check their temporal products; if these always match, and the operators always match, then reduction is possible. For example:

[X<Y].[Y<Z].[X<Z].[X|B]   against   [X<Y].[X|B]

1. Check [X<Y] and [X<Y]: -6 = -6, < = < ➔ match
2. Check [Y<Z] and [X|B]: -15 ≠ -22 ➔ no match
3. Check [X<Z] and [X|B]: -10 ≠ -22 ➔ no match
4. Check [X|B] and [X|B]: -22 = -22, | = | ➔ match
5. All doublets in the smaller CSQ match: CSQ1 contains CSQ2, so CSQ1 is redundant.

There are some complications here, however. Firstly, the CSQs may contain some unaffiliated events; the events in the smaller CSQ must match the larger CSQ, either in a doublet or as another unaffiliated event. Also, when comparing two doublets, the operator makes a difference: if the two doublets have the same temporal product, but the one from the smaller CSQ is a PAND while the other is a POR, then it still counts as a match (but not vice versa); if the two doublets are both SAND but the temporal products differ only in sign, this still counts as a match.

As a more complex example:

[X&Y].[Y<Z].[X<Z].[X<B]   against   [Y&X].[X|B]

1. Check [X&Y] and [Y&X]: ABS(-6) = ABS(6), & = & ➔ match
2. Check [Y<Z] and [X|B]: -15 ≠ -22 ➔ no match
3. Check [X<Z] and [X|B]: -10 ≠ -22 ➔ no match
4. Check [X<B] and [X|B]: -22 = -22, < ≠ | but PAND subsumed by POR ➔ match
5. All doublets in the smaller CSQ match: CSQ1 contains CSQ2, so CSQ1 is redundant.

Obviously, this temporal CSQ vs temporal CSQ checking is more time consuming, but it is necessary to ensure that the Absorption Law does apply correctly. If the doublets contained events in different orders, e.g. if CSQ2 in the last example contained [B|X] instead, then Absorption would not be possible.

Note the benefit of using the temporal products and prime products in this approach, however; otherwise, a complete check of every doublet's events, order, and operator versus every other doublet would be necessary all the time.

4.5.6 Summary and limitations of the Euripides Algorithm

A concise summary of the Euripides algorithm is as follows:

1	Binarboreal Stage	*Conversion of fault tree into binary form.*
2	Flattening Stage	*Flattening of binary fault tree into HTF.*
3	Encapsulation Stage	*Encapsulation and conversion of temporal gates into doublets.*
4	Minimisation Stage	*Minimisation of cut sequences to obtain MCSQs.*
4.1	Intra-CSQ Minimisation	*Reduction within cut sequences.*
4.2	Inter-CSQ Minimisation	*Reduction between cut sequences*.

The Euripides algorithm can cope with the majority of possible reductions and contradictions that you might expect to find in a temporal fault tree. However, as explained earlier, there is also the possibility of Completion-based reductions, and these are much harder to detect. Section **4.4.4** described one possible way of improving the detection of Completion possibilities by removing all AND gates from the fault tree and converting them into PANDs and SANDs according to the Conjunctive Completion Law. The problem with this approach (other than the complexity) is that it results in a large number of MCSQs, almost all of which will be temporal, because it cannot recombine the results according to the Completion laws. Thus, although Euripides can remove all contradictions and most other types of redundancies, it still struggles to produce results in the most concise form possible.

An alternative algorithm is presented next in section **4.6**: Archimedes. Archimedes is able to detect and solve the subtle Completion-based reductions but is even less efficient from a performance standpoint than Euripides. As will be explained later, they are best used in concert, as each is able to complement the strengths and weaknesses of the other.

Euripides suffers from other shortcomings as well, however. The ELRAFT-style prime number approach, with its temporal products and full/true prime products, offers great advantages during comparison of doublets and especially during comparison of CSQs, but it has a number of disadvantages too. Firstly, it requires a set of prime numbers that can be assigned to all basic events; if the number of basic events is very large, then a large amount of prime numbers will be required (though the penalty for this can be offset by pre-generating a sufficiently large set of prime numbers). Secondly, for large CSQs containing many doublets or events, the prime product can be excessively large, sometimes exceeding the capacity of common computer numeric representations (both normal limits for integers and accuracy limits for floating point representations). In these cases, the prime numbers would require the use arbitrary precision numbers, but doing so would have severe performance implications.

If prime numbers are not used at all, then the amount of checking required for minimisation is even greater. Although doublets make it easier to detect possible reductions by representing all temporal relations, they also increase the size of the cut sequences substantially and (without primes) require full combinatorial checking. This has problematic performance implications, particularly during inter-CSQ minimisation.

However, all of these issues must be balanced against the capabilities Euripides offers: without the doublets and BTF, checking for potential reductions and minimisations would require a virtually infinite number of temporal laws covering innumerable possibilities. By reducing the scope of the problem via encapsulating and representing all temporal relations as doublets, only a finite number of laws are required (though the number of applications of these laws increases as a result). These laws make it possible to detect all contradictions and most common types of minimisation.

4.6 Temporal Qualitative Analysis: Archimedes

"Eureka!"
- Archimedes, in his bath

4.6.1 Introduction to Archimedes

Archimedes[21] is the second Temporal Qualitative Analysis algorithm in Pandora. It serves as both an alternative and a complement to Euripides. Whereas Euripides is primarily a deductive technique based on the application of temporal and logical laws, Archimedes is an inductive technique that enumerates all possibilities by converting the fault tree into an alternative form (thus analogous to the BDD-style approach rather than the MOCUS-style approach).

The starting point of Archimedes is the **precedence tree**, which was described in Chapter 3. A precedence tree is a representation of a branching timeline that shows all possible sequences for a set of basic events. For example, the precedence tree for three events X, Y, and Z is shown in Figure 39. The root of a precedence tree is the situation where no events have occurred yet; its children represent the various possibilities of what might happen next, e.g. in this case, the occurrence of X, Y, or Z, the occurrence of two events simultaneously (represented as a simultaneous set {X, Y}), or the occurrence of all three events. Any events that have not yet occurred are represented as a POR set. Thus {X}|{Y, Z} represents the situation where X has occurred and Y and Z have yet to occur. The tree continues to branch until all events have occurred, so {X}|{Y, Z} will branch three times: {X}{Y}|{Z}, {X}{Z}|{Y}, and {X}{Y, Z}. The latter is a leaf node because all three events have occurred, whilst the first two each have an additional child – {X}{Y}{Z} and {X}{Z}{Y} respectively. At each stage, the events gain a sequence value indicating which simultaneous set they occur in; thus for {X}{Y, Z}, *S*(X) = 1 while *S*(Y) and *S*(Z) = 2.

An enumeration of all nodes in a precedence tree gives all possible cut sequences, while an enumeration of all leaf nodes provides all cut sequences in which every event occurs. These nodes are equivalent to the rows of a TTT and a CTTT respectively, and so the number of nodes is given by the Fubini numbers (i.e. for *n* events, there are 2*Fubini(*n*) nodes and Fubini(*n*) leaf nodes).

Precedence trees are useful chiefly because they provide a methodical process for generating all possible sequences for a set of events. For this reason, they are often used to generate TTTs.

[21] The idea came to me in the shower, much like Archimedes.

However, in Archimedes, their importance lies in the fact they can serve as a starting point for a **Dependency Tree**. Whereas a precedence tree is a branching timeline terminating in the cut sequences where all events occur, a dependency tree is a logical tree designed to represent the Completion Laws. Its leaf nodes are the nodes of the precedence tree for the same set of events (known as the **basic temporal nodes**), and it has one top node for each of those events (known as the **singleton nodes**).

The purpose of the dependency tree is to represent all possible cut sequences for a given set of events, but whilst a precedence tree shows all possible sequences of all events in a set, a dependency tree also shows all possible sequences of every possible subset of those events and also shows all possible conjunctions of those events (and subsets of those events). It is therefore considerably larger than the precedence tree on which it is based. The children for a given node are provided by applying the Conjunctive Completion Law (for conjunctions) and/or the Reductive Completion Law (for all nodes). It is possible for nodes – particularly the basic temporal nodes – to be shared by multiple branches.

The key to understanding dependency trees is that if all of the children of a given node are true, then that node is true and the children are redundant. Conversely, if a given node is true, then all of its descendants are also true (and redundant). Thus each node is 'dependent' on its children.

It is easier to understand a dependency tree with an example. Consider the dependency tree for two events {X, Y} in Figure 40:

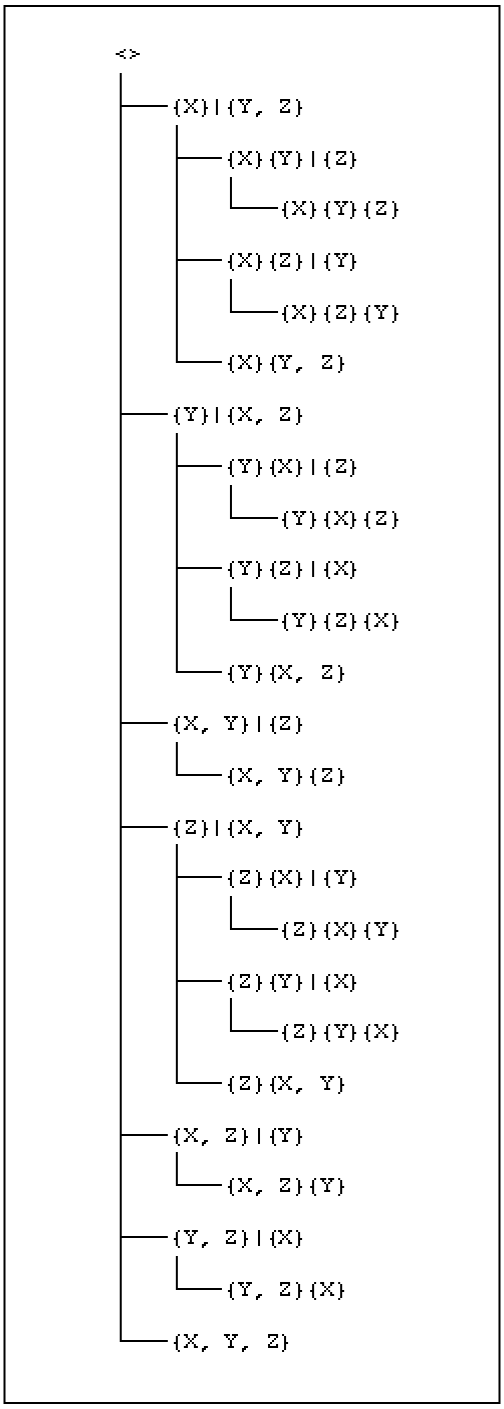


Figure 39 – Precedence Tree for {X,Y,Z}

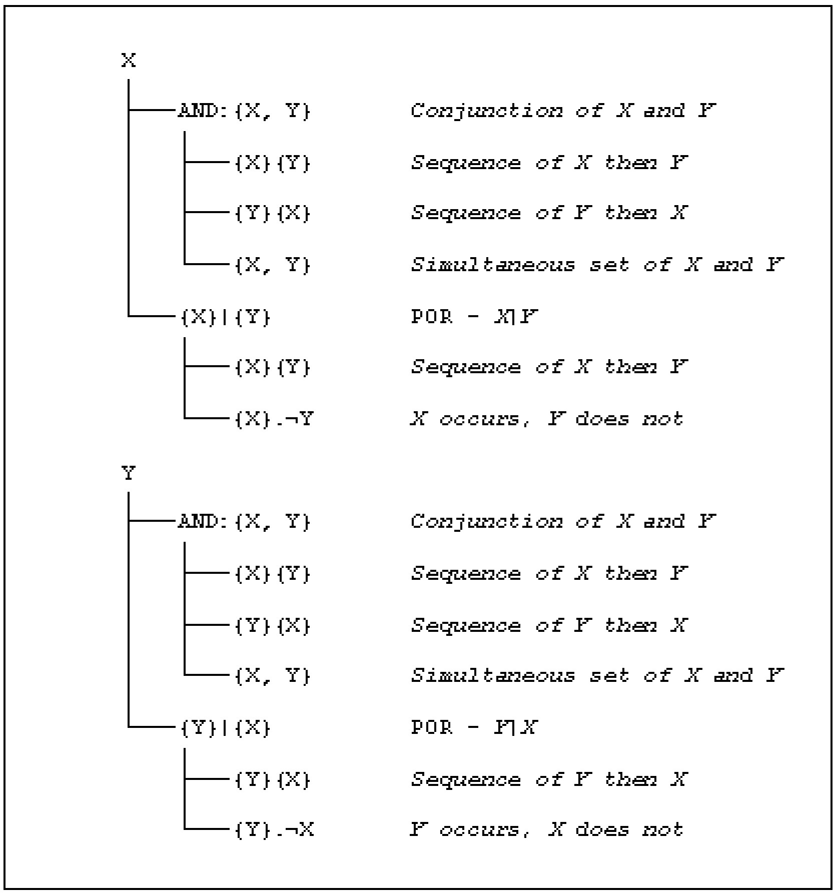


Figure 40 – Dependency Tree for {X, Y}

Although the trees for X and Y seem to be separate, they are in fact connected, as the nodes are shared. The top nodes – the singleton nodes – represent the occurrence of a single event. These are expanded using the second version of the Reductive Completion Law (RCL), i.e. X ⇔ X.Y + X|Y. If there were three events, then each singleton would have four children, e.g. for {X, Y, Z}, X would have the children X.Y, X.Z, X|Y, and X|Z. In this tree, conjunctions are then expanded according to the Conjunctive Completion Law (CCL), whilst PORs are expanded explicitly into a sequence – e.g. X<Y – and a negation – e.g. X.¬Y. The reason for this use of the dangerous NOT gate will be explained shortly.

In larger dependency trees, conjunctions and PORs are expanded further. In general, there are five types of node in a dependency tree:

- Singletons
- Pure Conjunctions
- Hybrid Conjunctions
- Partial Temporal Nodes
- Basic Temporal Nodes

Singletons, as explained above, represent the occurrence of a single event. These are equivalent to a cut set/CSQ containing one event. Pure conjunctions contain a number of basic events but no temporal relations; these are equivalent to standard cut sets. Hybrid conjunctions contain one or more unaffiliated events in addition to some temporal relations and are equivalent to a normal CSQ with unaffiliated events in it. Partial and basic temporal nodes are equivalent to normal CSQs with no unaffiliated events; partial temporal nodes contain only a subset of the total number of events, whereas basic temporal nodes – which always form the leaves of a dependency tree and which are obtained from the equivalent precedence tree – use all the events.

4.6.2 Evaluating a Dependency Tree

Dependency trees are used by evaluating them for a given expression. All nodes which are true for that expression are flagged, and these flagged nodes can then be collected to obtain the results – a fully reduced temporal expression. The dependency tree must contain each event that occurs in the expression.

The first step in this process is to evaluate the expression for each of the basic temporal nodes (BTNs). Because the BTNs are obtained from the equivalent precedence tree, they are guaranteed to represent every possible sequence of the events in question. Therefore, they also represent *every unique combination of sequence values*. This is evident from the fact that the nodes of a precedence tree correspond to the rows of a TTT. By using these sequence values, any expression can be evaluated to determine its own sequence value, because every gate has a function that returns its sequence value for given inputs (e.g. an AND gate returns the maximum of its inputs, as long as all inputs are greater than 0). This is effectively the same process as building a TTT for the expression.

The difference comes from the way these values are used in the tree. For the purposes of the dependency tree, the actual sequence value is irrelevant and only the truth value is required. For each basic temporal node, if the expression is true (i.e. non-zero), then a flag is set, and if the expression is false, then the flag is unset.

At this point it is necessary to explain the use of the NOT gates in the basic temporal nodes. Because each BTN must represent a unique combination of sequence values, equivalent to a single row of a TTT, it is necessary to differentiate between a POR that is true because its events occurred in a given sequence and a POR that is true because only its priority event occurred. For example, X|Y is true for two rows of the TTT for X and Y: the row where X = 1 and Y = 2, and the row where X = 1 and Y = 0. For a dependency tree to function, its BTNs must be true in exactly one case, so the POR nodes of the precedence tree (like X|Y) are treated as if the non-priority events (i.e. the right-most events) do not occur. The other case is represented already by the PAND. Thus for the purposes of the leaf nodes of the dependency tree[22], X|Y means X = 1 and Y = 0 and X<Y means X = 1 and Y = 2. These NOT-based basic temporal nodes do not appear in the final results; instead they are converted back to PORs[23].

Then a depth-first traversal of the dependency tree is performed. At each step, the number of child nodes with the truth flag set is counted; if *all* children of a node have their flag set, then the parent node has its flag set too. For example, if all of X<Y, X&Y, and Y<X are true, then X.Y is also true. The truth value thus propagates up the dependency tree until it either reaches the singletons or until not all children are true.

At this point we can collect the results, either as part of the same traversal (remembering, for every false node, every child node with its flag set) or by performing a second depth-first traversal, by collecting the top-most flagged nodes. This process is known as *capturing the flags*. When we select a node, we flag its children (recursively) as being redundant. If nodes are properly shared then there should be no duplicates, otherwise any duplicates will have to be separately removed.

As an example of this process, consider the expression X|Y + X&Y + Y<X – the constituents of the RCL. First, we check the BTNs:

[22] It is important to note that PORs may occur elsewhere in a dependency tree besides the leaf nodes, and in those cases they behave normally and do not involve NOT gates.

[23] This is not strictly necessary because in this limited situation the NOT does not introduce non-coherence. However, a NOT can never occur in a dependency tree without its corresponding PAND (e.g. X.¬Y cannot occur unless X<Y also occurs) because only a NOT-based expression could cause this (e.g. X.¬Y as an input). Thus it makes sense to combine them both as a POR instead.

X	**Y**	**BTN**	**X|Y + X&Y + Y<X**	**Flag**
0	0	¬X.¬Y	0	False
0	1	{Y}.¬X	0	False
1	0	{X}.¬Y	1	True
1	1	{X, Y}	1	True
1	2	{X}{Y}	1	True
2	1	{Y}{X}	2	True

As can be seen, four of the BTNs are true (and the situation where no events occur is ignored). Of the three second-level nodes in the {X, Y} dependency tree, two are true:

- AND:{X, Y} - all three children are true: {X}{Y}, {Y}{X}, and {X, Y}
- POR:{X}|{Y} - both children are true: {X}{Y} and {X}.¬Y
- POR:{Y}|{X} - only one child is true: {Y}{X}

Of the singletons, both children of {X} are true, namely AND:{X, Y} and POR:{X}|{Y}. Thus {X} is true also and has its flag set. The status of the dependency tree can be seen better in Figure 41:

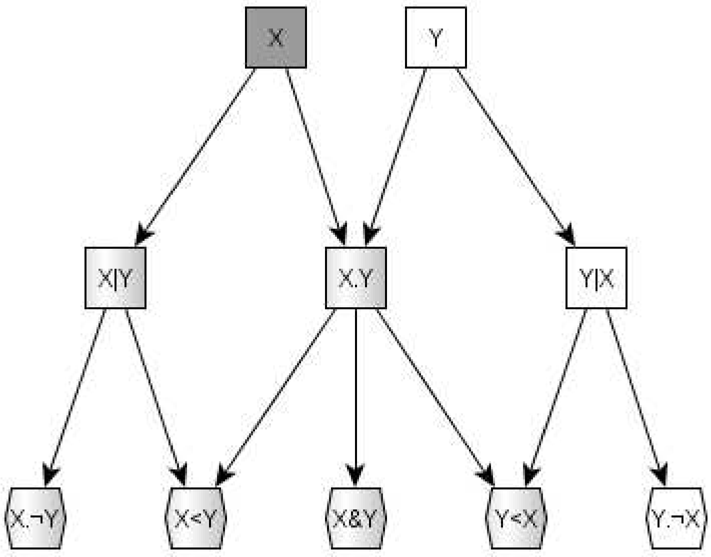


Figure 41 – Dependency tree for {X, Y} with flags set

The grey node (i.e. {X}) has its flag set, whilst the partially shaded nodes have flags set but are also redundant (because they are descendants of {X}). The hexagonal shaped nodes are BTNs.

A depth-first traversal immediately gives us {X} as a true node, so we flag all of its descendants as redundant. Moving to {Y}, which is not true, we move down to its children, AND:{X, Y} (which has been flagged as redundant) and POR:{Y}|{X} (which was not true). Moving down POR:{Y}|{X} reveals one true child ({Y}{X}) but that is also set as redundant. Thus only one node is selected for the results list: {X}. This also gives us the fully reduced result.

Note that there is an implicit disjunction between all nodes in the dependency tree, as each represents a different possible cut sequence, so if multiple nodes are selected, they are separated by ORs. For example, if we were to feed the expression X|Y + X&Y + Y|X into the dependency tree, all nodes would be true, including both singletons. Thus the result would be X + Y (which is the result of the Disjunctive Completion Law).

4.6.3 Creating Dependency Trees

Unfortunately, creating a dependency tree is considerably more complex than evaluating one. Dependency trees can be created in one of two ways. The crudest way (a kind of 'brute force' method) is to generate every possible node and then go through and assign children by evaluating each one as an expression and comparing it against every other possible node; all true nodes are added as children. Although this approach is guaranteed to represent every possible dependency, it also leads to larger than necessary dependency trees as nodes are added repeatedly at every level of the hierarchy. For example, in the {X,Y} dependency tree, rather than having just two nodes, each singleton would have six nodes, e.g.:

```
X
|
+---AND:{X, Y}          Conjunction of X and Y
|
+-- {X}{Y}              X before Y
|
+---{Y}{X}              Y before X
|
+---{X, Y}              X and Y occur simultaneously
|
+---{X}|{Y}             X occurs before Y or Y does not occur
|
+---{X}.¬Y              X occurs and Y does not occur
```

The four BTNs here would also be repeated under the AND and the POR too, leading to unnecessary duplication. This simplistic approach is also massively inefficient as every single possible node must be evaluated for every possible sequence (i.e. twice the Fubini number). For x nodes and n events, the worst case is $2x^2$ x Fubini(n) – a very large number for large x and n.

A far superior approach is to build the tree top-down by applying temporal laws at each stage as necessary. As already mentioned, this process begins with the singletons by applying the Reductive Completion Law: for every other event in the tree set, a conjunction and a disjunction are created, thus for {X, Y}, the singleton {X} has two children AND:{X, Y} and AND:{X}|{Y}. For a tree set of n nodes, there will be $2(n – 1)$ children for the singletons; half of these will be order 2 pure conjunctions and half order 2 PORs.

Pure conjunctions are expanded according to both the Conjunctive Completion Law and the Reductive Completion Law. The RCL part is easy and follows the same procedure used for the singletons; namely, for every remaining event not part of the conjunction, add a new pure conjunction child and a new POR child. For example, the RCL children for AND:{X, Y} out of a total set of {X, Y, Z} would be AND:{X, Y, Z} and AND:{X, Y}|{Z}. The CCL children are, however, somewhat more complex. Although generating the children for an order 2 conjunction is simply an application of the basic CCL, for higher orders, the process is more complex.

To generate the CCL contributors for a pure conjunction of size n, we need to represent every possible sequence of all n events that can lead to that conjunction. Although this could be done by generating the precedence tree for those n events and taking the leaf nodes, this would skip any intermediate hybrid conjunctions, e.g. X.(Y<Z) for AND:{X,Y,Z}. Generating hybrids can be done mechanically by taking each of the n events individually and inserting it in every possible position, e.g. for AND:{X, Y, Z}, using Z as the first 'temporal' event:

Y.(Z<X)	X.(Z<Y)
Y.(Z&X)	X.(Z&Y)
Y.(X<Z)	X.(Y<Z)

This would then be repeated with X and then Y as the 'temporal' event. Fortunately there is some overlap, e.g. when using X as the 'temporal' event, (Z<X).Y for instance has already been generated. Hybrids are further expanded until all non-temporal events are removed. The same sort of process is used if more than one unaffiliated event remains, taking one unaffiliated event at a time and inserting it into every possible position, e.g. for A.B.(C<D):

A.(B<C<D)	B.(A<C<D)
A.(B&C<D)	B.(A&C<D)
A.(C<B<D)	B.(C<A<D)
A.(C<B&D)	B.(C<A&D)
A.(C<D<B)	B.(C<D<A)

When only one unaffiliated event remains, the same process is applied, but this time the result is a set of partial or basic temporal nodes; for example, for X.(Y<Z) out of {X, Y, Z}, the five BTNs are:

X<Y<Z
X&Y<Z
Y<X<Z
Y<X&Z
Y<Z<X

But if the tree set was {W, X, Y, Z} these would only be partial temporal nodes.

Note that every hybrid contains only one temporal sequence; while it is technically possible for multiple temporal sequences to exist in the same hybrid conjunction, e.g. (A<B).(C<D), the single sequence form simplifies the results by ensuring that only one set of temporal relations is given at a time; multiple sequences can be confusing because the temporal relations between the individual events is not always clear, e.g. there is no relation between B and C in (A<B).(C<D), unlike a simple sequence such as B<C<D.

