To identify the smallest fault tree sections which contain dependencies

Huiling Sun and John Andrews
Department of Aeronautical and Automotive Engineering, Loughborough University, Loughborough, UK

Abstract

Since the early 1960's fault tree analysis has become the most frequently used technique to quantify the likelihood of a particular system failure mode. One of the underlying assumptions which justifies this approach is that the basic events are independent. However, many systems feature component failure events for which the assumption of independence is not valid. For example, standby dependency, maintenance dependency or sequential dependency can be encountered in engineering systems. In such situations, Markov analysis is required during the quantification process.

Since the efficiency of the Markov analysis largely depends on the size of the established Markov model, it is most effective to apply the Markov method only to the smallest possible fault tree sections containing dependencies. The remainder of the system assessment can be performed by the application of the conventional assessment techniques. The key of this approach is to extract from the fault tree the smallest sections which contain dependencies. This paper gives a brief introduction on some main existing dependency types and provides a method aimed at establishing the smallest Markov model for the dependencies contained within the fault tree.

Keywords: Fault tree analysis; Markov method; Dependency; Modularistion

1. Introduction

Since its conception in the 1960s, the fault tree analysis method has been increasingly employed by many industries to assess the rationality of system designs from a risk and reliability perspective, usually reflected by the likelihood of a particular system failure mode.

The fault tree method [1,2] has the assumption that the basic events, which usually represent component failure, occur independently. This assumption enables the fault tree quantification using Kinetic Tree Theory [3]. Features of some systems mean that the independence assumption is invalid. In this case an alternative method such as Markov Analysis should be employed [1,2].

Markov models become very large with relatively moderate numbers of components in a system. This results in models which are difficult to generate, validate or solve. They also lack the documentation which is a feature of the fault tree diagram.

As a result, hybrid fault tree/Markov methods have been evolved for systems where redundancies are encountered [4,7]. These retain the failure logic development of the fault tree diagram whilst enabling Markov analysis to be performed on the portions of the system containing the dependency.

This approach works efficiently for dependencies introduced for situations such as standby redundancy. In such circumstances the dependent events are generally located below a single

gate in the fault tree and the smallest part of the system which requires Markov analysis is easily determined. However, other types of dependency can be associated with any groups of basic events appearing in the fault tree. For efficient analysis the smallest section of the fault tree which contains the dependencies and must be solved using a Markov model has to be identified.

This paper describes some of the situations which introduce dependencies in the system failure model and develops a method to identify the smallest system sub-sections, fault tree sections, whose solution will be solved with a Markov model.

2. Types of dependency

Dependencies can arise between components in a system for a number of reasons such as: maintenance, standby, sequential failures and secondary failures. These types of dependency are discussed separately to identify the features they exhibit in the analysis.

Maintenance dependency rises from the situation where one maintenance engineer or team of engineers has to take responsibility for a group of components usually of the same or similar type. In this case when one component has failed and is under repair, components from the same group which fail subsequently have to go into a queue waiting for repair till the engineer has restored the first component. It may be that some failures are more critical than others and so are given higher priority by the maintenance team. Either way the queuing means that the failure of one component affects the repair times and therefore probability of failure of other components.

Function-related dependency can be considered in sub-categories defined in terms of what specific functional connection is involved between components. These 4 types of dependency discussed here are named as standby dependency, sequential dependency, secondary failure dependency and initiator-enabler dependency.

The use of **standby systems**, where the failure of the primary system activates the standby to take over the primary system duty, is also a cause of dependency. If the likelihood of the standby component failure increases as it experiences the operational load due to the failure of the primary operating component, there exists a dependency between the standby component failure probability and the operating component status (working or failed). Therefore failure to take into account the standby dependency will produce incorrect system unavailability and failure intensity. Usually 3 terms are used to describe different standby situations: hot standby, cold standby and warm standby [1], of which the latter two give rise to the standby dependency.

