Next Generation Fault Tree Analysis Methods

Prof John Andrews, University of Nottingham

Dr Silvia Tolo, University of Nottingham

Key Words: fault trees, binary decision diagrams, dependencies, Petri nets

SUMMARY

The tutorial presents a methodology, known as D^2T^2, Dynamic and Dependent Tree Theory, which overcomes some of the limitations of conventional fault tree analysis which restrict the ability of the approach to represent the performance of modern engineering systems. These include:

i. Components having constant failure and repair rates.
ii. All component failures being mutually independent.
iii. Limited maintenance strategies to be accommodated.

The D^2T^2 approach retains the fault tree structure to represent the causality of the system failure in terms of its component failures but the analysis process is changed to exploit the features of Binary Decision Diagrams, Petri nets and Markov models to overcome the limitations.

1. INTRODUCTION - FAULT TREE ANALYSIS CHARACTERISTICS AND ASSUMPTIONS

Fault Tree Analysis (FTA) is the most commonly used method in industry to analyse the reliability performance of systems. This is particularly true when the system is safety critical and its failure could result in injuries or fatalities to the workforce and the general public. The method has its origins back in the 1960s and 1970s in the nuclear and aerospace industries [1, 2]. However, modern systems are designed, operated and maintained very differently to those of the era when the method was conceived. There are several features and assumptions of the conventional FTA method, or its implementation in commercial codes, which limits its ability to represent the performance modern systems. These include:

i. all component failures being mutually independent.
ii. components having constant failure and repair rates.
iii. limited maintenance strategies to be represented.

It is common for modern systems to have features which violate these assumptions rendering the results obtained from a fault tree study unrepresentative of the system performance.
Independence between the basic events in the conventional fault tree is a fundamental requirement of the fault tree methodology. There are many aspects of system designs, their operation or maintenance which can violate this assumption. Component failure probability models, within the conventional implementation, feature constant failure and repair rates. Many systems have components whose condition is deteriorating over time and their failure results from some form of wear-out. As such their failure is not random and the failure rate is increasing not constant. It can be argued that, since maintenance is not a random process, the repair rate is never constant. Component failure probabilities, Q(t), are commonly evaluated using the following equations:

No Repair

$$Q(t) = F(t) = 1 - e^{-\lambda t} \leq \lambda t \tag{1}$$

where λ is the constant failure rate.

Revealed Failures

This is unscheduled, reactive, maintenance which takes place as soon as the component failure occurs. For components with constant failure rate, λ, and constant repair rate, υ, the failure probability at time t is given by:

$$Q(t) = \frac{\lambda}{\lambda + \nu}\left(1 - e^{-(\lambda+\nu)t}\right) \tag{2}$$

Unrevealed failure

When component failures are not revealed, tests are performed, at intervals of θ, to establish their state. The failure probability is then a function of λ, θ and the mean time to repair, τ. There are several models that can then be employed to calculate the average failure probability; a simple form is given in equation (3):

$$Q_{AV} = \lambda\left(\frac{\theta}{2} + \tau\right) \tag{3}$$

Maintenance processes are far more sophisticated than the three options represented in these equations. Examples include servicing, component replacement prior to failure and opportunistic maintenance.

This tutorial describes a framework (known as D^2T^2, Dynamic and Dependent Tree Theory) that will overcome the limitations, outlined above, whilst retaining the fault tree to express the system failure logic. The fundamental method employed for analysing the fault trees in this framework is the Binary Decision Diagram (BDD) method. It will be shown how to calculate the system failure probability and failure intensity. It builds on the material presented in the core tutorial 'Fault Tree Analysis'

2. BINARY DECISION DIAGRAMS

Binary Decision Diagrams provide a method for the efficient and accurate quantification of fault trees [3-9]. An example BDD is given in figure 1.

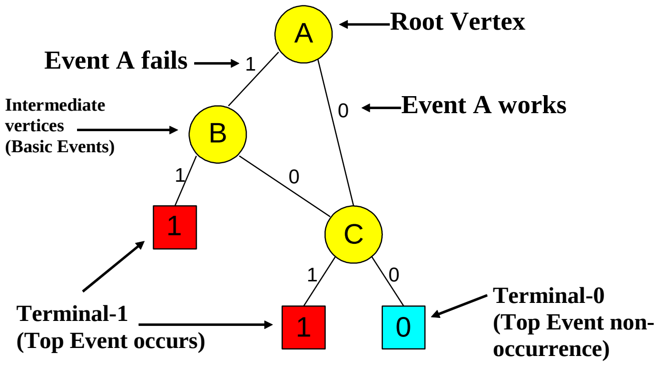

A
Root Vertex
Event A fails
1
0
Event A works
Intermediate vertices (Basic Events)
B
1
0
1
C
1
0
Terminal-1 (Top Event occurs)
1
0
Terminal-0 (Top Event non-occurrence)

Figure 1 Binary Decision Diagram Structure

The entry point on the BDD is at the root vertex. The intermediate nodes all correspond to components in the system and two paths leave each node, a '1' branch representing component failure and a '0' branch for component functioning. To construct the diagram an 'ordering' must be assigned to the basic events which specifies the order in which each component is considered on the diagram. A path on the diagram starts at the root vertex and progresses on a specific output branch from each node which determines the status of that particular component. As soon as the component conditions specified on any path establish the system status then the path is terminated with the appropriate terminal-0 or terminal-1 node for system functioning or system failure respectively. The concise nature of these diagrams is appealing from the mathematical perspective and results in efficient and accurate analysis. From the engineering perspective they are not documented and do not develop the failure logic in any structured way. As such, it is usually best to develop the system failure logic as a fault tree and then convert to a BDD for analysis. Two construction processes will be described. The first uses the logic function obtained from the fault tree, the second manipulates equations using a special **ite** (if-then-else) notation.

2.1 BDD construction using the system logic equations

Consider the fault tree illustrated in figure 2 as an example. To convert this into a BDD the variables (basic events) are placed in an order (A, B, C for this example).

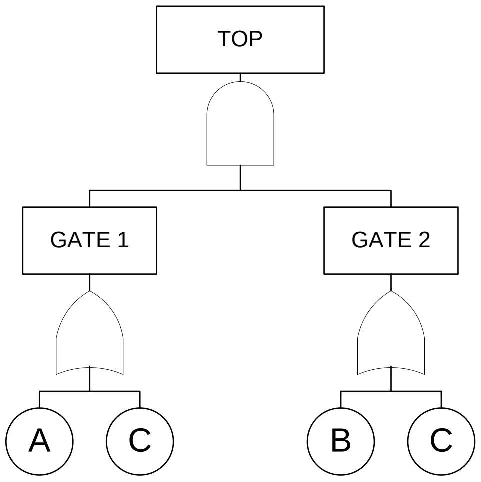


Figure 2 Simple Fault Tree

For the fault tree in figure 2 we get an expression for the top event:

$$\text{TOP} = (A+C).(B+C) \qquad (4)$$

(Note: in the logic equations given throughout this tutorial '+' will be used for OR and '.' for AND).

It does not matter what form this equation takes as long as it is only a function of the basic events. The construction of the BDD then starts with the first variable in the ordering, A. A is substituted as TRUE (failed) and FALSE (works) into the logic expression getting two new logic expressions to place on the output '1' branch and '0' branch respectively. These two functions are in turn developed by substituting in TRUE and FALSE values for the second variable in the list, B. This process is illustrated in figure 3 and, once completed, can be shown to produce the BDD illustrated in figure 1.

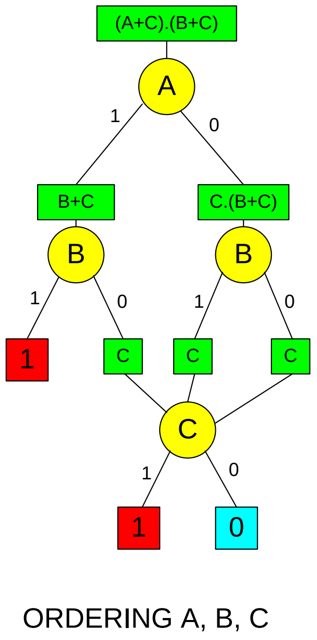


Figure 3 BDD Constructed from the System Logic Equation

*2.2 BDD construction using the **ite** form*

Since nodes on a BDD always have two exit edges there is a convenient mathematical form to express this. We consider the basic event represented by a node, say X.

IF the event X occurs

THEN we pass down the '1-branch' and consider the Boolean function represented on the BDD connected to this branch. (say f1).

ELSE (the event has not occurred) we pass down the '0-branch' and consider the Boolean function represented on the BDD connected to this branch (say f2).

This can be written in a very concise way:

ite (X, f1 , f2)

As an example consider the root vertex of the BDD illustrated in figure 3. This can be expressed as:

ite (A, f1 , f2)

where f1= B + C and f2 = C.(B + C).