Hybrids are also expanded according to the RCL if any unused events remain. For example, if we have A.(B<C) and the event D remains unused, we can also create two additional children:

A.D.(B<C)
A.(B<C|D)

Hybrid PORs like this are treated as hybrid conjunctions during CCL expansion. Pure PORs like those generated as children of the singletons are expanded according to the RCL too, e.g. the children of {X}|{Y} out of {X, Y, Z} are:

Z.(X|Y)
X|Y|Z

The former is treated as a hybrid conjunction while the latter is either a BTN or is expanded further according to the RCL. Any other partial temporal nodes are expanded according to the RCL into a hybrid conjunction and a POR form, e.g. for (A<B) out of {A, B, C}:

C.(A<B)
A<B|C

Thus, by following these processes, all types of nodes – pure and hybrid conjunctions, partial and basic temporal nodes – can be generated methodically. Ultimately, this process always culminates in the BTNs, which should be shared and stored separately in a list. This is to facilitate the evaluation processes, where the expression has to be evaluated for every BTN. The progression of nodes is as follows:

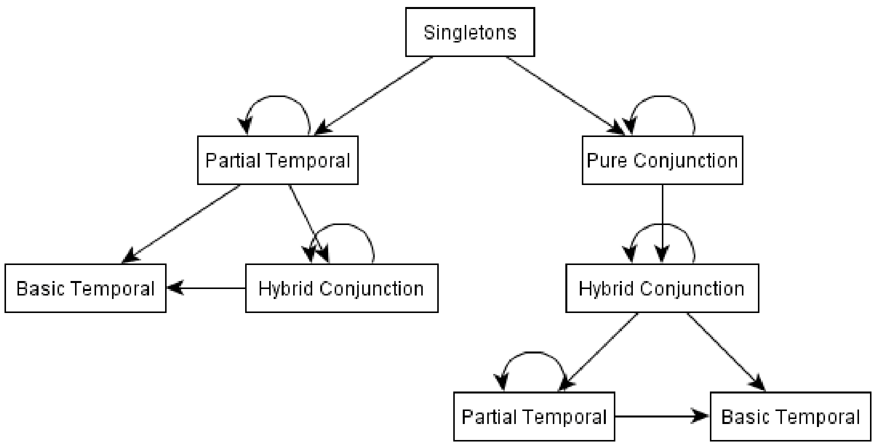


Figure 42 – Progression of nodes when creating Dependency trees

4.6.4 Summary and limitations of the Archimedes algorithm

The Archimedes algorithm can be concisely summarised as follows:

1 Generate a precedence tree for the events in the fault tree expression.
2 Using the nodes of the precedence tree as the BTNs, generate a dependency tree.
2.1 Starting with the singletons, generate two children (one POR, one AND) per additional event.
2.2 For each POR, generate a larger POR and a hybrid as per RCL.
2.3 For each hybrid, generate children according to CCL and, if any unused events remain, as per RCL.
2.4 For each pure conjunction, generate hybrid children according to CCL and larger pure conjunctions and PORs as per RCL (if unused events remain).
2.5 For each partial temporal node, apply RCL to obtain hybrid conjunction and POR.

2.6 Repeat until BTNs are reached.

3 Evaluate expression for each BTN, setting truth flag if the sequence value of the expression is non-zero.

4 Perform a depth-first traversal of the tree; if all children of a node are flagged, flag the node and set children (recursively) as redundant.

5 Perform a depth-first traversal to find top-most flagged (but non-redundant) nodes.

5.1 Remove duplicates if necessary.

The Archimedes algorithm is capable of producing exactly minimised cut sequences, including application of the Completion Law to reduce the total number of MCSQs where necessary. If the tree is created correctly using shared nodes, then no duplicates should be added to the results and thus no explicit law-based reduction (as in Euripides) is necessary[24].

However, the disadvantage of Archimedes' inductive approach is massive computational expense. Because a dependency tree represents every (or almost every) possible cut sequence, including subsets of the original set of events, it is inevitably very large, especially for large numbers of events. The number of nodes created is greater than the double Fubini number required to generate a TTT or precedence tree. In addition, evaluation of the tree uses TTT-style sequence values, which can be time consuming for large expressions (such as those used to represent large fault trees). Although Archimedes is still very fast for small numbers of events (e.g. 3 or 4), for larger numbers of events, the Fubini numbers involved rapidly reduce performance to unacceptable levels.

Despite the performance problems, Archimedes overcomes the limitations inherent in the Euripides algorithm: namely, its inability to handle Completion reduction satisfactorily. Furthermore, most of its conceptual complexity is in the construction of the dependency tree and, unlike Euripides, evaluation is merely a relatively simple tree traversal.

There are ways to overcome the limitations of both algorithms, however – in particular, they can be used together to maximise their strengths and minimise their weaknesses.

[24] The exception to this is the case where both POR and PAND BTNs are flagged, e.g. {X}|{Y} and {X}{Y}. In this case the PAND BTN can be discarded, but this is easily accomplished by linking these types of BTNs together – if {X}|{Y} is flagged then {X}{Y} is redundant. Note that this can only occur with BTNs.

4.7 Archipides/Euripemedes – Using both algorithms together

Euripides and Archimedes are two very different methods of solving the same basic problem: the reduction of Pandora fault trees to obtain minimal cut sequences. Euripides is a law-based deductive method that manipulates the expressions that represent the fault tree to remove contradictions and redundancies, whilst Archimedes is a tree-based inductive method that evaluates all possible cut sequences and ignores those that are redundant. Although both are capable of producing minimal cut sequences (with certain caveats for Euripides), both also suffer from performance issues and stumble at certain times.

Fortunately, when used together in concert, many of their disadvantages can be alleviated to a degree.

4.7.1 Obtaining the initial cut sequences with Euripides

Euripides is very effective at obtaining cut sequences. By converting a fault tree into Binary Hierarchical Temporal Form using a small number of Boolean and temporal laws, it effectively produces the complete set of cut sequences represented by the fault tree. By then creating doublets and checking for contradictions and absorption reductions, it can also reduce the number of cut sequences quite dramatically. Euripides's main problems stem from the difficulties of inter-CSQ minimisation, particularly Completion-based minimisation.

By creating the set of cut sequences, removing any contradictions and redundant doublets/events from within the cut sequences, and performing a simple check for absorption redundancies amongst the cut sequences, Euripides can relatively simply produce a set of minimised or partially minimised cut sequences. It is not necessary to check for Completion-based reduction at this point as the main goal is to reduce the size of the cut sequences and number of events in them.

The base temporal form expression representing the disjunction of those MCSQs can then be used as the input expression to Archimedes.

4.7.2 Analysing the cut sequences separately with Archimedes

Archimedes' main problem is that its performance decreases dramatically with each extra event. A tree for 3 or 4 events is much, *much* smaller than a tree for 6 or 8 events, as the Fubini numbers indicate. Therefore, it is imperative to limit the number of events needed to create the dependency trees. This can be achieved by examining the MCSQs produced by Euripides and grouping those that share events together. Each group is then analysed separately. Euripides's prime product technique also makes it very easy to check to see which events are present in which other cut sequences.

For example, consider the following MCSQs:

```
X<Y + X&Y + Y<X + A|B
```

The first three all share two events, X and Y. The fourth uses two more events, A and B, which are not used elsewhere. Thus the first three CSQs form one group with the event set {X, Y} and the other forms a second group (albeit of one CSQ) with the event set {A, B}. Archimedes can then be employed twice: first with a dependency tree for {X, Y} and then with a dependency tree for {A, B}. The total nodes created for both trees is still far less than the number of nodes necessary to create a single dependency tree for the combined event set {X, Y, A, B}. The Fubini number for $n = 2$ is just 4, whilst the Fubini number for $n = 4$ is 75.

Once Archimedes has produced its results for each group of MCSQs, the results can simply be combined.

4.7.3 Modularisation

The principle involved in combining Euripides and Archimedes is similar to the idea inherent in the modularisation algorithms used in standard FTA and in temporal FTA approaches like the DFT methodology. By limiting the scope of the problem, we make it easier for the algorithms to solve it. Euripides relies on Archimedes' superior capabilities for detecting and reducing instances of Completion whilst Archimedes relies on Euripides to reduce the problem space to a manageable size.

Modularisation can, however, be employed directly in Pandora in a similar way to how it is used in DFTs. In the Dynamic Fault Tree approach, modularisation is used to detect modules that contain temporal gates and those that do not (Gulati & Dugan, 1997). Temporal modules are then subjected to Markov analysis while static modules are analysed by traditional FTA methods (in DFTs, the BDD approach is normally used).

The linear time modularisation algorithm (LTMA) of Dutuit & Rauzy (1996) described in section **2.2.4** can be used for the same purpose in Pandora. The LTMA does not distinguish between different types of gates, only whether or not they have been visited yet. Thus it can still be used to determine the modules of Pandora fault trees. Even before the fault tree is converted into HTF by Euripides, independent sub-trees can be identified and it is a simple matter to check whether or not they contain any temporal gates. If they do not, these can be analysed separately according to traditional algorithms (e.g. MOCUS or MICSUP) or even by Euripides (since it includes standard Boolean analysis capabilities in addition to its temporal capabilities). The results for those modules can then be used directly as inputs for any higher modules or, if necessary, as inputs for temporal modules. Any temporal modules can then be analysed by Euripides & Archimedes.

As a very simple example, consider the fault tree represented by the following expression:

```
A&(B|D + B.D) + (C|(E.(F+E)))
```

This fault tree would produce four modules (not including the top OR):

1 A&(B|D + B.D)
2 (B|D + B.D)
3 C|(E.(F+E))
4 (E.(F+E))

The fourth module is static and can be analysed using normal FTA techniques. It would produce two cut sets (E.F + E.E) which would minimise to just one MCS: E. This can then be fed as an input to the third module, which is a top-level temporal module. In this case Euripides would quickly determine that the only MCSQ is C|E. Because there is only one, Archimedes would be unnecessary. (Alternatively, since there is only one small CSQ, Archimedes could also be used instead of Euripides). This MCSQ is final, since the third module is a top-level one and we know – because it is a module – that neither C nor E are used elsewhere in the fault tree.

The first module, however, is a more complex top-level temporal module containing another temporal module, the second one. Euripides could be used to determine the initial CSQs of module #2 but in this case the MCSQs are already present. Archimedes would preferably be used to determine that these two MCSQs can be reduced according to the RCL to obtain a single MCSQ: B. When fed into the first module, this would produce a larger MCSQ: A&B. As this is already minimal, it would not require any analysis by Euripides or Archimedes.

Thus the final results would simply be A&B + C|E. Using modules to analyse this tree has meant that Archimedes only had to analyse a two event set (B and D) and Euripides only had to determine one CSQ (i.e. C|E). If both techniques had been used to analyse the entire tree, it would have been much more expensive. Euripides would first have to generate the CSQs:

```
[A&B].[A|D].[B|D].[A&B].[B&D].[B&B].[B&D].[B|D].[B|D].[D|D] +
[A&B].[A|D].[B|D].[A&B].[B&D].[D<B].[B&B].[B|D].[B|D].[D|D] +
[A&B].[A|D].[B|D].[A&B].[B&D].[B<B].[B&D].[B|D].[B|D].[D|D] +
[A&B].[A|D].[B|D].[A&B].[D<B].[B&B].[B&D].[B|D].[B|D].[D|D] +
[A&B].[A|D].[B|D].[A&B].[D<B].[D<B].[B&B].[B|D].[B|D].[D|D] +
[A&B].[A|D].[B|D].[A&B].[D<B].[B<B].[B&D].[B|D].[B|D].[D|D] +
[A&B].[A|D].[B|D].[A&D].[B<D].[B&B].[B&D].[B|D].[B|D].[D|D] +
[A&B].[A|D].[B|D].[A&D].[B<D].[D<B].[B&B].[B|D].[B|D].[D|D] +
[A&B].[A|D].[B|D].[A&D].[B<D].[B<B].[B&D].[B|D].[B|D].[D|D] +
[A&B].[A|D].[B|D].[B|D]
     + [A&B].[B&D].[D<B]
     + [A&B].[D<B].[D<B]
     + [A&D].[B<D].[D<B]
     + [A&B].[B&D].[B&D]
     + [A&B].[D<B].[B&D]
     + [A&D].[B<D].[B&D]
     + [C|E].[C|E]
     + [C|E].[C|E]
     + [C|F].[C|E]
     + [C|F].[C|E]
```

And then minimise them as much as possible:

After intra-CSQ reduction:

```
[A&B].[A|D].[B|D]
+ [A&B].[D<B]
+ [A&B].[B&D]
+ [C|E]
+ [C|E]
+ [C|F].[C|E]
+ [C|F].[C|E]
```

After inter-CSQ reduction:

```
[A&B].[A|D].[B|D]
+ [A&B].[D<B]
+ [A&B].[B&D]
+ [C|E]
```

These would then be fed to Archimedes in two groups (the first three and the last one), which would create two dependency trees (or one, if the single MCSQ at the end was ignored). Archimedes would produce a three-event dependency tree for {A,B,D} and give the only result: {A,B} (i.e. A&B).

Thus the final result would be the same (namely, A&B + C|E), but the process required to achieve it would be much more costly without modules, involving the reduction of 20 CSQs (some of order 10) and the generation of a three-event dependency tree. Clearly, modularisation can provide significant benefits in Pandora, just as it can in other techniques.

4.8 Example

As a larger example of how a temporal fault tree can be qualitatively analysed, consider the simple example system used throughout the thesis:

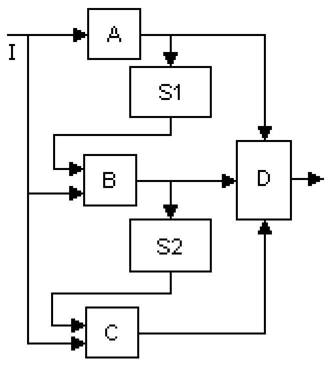


Figure 43 – The example system… again

At the end of Chapter 3, we annotated this system using Pandora's temporal gates to obtain the following temporal expression, representing the fault tree for the system. O-D is the top event and represents an omission of output from the system.

```
O-D = failureS1 < (failureA + O-I)
+ failureS1 & (failureA + O-I)
+ (failureB + O-I) < (failureA + O-I)
+ (failureB + O-I) & (failureA + O-I)
+ (failureS2 < ((failureA + O-I) < (failureB + O-I)))
+ (failureS2 & ((failureA + O-I) < (failureB + O-I)))
+ (failureC + O-I).((failureA + O-I)<(failureB + O-I)))
```

We can now attempt to perform a temporal qualitative analysis on this expression to obtain the minimal cut sequences. For the purposes of illustration, modularisation will not be used so that the full scope of minimisation using Euripides and Archimedes can be observed.

4.8.1 Applying Euripides

The first thing we have to do is to apply Euripides. The first stage of that is the Binarboreal Stage, i.e. converting the fault tree into binary form. Fortunately, it is almost already in binary form; only the top level OR is not in binary, and this can be omitted in this case (since it is an OR already in the top position).

```
  failureS1 < (failureA + O-I)
+ failureS1 & (failureA + O-I)
+ (failureB + O-I) < (failureA + O-I)
+ (failureB + O-I) & (failureA + O-I)
+ (failureS2 < ((failureA + O-I) < (failureB + O-I)))
+ (failureS2 & ((failureA + O-I) < (failureB + O-I)))
+ (failureC + O-I).((failureA + O-I)<(failureB + O-I)))
```

Next is the Flattening Stage, which flattens the fault tree into Binary Hierarchical Temporal Form. For this stage we need a number of laws to rearrange the gates properly.

AND Laws

```
X.(Y+Z) ⇔ X.Y + X.Z
(Y+Z).X ⇔ X.Y + X.Z
```

POR Laws

```
X|(Y+Z) ⇔ X|Y . X|Z
(Y+Z)|X ⇔ Y|X + Z|X
X|(Y.Z) ⇔ X|Y + X|Z
(Y.Z)|X ⇔ Y|X . Z|X
X|(Y|Z) ⇔ X|Y + X.(Z<Y) + X.(Z&Y)
(Y|Z)|X ⇔ Y|Z . Y|X
X|(Y&Z) ⇔ X.(Y|Z) + X.(Z|Y) + X|Y + X|Z
(Y&Z)|X ⇔ Y|X . Z|X . Y&Z
X|(Y<Z) ⇔ X|Z + X|Y + X.(Z<Y) + X.(Y&Z)
(Y<Z)|X ⇔ Y<Z . Z|X
```

PAND Laws

```
X<(Y+Z) ⇔ X|Z . X<Y + X|Y . X<Z
(Y+Z)<X ⇔ Y<X + Z<X
X<(Y.Z) ⇔ Y.(X<Z) + Z.(X<Y)
(Y.Z)<X ⇔ Y<X . Z<X
X<(Y|Z) ⇔ X|Y . Y|Z
(Y|Z)<X ⇔ Y<X . Y|Z
X<(Y<Z) ⇔ X<Z . Y<Z
(Y<Z)<X ⇔ Y<Z . Z<X
```

```
X<(Y&Z) ⇔ X<Y . X<Z . Y&Z
(Y&Z)<X ⇔ Z<X . Y<X . Y&Z
```

SAND Laws

```
X & (Y+Z) ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y)
(Y+Z) & X ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y)
X & (Y.Z) ⇔ (X&Y).(Y&Z) + (X&Y).(Z<Y) + (X&Z).(Y<Z)
(Y.Z) & X ⇔ (Y<X).(Z&X) + (Z<X).(Y&X) +
((X&Y).(X&Z)).(Y&Z)
X & (Y|Z) ⇔ ((X&Y) . (Y|Z)) . (X|Z)
(Y|Z) & X ⇔ X&Y . (Y|Z)
X & (Y<Z) ⇔ (X&Z . Y<Z) . Y<X
(Y<Z) & X ⇔ (Y<Z . Z&X) . Y<X
X & (Y&Z) ⇔ ((X&Y) . (X&Z)) . (Y&Z)
(Y&Z) & X ⇔ ((X&Y) . (X&Z)) . (Y&Z)
```

In the first branch, we need to rearrange it so that the OR is uppermost. For this we can use the law X<(Y+Z) ⇔ X|Z . X<Y + X|Y . X<Z:

```
failureS1 < (failureA + O-I)
⇔     failureS1|failureA . failureS1<O-I +
      failureS1|O-I . failureS1<failureA
```

The second branch can be rearranged using a SAND law, i.e. X & (Y+Z) ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y):

```
failureS1 & (failureA + O-I)
⇔     failureS1 & failureA . failureA & O-I +
      failureS1 & failureA . failureA | O-I +
      failureS1 & O-I . O-I | failure A
```

Next, the third branch uses two PAND laws, including (Y+Z)<X ⇔ Y<X + Z<X:

```
(failureB + O-I) < (failureA + O-I)
⇔     failureB < (failureA + O-I) +
      O-I < (failureA + O-I)
```

```
⇔     failureB|failureA . failureB<O-I +
      failureB|O-I . failureB<failureA +
      O-I|failureA . O-I<O-I +
      O-I|O-I . O-I<failureA
```

The next branch is similar but makes use of another SAND law, (Y+Z) & X ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y):

```
(failureB + O-I) & (failureA + O-I)
⇔     (failureA + O-I) & O-I . O-I & failureB +
      (failureA + O-I) & O-I . O-I | failureB +
      (failureA + O-I) & failureB . failureB | O-I
⇔     O-I & O-I . O-I & failureA . O-I & failureB +
      O-I & O-I . O-I | failureA . O-I & failureB +
      O-I & failureA . failureA | O-I . O-I & failureB +
      O-I & O-I . O-I & failureA . O-I | failureB +
      O-I & O-I . O-I | failureA . O-I | failureB +
      O-I & failureA . failureA . O-I | failureB +
      failureA & O-I . O-I & failureB . failureB | O-I +
      failureA & O-I . O-I | failureB . failureB | O-I +
      failureA & failureB . failureB | O-I . failureB | O-I
```

The fifth branch makes use of the PAND rules again, including some new ones, such as X<(Y.Z) ⇔ Y.(X<Z) + Z.(X<Y), X<(Y|Z) ⇔ X|Y . Y|Z, and X<(Y<Z) ⇔ X<Z . Y<Z:

```
(failureS2 < ((failureA + O-I) < (failureB + O-I)))
⇔     failureS2 < (failureA < (failureB + O-I)) +
      failureS2 < (O-I < (failureB + O-I))
⇔     failureS2 < (failureA|O-I . failureA<failureB) +
      failureS2 < (failureA|failureB . failureA<O-I) +
      failureS2 < (O-I|failureB . O-I<O-I) +
      failureS2 < (O-I|O-I . O-I<failureB)
⇔     failureA|O-I . failureS2 < (failureA<failureB) +
      failureA<failureB . failureS2 < (failureA|O-I) +
      failureA|failureB . failureS2 < (failureA<O-I) +
      failureA<O-I . failureS2 < (failureA|failureB) +
      O-I|failureB . failureS2 < (O-I<O-I) +
      O-I<O-I . failureS2 < (O-I|failureB) +
      O-I|O-I . failureS2 < (O-I<failureB) +
      O-I<failureB . failureS2 < (O-I|O-I)
```

```
⇔     failureA|O-I . failureS2<failureB . failureA<failureB +
      failureA<failureB . failureS2|failureA . failureA|O-I +
      failureA|failureB . failureS2<O-I . failureA<O-I +
      failureA<O-I . failureS2|failureA . failureA|failureB +
      O-I|failureB . failureS2<O-I . O-I<O-I +
      O-I<O-I . failureS2|O-I . failureA|failureB +
      O-I|O-I . failureS2<failureB . O-I<failureB +
      O-I<failureB . failureS2|O-I . failureA|O-I
```

The sixth branch is somewhat larger and expands as follows:

```
(failureS2 & ((failureA + O-I) < (failureB + O-I)))
⇔     failureS2 & (failureB + O-I) .
      (failureA + O-I) < (failureB + O-I) .
      (failureA + O-I) < failureS2
⇔     failureS2 & failureB . failureS2 & O-I .
      (failureA + O-I) < (failureB + O-I) .
      (failureA + O-I) < failureS2 +
      failureS2 & failureB . failureB | O-I .
      (failureA + O-I) < (failureB + O-I) .
      (failureA + O-I) < failureS2 +
      failureS2 & O-I . O-I | failureB .
      (failureA + O-I) < (failureB + O-I) .
      (failureA + O-I) < failureS2
⇔     failureS2 & failureB . failureS2 & O-I .
      failureA < (failureB + O-I) .
      failureA < failureS2 +
      failureS2 & failureB . failureS2 & O-I .
      O-I < (failureB + O-I) .
      failureA < failureS2 +
      failureS2 & failureB . failureS2 & O-I .
      failureA < (failureB + O-I) .
      O-I < failureS2 +
      failureS2 & failureB . failureS2 & O-I .
      O-I < (failureB + O-I) .
      O-I < failureS2 +
      failureS2 & failureB . failureB | O-I .
      failureA < (failureB + O-I) .
      failureA < failureS2 +
      failureS2 & failureB . failureB | O-I .
      O-I < (failureB + O-I) .
      failureA < failureS2 +
      failureS2 & failureB . failureB | O-I .
      failureA < (failureB + O-I) .
      O-I < failureS2 +
      failureS2 & failureB . failureB | O-I .
      O-I < (failureB + O-I) .
      O-I < failureS2 +
      failureS2 & O-I . O-I | failureB .
      failureA < (failureB + O-I) .
      failureA < failureS2 +
      failureS2 & O-I . O-I | failureB .
      O-I < (failureB + O-I) .
      failureA < failureS2 +
      failureS2 & O-I . O-I | failureB .
      failureA < (failureB + O-I) .
      O-I < failureS2 +
      failureS2 & O-I . O-I | failureB .
      O-I < (failureB + O-I) .
      O-I < failureS2
⇔     failureS2 & failureB . failureS2 & O-I .
      failureA | failureB . failureA < O-I .
      failureA < failureS2 +
      failureS2 & failureB . failureS2 & O-I .
      failureA | O-I . failureA < failureB .
      failureA < failureS2 +
      failureS2 & failureB . failureS2 & O-I .
      O-I | failureB . O-I < O-I .
      failureA < failureS2 +
      failureS2 & failureB . failureS2 & O-I .
      O-I | O-I  . O-I < failureB .
      failureA < failureS2 +
      failureS2 & failureB . failureS2 & O-I .
      failureA | failureB . failureA < O-I .
      O-I < failureS2 +
      failureS2 & failureB . failureS2 & O-I .
      failureA | O-I . failureA < failureB .
      O-I < failureS2 +
      failureS2 & failureB . failureS2 & O-I .
      O-I | failureB . O-I < O-I .
      O-I < failureS2 +
      failureS2 & failureB . failureS2 & O-I .
      O-I | O-I . O-I < failureB .
      O-I < failureS2 +
      failureS2 & failureB . failureB | O-I .
```

```
failureA | failureB . failureA<O-I .
failureA < failureS2 +
failureS2 & failureB . failureB | O-I .
failureA | O-I . failureA < failureB .
failureA < failureS2 +
failureS2 & failureB . failureB | O-I .
O-I | failureB . O-I < O-I .
failureA < failureS2 +
failureS2 & failureB . failureB | O-I .
O-I | O-I . O-I < failureB .
failureA < failureS2 +
failureS2 & failureB . failureB | O-I .
failureA | failureB . failureA < O-I .
O-I < failureS2 +
failureS2 & failureB . failureB | O-I .
failureA | O-I . failureA < failureB .
O-I < failureS2 +
failureS2 & failureB . failureB | O-I .
O-I | failureB . O-I < O-I .
O-I < failureS2 +
failureS2 & failureB . failureB | O-I .
O-I | O-I . O-I < failureB .
O-I < failureS2 +
failureS2 & O-I . O-I | failureB .
failureA | failureB . failureA < O-I .
failureA < failureS2 +
failureS2 & O-I . O-I | failureB .
failureA | O-I . failureA < failureB .
failureA < failureS2 +
failureS2 & O-I . O-I | failureB .
O-I | failureB . O-I < O-I .
failureA < failureS2 +
failureS2 & O-I . O-I | failureB .
O-I | O-I . O-I < failureB .
failureA < failureS2 +
failureS2 & O-I . O-I | failureB .
failureA | failureB . failureA < O-I .
O-I < failureS2 +
failureS2 & O-I . O-I | failureB .
failureA | O-I . failureA < failureB .
O-I < failureS2 +
failureS2 & O-I . O-I | failureB .
O-I | failureB . O-I < O-I .
```

```
      O-I < failureS2
      failureS2 & O-I . O-I | failureB .
      O-I | O-I . O-I < failureB .
      O-I < failureS2
```

The seventh branch is fortunately somewhat simpler:

```
(failureC + O-I).((failureA + O-I)<(failureB + O-I)))
⇔     failureC . ((failureA + O-I)<(failureB + O-I))) +
      O-I . ((failureA + O-I)<(failureB + O-I)))
⇔     failureC . failureA  < (failureB + O-I) +
      failureC . O-I < (failureB + O-I) +
      O-I . failureA < (failureB + O-I) +
      O-I . O-I < (failureB + O-I)
⇔     failureC . failureA | failureB . failureA < O-I +
      failureC . failureA | O-I . failureA < failureB +
      failureC . O-I | failureB . O-I < O-I +
      failureC . O-I | O-I . O-I < failureB +
      O-I . failureA | failureB . failureA < O-I +
      O-I . failureA | O-I . failureA < failureB +
      O-I . O-I | failureB . O-I < O-I +
      O-I . O-I | O-I . O-I < failureB
```

The next step is to convert these CSQs into doublets; fortunately, thanks to the flattening laws used, they are already prepared and in most cases can be converted directly into doublets. In some cases, additional doublets are generated according to the Law of Extension, e.g.:

```
failureS1 & failureA . failureA & O-I +
failureS1 & failureA . failureA | O-I +
failureS1 & O-I . O-I | failure A
➔
[failureS1 & failureA] . [failureA & O-I] . [failureS1 & O-I] +
[failureS1 & failureA] . [failureA | O-I] . [failureS1 | O-I] +
[failureS1 & O-I] . [O-I | failureA] . [failureS1 | failureA] +
```

Finally, we can begin the Minimisation Stage, starting with intra-CSQ minimisation.