Sequential dependency refers to the situation where a certain system level event will take place only when its causes occur in a specific sequence. Usually in the fault tree sequential dependency is represented by the ‘Priority AND gate’ as shown in Figure 1. With the priority AND gate, only when basic events occur in the order from left to right, i.e., a first, then b, finally c, will the output event represented by G0 occur.

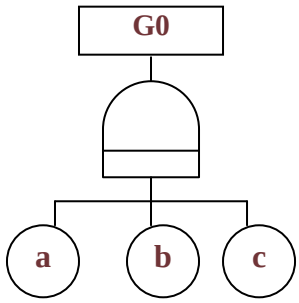


Figure 1. Priority AND gate

Secondary failure dependency can be introduced to a fault tree structure when it is developed by considering the state-of-component faults which are then represented by an OR gate with primary failure, secondary failure and command faults as input [5] (see figure 2 below). The definition of a secondary failure event is that they occur when the component is operating outside its expected working environment. This can occur due to the failure of another component in the system. To return the system to the fully functioning state requires the repair of both the components whose failure combined to cause the original problem (secondary failure) and the original problem itself [6]. Failure to account for the repair of the additional failure might lead to a serious underestimation of the system unavailability.

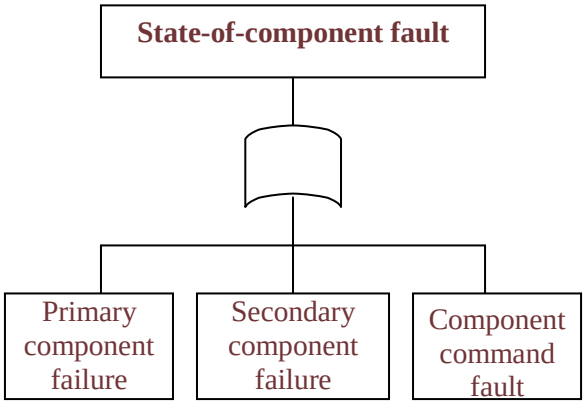


Figure 2. Fault tree development of state-of-component fault

Initiator-enabler dependency again is concerned with the order of component failure, but is less prescriptive than the sequential failure. This type of dependency usually occurs in the analysis of safety systems. The enabling events produce the condition which allows the initiating event to cause a hazard. Therefore the enabling events occur (in any order) prior to the initiating event. A good understanding of this type of dependency has to start with the definitions of initiating events and enabling events. As defined in Ref. 1, initiating events perturb system variables and place a demand on control/protective systems to respond, whilst enabling events represent the failure of inactive control/protective systems which permit initiating events to cause the top event. Below is a simple tank system (figure 3) to illustrate the distinction between the initiating and enabling events and the necessity to take into account the initiator-enabler dependency.

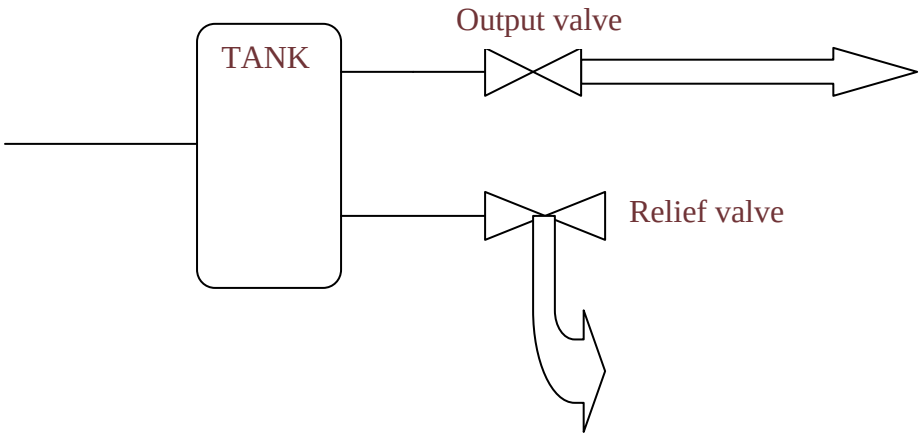


Figure 3. Pressure tank

In this system, when the tank is over-pressurised, the relief valve will open to reduce the pressure in the tank. Therefore the fault tree presentation for the top event 'Tank ruptures due to overpressure' will be drawn as shown in figure 4 (assuming that any overpressure will rupture the tank):