The method described above to generate the BDD using a top event equation does not lend itself to implementation on a computer. However, with the **ite** notation used to express each node on the BDD, mathematical rules can be established which are applied in a bottom up fashion to the fault tree and deliver the BDD form.

Initially each basic event in the fault tree is expressed in the **ite** notation. So for any basic event X we have:

X = **ite**(X, 1 , 0)

When we encounter a gate of logic type ⊕ which has two inputs G and H already expressed in **ite** form:

G = **ite** (X, g1, g2) and H = **ite** (Y, h1, h2)

then the following rules can be applied to get an **ite** expression for the gate event. It depends on where the basic events X and Y appear in the ordering. Either X < Y (X appears before Y) or X=Y (X and Y are the same variable). Then either:

$$G \oplus H = \mathbf{ite}(X, g1 \oplus H, g2 \oplus H) \quad \text{if } X<Y \qquad (5a)$$

Or

$$G \oplus H = \mathbf{ite}(X, g1 \oplus h1, g2 \oplus h2) \quad \text{if } X=Y \qquad (5b)$$

⊕ = AND (.) or OR (+)

Once an **ite** expression is formed for any gate it is simplified as much as possible using the Boolean identities:

$$G + 1 = 1 \quad (6a) \qquad G + 0 = G \quad (6b)$$
$$G \cdot 1 = G \quad (6c) \qquad G \cdot 0 = 0 \quad (6d) \qquad (6)$$

Also **ite** (X, f1, f1) =f1

As an example, consider the fault tree illustrated in figure 2. We will use the rules expressed in equations 5 and 6 and an ordering of the basic events A < B < C to convert this to a BDD. First all basic events are put into **ite** form:

ite (A, 1, 0) **ite** (B, 1, 0) **ite** (C, 1, 0)

Considering now GATE1:

GATE1=A+C

= ite(A,1 ,0)+ ite(C,1 ,0) [use equation 5a]

=ite(A, 1+ ite(C,1 ,0) , 0+ite(C,1 ,0)) [use equations 6a and 6b]

=ite(A, 1, ite(C,1 ,0))

Considering now GATE2:

GATE2=B+C

= ite(B,1 ,0)+ ite(C,1 ,0) [use equation 5a]

=ite(B, 1+ ite(C,1 ,0) , 0+ite(C,1 ,0)) [use equations 6a and 6b]

=ite(B, 1, ite(C,1 ,0))

Having now established expressions for GATE1 and GATE 2, an expression for the top event can be formulated:

TOP=GATE1.GATE2

= ite(A, 1, ite(C,1 ,0)) . ite(B, 1, ite(C,1 ,0))

Using equation 5a, gives:

=ite(A, 1. ite(B, 1, ite(C,1 ,0)) , ite(C,1 ,0) .ite(B, 1, ite(C,1 ,0)))

Applying equation 6c to the 1-branch and equation 5a to the term on the 0-branch of the above expression gives:

=ite(A, ite(B, 1, ite(C,1 ,0)) , ite(B, 1. ite(C,1 ,0) , ite(C,1 ,0). ite(C,1 ,0)))

Simplifying

=ite(A, ite(B, 1, ite(C,1 ,0)) , ite(B, ite(C,1 ,0) , ite(C,1 ,0)))

=ite(A, ite(B, 1, ite(C,1 ,0)) , ite(C,1 ,0)) (7)

Equation 7 represents the **ite** expression for the final BDD. It can be seen that the root vertex is A. On its '1-branch' it has the expression ite(B, 1, ite(C,1 ,0)) i.e. it is connected to the vertex B which has 1 and ite(C,1 ,0) on its 1 and 0 branches respectively. The '0-branch' of A is connected to vertex C. This produces the BDD illustrated in figure 4.

The **ite** structure lends itself well to computer implementation.

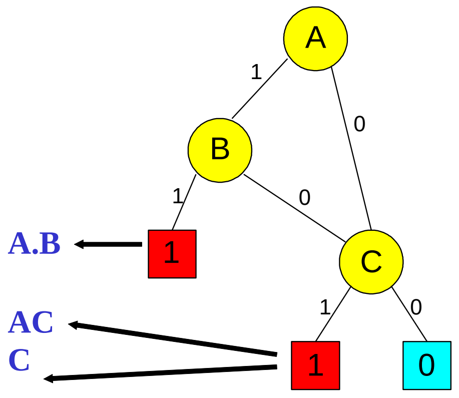


Figure 4 Minimal Cut Sets

2.3 BDD construction using the system logic equations

Whilst they are not needed in the quantification process, the BDD can be used to develop the minimal cut sets of the system. Consider the definition of a minimal cut set:

Cut Set: A list of component failed states which result in system failure

Minimal Cut Set**:** a cut set such that if any component failure is removed it no longer results in system failure.

Figure 4 shows a simple example BDD structure. Each path leading to a terminal-1 node contains the component conditions required for system failure. If the working component states are ignored and only the failed component states listed then this corresponds to a cut set. For the BDD shown in figure 4 there is a path to a terminal-1 passing through nodes A and B on their 1-branches. So AB is a cut set. There are two paths through the BDD to the second terminal-1 node. The first passes through the 1-branch of A, the 0-branch of B and the 1-branch of C, giving cut set AC. The second passes through the 0-branch of A and the 1-branch of C, giving cut set C. The cut sets are:

1. AB
2. AC
3. C

Since component A can be removed from cut set 2 and the system remains in the failed state, this is not a minimal cut set. Removing this from the list gives the minimal cut sets {AB}, {C}.

The above approach demonstrates the way minimal cut sets can be obtained in a system assessment however, for a full minimal cut set analysis, the efficient process would be to construct a zero-suppressed BDD which encodes only minimal cut sets. This is beyond the scope of this tutorial but details can be found in refs 6, 8.

2.4 Variable Ordering Schemes

The sequence specified for the component events when constructing the BDD structure plays a vital role in determining the size, and therefore the efficiency of the BDD produced to represent the logic function. Consider the fault tree given in figure 5.

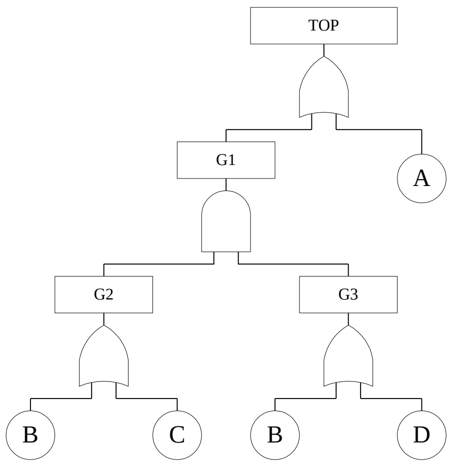


Figure 5 Fault Tree to Investigate Ordering Schemes

With ordering schemes of A<B<C<D and D<C<B<A the resulting BDDs are illustrated in figures 6a and 6b respectively.

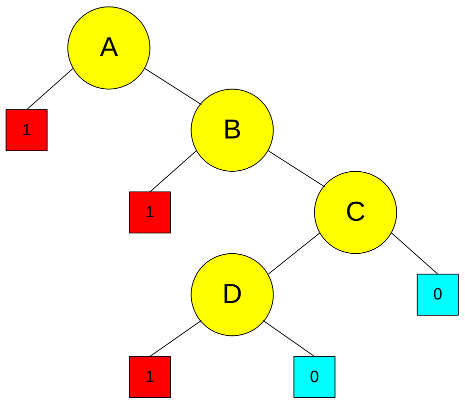


Figure 6a BBD for the Fault Tree in figure 5

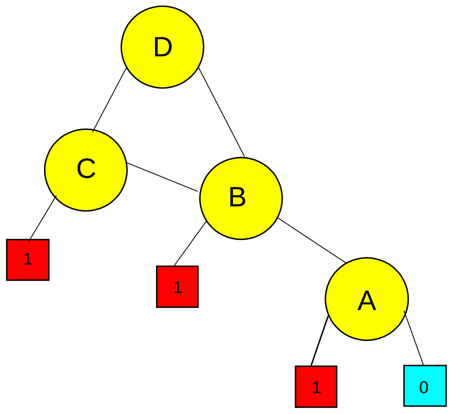


Figure 6b BBD for the Fault Tree in figure 5

The BDD in figure 6a has 4 intermediate nodes and produces 3 cut sets (all minimal). The BDD in figure 6b is not such an efficient representation with 4 intermediate nodes and 5 cut sets (2 are non-minimal). For a problem of this size the efficiency of the representation of the system failure logic is not important. However, as the size of the system grows this can be critical if a BDD is to be constructed. There is no universally accepted way in which the variable ordering can be specified and a number of approaches are possible [10]. The problem then becomes: given the characteristics of the original fault tree, select which strategy should be used to specify the variable ordering. Neural networks have been used for this selection process [11, 12].