The first branch has no redundancies so remains:

```
[failureS1|failureA] . [failureS1<O-I]
[failureS1|O-I] . [failureS1|failureA]
```

The second branch also has no redundancies yet:

[failureS1 & failureA] . [failureA & O-I] . [failureS1 & O-I]
[failureS1 & failureA] . [failureA | O-I] . [failureS1 | O-I]
[failureS1 & O-I] . [O-I | failureA] . [failureS1 | failureA]

The third branch allows us to detect a contradiction because two of its CSQs contain violations of Simultaneity:

[failureB|failureA] . [failureB<O-I]
[failureB|O-I] . [failureB<failureA]
~~[O-I|failureA] . [O-I<O-I]~~
~~[O-I|O-I] . [O-I<failureA]~~

The fourth branch provides great scope for Absorption and, in some cases, Contradiction:

~~[O-I & O-I] .~~ [O-I & failureA] . [O-I & failureB] . [failureA & failureB]
~~[O-I & O-I] .~~ [O-I | failureA] . [O-I & failureB] . [failureB | failureA]
~~[O-I & failureA] . [failureA | O-I] . [O-I & failureB]~~
~~[O-I & O-I] .~~ [O-I & failureA] . [O-I | failureB] . [failureA | failureB]
~~[O-I & O-I] .~~ [O-I | failureA] . [O-I | failureB]
~~[O-I & failureA] . [failureA | O-I] . [O-I | failureB]~~
~~[failureA & O-I] . [O-I & failureB] . [failureB | O-I]~~
~~[failureA & O-I] . [O-I | failureB] . [failureB | O-I]~~
[failureA & failureB] . [failureB | O-I]~~ . [failureB | O-I]~~

For example, [O-I&O-I] reduces to just O-I according to Simultaneity, and this unaffiliated event is then absorbed into the other doublets. In the case of the third, sixth, seventh, and eighth CSQs, Mutual Exclusion is violated, e.g. in the 8th we have both [O-I|failureB] and [failureB|O-I] – so all four of these CSQs can be discarded.

The other branches minimise in a similar way until we are left with thirty doubletised CSQs. Note that the four long CSQs (#23 - #26) are all that remains of the huge number of CSQs from the sixth branch once reduced. A certain amount of inter-CSQ minimisation can now take place, checking for obvious cases of Absorption, Priority, and POR Transformation:

1. [failureS1|failureA] . [failureS1<O-I]
2. [failureS1|O-I] . [failureS1<failureA]
3. [failureS1 & failureA] . [failureA & O-I] . [failureS1 & O-I]
4. [failureS1 & failureA] . [failureA | O-I] . [failureS1 | O-I]
5. [failureS1 & O-I] . [O-I | failureA] . [failureS1 | failureA]
6. [failureB|failureA] . [failureB<O-I]
7. [failureB|O-I] . [failureB<failureA]
8. ~~[O-I & failureA] . [O-I & failureB] . [failureA & failureB]~~ **absorbed, #13**
~~9. [O-I | failureA] . [O-I & failureB] . [failureB | failureA]~~ **absorbed, #14**
~~10. [O-I & failureA] . [O-I | failureB] . [failureA | failureB]~~ **absorbed, #15**
11. [O-I | failureA] . [O-I | failureB]
12. [failureA & failureB] . [failureB | O-I]
13. [O-I & failureA] . [O-I & failureB]
14. [O-I | failureA] . [O-I & failureB]
15. [O-I & failureA] . [O-I | failureB]
~~16. [O-I | failureA] . [O-I | failureB]~~ **duplicate #11**
~~17. [failureA & failureB] . [failureA | O-I] . [failureB | O-I]~~ **duplicate #12**
18. [failureA|O-I] . [failureS2<failureB] . [failureA<failureB]
19. [failureA<failureB] . [failureS2<failureA] . [failureA|O-I]
20. [failureA|failureB] . [failureS2<O-I] . [failureA<O-I]
21. [failureA<O-I] . [failureS2|failureA] . [failureA|failureB]
22. [O-I<failureB] . [failureS2|O-I] . [failureA|O-I]
23. [failureS2 & failureB] . [failureS2 & O-I] . [failureA < O-I] . [failureA < failureB] . [failureA < failureS2] . [failureB & O-I]
24. [failureS2 & failureB] . [failureB | O-I] . [failureS2 | O-I] . [failureA | O-I] . [failureA < failureB] . [failureA < failureS2]
25. [failureS2 & O-I] . [O-I | failureB] . [failureS2 | failureB] . [failureA | failureB] . [failureA < O-I] . [failureA < failureS2]
26. [failureS2 & O-I] . [O-I < failureB] . [failureS2 < failureB] . [failureA < O-I] . [failureA < failureB] . [failureA < failureS2]
27. failureC . [failureA | failureB] . [failureA < O-I]
28. failureC . [failureA | O-I] . [failureA < failureB]
29. [failureA | failureB] . [failureA < O-I]
30. [failureA < O-I] . [failureA < failureB]

It is not clear how else these remaining 'mostly minimal' cut sequences can be reduced, so now it is time to organise these 25 remaining CSQs and give them to Archimedes.

4.8.2 Applying Archimedes

The first step is to organise the results into groups of cut sequenced with shared events. In this case all the events are shared, so all 25 have to be given to Archimedes. As there are six events, Archimedes will then build a six-event dependency tree (with *n* = 6, there are 9367 nodes in the equivalent precedence tree, so this will be a very large dependency tree).

Since it is impossible to show how Archimedes analyses a dependency tree this large, we can look at an interesting subset of the CSQs to see what kind of results they produce. The CSQs in question are below and consist of those cut sequences that contain only `failureA, failureB, and O-I`.

```
[failureB|failureA] . [failureB<O-I]
[failureB|O-I] . [failureB<failureA]
[failureA & failureB] . [failureB | O-I]
[O-I | failureA] . [O-I | failureB]
[O-I & failureA] . [O-I & failureB]
[O-I | failureA] . [O-I & failureB]
[O-I & failureA] . [O-I | failureB]
[failureA | failureB] . [failureA < O-I]
[failureA < O-I] . [failureA < failureB]
```

When these nine CSQs are fed into Archimedes, it will produce a much smaller dependency tree. If we look at a particular part of this tree, we can see how the results are formed. For example, the subtree for the pure conjunction `failureA . failureB` is shown below, with basic temporal nodes in italics and the node names abbreviated (i.e. A = failureA, B = failureB, O = O-I). Nodes that are true for the above cut sequences are bold, whilst nodes which are true and *not* redundant are underlined. Thus the nodes that are bold and underlined are the top-most nodes that will be used to form the results.

```
A.B
+--- A.B|O
|      +--- A<B|O
|      |      +--- A<B.¬O
|      |      +--- A<B<O
|      +--- A&B|O
|      |      +--- A&B.¬O
|      |      +--- A&B<O
|      +--- B<A|O
|             +--- B<A.¬O
|             +--- B<A<O
```

```
+--- A.B.O
|     +--- A<B.O
|     |     +--- A<B<O
|     |     +--- A<B&O
|     |     +--- A<O<B
|     |     +--- A&O<B
|     |     +--- O<A<B
|     +--- A&B.O
|     |     +--- A&B<O
|     |     +--- A&B&O
|     |     +--- O<A&B
|     +--- B<A.O
|     |     +--- B<A<O
|     |     +--- B<A&O
|     |     +--- B<O<A
|     |     +--- B&O<A
|     |     +--- O<B<A
|     +--- A<O.B
|     |     +--- A<O<B
|     |     +--- A<O&B
|     |     +--- A<B<O
|     |     +--- A&B<O
|     |     +--- B<A<O
|     +--- A&O.B
|     |     +--- A&O<B
|     |     +--- A&B&O
|     |     +--- B<A&O
|     +--- O<A.B
|     |     +--- O<A<B
|     |     +--- O<A&B
|     |     +--- O<B<A
|     |     +--- O&B<A
|     |     +--- B<O<A
|     +--- B<O.A
|     |     +--- B<O<A
|     |     +--- B<A&O
|     |     +--- B<A<O
|     |     +--- A&B<O
|     |     +--- A<B<O
|     +--- B&O.A
|     |     +--- B&O<A
|     |     +--- A&B&O
|     |     +--- A<B&O
|       +--- O<B.A
|               +--- O<A<B
|               +--- O<A&B
|               +--- O<B<A
|               +--- O&B<A
|               +--- B<O<A
+--- A<B
|       +--- A<B.O
|       |       +--- O<A<B
|       |       +--- O&A<B
|       |       +--- A<O<B
|       |       +--- A<B&O
|       |       +--- A<B<O
|       +--- A<B|O
|               +--- A<B.¬O
|               +--- A<B<O
+--- A&B
|       +--- A&B.O
|       |       +--- O<A&B
|       |       +--- O&A&B
|       |       +--- A&B<O
|       +--- A&B|O
|               +--- A&B.¬O
|               +--- A&B<O
+--- B<A
        +--- B<A.O
        |       +--- O<B<A
        |       +--- O&B<A
        |       +--- B<O<A
        |       +--- B<A&O
        |       +--- B<A<O
        +--- B<A|O
                +--- B<A.¬O
                +--- B<A<O
```

Of all the nodes here, only two are true and non-redundant: A&B and B<A. The other top 'true' nodes (e.g. A<B.O) are made redundant by the singleton O, which is also true for this set of cut sequences. Thus Archimedes tells us that the MCSQs for the nine CSQs presented above are simply:

A&B

B<A

O

If we so wished, in this case we could use these three cut sequences to minimise the other cut sequences (though we would not normally do so). However, the results are quite illustrative:

1. failureA & failureB
2. failureB < failureA
3. O-I

~~4. [failureS1|failureA] . [failureS1<O-I]~~ **(from #3)**

5. ~~[failureS1|O-I] .~~ [failureS1<failureA] **(POR Transformation, from #3)**

~~6. [failureS1 & failureA] . [failureA & O-I] . [failureS1 & O-I]~~ **(from #3)**

7. [failureS1 & failureA]~~ . [failureA | O-I] . [failureS1 | O-I]~~ **(POR-T, #3)**

~~8. [failureS1 & O-I] . [O-I | failureA] . [failureS1 | failureA]~~ **(from #3)**

9. ~~[failureA|O-I] .~~ [failureS2<failureB] . [failureA<failureB] **(POR-T, #3)**

10. [failureA<failureB] . [failureS2<failureA]~~ . [failureA|O-I]~~ **(POR-T, #3)**

~~11. [failureA|failureB] . [failureS2<O-I] . [failureA<O-I]~~ **(from #3)**

~~12. [failureA<O-I] . [failureS2|failureA] . [failureA|failureB]~~ **(from #3)**

~~13. [O-I<failureB] . [failureS2|O-I] . [failureA|O-I]~~ **(from #3)**

~~14. [failureS2 & failureB] . [failureS2 & O-I] . [failureA < O-I] . [failureA < failureB] . [failureA < failureS2] . [failureB & O-I]~~ **(from #3)**

15. [failureS2 & failureB]~~ . [failureB | O-I] . [failureS2 | O-I] . [failureA | O-I]~~ . [failureA < failureB] . [failureA < failureS2] **(POR-T from #3)**

~~16. [failureS2 & O-I] . [O-I | failureB] . [failureS2 | failureB] . [failureA | failureB] . [failureA < O-I] . [failureA < failureS2]~~ **(from #3)**

~~17. [failureS2 & O-I] . [O-I < failureB] . [failureS2 < failureB] . [failureA < O-I] . [failureA < failureB] . [failureA < failureS2]~~ **(from #3)**

~~18. failureC . [failureA | failureB] . [failureA < O-I]~~ **(from #3)**

19. failureC . ~~[failureA | O-I] .~~ [failureA < failureB] **(POR-Trans., from #3)**

Thus minimising using these three cut sequences actually provides us with the full MCSQs for the example fault tree:

```
1. O-I
2. failureA & failureB
3. failureB < failureA
4. failureS2 < failureB . failureA < failureB
5. failureS2 < failureA < failureB
6. failureA < failureS2 & failureB
7. failureC . failureA < failureB
```

Archimedes will also provide the same results, although because it does not allow multiple sequences in the same conjunction, #4 will be expanded to:

```
failureS2 < failureA < failureB
failureS2 & failureA < failureB
failureA < failureS2 < failureB
```

according to the Completion Law. Thus there are 8 full MCSQs for the example system:

- an omission of input, which is a common cause failure as all of A, B, and C will fail;
- a failure of A and B at the same time, meaning B will never activate and will cause an undetectable omission of output;
- a dormant failure of B before A, meaning B will never activate and will cause an undetectable omission of output;
- a failure of sensor 2 before component A before a failure of B, meaning that C will never be activated;
- a failure of A before sensor 2 before a failure of B, meaning that C will never be activated;
- a simultaneous failure of sensor 2 and component A before a failure of B, meaning that C will never be activated;
- a failure of A before a simultaneous failure of sensor 2 and B, meaning that C will never be activated;
- a failure of all three components, assuming A fails before B (and thus B gets activated).

These results are considerably more accurate and informative than the purely static minimal cut sets produced in the **Introduction**.

5 Case Study

"The goal of all inanimate objects is to resist man and ultimately defeat him."
- Russell Baker

To see how Pandora might be used to analyse a less abstract system, Figure 44 shows a simplified version of an automotive braking system[25]. Electronic braking systems such as ESC (Electronic Stability Control) and ABS (Anti-lock Braking System) are increasingly used in vehicles due to the safety advantages they offer. However, due to the increasing complexity that such systems introduce, a more detailed safety analysis is often required.

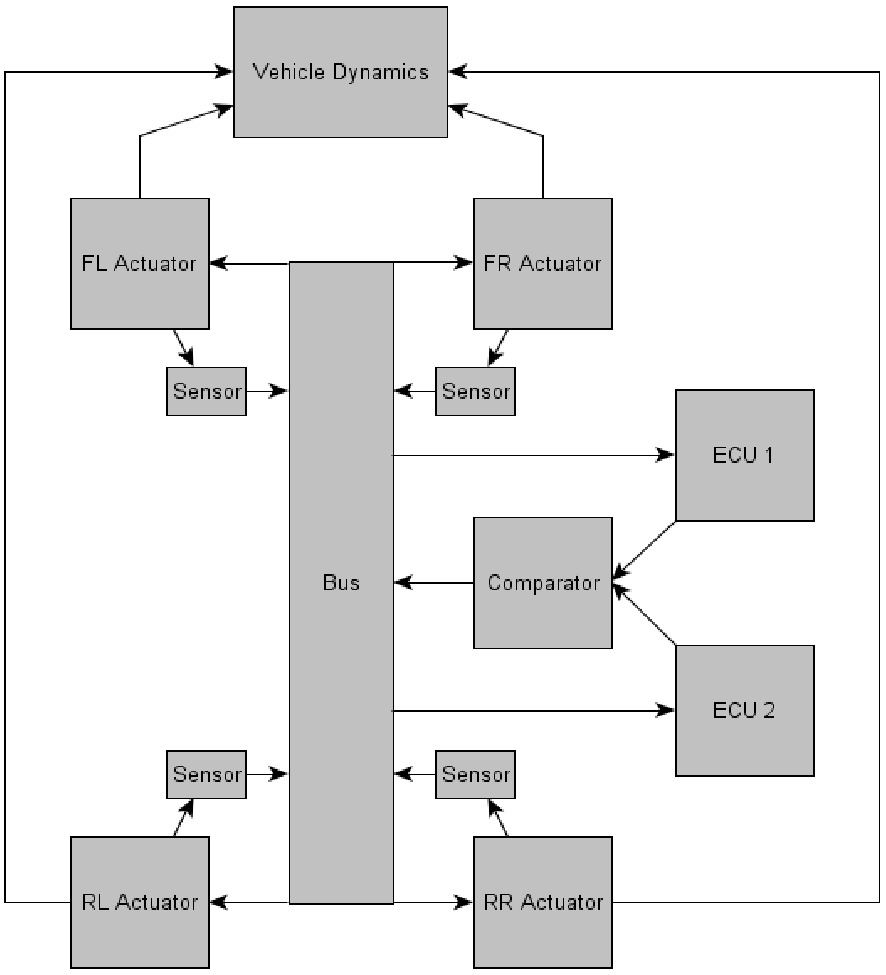


Figure 44 – Case Study: Automotive brake-by-wire system

The model consists of four separate brake actuators, one at each wheel, and each with a connected rotation sensor. The actuators are controlled via a central bus, which also carries the signals from the sensors. These signals are fed into a pair of electronic control units (ECUs) that control the brakes. The output of both ECUs must agree (as determined by a comparator) for the braking commands to be sent to the actuators. This is to prevent inadvertent braking caused by an error in one ECU. Finally, the 'vehicle dynamics' component is a virtual component representing the effect the brakes have on the handling of the vehicle. This can be thought of as the output of the braking system.

For the purposes of this case study, the model has been highly simplified in order to limit the number and complexity of results produced. The purpose is to show how Pandora can be used to model situations that may arise in a real-life system, not to demonstrate its capability to produce vast amounts of information. To that end, there is very little complex propagation of failure through the system, so that causes link almost directly with effects, and the only type of system-level failure that will be considered here is the permanent commission of braking pressure, i.e. the brakes locking. This type of failure can also be seen as a pessimistic extreme of failures involving excess braking pressure or temporary locking of the brakes.

[25] A slightly abbreviated version of this case study is in Walker *et al* (2009), in which the results of the FTA were presented in a form of 'temporal' FMEA.

5.1 Local failure data

"Events will take their course, it is no good of being angry at them; he is happiest who wisely turns them to the best account."
- Euripides, *Bellerophon*

Before analysis can proceed, the failure behaviour of the components of the system model must be modelled with local failure data, i.e. a Pandora expression for each possible failure that relates the failures to their causes (whether an internal failure of that component or another failure that has propagated to the component). The abbreviation 'C' stands for commission and the abbreviation 'Com' is used as a generalised component failure mode that can lead to a commission of output (e.g. ECUCom is an internal failure of an ECU causing a commission of output).

Furthermore, since the wheels, brakes, and sensors are all duplicated, we need only consider these once.

5.1.1 Actuators

Each actuator is responsible for applying braking pressure to the wheels in response to commands from the ECU. Actuator commission failures (i.e. they brake when they are *not* commanded to) can occur as a result of an error in the signal input from the bus, e.g. an incorrect command to brake, and also as a result of internal failures. For example, an actuator jamming can lead to a commission (or partial commission) if the jam occurs while currently applying brake pressure, because when the brake signal ceases, the actuator will still be in the 'brake' position.

Thus the local failure expression for each of the actuators is as follows:

C-Actuator = ActuatorCom + C-BrakeSignal

i.e. a commission of actuator braking pressure is caused by the actuator jamming in the braking position (an internal actuator commission) or by an incorrect command to brake from the bus.

5.1.2 Bus

The bus is responsible for communicating between the ECUs and the brake actuators and between the brake sensors and the ECUs. For the purposes of this case study, the only failure modelled as originating in the bus is a spurious signal, e.g. caused by EMI (electromagnetic interference) or by memory degradation in the bus leading to stuck bits. Commission failures are also propagated from the input, e.g. if an incorrect command to brake is sent to the bus, then the bus will forward this command to the actuators.

The local failure expressions for the bus are therefore as follows:

C-BusCommand = C-BrakeCommand
C-BusData = BusCom + C-SensorData

5.1.3 Sensors

The rotational speed sensors at the wheels are responsible for measuring the turn of the wheels and feeding this data back to the ECUs. If the data indicates that the car is out of control or nearly so, then the ECUs may issue commands to the brakes to attempt to regain stability. Each sensor can experience a high or low bias leading to a value error in its data, and the sensors can also fail entirely as a result of mechanical failure (however, this cessation of sensor data may be filtered out by plausibility checks in the ECUs). For the purposes of the case study, only high value errors are considered here (which will appear to the ECUs as the wheels turning faster than they really are).

The local failure behaviour for each sensor is thus simply:

C-Sensor = SensorCom

5.1.4 ECUs

The two ECUs are responsible for analysing the data returned by the four wheel sensors and determining whether the car needs stabilising. If so, the ECUs will send the appropriate commands to the four brake actuators. The ECUs are duplicated for redundancy and must agree for their commands to be communicated to the actuators. ECUs may fail as a result of both hardware problems and software errors, but often have some degree of resilience. For the purposes of the case study, it is assumed that ECUs fail silent in response to any internal errors (which would then be filtered out by the comparator), but that any value failures at their inputs may lead to an incorrect braking command being issued.

The failure data for the ECUs is as follows:

C-ECU = C-SensorData

5.1.5 Comparator

The comparator is necessary to compare the signals sent from both ECUs and ensure that both signals are in agreement before the command is sent to the brake actuators. This ensures that spurious commands from one ECU do not cause inadvertent braking. It also means that if one ECU fails, the comparator will assume both ECUs are in disagreement and will not forward the braking command. Thus the comparator seems to fail silent in response to an error from either ECU and any commission errors must be sent from both ECUs to be propagated.

The failure behaviour is thus:

C-Comparator = C-ECU1 . C-ECU2

5.1.6 Vehicle Dynamics

The final 'component' is the Vehicle Dynamics pseudo-component, which serves as the system output for the system. This is where the temporal semantics of Pandora are useful, because the order of brake failures can lead to different effects – with different severities – on the car. This level of detail is not normally possible to incorporate in FTA.

Because there are four wheels and four brakes, the failure of every sequence and every combination of these should ordinarily be taken into consideration. However, since the brakes are symmetrical, in practice many of these combinations will result in symmetrical duplicates, and so rather than considering left and right sides, we only refer to the near-side and far-side; thus these results are equally applicable to either side of the car. 'N' is used to indicate a near-side failure and 'O' for opposite side failures in cases where more than one brake failure is being considered; 'F' and 'R' represent front and rear respectively. Furthermore, combinations of more than two brake failures are treated only as simultaneous failures; any sequence of three brake failures will first result in a sequence of two brake failures, thus the two brake failures will normally have the initial and most noticeable effects.

The sequences of failures, together with their effects, are presented in the table below (using F = Front, R = Rear, N = Near side, O = Opposite side etc). Note that the effects assume front-wheel drive.