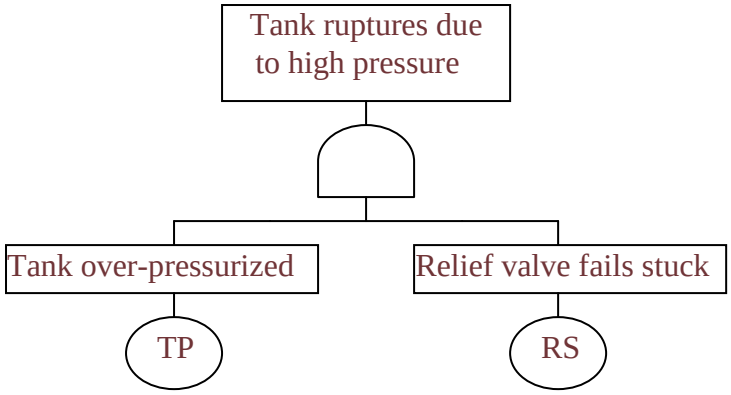


Figure 4. Pressure tank fault tree

Consider the situation where the tank is over-pressurised prior to the relief valve being stuck: since the relief valve will function to let out the fluid, the top event will not occur. On the contrary, when the relief valve has been stuck unrevealed, the subsequent tank over-pressurisation will lead to the occurrence of the top event and consequently reveal the failure of relief valve. Therefore the influence of the initiator-enabler dependency exists in two aspects: the first is that the order matters in distinguishing the effect of the failures on the system; the second is that the occurrence of the initiator will reveal any failures of the enablers and reduce their downtime causing a dependence in their failure probability.

3. Fault trees which contain dependencies

As is mentioned in the first section, the traditional fault tree methodology is not an appropriate approach for assessing systems where dependency exists between basic events. Meanwhile the Markov method provides a sound solution to the dependency problem [7]. The Markov method analyses the system by defining each state in which the system can reside and assigning a rate for every transition between these states. In this way it is able to

produce very accurate system risk and reliability measures for all types of dependency identified.

However, a problem comes along with the strength. Due to the way the models are formulated, the Markov method will produce a model which feature an exponential increase in size with the number of basic events included in it. This characteristic has severely restricted the effective application of the Markov method for moderate and large systems.

One approach to resolve this difficulty is to combine the Markov method and the conventional fault tree method by applying the Markov method only to fault tree sections where dependency between basic events exists. The key to the effectiveness of this solution process is to identify the smallest sections involving dependency within the given fault tree structure in order to produce a solvable Markov model. These sections which contain the dependencies are independent of the remainder of the fault tree.

As far as the five types of dependencies discussed in the previous section are concerned, initiator-enabler dependency, secondary failure dependency, standby dependency and sequential dependency tend to be grouped under a single OR or AND gate in the original fault tree structure which makes the corresponding smallest module easy to identify. Alternatively the maintenance dependency features dependent basic events which appear anywhere in the fault tree and the identification of the corresponding smallest module is even more necessary and important.

4. Identification of Independent Fault Tree Sections

The strategy for solving the dependency problem is to identify the smallest independent sections or modules of the fault tree. Those which contain dependencies will be solved using the appropriate Markov model. This paper presents the algorithm for identifying the independent modules. The algorith will be demonstrated by means of an example problem. An example fault tree is presented in figure 5 which is used to illustrate the whole of process of identifying smallest dependent modules. Due to its typical random distribution feature, the maintenance dependency is considered the worst case situation and as such will be considered in the fault tree as shown in table 1.