Potential strategies to order the fault tree basic events are:

- 'neighbourhood' methods where the fault tree is traversed in a systematic way and basic events listed as they are encountered. One of the most common ordering methods is in this category and is known as 'top-down, left right' where the tree is traversed from top to bottom and on each level encountered from left to right listing the variables as they are encountered for the first time. (see figure 7)
- 'Structural importance' methods allow nodes to be selected from anywhere in the tree structure. Nodes are

allocated a ‘weighting’ which indicates their contribution to the top event. Highest ‘weightings’ ordered first.

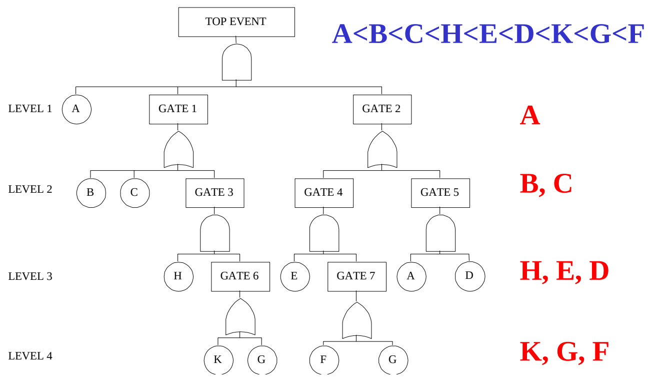


Figure 7 Top-down, Left-right Variable Ordering

2.5 Calculating the Top Event failure probability

It is with the system quantification that the BDD offers advantages over the conventional fault tree analysis method. Due to the way that the BDD is formed it has the characteristic that all paths through the BDD are disjoint. Consider any intermediate node X. A path will pass this node on either the 1-branch meaning that the event has occurred or the 0-branch meaning that it has not. These conditions are mutually exclusive and so anything which appears below the 1-branch, and therefore contains the component failed state, will be mutually exclusive to any path through the 0-branch. The branching at all other variables below this one results in all paths being disjoint.

The BDD therefore expresses the structure function in a disjoint form. The system failure probability, Q_{SYS}, is then simply the sum of the probability of all paths from the root node to a terminal-1 node (accounting for the failure/success of all components included in the path):

$$Q_{SYS} = \sum_{\text{all paths}} \text{P(path i to terminal - 1)} \tag{8}$$

Assuming an ordering of A<C<B<D, the BDD for the fault tree shown in figure 5 is given below (figure 8).

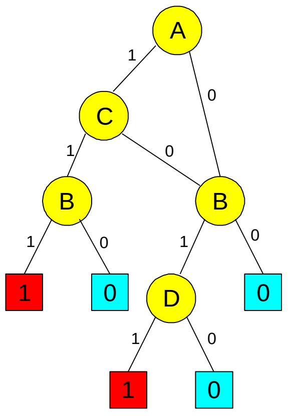


Figure 8 BDD for the Fault Tree in figure 5

The disjoint paths to a terminal-1 node are:

$$\begin{aligned} &1. \quad A.C.B \\ &2 \quad A.\overline{C}.B.D \\ &3 \quad \overline{A}.B.D \end{aligned} \tag{9}$$

Summing the probabilities of these paths gives:

$$\begin{aligned} Q_{SYS} &= q_a q_b q_c + q_a(1-q_c)q_b q_d + (1-q_a)q_b q_d \\ &= q_a q_b q_c + q_b q_d - q_a q_b q_c q_d \end{aligned} \tag{10}$$

The points to note about this process are:
1. the top event probability obtained is exact (approximations are not needed,
2. the minimal cut sets were not required as an intermediate step in the calculations.

This gives the BDD approach advantages in terms of both accuracy and efficiency over the conventional fault tree analysis approach.

2.6 Calculating the Top Event failure intensity

The system failure intensity, w_{sys}, the rate at which the system fails given that it was functioning at time t=0 can be calculated using the expression:

$$w_{SYS}(t) = \sum_{i=1}^{n} G_i(\underline{q}).w_i(t) \tag{11}$$

where w_i is the component failure intensity and $G_i(q)$ is the Criticality Function (Birnbaum’s measure of importance [13]). w_{sys}, is sometimes referred to as the unconditional failure intensity.

The ***Criticality Function***, $G_i(q)$, is the probability that the system is in a critical state for component i such that the failure of component i causes system failure. This can be evaluated using equation 12.

$$G_i(\underline{q}) = \frac{\partial Q_{sys}}{\partial q_i} = Q_{sys}(1_i, \underline{q}) - Q_{sys}(0_i, \underline{q}) \tag{12}$$

where $Q_{sys}(1_i,q)$ is the system failure probability with $q_i=1$ and $Q_{sys}(0_i,q)$ is the system failure probability with $q_i=0$.

3. DEPENDENCIES IN ENGINEERING SYSTEMS

Dependencies can occur between component failure events in many ways. As an example, a situation frequently encountered which introduces a dependency between its components is the use of standby redundancy. Generators used to replace a primary power source in the event that it fails is an example of such a system. Parallel pumping systems are another example. In standby systems, one component is operational and should it fail a backup component is activated. This type of system can be classified in three ways depending on how the backup component is considered to behave when it is non-operational. These classifications are known as *hot standby*, *warm standby and cold standby*.

Hot standby: components are considered as having the same failure rate in standby as in operation.

Warm standby: components are considered as having a lower failure rate in standby than in operation.

Cold standby: components are considered not to fail in standby.

Warm and Cold standby systems produce a dependency where the likelihood of failure of the standby component is dependent upon the state of the primary component.

Other examples of situations which introduce dependencies in engineering systems are shown in table 1.

When dependencies exist in a system, conventional fault tree analysis is not an appropriate means to predict its failure characteristics.

Type	Description	Example
Secondary Failure	When one component fails it increases the load on a second component which then experiences an increased failure rate	Two pumps both operational and sharing the load. Each pump has the capability to deliver the full demand should the other pump fail
Opportunistic Maintenance	A component fails which causes a system shutdown or that requires specialist equipment for the repair. The opportunity is taken to do work on a second component which has not failed but is in a degraded state	Components on a circuit board. Components in a sub-sea production module
Common Cause	When one characteristic (eg materials, manufacturing, location, operation, installation maintenance) causes the degraded performance in several components	Incorrect maintenance done on several identical sensors Impact breaks the circuit on cables routed in the same way to different redundant channels
Queueing	Failed components all needing the same maintenance resource are queued. Then repaired in priority order	Limited number of maintenance teams, equipment or spares

Table 1 Dependency types

4. MODELLING DEPENDENCIES

Two methods are considered for calculating the probability of systems which feature dependencies: Petri nets and Markov models. Petri nets are the most general of the two methods and can solve all situations that Markov models can solve and more beyond. However, in certain situations Markov can provide a more efficient solution than Petri nets and so the method has been incorporated for these situations. It is expected that most situations that involve complexities or dependencies in the system model will be solved using a Petri net.

4.1 Markov Analysis

A Markov model comprises two elements: nodes and transitions [14-16]. For system failure models, the nodes represent the possible states of the system. This is the complete set of mutually exclusive states in which a system can reside. The states are usually described by defining which components are working or failed. Transitions between states then represent component failures and repairs. Following the formulation of a Markov model its analysis will yield the probability of being in any of the states. The system failure probability is then determined by summing the probabilities of residing in those states which represent a system failure condition.

For the basic Markov approach to be applicable the system must be characterized by a lack of memory, that is, the future states of the system are independent of all past states except the immediately preceding one. So the future behaviour depends only on its present state, not the history of what has happened in the past. This is represented by the equation 13 where X_t is a random variable representing the state in which the system resides at time *t*.

$$P(X_{t+\mathrm{dt}} = k \mid X_t = j, X_{t-\mathrm{dt}} = i, X_{t-2\mathrm{dt}} = h, \dots, X_0 = a)$$
$$= P(X_{t+\mathrm{dt}} = k \mid X_t = j) \qquad (13)$$

Also, the system behaviour must not vary with time. The probability of making a transition from one state to another must be constant. This type of process is called stationary or homogenous. If the transition probabilities are functions of time then the process is not stationary and is known as non-Markovian.

With the lack-of-memory property, the likelihood of a component failure is only dependent on the fact that it is currently working. It does not matter that it may have been perfectly reliable up to this time or that it may have failed several times before.

The homogenous property means that the transitions between states are not dependent on time. They are, therefore, governed by a constant rate, and times between transitions are governed by the exponential distribution.

The first stage of the Markov analysis is to draw the state transition diagram. This takes the form of a directed graph where each node represents one of the discrete system states, and, for continuous time Markov models, the edges indicate the transition frequencies between the states in the direction indicated by an arrow drawn on the edge.

4.1.1 Example – Single-Component Failure/Repair Process

As an example consider the simple case of a single repairable component. The component undergoes an alternating sequence of failure and repair with constant rates. The component can exist in one of two states, working or failed. The state transition diagram for this situation is given in figure 9.