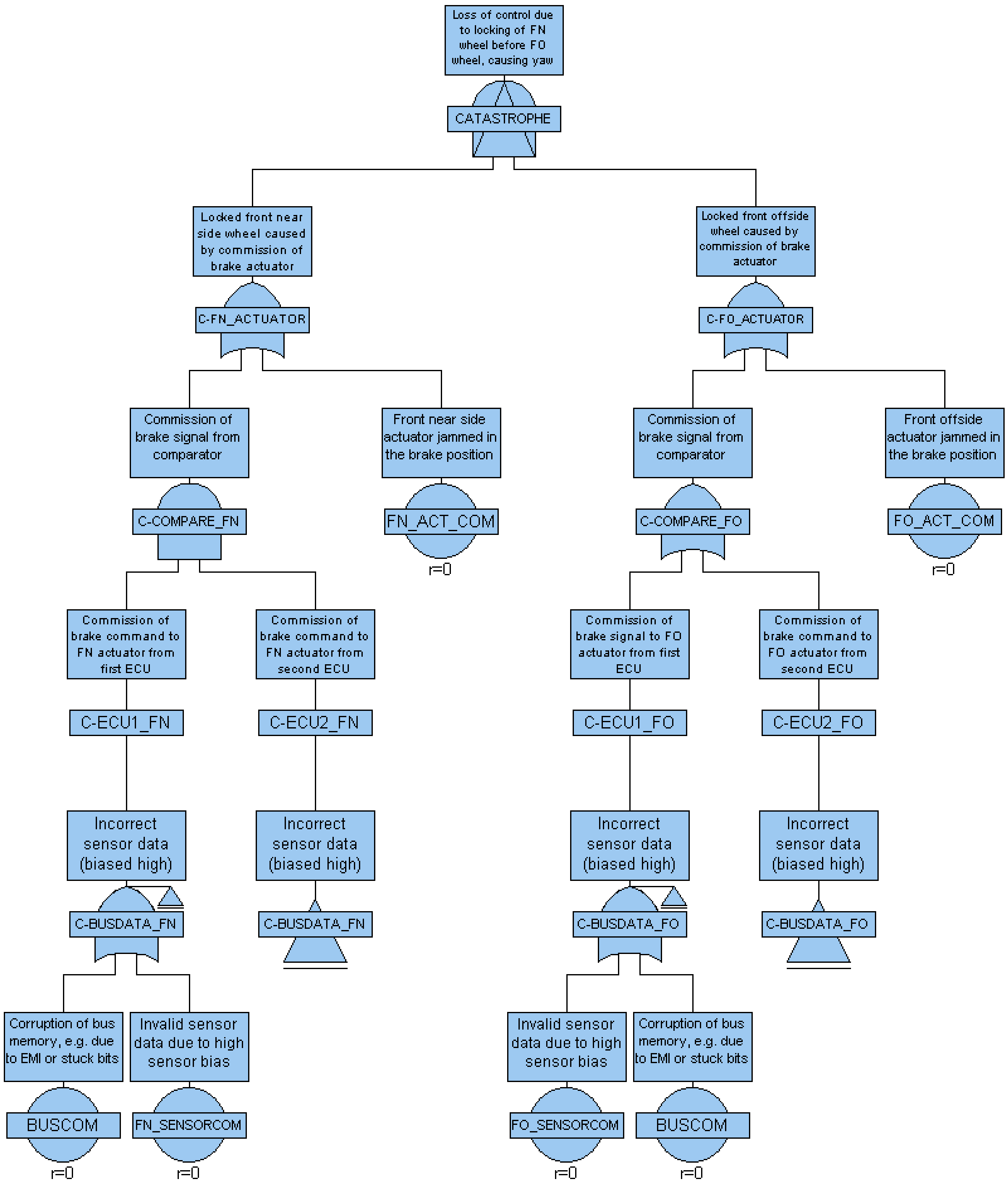
FAILURE	EFFECT	SEVERITY
C-BrakeF	Some loss of stability and control but supporting systems and other brakes can mitigate effect.	Moderate
C-BrakeR	Minor loss of stability, which can be mitigated, e.g. by increasing braking in the opposite front wheel	Marginal
C-BrakeFN < C-BrakeRN	Major loss of stability and control, potentially leading to a collision or going off the road. Ability to compensate limited.	Critical
C-BrakeRN < C-BrakeFN	Loss of stability but greater possibility of compensating.	Moderate
C-BrakeFN & C-BrakeRN	Major loss of stability and control; greater surprise, less possibility of mitigation. Will likely cause car to yaw in the direction of the near side.	Catastrophic
C-BrakeFN < C-BrakeRO	Some loss of control but locking of opposite wheel helps to stabilise vehicle with more chance of mitigation.	Moderate
C-BrakeRO < C-BrakeFN	Some loss of stability, but locking of opposite wheels results increases chances of successful compensation.	Marginal
C-BrakeFN & C-BrakeRO	Some loss of stability and control, with greater surprise, but locking of opposite wheels results is less severe.	Critical
C-BrakeFN < C-BrakeFO	Locking of both front wheels leads to sharper braking and loss of steering; vehicle will yaw to the near side.	Catastrophic
C-BrakeFO < C-BrakeFN	Locking of both front wheels leads to sharper braking and loss of steering; vehicle will yaw to the far side.	Catastrophic
C-BrakeFN & C-BrakeFO	Locking of both front wheels leads to sharper braking and loss of steering; less yawing due to simultaneous failure.	Catastrophic
C-BrakeRN < C-BrakeRO	Locking of rear wheels less severe than front, and more control is maintained; vehicle will yaw to the near side.	Critical

C-BrakeRO < C-BrakeRN	Locking of rear wheels less severe than front, and more control is maintained; vehicle will yaw to the far side.	Critical
C-BrakeRN & C-BrakeRO	Locking of rear wheels less severe than front, and more control is maintained; reduced loss of stability due to simultaneous rear braking.	Moderate
C-BrakeFN & C-BrakeFO & C-BrakeRN	Severe loss of both stability and control.	Catastrophic
C-BrakeRN & C-BrakeRO & C-BrakeFN	Severe loss of stability and major loss of control.	Catastrophic
C-BrakeFN & C-BrakeFO & C-BrakeRN & C-BrakeRO	All brakes lock; catastrophic loss of stability and control.	Catastrophic

Table 7 – Effects of braking commissions on the vehicle

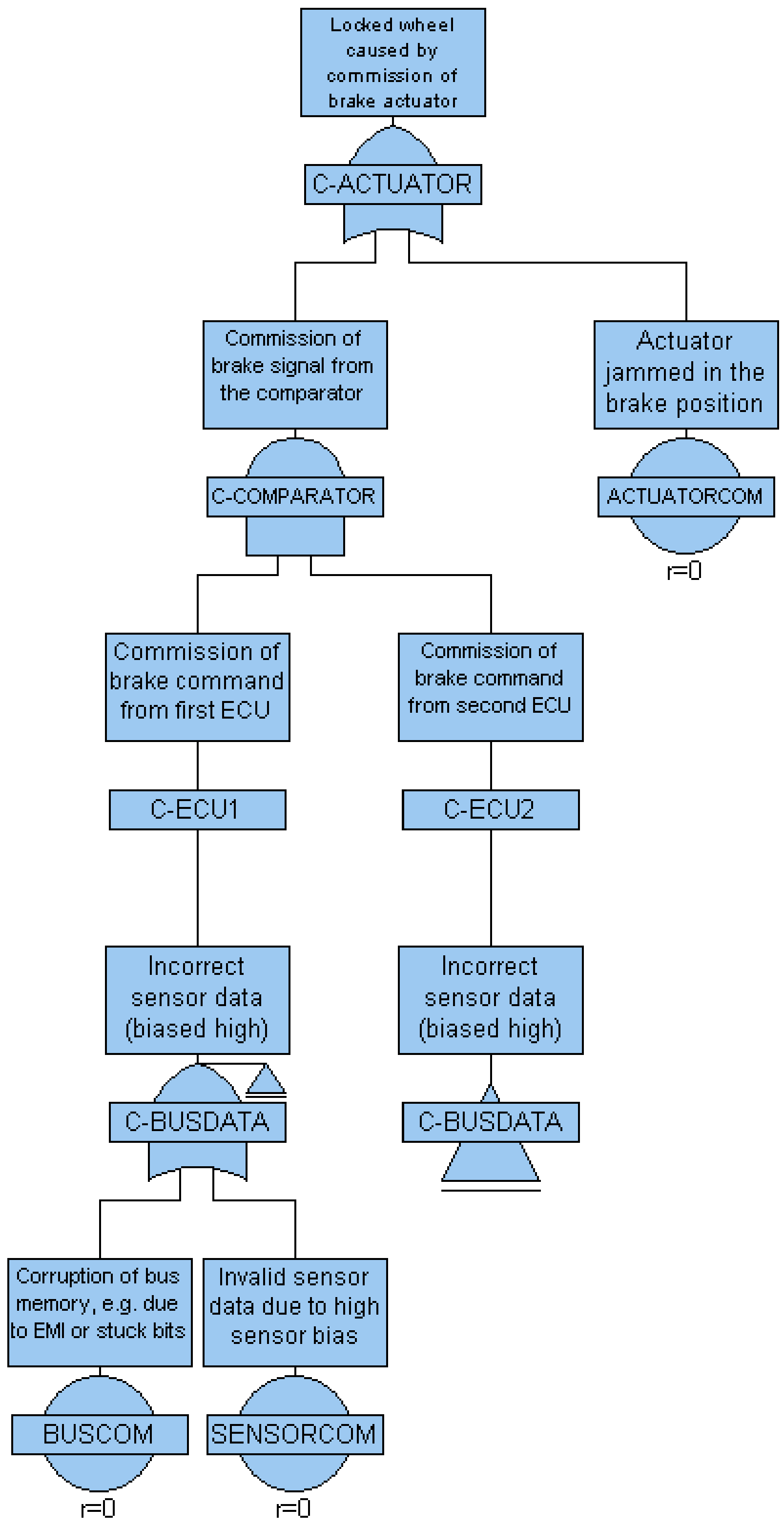
5.2 Fault Tree Analysis

There are 17 fault trees, one for each of the possible sequences of failure of the vehicle's braking system. However, for the most part, all of these share substantial common branches: namely, the commission of a brake actuator. The fault tree for C-ActuatorFN < C-ActuatorFO is shown below, in Figure 45, while the common fault tree for an indvidual actuator is shown in Figure 46.



Loss of control due to locking of FN wheel before FO wheel, causing yaw
CATASTROPHE
Locked front near side wheel caused by commission of brake actuator
C-FN_ACTUATOR
Locked front offside wheel caused by commission of brake actuator
C-FO_ACTUATOR
Commission of brake signal from comparator
C-COMPARE_FN
Front near side actuator jammed in the brake position
FN_ACT_COM
r=0
Commission of brake signal from comparator
C-COMPARE_FO
Front offside actuator jammed in the brake position
FO_ACT_COM
r=0
Commission of brake command to FN actuator from first ECU
C-ECU1_FN
Commission of brake command to FN actuator from second ECU
C-ECU2_FN
Commission of brake signal to FO actuator from first ECU
C-ECU1_FO
Commission of brake command to FO actuator from second ECU
C-ECU2_FO
Incorrect sensor data (biased high)
C-BUSDATA_FN
Incorrect sensor data (biased high)
C-BUSDATA_FN
Incorrect sensor data (biased high)
C-BUSDATA_FO
Incorrect sensor data (biased high)
C-BUSDATA_FO
Corruption of bus memory, e.g. due to EMI or stuck bits
BUSCOM
r=0
Invalid sensor data due to high sensor bias
FN_SENSORCOM
r=0
Invalid sensor data due to high sensor bias
FO_SENSORCOM
r=0
Corruption of bus memory, e.g. due to EMI or stuck bits
BUSCOM
r=0

Figure 45 - Fault tree for C-ActuatorFN < C-ActuatorFO



Locked wheel caused by commission of brake actuator
C-ACTUATOR
Commission of brake signal from the comparator
Actuator jammed in the brake position
C-COMPARATOR
ACTUATORCOM
r=0
Commission of brake command from first ECU
Commission of brake command from second ECU
C-ECU1
C-ECU2
Incorrect sensor data (biased high)
Incorrect sensor data (biased high)
C-BUSDATA
C-BUSDATA
Corruption of bus memory, e.g. due to EMI or stuck bits
Invalid sensor data due to high sensor bias
BUSCOM
SENSORCOM
r=0
r=0

Figure 46 – Generic fault tree for any C-Actuator

The minimal cut sets for this (static) fault tree are just:

- BusCom
- SensorCom
- ActuatorCom

Note however that while BusCom is a common cause failure, i.e. it will lead to commissions at all four actuators, SensorCom and ActuatorCom are specific to each actuator. Note that this relies on the assumption that the ECUs submit commands to an actuator at a wheel only in response to data from the sensor at that same wheel; situations where the ECU can issue a command in response to a sensor reading at another wheel (e.g. in an attempt to mitigate a failure elsewhere) are not considered here. Thus the cut sets for C-ActuatorFN, for example, are BusCom, SensorComFN, and ActuatorComFN, whereas for C-ActuatorRO, for example, they would be BusCom, SensorComRO, and ActuatorComRO.

These are also the minimal cut sets for the two single actuator failures (i.e. C-ActuatorFN and C-ActuatorRN). However, when sequences are involved, these minimal cut sets serve as inputs to the temporal gates, e.g. for C-ActuatorFN < C-ActuatorRN:

```
      (BusCom + SensorComFN + ActuatorComFN) <
      (BusCom + SensorComRN + ActuatorComRN)
```

Applying the PAND distributive law (X + Y) < Z ⇔ X<Z + Y<Z gives us:

```
      BusCom < (BusCom + SensorComRN + ActuatorComRN)
+     SensorComFN < (BusCom + SensorComRN + ActuatorComRN)
+     ActuatorComFN < (BusCom + SensorComRN + ActuatorComRN)
```

Then applying a second PAND distributive law, X < (Y+Z) ⇔ X<Y.X|Z + X<Z.X|Y, gives us the following nine cut sequences:

```
      BusCom < BusCom . BusCom | SensorComRN . BusCom |
      ActuatorComRN
+     BusCom < SensorComRN . BusCom | BusCom . BusCom |
      ActuatorComRN
+     BusCom < ActuatorComRN . BusCom | BusCom . BusCom |
      SensorComRN
+     SensorComFN < BusCom . SensorComFN | SensorComRN .
      SensorComFN | ActuatorComRN
```

```
+     SensorComFN < SensorComRN . SensorComFN | ActuatorComRN .
      SensorComFN | BusCom
+     SensorComFN < ActuatorComRN . SensorComFN | BusCom .
      SensorComFN | ActuatorComRN
+     ActuatorComFN < BusCom . ActuatorComFN | SensorComRN .
      ActuatorComFN | ActuatorComRN
+     ActuatorComFN < SensorComRN . ActuatorComFN |
      ActuatorComRN .  ActuatorComFN | BusCom
+     ActuatorComFN < ActuatorComRN . ActuatorComFN | BusCom .
      ActuatorComFN | ActuatorComRN
```

BusCom | BusCom and BusCom < BusCom are contradictions according to the Laws of Simultaneity, so the first three cut sequences are redundant. Removing these leaves six minimal cut sequences:

```
      SensorComFN < BusCom . SensorComFN | SensorComRN .
      SensorComFN | ActuatorComRN
+     SensorComFN < SensorComRN . SensorComFN | ActuatorComRN .
      SensorComFN | BusCom
+     SensorComFN < ActuatorComRN . SensorComFN | BusCom .
      SensorComFN | ActuatorComRN
+     ActuatorComFN < BusCom . ActuatorComFN | SensorComRN .
      ActuatorComFN | ActuatorComRN
+     ActuatorComFN < SensorComRN . ActuatorComFN |
      ActuatorComRN .  ActuatorComFN | BusCom
+     ActuatorComFN < ActuatorComRN . ActuatorComFN | BusCom .
      ActuatorComFN | ActuatorComRN
```

These MCSQs tell us that a commission of the front near-side brake followed by a commission of the rear near-side brake is caused by a sensor failure or an actuator failure in the front-near brake followed by a failure of the bus, rear near-side actuator, or rear near-side sensor. The BusCom CSQs are not present because a commission failure of the Bus *first* will in fact lead to a simultaneous commission of all actuators, as can be seen if we examine the results for C-ActuatorFN & C-ActuatorRN, for example:

```
      (BusCom + SensorComFN + ActuatorComFN) &
      (BusCom + SensorComRN + ActuatorComRN)
```

Applying Euripides will yield the following 27 MCSQs (shown here in a condensed format to save space):

```
ActuatorComFN & ActuatorComRN & SensorComFN & SensorComRN &
BusCom +
ActuatorComFN & ActuatorComRN & SensorComFN & SensorComRN |
BusCom +
ActuatorComFN & ActuatorComRN & SensorComRN & BusCom |
SensorComFN +
ActuatorComFN & ActuatorComRN & SensorComFN & BusCom |
SensorComRN +
ActuatorComRN & SensorComFN & SensorComRN & BusCom |
ActuatorComFN +
ActuatorComFN & SensorComFN & SensorComRN & BusCom |
ActuatorComRN +
ActuatorComFN & ActuatorComRN & BusCom | SensorComFN |
SensorComRN +
ActuatorComFN & ActuatorComRN & SensorComFN | BusCom |
SensorComRN +
ActuatorComFN & ActuatorComRN & SensorComRN | BusCom |
SensorComFN +
ActuatorComFN & SensorComFN & SensorComRN | ActuatorComRN |
BusCom +
ActuatorComRN & SensorComFN & SensorComRN | ActuatorComFN |
BusCom +
ActuatorComFN & SensorComRN & BusCom | ActuatorComRN |
SensorComFN +
ActuatorComRN & SensorComFN & BusCom | ActuatorComFN |
SensorComRN +
SensorComFN & SensorComRN & BusCom | ActuatorComFN |
ActuatorComRN +
ActuatorComFN & SensorComFN & BusCom | ActuatorComRN |
SensorComRN +
ActuatorComRN & SensorComRN & BusCom | ActuatorComFN |
SensorComFN +
ActuatorComFN & ActuatorComRN & BusCom | SensorComFN |
SensorComRN +
SensorComFN & SensorComRN & BusCom | ActuatorComFN |
ActuatorComRN +
ActuatorComFN & ActuatorComRN | BusCom | SensorComFN |
SensorComRN +
ActuatorComFN & SensorComRN | ActuatorComRN | BusCom |
SensorComFN +
ActuatorComRN & SensorComFN | ActuatorComFN | BusCom |
SensorComRN +
SensorComFN & SensorComRN | ActuatorComFN | ActuatorComRN |
BusCom +
BusCom & SensorComFN | ActuatorComFN | ActuatorComRN |
SensorComRN +
BusCom & SensorComRN | ActuatorComFN | ActuatorComRN |
SensorComFN +
BusCom & ActuatorComFN | SensorComFN | SensorComRN |
ActuatorComRN +
BusCom & ActuatorComRN | SensorComFN | SensorComRN |
ActuatorComFN +
BusCom | ActuatorComFN | ActuatorComRN | SensorComFN |
SensorComRN
```

These results show us that a simultaneous commission of braking of the front and rear near-side wheels is caused by a simultaneous occurrence of at least one front wheel failure and at least one rear wheel failure, or a failure of the bus before or at the same time as any wheel failure.

This pattern of results is the same for all two-brake failures, i.e. every PAND will result in six MCSQs and every SAND will result in 27 MCSQs.

For the three-brake failures, the number of MCSQs increases dramatically due to the increased number of basic events (7 as opposed to 5 events in the two-brake failures) and there are 113 MCSQs for each three-brake failure. However, they still follow the same pattern: a failure of the bus before anything else, a simultaneous failure of three out of the four brakes before the bus fails, and the combinations of both. For example, for a simultaneous failure of FN, FO, and RN brakes, the MCSQ describing the bus failing first is:

```
BusCom | ActuatorComFN | SensorComFN | ActuatorComFO |
SensorComFO | ActuatorComRN | SensorComRN
```

and the MCSQ describing simultaneous failure of all three sensors would be:

```
SensorComFN & SensorComFO & SensorComRN | BusCom | ActuatorComFN
| ActuatorComFO | ActuatorComRN
```

The same problem applies to the simultaneous failure of all four brake actuators, for which there are even more basic events (nine) and thus even more MCSQs – 373, in fact[26].

5.3 Conclusion

The MCSQs produced by the temporal FTA show the possible ways in which all sequences of one, two, three, or four brake failures may occur. In an ordinary static FTA, although it would be possible to look at the different combinations of brake failures, it would not be possible to look at the sequences in which those combinations may occur and thus the opportunity for a deeper insight into the failure behaviour of the system is lost. The minimal cut sets for the two near-side brakes failing would simply be:

- ActuatorComFN . ActuatorComRN
- ActuatorComFN . SensorComRN
- SensorComFN . ActuatorComRN
- SensorComFN . SensorComRN
- BusCom

Because it is not possible to look at the sequences in which these failures may occur in a traditional FTA, the analyst may not consider the effects of those sequences on the dynamics of the car and could thus underestimate the potential severity. For example, a simultaneous failure of the rear brakes is less severe than a sequential failure of them, because a sequence may lead to the car yawing and ultimately colliding with oncoming traffic or going off the side of the road. By looking at only static fault trees, the possibility of sequential failures may not be evident and thus their potential effects not considered.

At the same time, a static fault tree may lead the designer to overestimate severity. For example, Table 7 shows that in all two-brake failure cases, a failure of the front brake before or at the same time as the failure of a rear brake is more severe than a failure of the rear brake first. This level of detail is not available in a static fault tree. If the designer wanted to improve the reliability and safety of the braking system, they would have to treat all brakes equally.

[26] Given by the sum of all possible combinations, minus n –1 (because only BusCom is a CCF). Thus:

$$sum = \sum_{k=1}^{n} \frac{n!}{k!(n-k)!} - (n-1)$$

However, based on the results of the temporal FTA, the designer could choose instead to add extra resilience only to the front brakes (e.g. by duplicating the sensors) compared to the rear brakes. This would be a cost effective way to gain an appreciable increase in safety and reliability because it focuses on the parts of the system that have the most severe effects when they fail.

Therefore, the results of the temporal FTA on the case study are interesting not so much because of the way they represent sequences, but rather because of the fact that they can represent sequences at all. This grants the analyst (and thus the system designer) a much finer-grained view of the failure behaviour of the system, particularly with regard to the effect and severity of the various system failures, even in a simplified system such as the braking system presented here. This can help the designer draw conclusions about the system that would not otherwise be possible with static FTA.

6 Evaluation

"The greatest of faults, I should say, is to be conscious of none."
- Thomas Carlyle

6.1 Objectives achieved

"In this world second thoughts, it seems, are best."
- Euripides, *Hippolytus*

To evaluate Pandora, it is necessary to compare what has been achieved thus far against the objectives set out in the **Introduction**. It is also interesting to compare the Pandora approach in more detail against some of the other temporal FTA approaches described in the **Background** chapter.

6.1.1 Objective I: Retain, as far as is possible, the simplicity of FTA by requiring only a minimum of additional, temporal data

The intention behind this objective is to ensure that Pandora does not become so over-complicated that it makes its use as part of a fault tree analysis too expensive or too difficult to undertake. Several steps were taken in an attempt to achieve this.

Firstly, the choice of a simple model of time means that Pandora only requires relative temporal information – specifically, whether an event occurs after, before, or at the same time as another event. There is no metric for time, nor is branching time modelled, both of which can lead to considerable additional complexity. Pandora therefore uses a relative, linear system of time in which there are only three possible temporal relations between two events: *before*, *after*, and *simultaneous*.

Secondly, the definition of events in Pandora continues this quest for simplicity by stating that only the *occurrence* of an event is important, and that, for the purposes of relative time, only the moment of that occurrence matters – i.e. the moment at which the event takes place or has an effect. By choosing only to model instants in Pandora's limited relative system of time, the issue of point-based versus interval-based time becomes moot: if events do not have a duration, then they cannot 'overlap', and if events are represented as single instants, then the most that can be said is that they either occur at the same moment or they do not. Thus the issue of any 'gap' between moments is irrelevant because it is not possible to say that two events are meant to be consecutive.

Thirdly, the definition of the temporal gates aids simplicity by ensuring that each gate represents one of the three temporal relations possible in Pandora. The PAND and SAND gates are 'primitive' gates in that they directly represent the *after/before* and *simultaneous* relations, while the POR gate, although semantically a little more complex, also represents the *before* relation[27]. The central issue that plagued the TAND connective, namely the requirement for events to occur consecutively, is absent: because it is not possible to define an interval – or lack thereof – for an event, it is not possible in Pandora to state that two events must occur consecutively. Because the temporal gates in Pandora are defined to be exclusive, i.e. they do not represent both *before/after* and *simultaneous*, there is no 'overlap' between the priority gates (representing *before/after*) and the simultaneous gate. This helps to remove the ambiguity surrounding the original PAND gate, which could be interpreted in either an inclusive or an exclusive manner.

The choice of the PAND gate as the foundation stone of Pandora is no coincidence: by using the original PAND gate – FTA's pre-existing solution to the problem of representing sequences in fault trees – as the prototype for Pandora's temporal gates, it is hoped that Pandora remains faithful to the principles of FTA. The newly redefined PAND gate in Pandora also solves many of the other problems inherent in the original PAND. The sequence of events is explicitly stated to be left to right, removing the requirement for a separate conditioning event, and the occurrence of multiple events at the same time or the same event used multiple times results in a contradiction (as stated by the Law of Simultaneity). The issues of contradictions are also explicitly handled by the temporal laws, as explained below.

Although it has been suggested that the SAND gate – and by extension, the *simultaneous* relation – is superfluous, given the almost non-existent probability of a simultaneous occurrence of two events under normal circumstances, the SAND does serve a useful and valuable purpose. Firstly, although improbable, the occurrence of two events simultaneously is not impossible, and if Pandora lacked the ability to represent this scenario, it would lead to a 'semantic hole' in the logic. The common way of filling this hole is to make the *before* and *after* relations inclusive, i.e. before would really mean 'before or at the same time as'. But this leads to an overlap between *before* and *after* and this in turn has implications for the handling of contradictions. Secondly, in situations where a dependency or shared trigger exists, the simultaneous occurrence of two events is very common. This is particularly true of intermediate

[27] Technically, the POR and the PAND can be said to represent both the *after* or the *before* relations, depending on how they are read; e.g. X PAND Y can be read as both X *before* Y and as Y *after* X. However, the POR better reflects the *before* relation as it only specifies that its priority event must occur before any other, and does not necessarily mean that any of the subsequent events must occur afterwards.

events (i.e. gates) in cases where the same event occurs as an input to more than one gate, as in these instances the gates can all become true at the same time if that shared event occurs.

Finally, because Pandora allows a very flexible representation of time, the definition of 'simultaneous' is dependent more on effect than on strict time constraints; thus two events can be said to occur simultaneously if their occurrence close together (if not *exactly* at the same moment) is sufficient to cause a different effect to the effect of their occurrence at more widely separated times. This is possible in systems with mitigation devices that can cope with multiple failures widely spaced but not two failures occurring close together, e.g. an emergency pump on a ship may be able to cope with the floodwater from one leak while the leak is plugged, but two leaks occurring close together may overwhelm the pump. Also, as the case study shows, a simultaneous occurrence of two similar failures can have a more or less severe effect than the sequential occurrence of failures. In such systems, the SAND provides a way of representing this distinction.

The disadvantage of such a simple, general approach is that it becomes difficult to express more specific scenarios, e.g. "X must occur within 10 seconds of Y". Pandora cannot represent such scenarios easily. Extra information can be added as part of the event itself (e.g. by specifying the interval duration as a separate conditioning event, like in the CSDM), e.g. "(X PAND Y) AND '*Y occurs within 10 seconds of X*'", but in this case the additional temporal information has no support within Pandora itself and is thus not taken into account during analysis.

Nevertheless, it is believed that this price is worth paying. Pandora was not designed to be able to represent every possible temporal constraint but rather to be simple enough to be able to be applied in a majority of situations. Because it only requires a little extra information – the relative order of events, as indicated by the temporal gates – it is also easy to use and to understand. The result is that it becomes possible to include temporal information in a fault tree without the synthesis process becoming unwieldy or excessively time consuming.

6.1.2 Objective II: Remain compatible with the existing fault tree structure by minimising the impact of any extensions

Fault tree analysis has become popular for many good reasons, not least its ease of use and the readily understandable nature of its simple logical structure. Pandora is designed to stay in keeping with the aesthetics and semantics of traditional fault trees to ensure that it too is as easy to use and as understandable as possible. As a result, only a small number of new gates were introduced, each with relatively simple semantics, to augment the fault tree.