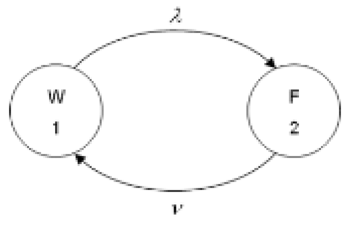


Figure 9. Repairable Component State Transition Diagram

The component can be considered to start in the working state at time t = 0. Transition from the working state, 1, to the failed state, 2, occurs with rate λ. Failure is immediately revealed and transition back to the working state, the repair process, occurs with rate v.

Let $P_w(t)$ denote the probability of the component working at time t and $P_f(t)$ the probability of the component being in the failed state at t. Differential equations relating these probabilities can be derived directly from the state transition diagram by considering the following rule for each state:

$$\frac{dP_{\text{state}}}{dt} = \text{(rate of entering state)} - \text{(rate of leaving state)} \tag{14}$$

Therefore:

$$\frac{dP_w(t)}{dt} = -\lambda P_w(t) + vP_f(t) \tag{15}$$

$$\frac{dP_f(t)}{dt} = \lambda P_w(t) - vP_f(t) \tag{16}$$

In matrix form this is:

$$\left[\dot{P}_w(t), \dot{P}_f(t)\right] = \left[P_w(t), P_f(t)\right]\begin{bmatrix} -\lambda & \lambda \\ v & -v \end{bmatrix} \tag{17}$$

i.e. $$\dot{\boldsymbol{P}} = \boldsymbol{P}\,\mathbf{A} \tag{18}$$

where **A** is the state transition matrix.

4.1.2 General Markov State Transition Model Construction

The two-state Markov model representing a single component described in the previous section is the simplest model possible. The method is more commonly used to model more complex systems comprising several components. States on the Markov model are required to be mutually exclusive and exhaustive. That is, they must be non-overlapping and represent every possible state in which the system can reside. A possible way to generate the system states is to identify the functionality or failure mode for each component in the system and list all possible combinations. Sometimes it is easy to write down all possible states; for example, for a system with two components which either work or fail, the Markov state transition diagram is show in figure 10. For components A and B, either both components work, state 1, or both components fail, state 4, or there are two ways in which one component works and one fails, states 2 and 3. The transitions rates λ_A, λ_B and repair rates v_A, v_B are incorporated on the diagram.

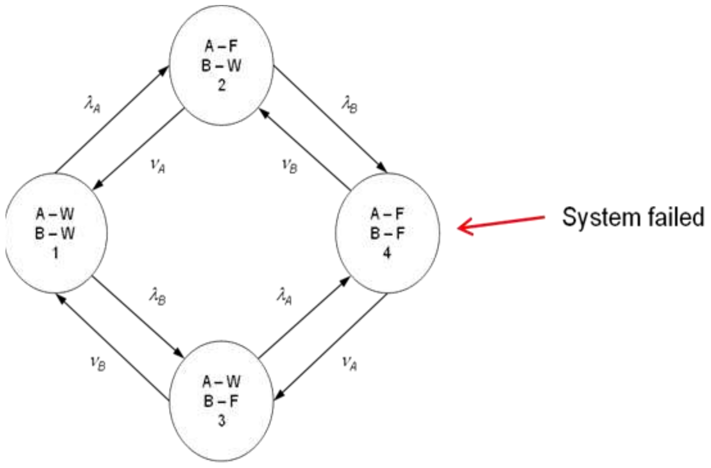


Figure 10. Two-Component Markov Model

The state transition matrix can be formulated directly from the transition diagram using the following rules:

(a) the dimensions of the matrix are equal to the number of states in the model;
(b) an off-diagonal element in row i column j represents the transition rate from state i to state j;
(c) a diagonal element row i, column i is the transition rate out of state i (always negative). (i.e. all rows sum to zero)

Using these rules to form the state transition matrix for the two-component system whose Markov diagram is illustrated in figure 10 gives:

$$\mathbf{A} = \begin{bmatrix} -(\lambda_A + \lambda_B) & \lambda_A & \lambda_B & 0 \\ v_A & -(v_A + \lambda_B) & 0 & \lambda_B \\ v_B & 0 & -(v_B + \lambda_A) & \lambda_A \\ 0 & v_B & v_A & -(v_A + v_B) \end{bmatrix} \tag{19}$$

with initial conditions:

$$P_1(0) = 1.0, \;\; P_2(0) = P_3(0) = P_4(0) = 0.0$$

4.2 Petri Nets

The Petri net is an alternative approach to Markov models for solving State-Space problems. It has been used to represent the dynamic processes in 'systems' occurring in science, engineering and business and was developed by Petri [17] in 1963.

A Petri Net [18] is a graphical model having fundamental elements: places, transitions, arcs and tokens. A place represents a condition or event in the system and is illustrated by a circle. A token, denoted by a dot, is located in a place to represent the existence of that condition. A transition allows the tokens to move between places in the model which represents the dynamic changes in condition of the system. The transition appears as a rectangle on the graph. Arcs are used to connect input places to transitions and transitions to output places. A number, known as the multiplicity, can be associated with any arc. If no multiplicity is stated then it has a default value of 1. The state of the system at any time in the simulation is characterised by the marking of the net which records the number of tokens residing in each place. A small example Petri net is shown in figure 11.

There are rules which govern the way that transitions 'fire' to move tokens around the network. In order for a transition to fire it must first be enabled. A transition is enabled when all of its input places contain at least the number of tokens as the multiplicity of the connecting arc. The transition has an associated time, specified by a value or a distribution. After a time delay specified or sampled from the appropriate distribution, the transition fires and removes the multiplicity of tokens from the input places and deposits the multiplicity of tokens to the output places. This is illustrated by the transition D_1 for the Petri net at the top of figure 11.

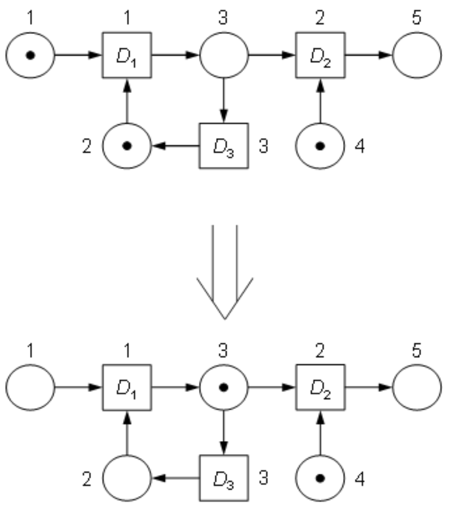


Figure 11. Simple Petri Net Transition Firing Process with Arc Multiplicities and an Inhibitor Arc.

Another feature of the basic Petri Net representation is the inhibitor arc which is used to prevent a transition from firing. As shown in figure 12, it appears on the graph as an arc (with associated multiplicity) with a round end connecting a place to a transition. When the arc input place contains at least the multiplicity of tokens of the arc the transition is inhibited and will not fire.

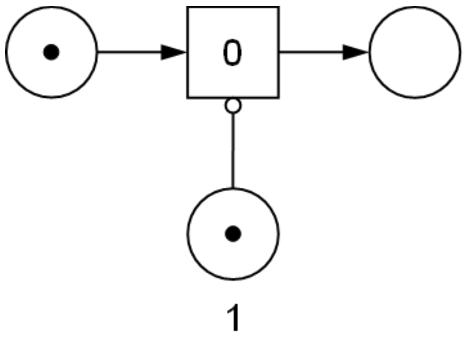


Figure 12. Inhibitor Arc

The solution of the Petri net is obtained using Monte Carlo simulation [19]. This performs a large number of model simulations deriving the times at which the transitions occur by taking random samples from their associated governing distributions. For each simulation the durations of residing in places, representing the system key performance parameters or the number of times the key places are entered, are recorded. In this way, distributions of the residence times, or the number of occurrences of these system states, can be established. It is common to use the averages of these distributions to judge the system performance.

The Petri nets used in the system modelling can extend the commonly used transition types to enhance the efficiency of the modelling capability.

4.2.1 Special Transitions

The enabling and firing processes for the newly defined transition types [20] are the same as those for the standard transitions used in the traditional Petri net method. However, the transitions also contain properties to execute additional tasks concisely.

- Periodic transition: This transition fires only when the system time is at a specified value. It can be used to represent the inspection process where the condition of an element is revealed through inspection performed at regular intervals.
- Reset transition: This transition resets the marking of specified places in the Petri net to some desired state. It has an associated list of places and the number of tokens that they will contain after the reset. Its use is in initialising the relevant Petri net places when actions such as a renewal are performed.
- Conditional transition: This transition type enables the firing time distribution to be dependent upon the number of tokens residing in another place on the network. A dashed line links this place to the transition. This enables degradation times to be linked to the number of prior interventions that have been performed.

In the stochastic models which will link the degradation and maintenance processes the transition times between the states must be specified. These are usually obtained from historic data collected to monitor the performance of this system or similar systems. Data for the times to any event occurrences are then used to define a statistical distribution which represents the transition times. It is possible to use any statistical distribution in this process, however, common choices for reliability type problems are the exponential, Weibull and lognormal distributions.

4.3 Dependency Modelling

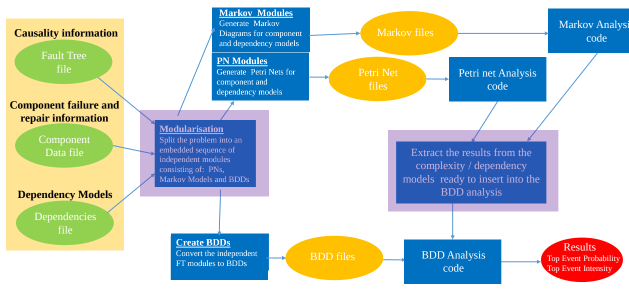
Whichever of the methods, Markov or Petri net, is used to address the dependencies, problems can be experienced if it is used to model the whole system. With Markov models there can be a state space explosion where the number of states grows exponentially with the number of components in the system. For a Petri net solved using Monte Carlo simulation, it can take substantial computer resources to achieve convergence for systems whose failure is rare.

Markov or Petri net models for the whole system also lack the causality structure of fault trees making peer review and auditing of the models difficult.

5. D^{2}T^2 MODELLING REQUIREMENTS

The approach taken in the Dynamic and Dependent Tree Theory (D^{2}T^2) modelling framework, which overcomes the limitations of conventional fault tree analysis and retains the fault tree structure to represent the failure, is shown in Figure 13. The objective is to perform the analysis efficiently which means that:

i. the dependency models are minimised containing only those events which are mutually dependent. (Rather than sections of the fault tree).
ii. the sizes of the BDDs are also minimised by using effective variable ordering and modularisation approaches.



Causality information
Fault Tree file
Component failure and repair information
Component Data file
Dependency Models
Dependencies file
Modularisation
Split the problem into an embedded sequence of independent modules consisting of: PNs, Markov Models and BDDs
Markov Modules
Generate Markov Diagrams for component and dependency models
PN Modules
Generate Petri Nets for component and dependency models
Markov files
Petri Net files
Markov Analysis code
Petri net Analysis code
Extract the results from the complexity / dependency models ready to insert into the BDD analysis
Create BDDs
Convert the independent FT modules to BDDs
BDD files
BDD Analysis code
Results
Top Event Probability
Top Event Intensity

Figure 13 D^{2}T^2 analysis framework

For a conventional fault tree analysis the input data will specify the structure of the fault tree along with the failure/repair characteristics for the components. For the D^{2}T^2 modelling framework, the fault tree structure file remains the same. The component failure/repair models are no longer limited to exponential failure time and repair time distributions. Any distribution can be accommodated for the state transition times. As shown in figure 13, these two files are supplemented by a third file which defines any dependencies between events and the models, either Markov or Petri net, which represent them.

The analysis then proceeds to identify the smallest modules that are independent of the rest of the system structure for analysis by either BDD, Petri net or Markov approaches. This part is achieved through the modularisation described below. The BDD, Petri net and Markov modelling solutions have been described in sections 2, 4.1 and 4.2 respectively. The results of these are integrated as described in section 7 below. The integration of the results will deliver the system failure probability or frequency.

6. FAULT TREE MODULARISATION

When the basic events in the fault tree are independent then there are two very effective approaches to finding independent modules of the tree. The first was used in the Faunet code from Riso [21], the second is a linear time algorithm which systematically transverses the fault tree [22]. Whilst very effective at reducing the complexity of the problem when the basic events are independent they do not provide the smallest possible independent modules. These approaches have been revised to accommodate the dependencies and are presented below. For an effective reduction in the problem, to achieve objective ii in section 5, they are applied one after the other. Prior to their application dependency groups are identified. These are groups of components which may experience at least one dependency between them but are mutually independent from all other component failures in the fault tree.

6.1 Fault Tree Factorisation

This approach repeatedly applies three operations until they can be no-longer applied. The operations are:

i. Contraction
Subsequent gates of the same type are contracted into a single gate

ii. Factorisation
Identifies factors, expressed as groups of events that always occur together in the same gate type. The factors can be any number of events if they satisfy the following:
- All events in the group are independent and either initiators or enablers [23].
- All events in the group feature a dependency and contain all events in the same dependency group.

iii. Extraction
Restructures fault tree structures where a repeated event can be extracted, as shown in Figure 14.

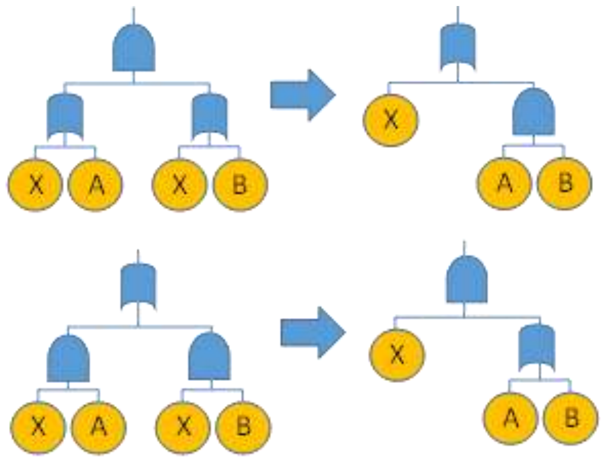


Figure 14 Extraction operation

For those independent factors identified in step ii their probabilities, Q_{cfi}, and frequencies, w_{cfi}, are calculated, in terms if element probabilities and intensities, q_j, w_j using:

For OR combinations $Cf_i = x_1+x_2+\ldots+x_n$

$$Q_{cf_i} = 1 - \prod_{j=1}^{n}(1 - q_{x_j}) \tag{20}$$

In the event that the factor contains only initiators:

$$w_{cf_i} = \sum_{j=1}^{n} w_j \prod_{\substack{k=1 \\ k\neq j}}^{n}(1 - q_{x_k}) \tag{21}$$

For AND combinations $Cf_i = x_1.x_2\ldots\ldots x_n$

$$Q_{cf_i} = \prod_{j=1}^{n} q_{x_j} \tag{22}$$

In the event that the factor contains only initiators:

$$w_{cf_i} = \sum_{\substack{j=1 \\ initiators}}^{n} w_j \prod_{\substack{k=1 \\ k\neq j}}^{n} q_{x_k} \tag{23}$$

For factors with dependencies that are solved using Markov or Petri net models, the required factor probabilities and intensities are extracted directly from the model.

6.2 Linear-time modularisation

Attempts are then made to further simplify the fault tree structure which remains after applying the factorisation modularisation. Each basic event appearing in a dependency group is then replaced in the fault tree structure with its dependency group label. The linear time algorithm is then applied as described in [22] and may result in the fault tree being broken down into further independent modules.

7. D^{2}T^2 DYNAMIC AND DEPENDENT FAULT TREE ANALYSIS

In utilising the BDD method, it is possible to exploit the fact that the paths are all mutually disjoint to integrate the results from the dependency models into the fault tree structures. The disjoint nature of the paths means that it is only necessary to deal with dependencies between the events occurring on any path to a terminal-1 and not the dependencies occurring between the paths. The system failure probability is then the sum of the probability of the paths to a terminal-1, which is a function of the probability of the event combinations in any dependency group k, $Dpath_j^k$, and the independent component failures, $Ipath_j$, evaluated using equation 24.

$$Q_{SYS} = \sum_{j=0}^{npath}\left[P\left(Ipath_j\right).\prod_{k=1}^{ndep} P(Dpath_j^k)\right] \tag{24}$$

In order to evaluate the system failure intensity, the probability equation, (24) is used to evaluate the criticality function which is used as expressed in equation 11.

8. CASE STUDY – PLANT COOLING SYSTEM

The above method is demonstrated through its application to a plant cooling system. The system is illustrated in figure 15.

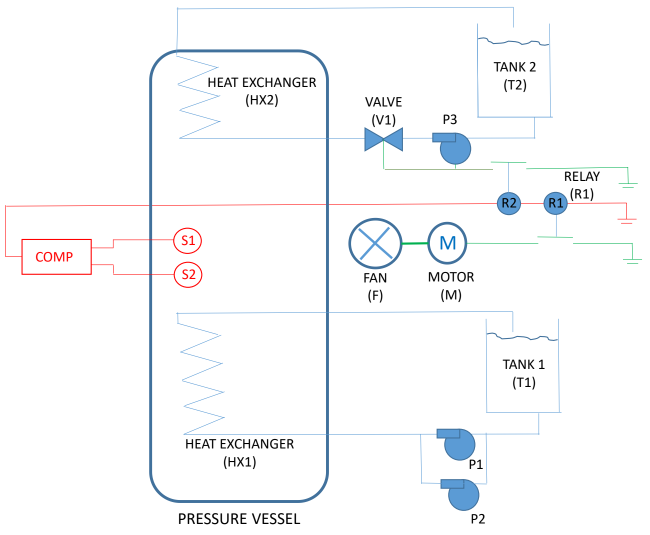


Figure 15 Plant Cooling System

The cooling system features a primary, normally operational, cooling sub-system, comprising a tank (T1), pumps (P1 and P2) and heat exchanger (Hx1). Both pumps normally function but should either fail, the coolant to the heat exchanger can be supplied by a single pump.