One important factor, however, is the coherent nature of the fault tree: if a fault occurs, it never *improves* the functioning of a system, and so a system cannot repair itself through failure. The logical underpinning of fault tree coherency is the structure function of the fault tree, which must be non-decreasing.

Pandora is meant to preserve the coherency of the fault tree, and as such it does not incorporate NOT gates. However, the POR gate offers some of the capabilities of the NOT gate without compromising the coherency of the fault tree. It allows the fault tree to specify conditions that are true only if one event occurs and another event has *not* happened – yet. Because it accommodates the potential occurrence of the negated event later on, it does not affect coherency.

Similarly, the PAND and SAND gates are only true once *all* of their inputs are true (and in the case of the PAND, this means the right-most input). Because Pandora assumes that events are non-repairable and that once an event or gate is true it remains true, once all events have occurred, it is not possible for a gate – or any type of event – to 'unoccur'.

Although this is in keeping with the semantics of fault trees as described in the *Fault Tree Handbook*, it does not necessarily preclude the possibility of representing repairable events. It is possible to specify the repair of a fault using a separate event (e.g. "X occurs" and "X repaired"). However, this approach means there is no logical link between the occurrence and repair of a fault; furthermore, it is difficult to specify a meaningful link without breaking the coherency of the fault tree, e.g. a failure Z occurs as long as X and Y occur and X has not been repaired – if X is repaired, then Z should become false. It is possible to model failure without repair by careful use of temporal gates, especially POR gates, but at the cost of a larger and more complex fault tree; e.g. to specify that Z occurs, the expression $Z = (X.Y)|X^R$ would mean that Z occurs as long as X and Y both occur before X is repaired.

Pandora also minimises the impact on the Boolean logical structure of the fault tree by ensuring that the temporal gates and existing Boolean gates are interrelated; by using the Completion Laws, it is possible to express Boolean gates in terms of temporal gates, and there are more temporal laws that allow both types of gates to be readily mixed and manipulated together. This helps remove any apparent seams between the different types of gates and – not unimportantly – also makes qualitative analysis truly possible.

Finally, on a purely aesthetic level, the symbols for the temporal gates – both in text form (e.g. &, <) and in diagrammatical form – are designed to be similar and familiar to the existing Boolean gate symbols and thus blend in more successfully.

6.1.3 Objective III: Allow qualitative analysis of the extended fault trees, producing results similar to those already provided by FTA

Pandora is designed to focus specifically on qualitative analysis of temporal fault trees, an area which seems to be somewhat neglected in the field of FTA. However, it is meant to make qualitative analysis of temporal fault trees possible, not replace qualitative analysis with another technique that serves a similar purpose, and it should therefore be possible to extract the same type of information and conclusions from the results of a temporal qualitative FTA as from a normal qualitative FTA.

Temporal qualitative analysis in Pandora is made possible through the use of the Euripides and Archimedes algorithms. Euripides closely resembles existing Boolean qualitative FTA techniques like MOCUS and MICSUP in nature, albeit considerably more complicated, whilst Archimedes more closely resembles BDD-style or Markov chain approaches that use alternative representations of the fault tree to perform the analysis. In both cases, the results of the analysis are presented in the form of minimal cut sequences, analogous to the traditional minimal cut sets. The same conclusions can be drawn from examining MCSQs, except with the addition of the information about the sequence in which the events must occur to cause the top event.

Although both algorithms are capable of providing useful results, they both have significant drawbacks. Euripides is more efficient, as it is a deductive algorithm that manipulates the fault trees using a limited (but still considerable) set of temporal laws; however, although it can remove redundancies and contradictions from the results it produces, it cannot fully minimise them in terms of simple size due to its inability to handle more difficult Completion problems. Although this issue is not the case in every fault tree, when it occurs, it can result in an increased number of MCSQs, making it more difficult to draw appropriate conclusions from the results.

Archimedes is perhaps the more problematic algorithm as it offers the best results but also has the worst performance. As an inductive algorithm, it generates every possible sequence and combination to determine the minimal cut sequences. Although it does not necessarily evaluate every possibility, performance is still subject to the Fubini numbers and thus does not scale well. However, as long as the dependency trees are generated, the results produced are exactly minimal – solving the Completion problems that Euripides struggles with.

At present, neither of the two algorithms is considered to be sufficiently mature to be applied to non-trivial real world systems and both require further research to refine and improve them.

Nevertheless, they are sufficient to demonstrate the potential for qualitative analysis of temporal fault trees and show how Pandora can be used to yield significant benefits in terms of precision and detail when analysing dynamic systems.

6.1.4 Objective IV: Provide a temporal logic that underlies the extensions to support qualitative analysis

To support its three new temporal gates, Pandora also introduces a new temporal logic to provide semantics for them and to link the new gates with the existing Boolean gates. As such, it can be viewed as a kind of temporal extension to Boolean logic. In traditional terms, Pandora logic is a form of linear, non-metricated, propositional temporal logic.

The significant difference between Pandora logic and Boolean logic (and, indeed, other similar propositional temporal logics) is the fact that Pandora temporal expressions can be evaluated to produce sequence values, analogous to normal Boolean truth values. The sequence values indicate the relative order in which events (both basic and intermediate) occur. Sequence values are a simple and useful way to represent the semantics of Pandora logic as temporal logical expressions can be treated as a kind of arithmetic function.

Sequence values are also particularly valuable when evaluating temporal expressions programmatically. Because temporal significance – the property of an event that means it must occur in a particular order – is supplied by the temporal gates and is not inherent to the event itself (i.e. an event can occur in different orders in different parts of the fault tree), sequence values are easily evaluated by a computer by determining what sequences of events are possible for a given temporal expression; indeed, this principle is the basis for the Archimedes algorithm.

Finally, sequence values make it possible for Pandora logic to be displayed – and proven – as part of temporal truth tables, listing the possible sequences for a given set of events and showing the value of a temporal expression for each sequence. Temporal truth tables provide another link to traditional Boolean logic (and indeed Boolean truth tables can be thought of as a subset of TTTs) and more importantly are readily understandable by anyone with a familiarity with normal truth tables.

However, such a fundamental change to Boolean logic comes with a price: in particular, a large number of new temporal laws. As with Boolean laws, there are a potentially infinite number of temporal laws, depending on how many events are taken into account. Fortunately, most laws use only two or three events and larger laws (with the possible exception of the Completion laws) can always be represented in terms of these smaller laws. This means the number of laws required for qualitative analysis can be reduced to a more manageable size (see **Appendix II: Boolean & Temporal Laws**).

6.1.5 Objective V: Define a simple, unambiguous definition of the meaning of the temporal relationships between events to resolve the issue of contradictions

As mentioned earlier, the temporal logic in Pandora does not include NOT gates to ensure it remains coherent, but it does account for the potential occurrence of contradictions. Introducing the concept of sequence also introduces the concept of contradiction, because it is possible to specify conjunctions of mutually exclusive expressions (like $X < Y \,.\, Y < X$). Pandora provides a set of temporal laws to detect such contradictions and allows for the use of existing Boolean laws (i.e. $X.0 = 0$ and $X + 0 = X$) to remove contradictions from an expression. In particular, the laws of Mutual Exclusion and Simultaneity identify any contradictions inherent in a temporal expression by detecting violations of the three temporal relations in Pandora (only one of which can be true at any time).

This contradiction handling is a major advantage in helping to overcome the ambiguities and dependencies inherent in dynamic systems. It is also something conspicuously absent from other temporal fault tree analysis techniques.

6.2 Comparison with other temporal FTA techniques

"In case of dissension, never dare to judge till you've heard the other side."
- Euripides, *Heraclidae*

The choices made during the development of Pandora have often meant that Pandora differs quite considerably from other techniques. It is therefore worthwhile to compare Pandora with these other techniques to contrast their relative advantages and disadvantages. It is important to note that these techniques are not necessarily mutually exclusive and it may be possible to use more than one technique to make use of the best features of them all.

6.2.1 Dynamic Fault Trees

Dynamic Fault Trees are primarily designed for quantitative analysis while Pandora is primarily designed for qualitative analysis, so a direct comparison is problematic. However, it is still possible to make some interesting observations.

The three main questions regarding qualitative analysis using DFTs are:

- How to represent the DFT's new dynamic gates from a qualitative perspective?
- How to handle simultaneous events (if indeed are they handled at all)?
- How are reductions performed and how are contradictions handled?

These questions also have implications for the quantitative analysis of DFTs.

The dynamic gates used in DFTs are the PAND gate (based on the original form, albeit usually with a left-to-right sequence), the SEQUENCE gate (defining a sequence of events but not strictly speaking a gate)[28], the FDEP gate (which specifies a functional dependency between one or more events and a trigger event), and the SPARE gate (which comes in COLD, WARM, and HOT varieties). Each of these gates implies some sort of sequence.

The original PAND gate, as has already been discussed, suffers from a number of ambiguities. These problems are frequently ignored from the perspective of quantitative analysis, where they are not deemed to be as important. The issue of simultaneity, for example, is often ignored in a quantitative situation because the probability of two events occurring at the same time is usually

[28] The SEQUENCE gate is frequently omitted from descriptions of the DFT approach – including the *Fault Tree Handbook with Aerospace Applications.* It appears that the SEQUENCE is not a true gate in that, similar to the FDEP gate, it does not have an output (Coppit *et al*, 2000). Thus the PAND is normally used to define sequences.

sufficiently small to be discarded. However, this assumes the events are independent, which is not always the case – particularly for intermediate events. For example, (X + Y) PAND (X + Z) features a dependency – if X occurs first, then both sides of the PAND will be true simultaneously. This situation can be explicitly modelled in DFTs using a FDEP to represent the dependency, e.g. assume A is the trigger for both B and C in an FDEP gate, and elsewhere we have B PAND C. If A occurs, then B and C will occur simultaneously as they are both dependent on A. Hence the probability of B and C occurring simultaneously is, at minimum, the same as the probability of A occurring and therefore it is no longer a valid assumption to assume that the probability of simultaneous occurrence is small enough that it can be ignored.

The issue of simultaneity in the PAND gate (and by extension, the SEQUENCE gate) also has bearing on whether the PAND is to be inclusive or exclusive, and thus whether or not contradictions are possible. If the PAND is taken to be exclusive, then it does not include the simultaneous occurrence of its inputs and in the FDEP scenario given above, it would be false. If it is inclusive, then it does include simultaneity and in the FDEP scenario it would be true. In both cases, any probabilistic evaluation of PAND would need to take this into account: an exclusive PAND would have to subtract the probability of input events occurring at the same time while an inclusive PAND would have to include it.

The quantitative situation is complicated further by contradictions, e.g. if PAND is inclusive, then (X PAND Y).(Y PAND X) is not a contradiction – instead, it defines a simultaneous occurrence of X and Y. However, the probabilistic calculation would have to be different, i.e.:

$$P((X \text{ PAND } Y).(Y \text{ PAND } X)) \neq P(X \text{ PAND } Y) \times P(Y \text{ PAND } X)$$
$$P((X \text{ PAND } Y).(Y \text{ PAND } X)) = P(\text{simultaneous occurrence of } X \text{ and } Y)$$

which means that the usual probabilistic multiplication used for the AND gate would be invalid here. Similarly, if PAND is exclusive, the probability is 0 and definitely not the product of X PAND Y and Y PAND X. The same questions also apply in cases like X PAND X where the same event is used more than once as an input.

The semantics of the PAND gate can therefore have important implications regarding the probabilistic calculations needed to solve a DFT. As a result, in the DFT approach, the PAND has sometimes been redefined to avoid these problems. In Coppit *et al.* (2000), the PAND is defined to be inclusive and also to ignore repeated events. Thus the simultaneous occurrence of events is included in the PAND, and when the same input is used more than once, only its first appearance is used, i.e. X PAND Y PAND X is equivalent to X PAND Y and the second appearance of X is discounted. However, although this definition resolves the ambiguity, it still complicates the probabilistic calculations, as described above.

The FDEP gate is used to define a 'trigger' event that will simultaneously cause the occurrence of one or more other events (Coppit *et al.*, 2000). In qualitative terms, the FDEP represents a kind of logical implication, i.e. X FDEP Y means that if X is true, then Y will also be true, but if X is false, then Y can be either true or false. However, the FDEP is not a true gate in that it has no output; instead it serves purely as a horizontal link between multiple events, avoiding the regular hierarchy of the fault tree. This means the FDEP itself does not take part in the logical evaluation of the fault tree; instead, it simply means that one or more basic events have a potential antecedent cause (the trigger event) in addition to the possibility of independent occurrence. Thus the dependency modelled by the FDEP is functionally equivalent to an OR: X FDEP Y means that Y may occur as a result of X or simply by its own occurrence, i.e. X FDEP Y means X OR Y. If Y has already happened, then X has no effect (unless it also serves as the input to another fault tree gate). In Pandora terms, this could be described as X|Y + Y, but according to the Law of POR Transformation this is equivalent to X + Y anyway.

FDEP gates can also introduce circular logic, similar to the way PAND gates can introduce circular dependencies (although the FDEP does not result in contradictions). If X triggers Y and Y triggers Z and Z also triggers X, then the result is that the occurrence of any one of X, Y, or Z will trigger all of the others. Thus each of X, Y and Z would have to be equivalent X+Y+Z in logical terms.

SPARE gates also introduce a number of complexities in the DFT methodology, particularly when combined with the possibility of simultaneous events. For example, if two SPARE gates share the same standby, e.g. X SPARE Z and Y SPARE Z, where Z is the standby, then a simultaneous occurrence of X and Y means that both spares will try to activate Z. Since only one can succeed, it leads to a non-deterministic situation. Secondly, if the first SPARE was a cold SPARE and the second was a HOT spare, then Z would need two different probabilistic values as it would simultaneously be a dormant standby and an active standby. As a result, in Coppit *et al.* (2000) the common hot/cold SPARE gates are replaced in favour of a single SPARE gate which can only have basic events as inputs and in which no sequence of failure is defined (i.e. there is no difference between the primary and the standby events). The issue of non-determinism remains, however.

In qualitative terms, the SPARE gates (in both their original form and the constrained, redefined form) define a type of conjunction: if at least one standby remains operational, then the SPARE gate is false, and when all standby events fail, then the SPARE becomes true. It is not necessary to derive a sequence (e.g. primary first) in a normal SPARE because the order of failure does not matter. However, if two or more SPAREs share standby events, then the issue of non-determinism appears and it becomes more complex – which SPARE will fail if several try to use the same standby at the same time?

Thus the main DFT gates can be translated into Pandora equivalents as follows:

- The DFT's redefined PAND is roughly equivalent to Pandora's PAND and SAND, e.g. X PAND Y in a DFT would be the same as (X < Y) + (X & Y) in Pandora. However, the DFT PAND can have the same event occurring as multiple inputs, which means the conversion is not direct (the repeated inputs would have to be removed first).
- The FDEP gate is effectively equivalent to an OR. All triggered events would be replaced by an OR containing the trigger and the triggered events. However, circular logic would have to be resolved first.
- The SPARE gate is effectively equivalent to an AND. All standby events and the primary must fail for a SPARE gate to fail. However, the non-determinism problem for shared standby events means that this scenario – and the sharing of standby events – should be avoided in favour of more explicit modelling using PANDs, SANDs and PORs.

With these conversions it is possible to convert most DFTs into a Pandora-like fault tree and thus apply a form of qualitative analysis on it. This analysis would be able to deal with the issues of simultaneity more explicitly (due to the presence of the SAND) and would be able to produce equivalent MCSQs.

However, the various problems with the DFT gates described above show that it is difficult to develop methods for quantitative analysis in the absence of consideration for qualitative analysis, because doing so can lead to a number of logical ambiguities. By defining gates for use with a quantitative methodology first, the DFT approach has had to later define or redefine its gates to resolve the ambiguities and overcome the resultant problems. Pandora does not suffer the same problems as these ambiguities are addressed initially, potentially providing a sounder foundation for any future quantitative analysis methodology.

6.2.2 CSDM-style approach

CSDM and similar approaches like the Durational Calculus represent temporal information as part of an event rather part of a gate. This does not mean that they do not use a temporal logic – they do – but simply that the temporal logic is kept separate from the Boolean logic that comprises the structure of the fault tree. As a result, any qualitative analysis methodology is complicated by the fact that it must take into account two forms of logic in two separate places.

It is not even clear that such an analysis is meaningful, however. The temporal logics used in CSDM and Durational Calculus, for example, are primarily designed to represent *constraints*. Both approaches are mainly used to describe the specifications and requirements of real-time systems, in which the exact timing of events becomes critical. As a result, both feature a metricated, quantifiable form of time. Furthermore, particularly in CSDM, the notion of an event is different to the definition of an event in Pandora: in Pandora, events are persistent and instantaneous, whereas in CSDM, events are finite and can occur more than once. Pandora's events are closer in definition to the actions of CSDM, which represent individual instances of events, but even actions have start and end times, something which is difficult to represent with Pandora.

Pandora can be used to represent durations by defining events to represent start and end times. It is also possible to represent multiple durations the same way. Then the temporal gates can be used to specify that an event occurred within a particular interval, e.g. if START is the start event of the interval and END the end event, then X occurs within the interval if:

$$(\text{START} < \text{X} + \text{START} \ \& \ \text{X}).(\text{X} \mid \text{END} + \text{X} \ \& \ \text{END})$$

However, the most important difference between the CSDM-style approaches and Pandora is that those approaches can *measure* intervals. It is not possible in Pandora to say that an interval (or an event) lasts a certain amount of time. The best that can be done is to embed a time in the end event of an interval, e.g. if START is the start of an interval, then END would be "10 seconds elapsed since START", but this is separate from the temporal logic.

Thus performing a qualitative analysis using CSDM-style logic is difficult because it was not designed to be used in that way; similarly, trying to specify real-time constraints using Pandora is difficult and inexact because Pandora does not use a measurable system of time.

This does not mean that both types of approaches cannot complement each other. It is possible to use CSDM-style logic inside the basic events of Pandora to represent a finer-grained, real-time view of events that can be used in validation and verification, whilst Pandora can be used as the logic that joins those events, which would enable some form of preliminary qualitative analysis to take place. However, any conclusions drawn must take into account both types of information if a true understanding of the system is to take place; e.g. the failure of a fire-suppression system might be modelled using Pandora to represent the sequence of events (e.g. alarm fails before fire occurs) but CSDM to represent the occurrence of the fire (e.g. gas leak AND naked flame AND overlap for 10s).

An alternative approach would be to extend Pandora with quantifiable operators, i.e. to allow PAND, SAND, and optionally POR gates to be given durations. Thus X PAND10 Y might mean that X must occur 10 time units before Y, and X SAND30 Y might mean that X and Y must occur together for 30 seconds. This type of measurable interval would still not be part of the qualitative temporal logic, however, and would be purely descriptive.

6.2.3 TFT approach

The Temporal Fault Tree (TFT) approach is designed for fault diagnosis, and as such uses a more precise temporal logic than Pandora. TFT's PLTLP is a past-orientated logic that allows for the quantification of time (e.g. the WITHIN operator allows the analyst to specify that two events much occur within *n* instants). Pandora's temporal gates can be crudely represented by PLTLP, e.g. the PAND gate can be expressed by the SOMETIME-PAST operator: if the current time is the time at which the PAND became true (i.e. its sequence value), then X PAND Y means that X occurred SOMETIME-PAST before Y. The SAND can be represented by giving two events the same starting point, e.g. X occurred *n* instants previously and Y occurred *n* instants previously too. However, it is difficult to express the POR because it is inherently a future-oriented operator and can only really be used in a TFT context to describe situations where one event has occurred but another event has not (yet).

Qualitative and quantitative analysis are both possible with TFTs, but these analyses are meant to be carried out on a trace of a system operation to determine the cause of failure. The nature of the time model used is point-based, because it assumes that observations of the system are taken at regular times. This means that intervals can be defined as collections of points and that measuring the time between events is possible; this is not possible using Pandora.

Both techniques are developed in very different ways for very different purposes and it is difficult to see how they could be used together. Pandora does not offer the exact precision or the model of time needed to represent the trace of an operational system (just as it cannot easily express real-time constraints), whereas TFTs do; but TFTs do not offer the flexibility necessary to model more general sequences of events with unknown times of occurrence.

6.2.4 TAND connective

The TAND is a single logical connective designed to represent the situation where one event (or state) immediately follows another, i.e. it defines a sequence. In purpose it is similar to the PAND, but in practice it is very different. The TAND is meant to show how a sequence of states can lead to a failure, and X TAND Y specifies that X was true at first, then it stopped at the same time Y occurred, and then Y ceases some time later. The only interpretation in which this really makes sense is for X and Y to represent states, not the occurrence of events. This is fundamentally different to Pandora's approach of representing instantaneous occurrence of events with persistent effects.

However, the TAND-style state based approach offers one significant benefit not present in Pandora – the possibility of repeated and/or intermittent events. A sequence like X TAND Y TAND X is not impossible if X and Y are states, and the TAND can represent repeating sequences like this very easily. By contrast, Pandora cannot represent the repeated occurrence of the same event – it is, by definition, a contradiction in Pandora's logic, since it violates Simultaneity.

Despite this advantage, the TAND is very difficult to use in general. Because it is intended for use with states, it is ill-suited to the type of events normally found in fault trees, where events typically represent the occurrence of faults (not the *existence* of faults). If events represent states, then it has many semantic implications, e.g. for repairable systems and in terms of coherency and non-coherency. If an event can go from true to false multiple times, then a statement like X AND NOT Y can be true at some times and not at others.

Pandora, by contrast, is much better suited to representing non-repairable systems in which events represent failures with persistent effects and in which the sequence of occurrence is important. This type of scenario is something the TAND essentially cannot represent, because if an event cannot 'end' then the TAND is meaningless.

As a result, Pandora and the TAND are – to all intents and purposes – utterly incompatible.

6.2.5 Formal approaches

The other approaches described in Chapter **2** – i.e. the work of Bruns & Anderson, Schellhorn *et al.*, Xiang, and Güdemann – are based upon formalisations of the fault tree logic. Their intention is usually to produce a more rigorous semantics for fault trees for the purposes of a more accurate representation of system failure behaviour, and is not typically intended to produce a 'temporal' fault tree, particularly not for the purposes of qualitative analysis. Typically, the formalisation process focuses on the definition of an event rather than the introduction of new gates.

Bruns & Anderson attempt to solve some of the ambiguities in the *Fault Tree Handbook* by defining events as conditions having a duration (rather than being instantaneous) – thus making them functionally if not semantically equivalent to states – and then defining the AND as a simultaneous conjunction of these events, i.e. it is true if all input conditions are true. The fact that events have durations implies that an AND gate can later become false, and it also means that the AND gate will never be true if all of its inputs are true at different times, even if they all occur at least once. Temporal semantics are introduced using an *even* operator to represent a kind of sequence (that an event will be eventually true in the future) or a *prev* operator (to indicate that an event was true at some point in the past); however, these operators are semantic operators designed to formalise the meanings of the fault tree and are not meant to be used as explicit logical gates along the lines of AND and OR and Pandora's temporal gates. There is therefore no means of using these as part of an analysis. This approach is therefore only a formalisation of existing fault tree functionality for the purposes of verification and validation, and is not meant to extend fault trees with new temporal functionality; indeed, Bruns & Anderson explicitly acknowledge that if temporal semantics are introduced, then existing Boolean laws are insufficient to perform analysis.

The approach of Schellhorn *et al.* is another formalisation but one that makes explicit use of temporal logic – in this case, a hybrid form of ITL and Duration Calculus. Each gate is assigned temporal semantics, and the existing gates are split into decompositional gates and consequential gates. This distinction is made to separate the notion of causality from the notion of decomposition. Some gates – specifically the AND gate – are further divided into synchronous (i.e. simultaneous) and asynchronous (i.e. sequential) forms. In this approach, these advanced gates can be assigned their own semantics expressed in the ITL/Duration Calculus logic, e.g. to specify durations. Thus these gates are treated in some respects as 'temporal events' and not merely as logical operators, since, in order to be true, additional conditions must be fulfilled beyond the logical meaning of the gates. Again, however, this approach is not designed to support qualitative analysis and is meant to form a more rigorous description of system behaviour.

Xiang's approach is similar but defines events as being based on occurrence instead. New temporal operators are introduced, e.g. *next*, *eventually* to describe the occurrence of events in the discrete, linear model of time used. Again, however, the intention is to represent more precisely the behaviour of dynamic systems rather than to enable the analysis (qualitative or quantitative) of such systems. Furthermore, in this approach, the temporal semantics are represented by conditioning events rather than by gates.

In all three of these cases, the intention is to produce a more formal fault tree methodology with more precise semantics that can be used to verify a system specification. This is a very different goal from Pandora, which is comparatively informal and designed instead to extend fault trees to allow more advanced forms of analysis. The lack of quantifiers in Pandora's gates (meaning that precise durations cannot be expressed) and its strictly relative model of time mean that Pandora is not suitable for this sort of task; instead, it is deliberately imprecise to enable the fault tree to describe sequences of failures without having to specify exactly *when* those failures occur.

The work of Güdemann *et al.* on Deductive Failure Order Analysis (DFOA) is, however, somewhat different: this approach is designed with analysis in mind. It is meant to produce minimal cut sequences containing Pandora's PAND and SAND gates by using an inductive analysis technique to obtain normal cut sets and then applying an ordering to them on the basis of temporal semantics based on CTL* and LTL. Although the result is similar to Pandora, the process is very different. The primary difference is that DFOA is a two stage process – in the first stage, the combinations of events are produced, and in the second stage, the temporal semantics are added to the results. In Pandora, the temporal analysis takes place as part of the normal analysis – there is no distinction.

DFOA is still very new and the details about how it works are not yet fully clear, but it seems that DFOA serves as an alternative algorithm of obtaining Pandora-style minimal cut sequences, since DFOA uses the same gates (except currently lacking a POR). However, DFOA also makes use of classical temporal logics like CTL* and LTL to provide the full temporal semantics, which presumably introduces a much higher level of complexity compared to Pandora's very limited relative temporal logic.