The temperature of the pressure vessel is monitored by a sub-system featuring two temperature sensors (S1 and S2). The signals from the sensors feed into a computer (Comp) and should either sensor indicate a higher than expected vessel temperature, the computer will activate an alternative means of cooling. This is achieved by de-energising an output which removes power from the two relays (R1 and R2).

When R1 de-energises its contacts close and powers the motor for the secondary cooling fan system. R2 acts similarly to open motorized valve V1 and power pump P3 in the secondary cooling water system which draws water from tank (T2) and delivers it to heat exchanger (Hx2).

The valve V1 and pumps P1, P2 and P3 all have a common power supply, PoW.

There are three dependencies which feature in the system:

i. Pumps P1 & P2 – if one fails it puts increased load on the other (and increases its failure rate).
ii. Heat Exchangers Hx1 & Hx2 – since specialist equipment is needed when one needs replacement, the opportunity is taken to replace both.
iii. Pump P3 - can fail to start when the demand occurs (P3S) and fail once running to provide cooling for the required time (P3R). The two events P3S and P3R are clearly dependent.

Failure and repair data for the basic events in the fault tree, accounting for the dependent failures, is given in table 2.

There is an additional complexity. As can be seen from table 2, the motor does not have constant failure and repair rates. Its failure times are governed by a Weibull distribution and its repair times are given by a lognormal distribution.
The dependencies and the complexity are solved using appropriate Petri nets and Markov models as discussed in the next section.

8.1 Complexity and dependency models

8.1.1 Motor failure model

The motor has failure times from a Weibull distribution, Weib(β=1.5, η=1200 hours), and repair times distributed by LogN(μ=24.0 hours, σ=4.8 hours). This is solved using a very simple Petri net as shown in Figure 16 and the results shown in table 3.

Event Code	Description	I / E	D-Group	Failure rate (/hour)	Mean time to repair (hours)	Inspect interval (hours)	q	w
P1, P2	Pumps fail when running	I	D1	Failure rate $\lambda_1 = 2 \times 10^{-5}$ /h under normal load $\lambda_2 = 5 \times 10^{-3}$/h under full load Repair rate ν= 0.041667 (MTTF = 24hrs)				
T1	Water Supply failure	I		$1x10^{-5}$	24		$2.4x10^{-5}$	9.99976 $x10^{-6}$
Hx1	Heat Exchanger fails	I	D2	Failure time = W(β=2.5, η=30,000h) The system is shut down when the repair is undertaken				
PoW	Power supply failure	I		$1x10^{-4}$	10		$1x10^{-3}$	9.99 $x10^{-5}$
S1, S2	Sensor fails to detect a high temperature	E		$5x10^{-4}$	5	730	0.185	
Comp	Computer fails to process sensor signals	E		$5x10^{-5}$	5	2190	0.055	
R1 / R2	Relay contacts fail to close	E		$1x10^{-5}$	24	2190	0.0112	
Fan	Fan fails	E		$2x10^{-6}$	8	2190	2.206 $x10^{-3}$	
Motor	Fan motor fails	E	C1	Failure time = W(β=1.5, η=12,000h) Repair time = LogN(μ=24hrs, σ=4.8h)				
P3S	Pump fails to activate	E	D3				0.05	
P3R	Pump fails when running	E	D3	$1x10^{-4}$				
T2	Water Supply failure	E		$1x10^{-5}$	24	2190	0.0112	
Hx2	Heat Exchanger fails	E	D2	Failure time = W(β=2.5, η=30,000h) The system is shut down when the repair is undertaken				
V1	Valve fails to open	E		$5x10^{-5}$	30	2190	0.05625	

Table 2 Component failure and repair data.

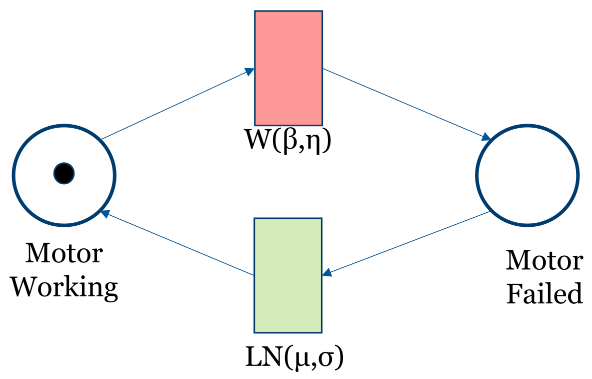


Figure 16 Motor failure and repair Petri net

STATE	Probability	Frequency (per hour)
Motor Failed	0.0058389642	8.686868 x 10^{-5}

Table 3 Quantification results from the Motor Petri net in figure 16

8.1.2 Hx1 and Hx2 failure dependency Petri net model

The Petri net used to model the maintenance repair dependency between the two heat exchangers is shown in figure 17.

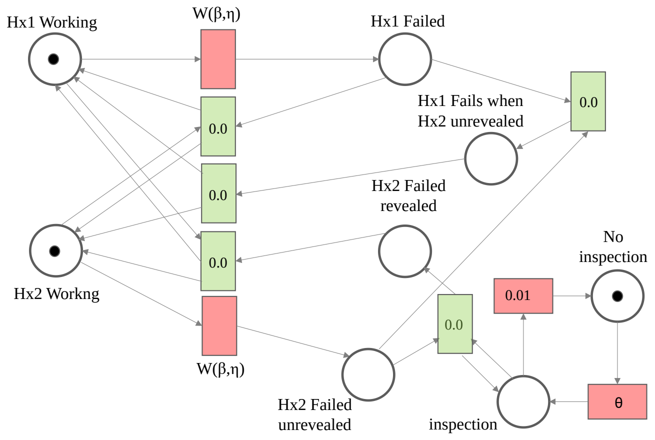


Figure 17 Heat exchanger dependency Petri net model

The results of simulating the Petri net model are:

State Probabilities:

$P(Hx1_W, Hx2_W)=0.98646987828725829$
$P(Hx1_W, Hx2_F)=0.0135301$
$P(Hx1_F, Hx2_F)=0.0$
$P(Hx1_F)=0.0$
$P(Hx2_F| Hx1_F)=0.0$
$P(Hx2_F| Hx1_W)= 0.0135301$

(25)

State Failure Intensities

$w(Hx1_F, Hx2\_unrevealed)=3.1709792 \times 10^{-07}$ /hour
$w(Hx1_F, Hx2_W)=1.8161063 \times 10^{-05}$ /hour
$w(Hx1_F)=1.8478161 \times 10^{-05}$ /hour

8.1.3 Pumps P1 and P2 failure Markov model

Since pumps P1 and P2 in the primary cooling circuit have constant failure and repair rates, both Petri nets and Markov methods could be used to model their dependency. The Markov method is selected in this case since it offers a more efficient solution.

The Markov state transition diagram is shown in figure 18.

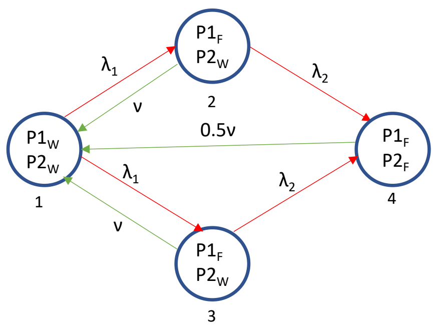


Figure 18 Markov model of the P1 and P2 dependency

The results from the Markov analysis are shown in table 4.

State Number	State	State Probability	Intensity Expression	State Intensity
1	$P1_W P2_W$	0.99743518	$w1 = (Q2 + Q3).\nu + Q4.0.5\nu$	7.12456×10^{-5}
2	$P1_F P2_W$	0.00042747	$w2 = Q1.\lambda_1$	1.99487×10^{-5}
3	$P1_W P2_F$	0.00042747	$w3 = Q1.\lambda_1$	1.99487×10^{-5}
4	$P1_F P2_F$	0.00170988	$w4 = (Q2 + Q3).\lambda_2$	4.2747×10^{-6}

Table 4 results from the Markov model shown in figure 18

8.1.4 Dependent failure modes for pump P3

The probabilities of the two dependent failure modes for pump P3, failure to start, P3S, and failure once running, P3R, can be used to deliver the failure probability of P3 to work for the required period using:

$$q_{P3} = q_{P3S} + (1.0 - q_{P3S})\lambda_{P3R}.t_{period} \qquad (26)$$
$$= 0.05 + 0.95 \times 10^{-4} \times 30 = 0.05285$$

8.2 Failure of the Plant Cooling System Fault Tree

The fault tree for the pressure vessel cooling system failure is shown in figure 19. The branches of the fault tree are terminated in basic events, component failure events listed in table 2. All events on the left-hand branch of the fault tree represent the causes of failure of the primary cooling system. These are all initiators [23]. Events on all other branches, representing the failure of safety systems to respond, are enablers.