6.2.6 Conclusions

The above approaches can be divided into four main categories, depending upon the purpose they were designed for:

• Analysis approaches	-	DFTs, TAND, DFOA, Pandora
• Real-Time Specification	-	CSDM, Duration Calculus
• Fault Diagnosis	-	TFTs
• Formal Verification	-	Schellhorn, Bruns & Anderson, Xiang

The category of the approach has the biggest impact upon what it contains and how it works. The specification and verification categories both provide very specific, detailed, and formal logics designed for precise representation of system behaviour; however, they tend not to be designed with analysis in mind and in most cases, there is no indication of how the resultant fault tree is to be analysed (either qualitatively or quantitatively). The TFT approach is designed for fault diagnosis - itself a form of analysis – and falls somewhere between the formal approaches and the analysis approaches in that it offers a comprehensive and precise temporal logic but that it also takes into account the possibility of analysis (albeit based on a trace of the system operation). The analysis approaches, by contrast, are designed to extend fault trees with temporal semantics for general usage and it is into this category that Pandora falls.

Of these approaches, Pandora bears the strongest similarity to the TAND approach in that it is designed to allow fault trees to represent and qualitatively analyse the sequence of events as part of the fault tree structure by introducing new temporal gates. However, the method used is very different, and thus Pandora's semantics are more similar to those of the DFT methodology in that its events represent occurrences rather than states and its gates impose sequential constraints on the order of those events. The DFOA approach actually uses Pandora gates in its results, but uses a more complex and formal style of temporal logic during the analysis (due to its use of DCCA – a formal model checking technique – as the engine for producing the results).

Pandora's temporal gates resolve many of the ambiguities present in the TAND and DFT gates by enabling the explicit representation of simultaneity and also by directly acknowledging the possibility of contradictions arising from the sequence of events. Its focus on qualitative analysis means that it is possible to analyse the temporal information contained within the logical structure of the fault tree, whereas other techniques (e.g. DFTs and to a certain degree also DFOA) must separate the logical and temporal information, typically performing a standard qualitative analysis first and then restoring the temporal information afterwards. On the other hand, both DFTs and the DFOA method are better equipped for quantitative analysis, either through the use of Markov chains in the case of DFTs or via model checking in DFOA. Quantitative analysis is not possible using the TAND.

Pandora is unique in providing a set of new laws that combine a temporal logic allowing sequences of events to be represented and the normal Boolean logic of the fault tree. In other approaches that introduce temporal gates, the new gates are generally treated separately.

6.3 Further Research

"Time will explain it all. He is a talker, and needs no questioning before he speaks."
- Euripides, *Aeolus*

Although Pandora's box has now been opened, the full contents have not yet been explored. There are many possible avenues for further research, and also a number of areas that would benefit from further development. The three main directions for further research are improvements to the algorithms and logic, quantitative analysis, and further investigation into possible applications of Pandora.

6.3.1 Improving Pandora

Pandora enables temporal qualitative analysis in two parts: by introducing new gates with a logic to support them and by suggesting algorithms that can be used to analyse them. Although the logic is relatively mature and unlikely to be extended further (e.g. with any more gates), it is possible that useful new laws remain to be discovered. One possible avenue worthy of further investigation – though admittedly unlikely to yield any simple results – is the inclusion of the NOT gate in Pandora. As explained earlier, the NOT gate is problematic, but a non-coherent variation of Pandora may allow for the analysis of a greater range of systems.

Alternatively, if Pandora's temporal logic is stable, then the spotlight falls upon the algorithms, which are in greater need of improvement. The two temporal qualitative FTA algorithms presented here, Euripides and Archimedes, both suffer from a number of shortcomings – particularly performance related difficulties. Currently, these problems mean that both algorithms are limited to small applications; using them to analyse a non-trivial fault tree of hundreds of basic events is not practical. Until a more efficient method is developed, Pandora's applicability to real world systems is limited.

Of the two approaches, it is Euripides that has perhaps the best chances of becoming a practical algorithm. Euripides uses the manipulation of Boolean and temporal laws to determine the minimal cut sequences and at present its main weakness is in dealing with Completion Problems. Further research may highlight more suitable methods of dealing with this problem, whether by using new or different laws or by finding a way to reduce the problem to a manageable size (e.g. in the same way doublets limit the number of laws needed during the rest of the analysis). It may be possible to find some way of recognising when elements of a

Completion law are present in order to be able to effect some additional reduction, e.g. in its simplest form, to recognise that X<Y + X&Y + Y<X are all the constituents of X.Y.

Archimedes has less scope for improvement because it is fundamentally limited by the size of its dependency trees. As an inductive technique, it must generate all possible cut sequences, and the number of cut sequences increases dramatically according to the Fubini numbers. Further research into Archimedes is likely therefore to focus on more efficient ways of applying it or identifying situations where it can be applied, e.g. by limiting the number of events it must deal with.

There may of course also be entirely different algorithms possible for use with Pandora. FTA itself has seen many new algorithms developed over its history, from simple Boolean-based techniques like MOCUS to modern approaches like the BDD method. It is not impossible, therefore, that more radical new techniques may be found.

6.3.2 Quantitative Analysis

Qualitative analysis is only one half of FTA: to be truly useful, quantitative analysis should also be possible. While it has proven valuable to focus on qualitative analysis first, thereby dealing with the problems and ambiguities that may arise from introducing time to the logical structure of the fault tree, there is currently no method for quantitative analysis to take advantage of this solid foundation.

This is one area where combination with or at least inspiration from other temporal FTA techniques like DFTs may prove useful. Other temporal FTA techniques generally focus on quantitative analysis rather than qualitative analysis and this has yielded a number of different methods. In particular, it may be possible to combine Pandora's temporal gates with the Markov chains used in DFTs; alternatively, more traditional calculation methods such as the PAQ or SFL methods may prove more effective.

The first step in providing quantitative analysis capabilities in Pandora is to find a way to quantify each of the temporal gates. As explained earlier, quantifying simultaneous and sequential events is not necessarily a simple matter and can be highly dependent on context.

6.3.3 Applications of Pandora

Although Pandora is designed to be used with dynamic systems, currently only non-repairable persistent failures are modelled, and this can sometimes limit the types of systems Pandora can analyse. Extending Pandora with the capability of modelling additional types of failure could be a fruitful avenue. It may also be possible to use Pandora in different ways or to analyse systems in conjunction with other techniques.

For example, many dynamic systems are modelled using states and transitions – currently something that fault trees cannot adequately represent (in their standard form, at least). In particular, it is not sufficient to model a path through a state transition diagram as a conjunction of states as transiting through the same states but in a different order may result in a different end state. Pandora may be one way of extending fault trees to represent states in such systems.

However, if Pandora is to be used to model states and transitions, then it may require changes or at least additions to the semantics of events and gates in Pandora. Currently events are persistent – they do not end. States are not persistent and the system can transition from one state to another. In this case, perhaps the lessons learnt from the TAND connective may prove useful in defining a variation of Pandora that can cope with states and non-persistent events without also introducing the ambiguities and inconsistencies experienced by the TAND. It may be a sufficient abstraction to assume that state-transition events are mutually exclusive and that the occurrence of one event means that the previous state has come to an end, in which case $X<Y$ would be more or less equivalent to $X \Pi Y$. The full ramifications of such alterations must be studied in greater detail first, however.

Another likely direction of research for Pandora is investigating how it can be combined with automated fault tree synthesis & analysis techniques like HiP-HOPS (Hierarchically-Performed Hazard Origin & Propagation Studies) (Papadopoulos *et al.*, 2001). This would mean enabling a system model to be annotated with Pandora failure logic and then synthesising and analysing temporal fault trees from this data.

As an example of how Pandora may prove useful for other safety analysis techniques, Pandora has already been used as part of a temporal/combinatorial FMEA (Failure Modes and Effects Analysis), e.g. in Walker *et al.*, 2009; this builds upon earlier work on combinatorial FMEA (Parker *et al.*, 2006) and combines it with Pandora to enable the production of FMEA tables that contain not only the effects of single failures but also the effects of multiple failures as well as sequences of (or simultaneous) failures.

In addition to the above, it would also be valuable simply to apply Pandora to a greater range of dynamic systems; the problems encountered and lessons learnt would no doubt reveal new areas of possible further research for Pandora.

Finally, Pandora may also have other applications entirely, in the same way Boolean logic is not exclusive to fault trees. It could find wider application in fields beyond safety analysis, e.g. in areas where Boolean logic is used currently to represent and solve problems but where there is a need to move beyond a combinatorial model of analysis, such as electronic design.

7 Conclusion

"Success consists of going from failure to failure without loss of enthusiasm."
- Winston Churchill

7.1 Opening Pandora's Box

Over its long history, Fault Tree Analysis has proven to be a useful and valuable safety analysis technique; it allows a deeper examination of the failure behaviour of a system and the results it produces are a vital aid in helping to ensure that system failures are fully recognised and either anticipated or prevented.

However, FTA's long history makes it all the more surprising that one of its central deficiencies – its inability to model the effects of *sequences* of failures – has gone uncorrected for so long, particularly given the increasingly complex dynamic behaviour exhibited by modern systems. Such systems frequently employ techniques such as monitoring, standby components, and automated recovery to improve reliability and safety and reduce the probability and severity of failure, but by doing so they also introduce sequence-dependent behaviour that cannot be modelled accurately in a fault tree.

Although there have been other attempts to remedy this problem, such as those described in Chapter **2**, virtually all have focused solely on quantitative analysis and have ignored or omitted the possibility of temporal qualitative analysis. This is often because it is thought that temporal quantitative analysis alone is sufficient and that the existing qualitative analysis capabilities of fault trees can be stretched or generalised to cover dynamic situations, but this is not always the case: a prior qualitative analysis will often reveal information that a quantitative analysis alone cannot disclose, particularly in terms of contradictions and simultaneous events. Introducing sequences of events also inevitably introduces the possibility of mutually exclusive sequences and thus contradictions, which must be avoided or dealt with appropriately, and if it is possible for events to occur in a certain sequence then it is also possible for events to occur simultaneously, particularly if those events share a common cause or dependency. Techniques that cater only for quantitative analysis often do not consider such possibilities at all and this can adversely affect or even invalidate the quantitative results, as pointed out in Chapter **6**.

Therefore there is a clear omission in the vocabulary of fault tree analysis that limits its efficacy and applicability, and it has hitherto received surprisingly little attention. Fortunately, this omission in FTA is easily remedied by introducing a technique for temporal qualitative analysis of dynamic fault trees. Pandora is designed to accomplish exactly that: it extends fault trees with new temporal capabilities to represent events that must occur in sequence or simultaneously in order to cause a further fault, and it provides a temporal logic to support these gates and allow qualitative analysis to take place. It is this fundamental temporal logic underlying its gates that makes Pandora unique amongst temporal fault tree analysis techniques, whether quantitative or qualitative.

7.2 What emerged from Pandora's Box

The first part of the solution is to represent temporal information as an integral part of the fault tree, and the second part is to provide a method to enable that information to be analysed. Both halves of the solution depend on the formulation of an appropriate temporal logic.

Although the idea of a temporal logic is far from new, most classical temporal logics are difficult to use with fault trees. Classical temporal logics are designed to formally reason about knowledge of time, and as such are able to take into account *when* something is true; however, if the only requirement is to represent the relative order or sequence of a series of events, then these logics are often overpowered and inefficient: most of their expressive power is wasted and it is comparatively awkward to describe something as simple as 'X occurs before Y'.

There are some temporal logics introduced specifically for use with fault trees, but these often experience problems of their own. Fault trees have commonly understood semantics (though there seems to be no shortage of possible formalisms for those semantics) and if it is to be used in a fault tree successfully, any temporal logic should attempt to be compatible with those semantics. Temporal logics that introduce states or differentiate between decomposition and causation complicate the semantics and force the fault tree analyst to think in particular – and potentially unfamiliar – ways.

Pandora introduces a new logic that is designed to follow the traditional semantics of the fault tree by modelling events as occurrences of faults and basing its gates on the staple Boolean gates that comprise the core of standard fault tree logic. Events in Pandora do not require durations or complex temporal relations to describe overlaps or intervals, and Pandora's gates do not represent instances of a complicated classical temporal logic; instead, Pandora's PAND, SAND, and POR gates represent only the simple and easily understood relative temporal order between events.

The PAND gate itself is based on an original fault tree gate but updates its definition to avoid the problems that have prevented that gate from being used effectively in the past. PAND represents the 'before' or 'after' relation – the simplest and most fundamental temporal relation possible. As such, it forms the core of Pandora and is the basis from which all else has developed. The related POR gate is a development of the PAND and better expresses the notion of priority: that one event takes precedence over others and must occur first. As such, the POR gate can be used as a temporal version of the Boolean NOT gate but without the associated cost of introducing non-coherence to the fault tree.

However, if event X does not occur after event Y, and it does not occur before event Y, but both X and Y occur, then another temporal relation must be true – that of simultaneity. The SAND gate represents this case explicitly. Other techniques frequently choose to ignore simultaneity, dismissing it as unnecessary on the basis of probability, but this is a flawed decision based on the assumption of independence of events. If two or more events share a common cause, then simultaneity is not only possible, it is probable; furthermore, explicitly modelling simultaneity helps create a balanced temporal logic by avoiding ambiguities and ensuring that the three basic temporal relations are consistent and mutually exclusive – in Pandora, either two events occur at the same time, or one must occur before the other.

Underlying these three simple gates and the basic temporal relations are a system of temporal laws – laws of logic that represent equivalence between different logical expressions. These temporal laws allow Pandora to manipulate and simplify temporal expressions, highlight any contradictions that may arise from the introduction of sequences, and perhaps most importantly, link the new temporal gates to the existing Boolean gates. This latter task is accomplished by the Completion Laws, the jewels in Pandora's crown, which demonstrate how conjunctions and disjunctions and even single events can be expressed as a set of temporal relations.

These temporal laws are also what make temporal qualitative analysis possible in Pandora. In the same way Boolean laws can be used to reduce a traditional fault tree, temporal laws can be used to reduce a temporal fault tree by identifying instances of the three types of logical redundancy: absorption, contradiction, and completion. Absorption is the removal of redundant collections or sequences of events from the results, contradiction involves the removal of sequences that are impossible, and completion allows multiple sequences of events to be converted into a simplified equivalent form.

However, before temporal laws can be used, they have to be proved to be valid. Fortunately, just as all Boolean laws can be proven by means of a truth table expressing every combination of truth values, all temporal laws can be proven by means of temporal truth tables that express every possible sequence of events. This introduces the associated concept of a sequence value, analogous to a truth value, which describes the relative order in which events occur. All gates and events can be evaluated in terms of their sequence values, allowing them to be used as part of a temporal truth table. All possible sequences of events can be efficiently generated by means of a precedence tree, which takes the form of a branching timeline in which time continues with new events occurring at each node until all events occur at the leaf nodes. The contents of the precedence tree can then be used to populate the temporal truth table, and each row is assigned a different but unique set of sequence values. If these sequence values of two expressions are identical on every row of their temporal truth table, then those two expressions are proved to be equivalent.

There are, however, many possible temporal laws and it would not be possible to use them all as part of an analysis. To reduce the number of laws necessary for a qualitative analysis, Pandora introduces the idea of doublets – an encapsulation of a single temporal relation between two events, allowing temporal relations to be treated atomically as with any other event. Doublets greatly facilitate qualitative analysis by reducing the number of temporal laws needed for reduction and manipulation, and they better represent the many possible temporal relations between the basic events involved in a typical temporal analysis.

The analysis of a temporal fault tree by means of doublets and applying temporal laws is called Euripides, but this is only one possible method; it is also possible to make use of a dependency tree, which shows how one possible sequence can be part of and therefore made redundant by another sequence or combination. In one sense a dependency tree is the graphical representation of the Completion Laws, and as such the dependency tree is an ideal way of solving the Completion Problem – identifying all the possible sequences that comprise a given expression and reducing them to their simplest form. This form of analysis is called Archimedes. The two algorithms are not mutually exclusive and indeed both can successfully be used in concert to obtain the minimal cut sequences that are capable of causing the top system failure event of the fault tree.

These minimal cut sequences then form the basis for the decisions the analyst makes about the system; they may highlight weak points in the system, e.g. places where only one or two events are required to cause the system to fail, or may suggest strategies for preventing or mitigating failures, e.g. if one sequence of events can lead to a critical failure, then preventing that sequence – perhaps by ensuring that events fail in a different order or simultaneously – may lead to a lesser failure or even no failure at all. The different criticality that can result from different sequences of events in particular is highlighted in the case study presented earlier.

Pandora therefore allows fault trees to incorporate temporal information and it then allows this information to be subjected to a temporal analysis; the results of that analysis can provide a better insight into how a dynamic system may fail or react to failure. It improves on existing qualitative analysis because it can produce more accurate results that avoid unnecessary optimism or pessimism and furthermore can represent situations that simply cannot be modelled correctly using only traditional static techniques.

7.3 What remains in the Box

Although Pandora now makes temporal qualitative analysis possible, this is only the beginning. As explained in Chapter **6**, the Evaluation chapter, Pandora can be improved and expanded in many directions. In particular, to enable a full FTA, it should also be possible to conduct a quantitative analysis and this is one major avenue for further research with Pandora. Another promising avenue is the use of Pandora for modelling states and transitions in dynamic systems. Finally, further investigation and enhancement of the algorithms and laws used in Pandora would no doubt prove to be a fruitful line of enquiry, particularly if Pandora is ever to be used effectively in real-world systems.

But those are ideas for another day, and now it is time to concentrate on the present. Temporal qualitative analysis is a virtually unexplored field, and those pioneers who have ventured into it have either given it a low priority and consequently achieved little or alternatively achieved their goals only by altering the meaning of fault trees to evade the problems they encountered. Pandora was designed from the start to retain as many of the good qualities of FTA as possible – particularly its ease of use and simple logical structure. Pandora only requires the analyst to consider the relative order of occurrence of events and does not require any timings or intervals or states; by adopting this simple philosophy it minimises the necessary additions and retains the logical coherence of fault trees without compromising its ability to express dynamic behaviour. Pandora therefore offers new capabilities for the representation and analysis of temporal sequences of events that have hitherto been at best difficult and at worst impossible, and it is these capabilities that make Pandora unique in the pantheon of FTA.

References

"I begin by taking. I shall find scholars later to demonstrate my perfect right."
- Euripides, *Suppliants*

1. ADAMYAN A, HE D. 2002. "Analysis of sequential failures for assessment of reliability and safety of manufacturing systems." *Reliability Engineering & System Safety*, Vol. 76, 227-236

2. ALLEN J.F. 1983. "Maintaining knowledge about temporal intervals*." Communications of the ACM*, Issue 26 Vol. 11, pages 832-843, November 1983.

3. ALUR R., HENZINGER T.A. 1991. "Logics and Models of Real Time: A Survey." *In Real-Time: Theory in Practice*, REX Workshop, LNCS 600, pp. 74-106, 1991.

4. ALUR R., HENZINGER T.A. 1992. "A Really Temporal Logic." *Journal of the ACM* 41, 1994, p181-204.

5. ALUR R., HENZINGER T.A. 1993. "Real-time logics: complexity and expressiveness." *Information and Computation* 104(1):35-77, 1993

6. ALUR R., HENZINGER T.A., KUPFERMAN O. 1997. "Alternating-time Temporal Logic." *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science* (FOCS '97). p100-109.

7. ALUR R., ETESSAMI K., MADHUSUDAN P. 2004. "A Temporal Logic of Nested Calls and Returns." *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS), March 29 - April 2, 2004 Barcelona, Spain.

8. ALUR R., ARENAS M., BARCELO P., ETASSAMI K., IMMERMAN N., LIBKIN L. 2007. "First-order and temporal logics for nested words*." 22nd IEEE Symposium on Logic in Computer Science*, Wroclaw, Poland, 2007.

9. AMARI S., DILL G., HOWALD E. 2003. "A new approach to solve dynamic fault trees." In *Proceedings of the Annual Reliability and Maintainability Symposium*, 2003. ISBN 0-7803-7717-6, pp 374-379.

10. ANDREWS J.D. 2000. "To not or not to not." *Proceedings of the 18th International System Safety Conference*, Fort Worth, Sept 2000. pp 267-275

11. ASSAF T., DUGAN J.B. 2003. "Diagnostic Expert Systems from Dynamic Fault Trees." *Proceedings of the Annual Reliability and Maintainability Symposium*, Jan 2004. Los Angeles, USA. pp 444-450.

12. BARTLETT L.M., and ANDREWS J.D. 2000. "An ordering heuristic to develop the binary decision diagram based on structural importance." *Reliability Engineering & System Safety*, Vol 72 (2001), pages 31-38.

13. BELLINI P., MATTOLINI R., NESI, P. 2000. "Temporal Logics for Real-Time System Specification." *ACM Computing Surveys*, Vol 32, No 1, March 2000.

14. BOUDALI H., CROUZEN P., STOELINGA M. 2007. "Dynamic Fault Tree Analysis using Input/Output Interactive Markov Chains." *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks*, pp 708-717. IEEE Computer Society, USA. ISBN: 0-7695-2855-4

15. BOUDALI H., CROUZEN P., STOELINGA M.I.A. 2007 "A compositional semantics for Dynamic Fault Trees in terms of Interactive Markov Chains." In *5th International Symposium on Automated Technology for Verification and Analysis* (ATVA'07), October 22-25, 2007, Tokyo, Japan. pp. 441-456. Lecture Notes in Computer Science 4762. Springer. ISSN 0302-9743 ISBN 978-3-540-75595-1

16. BOUISSOU M., BON J-L. 2003. "A new formalism that combines advantages of fault trees and Markov models: Boolean logic driven Markov processes*." Reliability Engineering & System Safety*, Vol 82, pp 149-163.

17. BRUNS G. and ANDERSON S. 1993. "Validating Safety Models with Fault Trees." In *Proceedings of the 12th International Conference on Computer Safety*, Reliability, and Security, Janusz Górski, editor, pages 21-30. Springer-Verlag, 1993

18. BUCCI P., KIRSCHENBAUM J., MANGAN L.A., ALDEMIR T., SMITH C., WOOD T. 2008. "Construction of event tree/fault tree models from a Markov approach to dynamic system reliability." *Reliability Engineering & System Safety*, Vol 93, pp 1616-1627.

19. BUCHACKER K. "Modeling with extended fault trees." *Fifth IEEE International Symposium on High Assurance Systems Engineering* (HASE'00), Nov 15-17, 2000, Alberquerque, New Mexico, p 238.

20. CHOMICKI J., TOMAN D. 1997. "Temporal Logic in Information Systems." *Basic Research In Computer Science* (BRICS) Lecture Series LS-97-1, ISSN 1395-2048, November 1997.

21. CLEUGH M.F. 1937. *Time*. Methuen & Co Ltd, London, UK.

22. CODETTA-RAITERI D. 2005. *Extended Fault Trees Analysis supported by Stochastic Petri Nets.* PhD Thesis, Dipartimento di Informatica, Università degli Studi di Torino, Italy.

23. COPPIT D., SULLIVAN K.J, DUGAN J.B. 2000. "Formal Semantics of Models of Computational Engineering: A Case Study on Dynamic Fault Trees." *Proceedings of the 11th International Symposium on Software Reliability Engineering*, San Jose, California, USA, 8-11 Oct 2000 (ISSRE 2000). ISBN: 0-7695-0807-3. pp 270-282

24. COPPIT D. 2003. *Engineering Modeling and Analysis: Sound Methods and Effective Tools*. PhD Thesis, University of Virginia, USA.

25. DUGAN J.B., VENKATARAMAN B., GULATI R. 1997. "DIFTree: A software package for the analysis of dynamic fault tree models." *Proceedings of the Annual Reliability and Maintainability Symposium,* 13-16 Jan 1997, USA, pp 64-70.

26. DUGAN J.B., SULLIVAN K.J., COPPIT D. 2000. "Developing a Low-Cost, High-Quality Software Tool for Fault Tree Analysis." *IEEE Transactions on Reliability*, Vol 49 Issue 1, Mar 2000, ISSN 0018-9529. pp 49-59.

27. DUGAN J.B., ASSAF T. 2001. "Dynamic Fault Tree Analysis of a Reconfigurable Software System." *19th International System Safety Conference*, Huntsville, Alabama, USA. Sept 2001.

28. DUTUIT Y., and RAUZY A. 1996. "A linear-time algorithm to find modules of fault trees." *IEEE Transactions on Reliability*, Sept 1996, Volume R-45/3, pp 422-425.

29. EMERSON E. A. 1990. "Temporal and Modal Logic." *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics* 1990, J. van Leeuwen, ed., North-Holland Pub. Co./MIT Press, Pages 995-1072.

30. ERICSON C. 1999. "Fault Tree Analysis – A History." *Proceedings of the 17th International System Safety Conference* 1999.

31. FUSSEL J.B., VESELY W.E., 1972. "A new methodology for obtaining cut sets for fault trees." *Transactions of the American Nuclear Society*, vol 15, p262-263.

32. FUSSEL J.B., ABER E.F., RAHL R.G. 1976. "On the quantitative analysis of Priority-AND failure logic." *IEEE Transactions on Reliability*, 1976, Volume R-25/5, pp 324-326.

33. GALTON A. 2003. "Temporal Logic." [Online]. Stanford Encyclopaedia of Philosophy, last modified 11th Dec 2003. Accessed at http://plato.stanford.edu/entries/logic-temporal/ in Jan 2004.

34. GIRARD J.Y. 1987. "Linear Logic." *Theoretical Computer Science*, Vol 50 Issue 1, M. Sintzoff (ed.). North-Holland, Netherlands, 1987. ISSN 0304-3975. p 1-102.

35. GORSKI J. 1994. "Extending safety analysis techniques with formal semantics." *In Technology and Assessment of Safety Critical Systems*, pages 147-163, Springer-Verlag, London, 1994. F.J Redmill and T. Anderson (eds.).

36. GORSKI J., WARDZINSKI A. 1996. "Deriving Real-Time Requirements for Software from Safety Analysis." *Proc. 8th Euromicro Workshop on Real-Time Systems*, IEEE CS Press (1996) 9-14.

37. GORSKI J., WARDZINSKI A. 1997. "Timing aspects of fault tree analysis of safety critical systems." *Proceedings of the Safety Critical Systems Symposium*, Brighton, UK, Feb 1997. pp 231-244.

38. GULATI R., DUGAN J.B. 1997. "A Modular Approach for Analyzing Static and Dynamic Fault Trees." *Proceedings of the Annual Reliability and Maintainability Symposium*, Philadelphia, Pennsylvania, 13-16 Jan 1997, pp 57-63.

39. GÜDEMANN M., ORTMEIER F., REIF W. 2008. "Computing Ordered Minimal Critical Sets." *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems* (FORMS / FORMAT 2008) (eds. G. Tarnai & E. Schnieder)

40. HANSEN K.M., RAVN A.P, STAVRIDOU V. 1998. "From Safety Analysis to Software Requirements." *IEEE Transactions on Software Engineering*, Vol 24, No 7, July 1998 p573.

41. HUANG C-Y., CHANG Y-R. 2006. "An improved decomposition scheme for assessing the reliability of embedded systems by using dynamic fault trees." *Reliability Engineering & System Safety*, Vol 92, pp 1403-1412.

42. ISOGRAPH SOFTWARE, 2002. *Fault Tree+ v10.1*. Software tool (http://www.isograph-software.com/index.htm)

43. KAISER B., GRAMLICH C., FÖRSTER M. 2007. "State/event fault trees – A safety analysis method for software controlled systems." *Reliability Engineering & System Safety*, vol 92, pp 1521-1537.

44. KESTEN Y., MANNA Z., McGUIRE H., PNUELI A. 1993. "A Decision Algorithm for Full Propositional Temporal Logic." *Proceedings of the 5th International Conference on Computer Aided Verification* (CAV '93), volume 697 of Lecture Notes in Computer Science, Spinger-Verlag. 1993

45. LAMPORT L. 1980. "'Sometime' is sometimes 'Not Never.'" *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN, Jan 1980.

46. LAMPORT L. 1983. "What good is temporal logic?" *Information Processing* 83 (R.E.A Mason, ed.). Elsevier Science, 1983. p657-668.

47. LAMPORT L. 1994. "The Temporal Logic of Actions." *ACM Transactions on Programming Languages and Systems* 16:3. May 1994, p872-923.

48. LAMPORT L. 1994. "Introduction to TLA." SRC Technical Note 1994-001, Dec 16, 1994. Digital Systems Research Center, Palo Alto, California, USA, 1997.

49. LICHTENSTEIN O., PNUELI A. 2000. "Propositional Temporal Logics: Decidability and Completeness." *International Journal of the Interest Group in Pure and Applied Logic* (IGPL), Vol 8 No. 1, pp 55-85, Oxford University Press, UK, 2000.

50. LIU D., ZHANG C., XING W., LI R., LI H. 2007. "Quantification of Cut Sequence Set for Fault Tree Analysis." *High Performance Computing Conference* (HPCC) 2007, LNCS 4782, pp 755-765, Springer-Verlag, Berlin, 2007.

51. LONG W, SATO Y. 1998. "A Comparison between probabilistic models for quantification of Priority-AND gates." *Proceedings of the PSAM-4*, 2, p1215-20.

52. LONG W., SATO Y., HORIGOME M. 1999. "Quantification of sequential failure logic for fault tree analysis." *Reliability Engineering & System Safety* Vol 67 (2000), pages 269-274.

53. MANIAN R., COPPIT D., SULLIVAN K.J., DUGAN J.B. 1999. "Bridging the gap between Fault Tree Analysis Modelling Tools and the Systems being Modelled." *Proceedings of the Annual Reliability and Maintainability Symposium*, 1999. pp 105-111.

54. MANIAN R., DUGAN J.B., COPPIT D., SULLIVAN K.J. 1998. "Combining various solution techniques for dynamic fault tree analysis of computer systems." *Third IEEE International High-Assurance Systems Engineering Symposium*, Nov 13-14 1998, Washington DC, pp 21-28.

55. McARTHUR, R.P. 1976. *Tense Logic*. Reidel Publishing, Dordrecht, Holland.

56. MERLE G., ROUSSEL J-M. 2007. "Algebraic modelling of temporal fault trees." *DCDS'07*, 1st IFAC Workshop on Dependable Control of Discrete Systems, June 2007.

57. MESHKAT L., DUGAN J.B., ANDREWS J.D. 2002. "Dependability analysis of safety systems with on-demand and dynamic failure modes." *IEEE Transactions on Reliability*, Vol 51 Issue 2, Jun 2002. ISSN: 0018-9529. pp240-251.

58. NORRIS J.R., *Markov Chains*. Cambridge University Press, 1997, reprinted 2006, USA.

59. ON-LINE ENCYCLOPEDIA OF INTEGER SEQUENCES. *A000670: Number of preferential arrangements of n labeled elements; or number of weak orders on n labeled elements.* [online] Available at: http://www.research.att.com/~njas/sequences/A000670 [Accessed April 2007]

60. ON-LINE ENCYCLOPEDIA OF INTEGER SEQUENCES. *A034172 Nearest integer to n!/(2*log(2)^(n+1)).* [online] Available at: http://www.research.att.com/~njas/sequences/A034172 [Accessed April 2007]

61. ORTMEIER F., REIF W., SCHELLHORN G. 2005. "Deductive Cause-Consequence Analysis (DCCA*)." Proceedings of the 16th IFAC World Congress* Elsevier Jun-2006 ISBN: 978-0-08-045108-4 and 0-08-045108-X

62. ORTMEIER F., BALSER M., DUNETS A., BÄUMLER S. 2008. *Embedding CTL* in an Extension to Interval Temporal Logic (ITL)*. Technical Report, Institute of Computer Science, University of Augsburg, October 2008

63. OU Y., DUGAN J.B. 2000. "Sensitivity Analysis of Modular Dynamic Fault Trees." *Proceedings of the Computer Performance and Dependability Symposium* (IPDS'00), 2000, pp 35 – 43.

64. PALSHIKAR G.K. 2001. "Temporal Fault Trees." *Information and Software Technology #44* (2002), pages 137-150.

65. PAPADOPOULOS Y., McDERMID J.A., SASSE R., HEINER G. 2001. "Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure." *Reliability Engineering & System Safety* Vol 71 (2001), pp 229-247.

66. PARKER D., WALKER M., PAPADOPOULOS Y., GRANTE C. 2006 "Component-Based, Automated FMEA of Advanced Active Safety Systems." *FISITA'06*, 31st World Automotive Congress, Yokohama, Published by JSAE, ISBN: 4-915219-83-6, 2006.

67. PNUELI, A. 1977. "The temporal logic of programs." *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press. USA. pp 46-57.

68. PRIOR, A. 1957. *Time and Modality*. Clarendon Press, Oxford.

69. PRIOR, A. 1967. *Past, Present and Future*. Clarendon Press, Oxford.

70. PRIOR, A. 1969. *Papers on Time and Tense*. Clarendon Press, Oxford.

71. RESCHER N., URQUHART A. 1971 *Temporal Logic*. Springer-Verlag/Wien, Austria. ISBN 0-387-80995-3

72. SCHELLHORN G., THUMS A., REIF W. 2002. "Formal Fault Tree Semantics." *Integrated Design and Process Technology*, IDPT-2002, USA, June 2002.

73. SEMANDERES S N. 1971. "ELRAFT: A computer program for the Efficient Logic Reduction Analysis of Fault Trees." *IEEE Transactions on Nuclear Science* Vol18/1 pp 481-487.

74. SHARVIA, S., PAPADOPOULOS Y.I. 2008. "Non-coherent modelling in Compositional Fault Tree Analysis." *17th World Congress, International Federation of Automatic Control (IFAC)*, Seoul, Korea. July 2008.

75. SINNAMON R.M. and ANDREWS J.D. 1996. "New approaches to evaluating fault trees." *Reliability Engineering and System Safety* 58 (1997), pages 89-96.

76. SULLIVAN K., DUGAN J., COPPIT D. 1999 "The Galileo Fault Tree Analysis Tool." *Proceedings of IEEE International Symposium of Fault Tolerant Computing*, FTC-29, June 1999, pp 232-235.

77. TANG Z., DUGAN J.B. 2004. "An Integrated Method for Incorporating Common Cause Failures in System Analysis." *Proceedings of the Annual Reliability and Maintainability Symposium* (RAMS), 26-29 Jan 2004, pp 610-614. ISBN: 0-7803-8215-3.

78. TANG Z., DUGAN J.B. 2004. "Minimal cut set/sequence generation for dynamic fault trees." *Proceedings of the Annual Reliability and Maintainability Symposium*, Los Angeles, USA, Jan 2004. pp 207-213

79. TOMAN D. 1996. "Point vs Interval-based Query Languages for Temporal Databases." *Proceedings of the ACM Symposium on Principles of Database Systems (PODS) '96*, pp 58-67.

80. VARDI M.Y. 2001. "Branching vs Linear Time: Final Showdown." *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS Vol 2031, Springer-Verlag, UK, 2001. ISBN 3-540-41865-2. pp1-22.

81. VESELY W.E., NARUM R.E. 1970. *PREP and KITT: Computer codes for the Automatic Evaluation of a Fault Tree*. IN-1349 (1970) pp 188.

82. VESELY W.E., GOLDBERG F.F., ROBERTS N.H., HAASL D.F. 1981. *Fault Tree Handbook*. Washington D.C., USA. US Nuclear Regulatory Commission

83. VESELY W.E., STAMATELATOS M., DUGAN J.B., FRAGOLA J., MINARICK J., RAILSBACK J., 2002. *Fault Tree Handbook with Aerospace Applications*. NASA Office of Safety and Mission Assurance.

84. VILLEMEUR A. 1992. *Reliability, Availability, Maintainability and Safety Assessment: Volume 1*. Chichester, UK. John Wiley & Sons.

85. VILLEMEUR A. 1992. *Reliability, Availability, Maintainability and Safety Assessment: Volume 2*. Chichester, UK. John Wiley & Sons.

86. WALKER M.D. 2005. *Project Pandora: Temporal Fault Tree Analysis*. MSc Thesis, University of Hull, UK, 2005.

87. WALKER M.D., PAPADOPOULOS Y.I. 2006. "Pandora: The Time of Priority-AND gates." 12th *IFAC Symposium on Information Control Problems in Manufacturing (INCOM'06)*, St Etienne, France. pp 237- 242. (Best Paper in Track Award)

88. WALKER M.D., BOTTACI L., PAPADOPOULOS Y.I. 2007. "Compositional Temporal Fault Tree Analysis." In *Computer Safety, Reliability, and Security* - SAFECOMP'07, (eds) Saglietti, Oster, Norbert, Lecture Notes in Computer Science 4680:105-119, Springer, ISBN 978-3-540-75100-7, ISSN 0302-9743.

89. WALKER M.D., PAPADOPOULOS Y.I. 2007. "PANDORA 2: The Time of Priority OR gates." DCDS'07, *1st IFAC Workshop on Dependable Control of Discrete Event Systems*, Paris, 2007, pp. 169-174. Elsevier Science

90. WALKER M.D, PAPADOPOULOS Y.I. 2008. "Synthesis and analysis of temporal fault trees with PANDORA: The time of Priority AND gates", *Nonlinear Analysis: Hybrid Systems*, 2(2):368-382, Elsevier Science, ISSN 1751-570X

91. WALKER M.D., PAPADOPOULOS Y.I. 2009. "Qualitative Temporal Analysis: Towards a full implementation of the Fault Tree Handbook." *Control, Engineering & Practice*.

Elsevier Science, DOI 10.1016/j.conengprac.2008.10.003, ISSN 0967-0661 [Print version forthcoming, available online Nov 2008].

92. WALKER M.D., PAPADOPOULOS Y.I., PARKER D.J., LÖNN H., TÖRNGREN M., CHEN D., JOHANNSON R., SANDBERG A. 2009. "Semi-automatic FMEA supporting complex systems with combinations and sequences of failures." *SAE World Congress*, Detroit, April 2009.

93. WIJAYARATHNA P.G., and MAEKAWA M. 2000. "Extending Fault Trees with an AND-THEN gate." *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)* 2000.

94. WIJAYARATHNA P.G., KAWAT, Y., SANTOSA A., ISOGAI K. 1997. "Representing relative temporal knowledge with the tand connective." *Eight Ireland Conference on Artificial Intelligence (AI-97)*, Vol 2, pp 80-87, Sept 1997.

95. WOLFORTH I.P. 2005. *Extensions to a fault tree synthesis tool to enable efficient evaluation of synthesised fault trees.* MSc Thesis, University of Hull, 2005.

96. WORRELL R.B, and STACK D.W. 1978. *A SETS User Manual for the Fault Tree Analyst.* NUREG CR-04651, 1978.

97. XIANG J. 2005. *Fault Tree Analysis and Formal Methods for Requirements Engineering*. PhD Thesis, Japan Advanced Institute of Science and Technology, Sept 1996.

98. XING L., DUGAN J.B. 2002. "Reliability analysis of static phased mission systems with imperfect coverage". *IEEE Transactions on Reliability*, Vol 51, Issue 2; Jun 2002. ISSN: 0018-9529. pp 199-211

99. XU H., DUGAN J.B. 2004. "Combining Dynamic Fault Trees and Event Trees for Probabilistic Risk Assessment." *Proceedings of the Annual Reliability and Maintainability Symposium*, 26-29 Jan 2004, Los Angeles, USA. pp 214-219

100. YUGE T., YANAGI S. 2008. "Quantitative analysis of a fault tree with priority AND gates." *Reliability Engineering & System Safety*, Vol 93, pp 1577-1583.

Appendix I: Glossary

"No one who lives in error is free."
- Euripides

Absolute time
A model of time in which time is measured against a single absolute timeline, such as a calendar. Poses difficulty for FTA because we do not know exactly when faults will occur. Absolute time is almost always quantifiable.

Archimedes
Greek scientist, c. 287 – 212 BC, noted particularly for his mathematical and engineering genius. Famous for shouting "Eureka!" while in the bath. In the context of this thesis, Archimedes is a temporal qualitative FTA technique involving the generation and analysis of dependency trees.

Availability
The proportion of time that a system will be operational (or alternatively, the probability that the system will be operational at any given moment). Opposite of **unavailability**, and for non-repairable systems this is effectively the same as **reliability**.

Base Temporal Form (BTF)
A form of temporal expression in which all temporal operators are encapsulated in doublets. A refinement of **hierarchical temporal form.** Maximum number of nested gates is 3, e.g. `X + [Y<Z].[X<Z]` is in BTF.

Basic event
A simple contributing fault in a fault tree, usually a component failure. A basic event has no children.

Basic temporal node
The leaf nodes of a Dependency tree; the leaf nodes for a given Dependency tree are the same as the nodes of the equivalent Precedence tree. Basic temporal nodes contain all the events for a given tree and cannot be expanded further as they represent exactly one possible sequence. They contain only temporal operators.

Binary fault tree

A fault tree in which every gate has at most two inputs. Adjectival form is 'binarboreal'.

Boolean law

A Boolean law expresses an equivalence between two logical expressions. Boolean Laws can be used to manipulate and reduce logical expressions. Commonly used laws include the Distributive Laws and the Absorption Law.

Branching time

A system of time which branches whenever there is a choice (e.g. one branch may be "Fault A occurred" and another may be "Fault A did not occur"). The timeline may branch in the future, in the past, or both. It is also known as *non-deterministic* time (since there are many possible futures).

Coherent fault tree

A fault tree with a non-decreasing **structure function**, i.e. a fault occurring will not help to repair the system. A fault tree containing only AND and OR gates is always coherent.

Completion Laws

The Completion Laws are **temporal laws** in Pandora that relate the three temporal gates PAND, SAND and POR to the two logical gates AND and OR. They can be used for a number of purposes, including the reduction of the fault tree. The three laws are the **Conjunctive Completion Law** (or 1st Completion Law), **Disjunctive Completion Law** (or 2nd), and **Reductive Completion Law** (or 3rd).

Completion Problems

These are a type of reduction problem where reduction is only possible (or at least apparent) after an initial application of the Completion Laws. A simple example is X.Y.Z + X<Y + X&Y + Y<X, which reduces to just X.Y. When more than two events are involved it becomes very difficult to determine whether or not a Completion Law is applicable.

Conjunctive Completion Law (CCL)

The 1st Completion Law: `X<Y + X&Y + Y<X` $\Leftrightarrow$ `X.Y`. Links the **Conjunctive Temporal Gates** to the AND gate.

Conjunctive Temporal Truth Table

A form of TTT in which all events must occur (i.e. it contains no 0 rows). Formed by the leaf nodes of the equivalent precedence tree and is half the size of the equivalent normal TTT. In practice it omits all POR gates and works only with the conjunctive gates, i.e. AND, PAND, and SAND.

Continuous time

A system of time in which there are no 'gaps' between whatever atomic unit the time is measured in. A good analogy is the set of real numbers as opposed to the set of integers. Also known as *dense* time.

Contradiction

A contradiction is an impossibility, e.g. X must occur before Y and Y must occur before X; if it occurs under an AND gate, then the AND is also impossible. Used to help reduce temporal fault trees. Also a type of reduction which is based around the detection and removal of contradictions.

Cut sequence (CSQ)

A conjunctive sequence of basic events, the temporal analogue to the **cut set**. The difference is that two or more events in the set are in a certain order.

Cut set

A cut set is a conjunctive set of basic events used in **qualitative analysis**. The number of events in a cut set is the cut set's **order**.

Cyclic contradiction

A cyclic contradiction is a contradiction across more than one temporal gate which, individually, are not contradictions themselves. Example: X<Y . Y<Z . Z<X. Detected by means of the Laws of Extension.

Dependability

Dependability is the general term for reliability in its wider sense, i.e. how trustworthy and unlikely to fail something is.

Dependency Tree

A type of tree that models every possible sequence or combination for a given set of events and arranges them hierarchically according to logical Absorption and temporal Completion laws so that if all children of a node are true, then that node will also be true and the children are all redundant. For example, X<Y, X&Y, and Y<X are all children of X.Y and if all three are true then so is X.Y, which renders the children redundant. Dependency trees are used in

Archimedes. Technically Dependency trees are not trees however as they possess more than one top or root node.

Discrete time

Discrete time is the opposite of **continuous time**; it is a model of time in which there are 'gaps' between the atomic units that time is measured in. A good analogy is the set of integers as opposed to the set of real numbers. Discrete time is often used in **point-based time** models.

Disjunctive Completion Law (DCL)

The 2nd Completion Law: `X|Y + X&Y + Y|X` $\Leftrightarrow$ `X.Y`. Links the POR and SAND gates to the OR gate.

Doublet

An encapsulation of a single temporal relation between exactly two basic events, e.g. [X<Y]. Distinguished using square brackets. See also **base temporal form**. Also a word puzzle invented by Lewis Carroll.

Dynamic Fault Tree (DFT)

A type of fault tree augmented with temporal gates designed to facilitate temporal quantitative analysis using Markov chains. Pioneered by J. B. Dugan, K. J. Sullivan, and D. Coppit.

Dynamic systems

Systems which have behaviour that varies over time. In the context of this thesis, a dynamic system is one that has failure behaviour dependent on the occurrence of particular sequences of events (rather than just a combination of events).

Euripides

Greek dramatist (c. 480 BC – 406 BC) famed for his tragedies. In the context of this thesis, Euripides is a deductive temporal qualitative FTA method that uses temporal laws to manipulate and reduce temporal expressions to obtain minimal cut sequences.

Event (Pandora)

An event in Pandora represents the occurrence of a fault or other change in the system. Before it occurs, an event is false; once it occurs, it becomes true and remains true thereafter.

Event sequences

A set of events that have to occur in a certain order.

Event-based temporal fault tree

A type of temporal fault tree approach that models temporal or sequential information by means of events rather than gates. An example is the CSDM technique. Cf. **gate-based temporal fault tree**.

Failure

A failure occurs when a component or system is no longer capable of performing its intended function. It is slightly different to a **fault** but the two are often used interchangeably.

Failure rate

The frequency with which failures occur for a given component or system. Represented by λ.

Fault

A fault occurs when a component or system either fails to perform its intended function, or performs its intended function but at the wrong time. Also a general catch-all term for things going wrong.

Fault tree analysis (FTA)

Fault tree analysis is a deductive systems analysis technique used in reliability engineering to determine the root causes of events. It uses a graphical diagram (the fault tree itself) to connect the **top event**, which is a system fault, to a set of **basic events** via a network of logic **gates**.

Fubini numbers

The Fubini numbers are a series of numbers that describe the possible temporal sequences for a given set of events in Pandora. They increase dramatically with increasing numbers of events, imposing performance constraints on algorithms.

Gate

A gate is the term used to describe a logical (or temporal) operator in fault trees, e.g. an OR gate. Gates are intermediate events in a fault tree and may have children.

Gate-based temporal fault tree

A type of temporal fault tree approach that models temporal or sequential information by means of new temporal gates. DFTs and Pandora are both gate-based approaches. Cf. **event-based temporal fault tree**.

Hierarchical Temporal Form (HTF)

Also known as '**cut sequence** form', this is a form of expression in which operators are arranged hierarchically (hence the name): ORs first, ANDs second, then PORs, PANDs, and SANDs. It is a form of temporal disjunctive normal form. For example, the expression `X<Y + Y.(X&Z)` is in HTF. An expression in HTF contains one or more cut sequences (in the example, X<Y and Y.(X&Z) are the two cut sequences). Equivalent to disjunctive normal form / sum-of-products form in normal static fault trees.

Implicants

Implicants are the non-coherent equivalent to **cut sets**. Implicants include not only those events necessary to cause the top event but also which events must *not* occur in order to cause the top event.

Intermediate event

In a fault tree, an intermediate event is a gate. It is an event that can be decomposed or caused by other events. Intermediate events will have one or more children representing these events.

Interval-based time

Time which is measured in atomic intervals. For example, we measure time in seconds, which are intervals of time in themselves; cf. **point-based time.** Interval-based time is often preferred because it can be seen as a generalisation of point-based time, but it can also have more complex semantics if intervals can overlap.

Linear Temporal Logic (LTL)

A general term for temporal logics that use a linear model of time. The typical example is PTL (Propositional Temporal Logic). Most common operators include *sometime, always, until,* and *since*. Can include past or future temporal operators or both.

Linear time

Time which does not branch, i.e. there is only one timeline of events. This is also known as deterministic time as there is only one possible future (and one possible past).

Logical equivalence

Logical equivalence holds for two expressions if they are always true at the same time and always false at the same time. This means the two expressions can be substituted for each other, a fact put to great use in FTA.

Mean Time Between Failures (MTBF)

The average time between/before failures occurring in a system. Inverse of the **failure rate**.

Mean Time To Repair (MTTR)

The average time a system is out of operation, i.e. the time it takes to repair it.

Minimal cut sequence (MCSQ)

A minimal cut sequence is a **cut sequence** in which every event must occur to cause the top event, i.e. a cut sequence containing no redundant events.

Minimal cut set (MCS)

A **cut set** in which every event must occur to cause the top event, i.e. a cut set containing no redundant events.

Non-coherent fault tree

A fault tree with a decreasing **structure function**, i.e. one in which the occurrence of a fault can improve the functioning of a system. Often the case when NOT gates are introduced to the fault tree.

Omniscient view

A concept in Pandora in which the analyst is presumed to have full knowledge of exactly when events occur; cf. **restricted view**. An omniscient view could be applied to a system at design time or after it had failed and ceased operation.

Order (cut set)

The order of a **cut set** or **cut sequence** is the number of unique events it contains. In cut sequences the same event may occur more than once in multiple doublets or temporal gates but is only counted once for purposes of defining the order of the cut sequence; e.g. [X<Y].[X<Z] is order 3.

Pandora

Pandora is what this document is all about and it would be inefficient to repeat the entire document in the Glossary, quite apart from the infinite loop it would lead to.

Period of observation

The period of observation is a concept in Pandora in which the failure behaviour of a system is observed only for a finite period of (bounded) time; used in connection with a **restricted view** to allow for NOT gates to firmly say that an event has not occurred within the period.

Persistence

Persistence is a quality that is vital for a person who has to write a thesis. It is also used to describe events which, once true, can no longer become false, i.e. they stay true forever.

Point-based time

A system of time in which the atomic unit of time is a point. Typically used with **discrete** models of time and causes semantic problems when one event ends as another begins (do they end on the same point, in which case, do they overlap? Or do they end on different points, in which case isn't there a gap?); cf. **interval-based time**.

Precedence Tree

A type of tree representing a branching timeline for a given set of events. The nodes of the precedence tree give every possible sequence of those events (including cases where events do not occur). The leaf nodes give every possible sequence in which all those events occur. Useful in generating **temporal truth tables**.

Prime implicant

An **implicant** which contains no redundant events or complements of events; analogous to a **minimal cut set** except with complements of events included.

Priority gates

The PAND and POR gates; gates which impose a sequence rather than a simultaneity on their inputs.

Priority-AND gate (PAND)

The original temporal fault tree gate, it is true if its input events occur in a certain sequence. It is used in a number of temporal FTA techniques including both DFTs and Pandora. Can be either inclusive (includes simultaneous events) or exclusive (does not include simultaneous events). In Pandora, the PAND is exclusive and is true if all of its inputs occur in sequence from left to right. Its sequence value is then the same as the right-most input event.

Priority-OR gate (POR)

The Priority-OR gate is a temporal gate in Pandora used to represent the notion of *priority*, i.e. that one event has priority over a number of others and must occur first. The POR gate is true if its priority (left-most) event occurs before any other input, regardless of whether any other input occurs or not. Its sequence value is the same as the sequence value of its left-most input.

Qualitative analysis

Also known as logical analysis. In FTA qualitative analysis is the process of reducing the fault tree to obtain its minimal cut sets (or sequences) which allows the analyst to see which failures have the biggest impact on the system by looking at the number of events necessary to cause the top event. Single events are the worst and are known as **single points of failure**. The qualitative analysis algorithms in Pandora are **Euripides** and **Archimedes**.