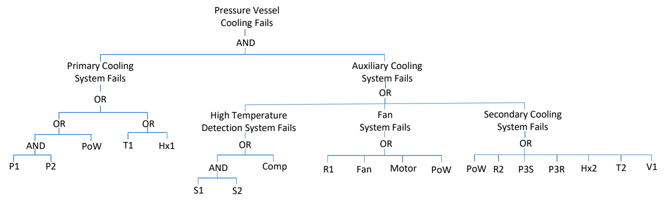


Figure 19 Pressure Vessel Cooling System Failure Fault Tree

8.3 Fault Tree modularisation

8.3.1 Fault Tree Factorisation

Applying the Factorisation algorithm, shown in section 6.1, to the fault tree in figure 19 is accomplished in the following steps:

Contraction 1

The OR gates which lead into OR gates for both the 'primary cooling system fails' and 'auxiliary cooling system fails' events are contracted into a single OR gate structure. The resultant fault tree is shown in figure 20.

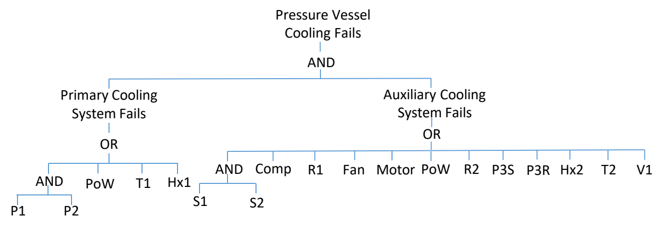

Pressure Vessel Cooling Fails
AND
Primary Cooling System Fails
OR
AND
P1
P2
PoW
T1
Hx1
Auxiliary Cooling System Fails
OR
AND
S1
S2
Comp
R1
Fan
Motor
PoW
R2
P3S
P3R
Hx2
T2
V1

Figure 20 Contracted fault tree structure

Factorisation 1
Following the rules for the identification of factors given in section 6.1, the following factors can be identified in the fault tree in figure 20.

$Cf_1 = P1.P2$ (dependency group D1 – initiators)
$Cf_2 = S1.S2$ (independent enablers)
$Cf_3 = Comp + R1 + Fan + Motor + R2 + T2 + V1$ (independent enablers)
$Cf_4 = P3S + P3R$ (dependency group D3 – enablers) (27)

Substituting these factors into the fault tree reduces it to that shown in figure 21.

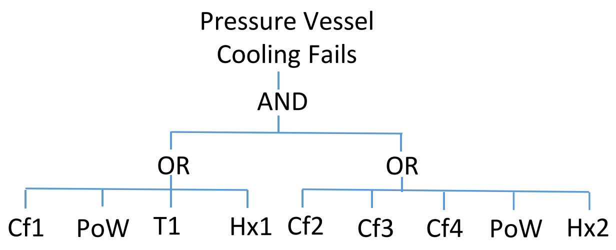


Figure 21 Factorized fault tree

Extraction 1
The fault tree in figure 21 can be seen to have a repeated basic event representing the power failure to the pumps and valve, PoW. Restructuring the fault tree as shown in the top diagram in figure 14, produces the structure shown in figure 22.

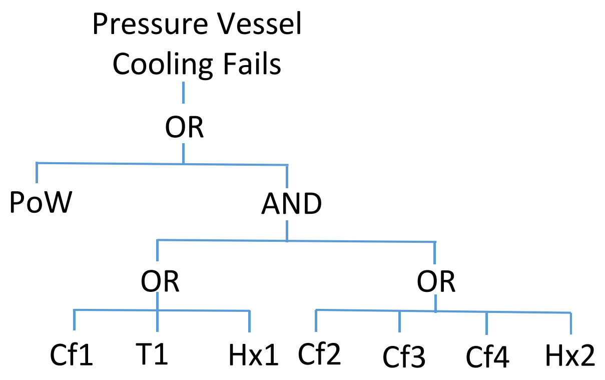


Figure 22 Fault Tree after Extraction 1 stage

The three stages are now applied for a second time. Since the fault tree is now already in an alternating sequence of AND and OR gates the application of Contraction 2 does not change anything.

Factorise 2
This second application of factorization picks out a second list of factors:

$Cf_5 = Cf_1 + T1$ (initiator) (28)
$Cf_6 = Cf_2 + Cf_3 + Cf_4$ (enabler)

Substituting these factors into the fault tree structure gives the fault tree illustrated in figure 23. This is the minimal fault tree structure as far as the factorisation algorithm is concerned. This has significantly reduced the complexity of the fault tree structure from the original tree, shown in figure 19 which had:

19 basic events, 10 gates
34 min cut sets (1 order 1, 20 order 2, 12 order 3 and 1 order 4)

to the tree in figure 23 which has:

5 basic events, 4 gates
5 min cut sets (1 order 1, 4 order 2)

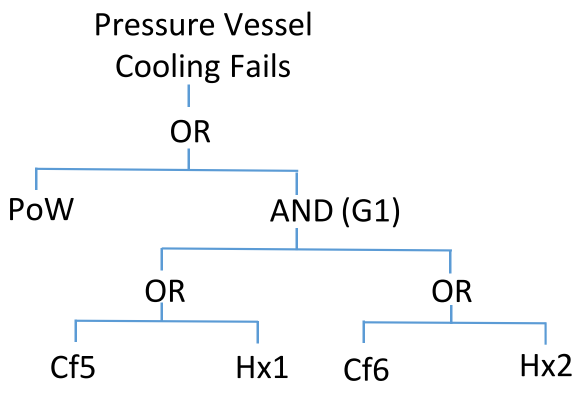


Figure 23 Final fault tree from the factorization algorithm

8.3.2 Linear Time Modularisation Algorithm

Application of the linear time algorithm of Rauzy and Dutuit [22] reduces the tree still further. It identifies gates top event and gate G1 in the fault tree in figure 23 to be modules. These modules are shown in figure 24. These are the fault trees which can be solved using the BDD method (or as a simple factor Cf_7 = PoW + G1).

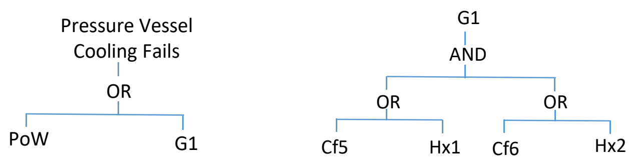


Figure 24 final fault tree independent modules

8.4 Quantifying the independent basic events and factors

The quantification of the fault tree requires the probability of the independent events and factors to be calculated. For those basic events that are not involved in any dependency or complexity models, then equations 1-3 are used, as appropriate. The results are provided in table 2.

The modularisation process identified 6 independent factors. The probability of these are calculated using the appropriate equations 20, 22. Factor Cf2 is an AND combination of independent basic events, S1 and S2, and so equation 22 is used. Cf3, Cf5 and Cf6 are OR combinations of independent events and so equation 20 is appropriate. It should be noted that *motor* is a basic event input to Cf3 and, although this is independent from the other inputs to Cf3, it features a complexity which means its likelihood is evaluated using the Petri net in figure 16. The probability of Cf1, which is the probability that P1 and P2 both fail, is extracted from the Markov model shown in figure 18. The final factor, Cf4, also comprised of dependent events, in this case P3R and P3S. The probability of P3 failing to function on demand and continue for the required duration is given by equation 26.

A list of the probabilities of factors Cf1-Cf6 are provided in table 5.

Event Code	**Description**	**I / E**	**D-Group**	**q**
Cf1	P1. P2	I	D1	0.00170988
Cf2	S1. S2	E		0.00034225
Cf3	Comp + R1 + Fan + Motor + R2 + T2 + V1	E		0.6035094
Cf4	P3S + P3R	E	D3	0.05285
Cf5	Cf_1 + T1	E		0.0017338
Cf6	$Cf_2 + Cf_3 + Cf_4$	E		0.6246519
G1	BDD	I		0.001091749
Cf7	PoW + G1			0.0020906577

Table 5 factor probabilities

8.5 Analysis of the Fault Tree

8.5.1 Top Event failure probability

The fault tree which needs to be analysed is that illustrated in figure 24 with the top event G1. It contains independent events Cf5 and Cf6 along with dependent events Hx1 and Hx2.

To perform the analysis of this fault tree first requires its conversion to a BDD. An ordering is therefore placed on the basic events, Cf5 < Hx1 < Cf6 < Hx2. This produces the BDD shown in figure 25.