Quantitative analysis

Also known as numerical or probabilistic analysis. In FTA quantitative analysis is the process of calculating the probability of the top event occurring, given individual probabilities for all the basic events. It can be used either with or without a prior qualitative analysis.

Quantitative time

Also known as metricated time. In these models of time, it is possible to measure time; in practice this means measuring time in real units. Valuable for modelling real-time systems.

Reductive Completion Law (RCL)

The 3rd Completion Law: `X|Y + X&Y + Y<X` ⇔ `X`. Allows an event to be entirely removed if it does not matter which order it occurs in or even if it occurs at all.

Relative time

A model of time in which time is measured against other events rather than against an absolute timeline. 'Last week' and 'tomorrow' are statements that use a relative model of time, because they state only when something happened relative to the current moment in time.

Reliability engineering

A field of engineering dedicated to improving the reliability and safety of engineering systems. Fault tree analysis is one technique capable of doing this.

Repair rate

The frequency at which a system is repaired, represented by 'μ'.

Restricted view

An esoteric concept in Pandora in which the analyst only has a limited view of when events occur, so that at most it is possible to say that a fault "has not occurred *yet*". A restricted view would be the case in a monitored system, for example. Cf. **omniscient view** and **period of observation**.

Risk

A product of reliability and severity, so that a risky system can be reliable but cause great harm if it breaks down or relatively unreliable but not capable of doing much damage if it breaks.

Safety critical systems

Systems in which safety is an important factor; typically, safety critical systems are those which have the potential to harm people or the environment if they suffer a fault, hence the need for reliability engineering to improve their reliability and reduce the likelihood of failure.

Sequence value

An integer indicating the order in which an event occurred, or if 0, that an event did not occur. For example, a sequence value of 3 means that the event was the third to occur. Often indicated by *S*, e.g. *S*(X) is the sequence value of event X. Sequence values in temporal expressions are analogous to truth values in Boolean expressions.

Simultaneity

The property of two or more events that occur at the same time. It is often ignored in temporal FTA but not in Pandora.

Simultaneous-AND (SAND)

A temporal gate in Pandora which is true if all its input events occur at the same time. The SAND has the same sequence value (if true) as all of its inputs.

Single point of failure

A cut set or sequence containing a single event; this event is sufficient to cause the top event to occur on its own and is therefore a vulnerability in the reliability of the system. See also **singleton node**.

Singleton node

The top level nodes of a dependency tree which represent the occurrence of single events.

Static fault tree (SFT)

A traditional fault tree containing only AND and OR gates, as opposed to a **temporal fault tree**.

Structure function

A function which describes the top event of a fault tree in terms of its constituent basic events (effectively a logical expression); if it is non-decreasing, it means all faults contribute to the overall failure of the system, but if it is decreasing, it means that some faults may prevent the system from failing or even cause it to become operational again.

Systems analysis

The process of studying a system and how it works in order to gain information about the system, which can then be used to draw conclusions about the system, e.g. how it can be improved.

Temporal conjunctive gates

The PAND and SAND gates, which are subsets of the AND gate and part of the **Conjunctive Completion Law**.

Temporal disjunction gate

The POR gate, which is part of the **Disjunctive Completion Law**.

Temporal fault tree / Temporal fault tree analysis

A fault tree (or FTA) that has been extended or augmented with additional features capable of representing (or analysing) time-based or sequential information. Pandora is a temporal fault tree methodology. Not to be confused with TFTs (Temporal Fault Trees) which are a specific temporal fault tree analysis approach proposed by G.K. Palshikar.

Temporal law

A logical law in the style of Boolean laws except that it contains temporal gates. Used in a similar manner to Boolean laws but a temporal law is often more complicated. Important temporal laws include the **Completion Laws** and the Laws of Mutual Exclusion.

Temporal product

The value of a doublet based on the product of its events' unique prime numbers, the sign of which is given by the subtraction of the second event from the first. For example, in [X<Y], assuming X = 3 and Y = 5, the product is 3 * 5 = 15 combined with the sign 3 – 5 = -2, so the temporal product is –15 (the value from the first calculation and the sign from the second). The temporal product is a way of quickly determining what events a doublet contains and in what order they occur. Further checks are still necessary to compare the different operators, however.

Temporal truth table (TTT)

A truth table which proves temporal equivalence between two (or more) expressions; whereas a Boolean **truth table** shows only true and false values, a temporal truth table shows **sequence values** instead, showing the order in which events occur. See also **Conjunctive TTT**.

Temporally equivalent

A condition which means that two logical expressions are not only true at the same time and false at the same time, but which also have the same **sequence value** (i.e. they become true in the same order).

Temporal operator

A (temporal) logical operator that performs a function or operation involving time or sequence. In classical temporal logics, temporal operators are usually either future- or past-oriented. In Pandora, there are three temporal operators (PAND, POR, SAND), each of which can be seen as functions that operate on sequence values.

Temporal significance

The property of a basic event or intermediate event that means it must occur in a certain order. Temporal significance is conferred by a temporal gate, meaning that the same event can be temporally significant in one branch of a fault tree but not in another branch.

Top event

The top event of a fault tree is its 'root' event and usually represents a system-level fault. It is the starting point of a fault tree analysis as the goal is to determine what combinations (or sequences) of events may cause the top event to occur.

Truth table

A table showing all possible truth values for a given set of events in Boolean logic. If two logical expressions have the same truth tables, then they are logically equivalent.

Unaffiliated events

Events in a cut sequence which are not part of a doublet or temporal gate, e.g. Z in the cut sequence `[X<Y].Z`.

Unavailability

Equal to 1 – **availability**; the proportion of time that a system is likely to spend out of action.

Appendix II: Boolean & Temporal Laws

"Time is the cruellest teacher: first it gives the test, then it teaches the lesson."

- Unknown

Commutative Law

```
X.Y ⇔ Y.X
X+Y ⇔ Y+X
```

Associative Law

```
X.(Y.Z) ⇔ (X.Y).Z ⇔ X.Y.Z
X+(Y+Z) ⇔ (X+Y)+Z ⇔ X+Y+Z
```

Distributive Law

```
X.(Y+Z) ⇔ (Y+Z).X ⇔ X.Y + X.Z
X+(Y.Z) ⇔ (Y.Z)+X ⇔ (X+Y).(X+Z)
(A.B)+(C.D) ⇔ (A+C).(A+D).(B+C).(B+D)
(A+B).(C+D) ⇔ (A.C)+(A.D)+(B.C)+(B.D)
```

Idempotent Law

```
X.X ⇔ X
X+X ⇔ X
```

Absorption Law

```
X.(X+Y) = X
X+(X.Y) = X
```

Temporal Commutative Laws

```
X&Y ⇔ Y&X
```

Temporal Associative Laws

```
(X<Y)<Z ⇔ X<Y<Z
X&(Y&Z) ⇔ (X&Y)&Z ⇔ X&Y&Z
(X|Y)|Z ⇔ X|Y|Z
```

Temporal Distributive Laws

```
X < (Y.Z) ⇔ Y.(X<Z) + Z.(X<Y)
X < (Y+Z) ⇔ (X|Z).(X|Y).(Y+Z)
X < (Y<Z) ⇔ (X<Z).(Y<Z)
X < (Y&Z) ⇔ (X<Y).(X<Z).(Y&Z)
X < (Y|Z) ⇔ (X<Y).(Y|Z)
X < (Y|Z) ⇔ (X|Y).(Y|Z)
X & (Y+Z) ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y)
X & (Y.Z) ⇔ (X&Y).(Y&Z) + (X&Y).(Z<Y) + (X&Z).(Y<Z)
X & (Y<Z) ⇔ (Y<Z)&(Y<X)
X & (Y<Z) ⇔ (X&Z).(Y<Z).(Y<X)
X & (Y&Z) ⇔ X&Y&Z
X & (Y|Z) ⇔ (X&Y).(Y|Z).(X|Z)
X & (Y|Z) ⇔ (X|Z)&(Y|Z)
X | (Y+Z) ⇔ (X|Y).(X|Z)
X | (Y.Z) ⇔ X|Y + X|Z
X | (Y<Z) ⇔ (X|Z) + (X|Y) + X.(Z<Y) + X.(Y&Z)
X | (Y&Z) ⇔ X.(Y|Z) + X.(Z|Y) + (X|Y) + (X|Z)
X | (Y|Z) ⇔ (X|Y) + (X.Z<Y) + (X.Y&Z)

(Y+Z) < X ⇔ (Y<X) + (Z<X)
(Y.Z) < X ⇔ (Y<X).(Z<X)
(Y<Z) < X ⇔ (Y<Z).(Z<X)
(Y&Z) < X ⇔ (Z<X).(Y<X).(Y&Z)
(Y|Z) < X ⇔ (Y<X).(Y|Z)
(Y+Z) & X ⇔ (X&Y).(Y&Z) + (X&Y).(Y|Z) + (X&Z).(Z|Y)
(Y.Z) & X ⇔ (Y<X).(Z&X) + (Z<X).(Y&X) + X&Y&Z
(Y<Z) & X ⇔ (Y<Z).(Z&X).(Y<X)
(Y&Z) & X ⇔ Y&Z&X
(Y|Z) & X ⇔ (X&Y).(Y|Z)
(Y+Z) | X ⇔ (Y|X) + (Z|X)
(Y.Z) | X ⇔ (Y|X).(Z|X)
(Y<Z) | X ⇔ (Y|Z).(Z|X)
(Y&Z) | X ⇔ (Y|X)&(Z|X)
(Y&Z) | X ⇔ (Y|X).(Z|X).(Y&Z)
```

```
(Y|Z) | X ⇔ (Y|Z).(Y|X)

(A+B) & (C+D) ⇔ A&B&C&D + A&C|B|D + A&D|B|C + B&C|A|D
                + B&D|A|C + A&B&C|D + A&B&D|C + A&C&D|B
                + B&C&D|A
(A.B) & (C.D) ⇔ A&B&C&D + A<B&C&D + B<A&C&D + C<A&B&D
                + D<A&B&C + (A.C)<B&D + (A.D)<B&C +
                + (B.C)<A&D + (B.D)<A&C
(A<B) & (C<D) ⇔ (A<B).(C<D).(B&D)
(A|B) & (C|D) ⇔ (A&C).(A|B).(C|D)
(A+B) < (C+D) ⇔ (A|C).(A|D).(C+D) + (B|C).(B|D).(C+D)
(A.B) < (C.D) ⇔ (A<D).(B<D).(C<D) + (A<C).(B<C).(D<C)
                + (A<C).(A<D).(B<C).(B<D).(C&D)
(A&B) < (C&D) ⇔ (A<C).(B<C).(A<D).(B<D).(A&B).(C&D)
(A|B) < (C|D) ⇔ (A<C).(A|B).(C|D)
(A+B) | (C+D) ⇔ (A|C).(A|D) + (B|C).(B|D)
(A.B) | (C.D) ⇔ (A|C).(B|C) + (A|D).(B|D)
(A<B) | (C<D) ⇔ (A<B).(B|C) + (A<B).(B|D) + (A<B).(D<C)
                + (A<B).(C&D)
(A&B) | (C&D) ⇔ (A&B).(C|D) + (A&B).(D|C) + (A&B).(B|C)
                + (A&B).(B|D)
```

Temporal Absorption Laws

```
X . (X < Y) ⇔ X < Y
X . (X & Y) ⇔ X & Y
X . (X | Y) ⇔ X | Y
X < (X . Y) ⇔ X < Y
X & (X . Y) ⇔ X&Y + Y<X
X | (X . Y) ⇔ X | Y
X < (X + Y) ⇔ 0
X & (X + Y) ⇔ X&Y + X|Y
X | (X + Y) ⇔ 0
(X . Y) < X ⇔ 0
(X . Y) & X ⇔ X&Y + Y<X
(X . Y) | X ⇔ 0
```

```
(X + Y) < X ⇔ Y < X
(X + Y) & X ⇔ X&Y + X|Y
(X + Y) | X ⇔ Y | X
X + (X < Y) ⇔ X
X + (X & Y) ⇔ X
X + (X | Y) ⇔ X
Y + (X < Y) ⇔ Y
Y + (X & Y) ⇔ Y
Y + (X | Y) ⇔ X + Y
```

The Completion Laws

```
X.Y ⇔ X<Y + X&Y + Y<X
X+Y ⇔ X|Y + X&Y + Y|X
X ⇔ Y<X + X&Y + X|Y
```

Laws of Mutual Exclusion

```
X<Y . Y<X ⇔ 0
X<Y . X&Y ⇔ 0
Y<X . X&Y ⇔ 0
X|Y . Y<X ⇔ 0
Y|X . X<Y ⇔ 0
X|Y . X&Y ⇔ 0
Y|X . X&Y ⇔ 0
```

Laws of Simultaneity

```
X<X ⇔ 0
X|X ⇔ 0
X&X ⇔ X
```

Law of Extension

```
X<Y . Y<Z ⇔ X<Y . Y<Z . X<Z ⇔ X<Y<Z
X&Y . Y&Z ⇔ X&Y . Y&Z . X&Z ⇔ X&Y&Z
X|Y . Y|Z ⇔ X|Y . Y|Z . X|Z
```

Extended Laws of Extension

X&Z . Y&Z ⇔ X&Z . Y&Z . **X&Y**

Y&X . Y&Z ⇔ Y&X . Y&Z . **X&Z**

Z<X . Y<Z ⇔ Z<X . Y<Z . **Y<X**

Z&X . Y&Z ⇔ Z&X . Y&Z . **X&Y**

Z|X . Y|Z ⇔ Z|X . Y|Z . **Y|X**

XpY . YqZ

X&Y . Y<Z ⇔ X&Y . Y<Z . **X<Z**

X&Y . Y|Z ⇔ X&Y . Y|Z . **X|Z**

X<Y . Y&Z ⇔ X<Y . Y&Z . **X<Z**

X<Y . Y|Z ⇔ X<Y . Y|Z . **X|Z**

X|Y . Y&Z ⇔ X|Y . Y&Z . **X|Z**

X|Y . Y<Z ⇔ X|Y . Y<Z . **X<Z**

XpZ . YqZ

X&Z . Y<Z ⇔ X&Z . Y<Z . **Y<X**

X&Z . Y|Z ⇔ X&Z . Y|Z . **Y|X**

X<Z . Y&Z ⇔ X<Z . Y&Z . **X<Y**

X|Z . Y&Z ⇔ X|Z . Y&Z . **X|Y**

YpX . YqZ

Y&X . Y<Z ⇔ Y&X . Y<Z . **X<Z**

Y&X . Y|Z ⇔ Y&X . Y|Z . **X|Z**

Y<X . Y&Z ⇔ Y<X . Y&Z . **Z<X**

Y|X . Y&Z ⇔ Y|X . Y&Z . **Z|X**

ZpX . YqZ

Z&X . Y<Z ⇔ Z&X . Y<Z . **Y<X**

Z&X . Y|Z ⇔ Z&X . Y|Z . **Y|X**

Z<X . Y&Z ⇔ Z<X . Y&Z . **Y<X**

Z<X . Y|Z ⇔ Z<X . Y|Z . **Y<X**

Z|X . Y&Z ⇔ Z|X . Y&Z . **Y|X**

Z|X . Y<Z ⇔ Z|X . Y<Z . **Y|X**

The Law of SAND Substitution

```
X&Y . X<Z ⇔ X&Y . Y<Z
X&Y . X|Z ⇔ X&Y . Y|Z
X&Y . Z<X ⇔ X&Y . Z<Y
X&Y . Z|X ⇔ X&Y . Z|Y
```

or generally, where ? is another temporal operator:

```
X&Y . X?Z ⇔ X&Y . Y?Z
```

The Laws of POR Transformation

```
X|Y . Y ⇔ X<Y
X|Y + Y ⇔ X + Y
```

The Laws of Priority

```
X<Y + X|Y ⇔ X|Y
X<Y . X|Y ⇔ X<Y

X<Y + X.Y ⇔ X.Y
X&Y + X.Y ⇔ X.Y
X|Y + X.Y ⇔ X
```

The Binary Laws

```
A.B.C. ... .N ⇔ ((((A.B).C)...).N)
A+B+C+ ... +N ⇔ ((((A+B)+C)...)+N)
A<B<C< ... <N ⇔ ((((A<B)<C)...)<N)
A&B&C& ... &N ⇔ ((((A&B)&C)...)&N)
A|B|C| ... |N ⇔ A|(((B+C)...)+N)
```

The Encapsulation Laws

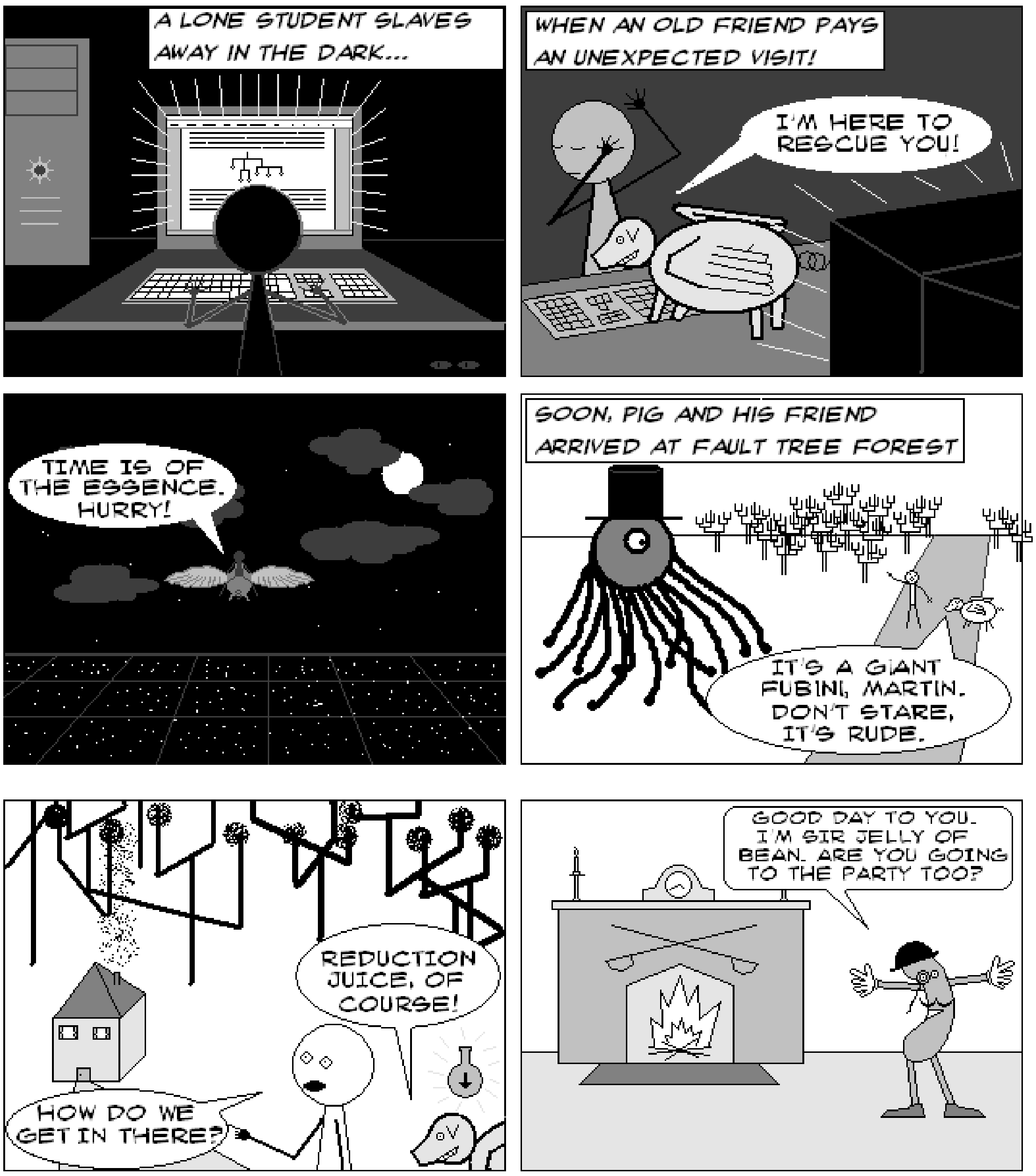
```
X < (Y < Z) ⇔ [X<Y] . [Y<Z] . [X<Z] +
              [Y<X] . [Y<Z] . [X<Z] +
              [X&Y] . [Y<Z] . [X<Z]
X < (Y&Z) ⇔ [X<Y].[X<Z].[Y&Z]
X | (Y<Z) ⇔ [X|Z] + [X|Y] + X.[Z<Y] + X.[Y&Z]
X | (Y&Z) ⇔ X.[Y|Z] + X.[Z|Y] + [X|Y] + [X|Z]
X | (Y|Z) ⇔ [X|Y] + X.[Z<Y] + X.[Y&Z]
```

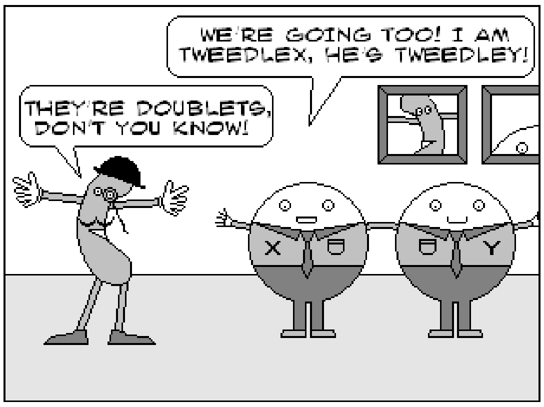
```
(Y<Z) < X ⇔ [Y<Z].[Z<X].[Y<X]
(Y&Z) < X ⇔ [Z<X].[Y<X].[Y&Z]
(Y&Z) & X ⇔ [X&Y].[Y&Z].[X&Z]
(Y<Z) | X ⇔ [Y<Z].[Z|X].[Y|X]
(Y&Z) | X ⇔ [Y|X].[Z|X].[Y&Z]
(Y|Z) | X ⇔ [Y|Z].[Y|X]
```

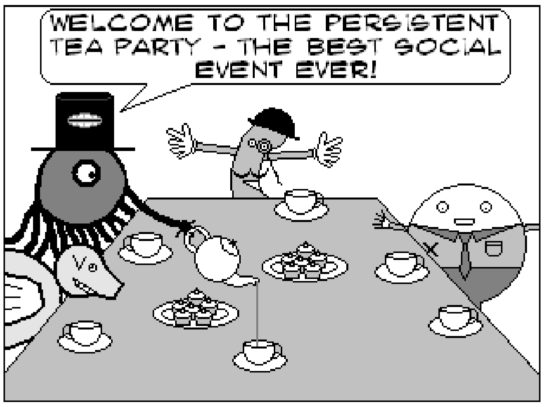
Appendix III: The Adventures of Martin in Pandora Land

"This is slavery, not to speak one's thought."

- Euripides

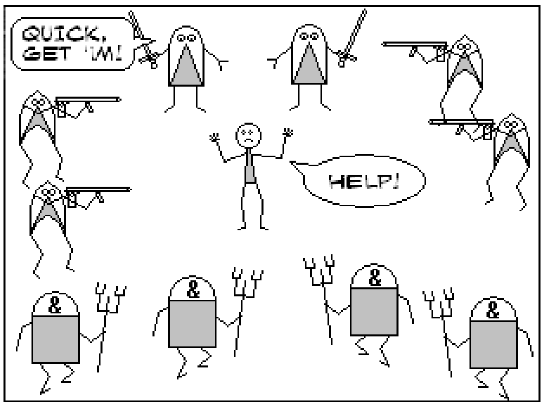

A LONE STUDENT SLAVES AWAY IN THE DARK...
WHEN AN OLD FRIEND PAYS AN UNEXPECTED VISIT!
I'M HERE TO RESCUE YOU!
TIME IS OF THE ESSENCE. HURRY!
SOON, PIG AND HIS FRIEND ARRIVED AT FAULT TREE FOREST
IT'S A GIANT FUBINI, MARTIN. DON'T STARE, IT'S RUDE.
HOW DO WE GET IN THERE?
REDUCTION JUICE, OF COURSE!
GOOD DAY TO YOU. I'M SIR JELLY OF BEAN. ARE YOU GOING TO THE PARTY TOO?

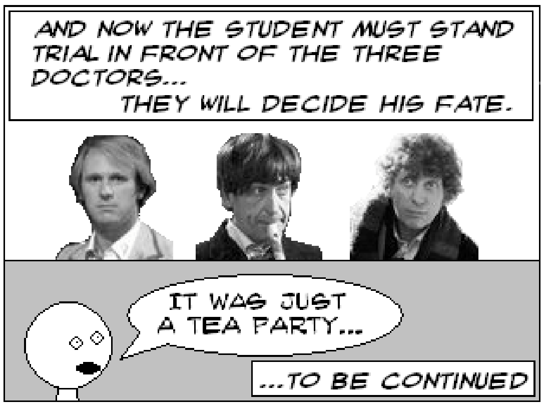

WE'RE GOING TOO! I AM TWEEDLEX, HE'S TWEEDLEY!
THEY'RE DOUBLETS, DON'T YOU KNOW!
X
Y


WELCOME TO THE PERSISTENT TEA PARTY - THE BEST SOCIAL EVENT EVER!

BUT THE STUDENT HAD WORK TO DO - HE HAD TO GET BACK HOME...
I AM EURIPIDES! HOW DARE YOU LEAVE THE PERSISTENT PARTY? TO THE QUEEN WITH YOU!
WHO?!

I AM QUEEN PANDORA, AND YOU WILL ADDRESS ME AS 'YOUR MAJESTY!'
ORF WITH HIS HEAD!!!


QUICK, GET 'IM!
HELP!


AND NOW THE STUDENT MUST STAND TRIAL IN FRONT OF THE THREE DOCTORS... THEY WILL DECIDE HIS FATE.
IT WAS JUST A TEA PARTY...
...TO BE CONTINUED