The quantification of the BDD probability requires that the dependency between Hx1 and Hx2 is accounted for when evaluating equation 24. There are 4 paths through the BDD, as listed in table 6, the probability of each of the paths is evaluated, giving:

$$Q_{path1} = P(Cf5_1).P(Cf6_1) \; = \; 0.0010830 \qquad (29)$$

$$Q_{path2} = P(Cf5_1).(1 - P(Cf6_1)).P(Hx2_1) = 8.8052957\text{e-}06 \qquad (30)$$

$$Q_{path3} = (1 - P(Cf5_1)).P(Cf6_1).P(Hx1_1) = 0.0 \qquad (31)$$

$$Q_{path4} = (1 - P(Cf5_1)).(1 - P(Cf6_1)).P(Hx1_1, Hx2_1) = 0.0 \qquad (32)$$

j	*path$_j$*	*Ipath$_j$*	$Dpath_j^1$
1	*Cf5$_1$, Cf6$_1$*	*Cf5$_1$, Cf6$_1$*	
2	*Cf5$_1$, Cf6$_0$, Hx2$_1$*	*Cf5$_1$, Cf6$_0$*	*Hx2$_1$*
3	*Cf5$_0$, Hx1$_1$, Cf6$_1$*	*Cf5$_0$,Cf6$_1$*	*Hx1$_1$*
4	*Cf5$_0$, Hx1$_1$, Cf6$_0$, Hx2$_1$*	*Cf5$_0$, Cf6$_0$*	*Hx1$_1$, Hx2$_1$*

Table 6 paths through the BDD in figure 25

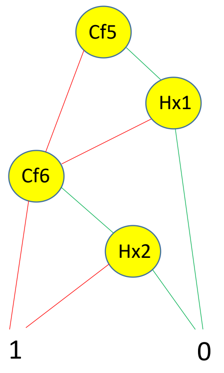


Figure 25 BDD for the fault tree with top event G1

Summing equations 29 to 32 gives the probability of event G1 as 0.00109175.

Finally, the probability that the pressure vessel temperature protection system fails is calculated from:

$$P(Top) = P(G1 + PoW) = 0.0020906577 \qquad (33)$$

8.5.2 Top Event failure intensity

For the top event failure intensity to be evaluated then the criticality function for each initiating event (P1, P2, PoW, Hx1 and T1) is calculated using equations 11 and 12. The results obtained are shown in table 7.

Variable	Q(var=F)	Q(var=W)	G_i(var)	$G_{i\ (var)}$ w
Hx1				$1.152147238 \times 10^{-5}$
T1	0.6300421	0.0020756	0.6279665	6.2795143×10^{-6}
P1	0.5042367	0.1268205	0.3774162	8.3356331×10^{-6}
P2	0.5042367	0.1268205	0.3774162	8.3356331×10^{-6}
PoW	1.0	0.0010918	0.9989082	9.979093×10^{-5}

Table 7 calculating the system failure intensity

Summing the intensities over the initiators gives the failure rate of the pressure vessel cooling system as *1.342632×10^{-4} / hour*.

9. CONCLUSIONS

- Dynamic and Dependent Tree Theory, D^2T^2, enables the evaluation of fault trees which are not limited by the restrictions which apply to conventional fault trees solved by traditional Kinetic Tree Theory.
- The analysis algorithm utilises BDDs, Petri Nets and Markov Models.
- Retains the familiar and popular fault tree causality structure.
- The Petri net and Markov models dedicated to solve the complexities and dependencies are minimal in size.
- Modularisation of the fault tree minimises the size of the BDD utilised in the system evaluation (and therefore the number of paths).

10. REFERENCES

1. W.E. Vesely, 'A Time Dependent Methodology for Fault Tree Evaluation', *Nuclear Design and Engineering*, no. 13 (1970): 337-360.
2. Watson H.A., Launch Control Safety Study, Bell Telephone Laboratories, Murray Hill, N.J. USA, 1961.Akers B, 'Binary Decision Diagrams', *IEEE Trans on Computers*, 27(6), 509-516, 1978.
3. Bryant R, 'Graph Based Algorithms for Boolean Function Manipulation', *IEEE Trans on Computers*, 35(8), 677-691, 1986.
4. Schneeweiss W., 'Fault Tree Analysis Using Binary Decision Diagrams', *IEEE Trans on Reliability*, 34(5), 453-457, 1985.
5. Rauzy A, 'New Approaches for Fault Tree Analysis', *Reliability Engineering and System Safety*, 05(59), 203-211, 1993.
6. Sinnamon R.M. and Andrews J.D., 'Quantitative Fault Tree Analysis Using Binary Decision Diagrams', *European Journal of Automation*, 30 (8), 1996, 1051-1071.
7. Sinnamon R.M and Andrews J.D., 'Improved Efficiency in Qualitative Fault Tree Analysis', *Quality and Reliability Engineering International*, Vol 13, 1997, pp293-298.
8. Sinnamon R.M and Andrews J.D., 'Improved Accuracy in Quantitative Fault Tree Analysis', *Quality and Reliability Engineering International*, Vol 13, 1997, pp285-292.
9. Bartlett L.M. and Andrews J.D, 'An Ordering Heuristic to Develop the Binary Decision Diagram based on Structural Importance', *Reliability Engineering and System Safety*, Vol 72, 2001, pp31-38.
10. Bartlett L.M. and Andrews J.D., 'Efficient Basic Event Ordering Schemes for Fault Tree Analysis', *Quality and Reliability Engineering International*, Vol 15, No 2, 1999, pp95-103.
11. Bartlett L.M. and Andrews J.D, 'Selecting an Ordering Heuristic for the Fault Tree to Binary Decision Diagram Conversion process using Neural Networks', *IEEE Transactions on Reliability*, Vol 51 No 3, Sept 2002, pp 344-349.
12. Z.W.Birnbaum, 'On the importance of different components in a multi-component system', *Multivariate Analysis 11*, P.R.Krishnaiah, ed.,Academic Press, 1969
13. Andrews, J.D. and Moss, T.R, Reliability and Risk Assessment, Second Edition, Professional Engineering Publishers, Ltd, 2003.
14. O'Connor, P.D.T and Kleyner, A, Practical Reliability Engineering, 5th Edition, Wiley, 2012.
15. Rausand M, Reliability of Safety-Critical Systems: Theory and Applications, Wiley, 2014.
16. Elsayed, E., Reliability Engineering, 2nd Edition, Wiley, 2012.
17. Petri, C.A., Fundamentals of a Theory of Asynchronous Information Flow. Proc. of IFIP Congress 62. Amsterdam: North Holland Publ. Comp., 1963: p. 386-390.
18. Schneeweiss, W.G., Petri Nets for Reliability Modelling, LiLoLe- Vrelag Publishing Company Limited, 1999.
19. Rubinstein, R and Kroese, D, Simulation and the Monte Carlo Method, 2nd Edition, Wiley, 2011.
20. Andrews, J., A Modelling Approach to Railway Track Asset Management, Proceedings of the Institution of Mechanical Engineers, Part F: Journal of Rail and Rapid Transit, 2013. 227(1): p. 56-73.
21. Platz, O. and Olsen J. V. "FAUNET: A Program Package for Evaluation of Fault Trees and Networks", Research Establishment Riso, Report No 348, DK-4000 Roskilde, Denmark, Sept. 1976.
22. Dutuit, Y. and Rauzy, A. A Linear-Time Algorithm to find Modules of Fault Trees, IEEE Trans. Reliability, 45, No. 3, 1996.

BIOGRAPHIES

John Andrews, Ph.D, FIMechE, CEng, MIMA, CMath, MSaRS
Professor of Infrastructure Asset Management
Head of the Resilience Engineering Research Group
University of Nottingham
Faculty of Engineering,
University Park
Nottingham, NG7 2RD, England

email: john.andrews@nottingham.ac.uk

John Andrews is Professor of Infrastructure Asset Management in the Faculty of Engineering at the University of Nottingham, UK. He is also the Head of the Resilience Engineering Research Group. He moved to Nottingham in 2009 having previously worked for 20 years at Loughborough University. The focus of his research has been on methods for predicting system reliability and availability in terms of the component failure probabilities and a representation of the system structure. Much of his early work has concentrated on the Fault Tree technique and the use of the Binary Decision Diagrams (BDDs) as an efficient and accurate solution method. More recently his main interest has been on modelling the effects of maintenance in order to identify the optimal strategy for asset management. He is the author of around 350 research papers on this topic and is joint author, with Bob Moss, of a text book, Reliability and Risk Assessment, now in its second edition, published by ASME. John was the founding Editor of the Journal of Risk and Reliability and is a member of the Editorial Boards for Reliability Engineering and System Safety, and Quality and Reliability Engineering International.

Silvia Tolo, BSc, Ph.D
Research Fellow in Risk and Reliability Engineering
Resilience Engineering Research Group
University of Nottingham
Faculty of Engineering,
University Park
Nottingham, NG7 2RD, England

email: silvia.tolo@nottingham.ac.uk

Dr Silvia Tolo gained an M.Sc. in Energy and Nuclear Engineering from the University of Bologna, and subsequently collaborated with the Institute for Risk and Uncertainty at the University of Liverpool, where she was awarded a PhD. She is currently undertaking research within The Resilience Engineering Research Group at the University of Nottingham on the development of theoretical and computational tools for the efficient modelling of complex systems.