Dependency group no.	Dependency type	Number of basic events	Number of repairs	Basic event list
1	Maintenance	2	1	5 and 8
2	Maintenance	3	1	1, 2 and 7

Table 1. Dependency information

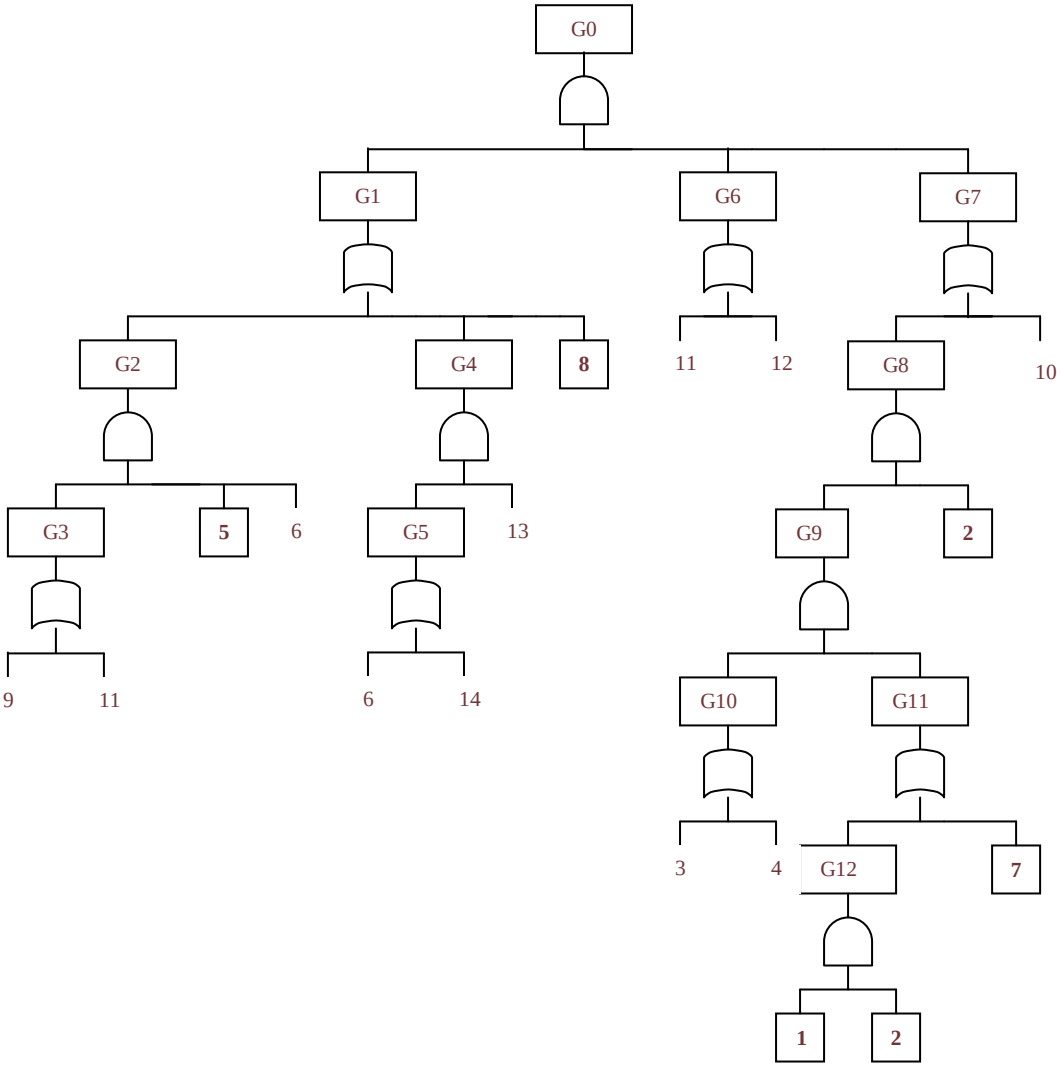


Figure 5. Example fault tree

This fault tree includes 12 gates and 14 basic events, of which 3 (2, 6 and 11) appear more than once. Maintenance dependency exists between basic events 5 and 8, plus 1, 2 and 7 as for each group there is only one engineer responsible for the maintenance.

The process of identifying the smallest independent section containing a certain dependency can be broken down into the following stages:
1) Re-organize dependency information
2) Fault tree simplification
3) Form the dependency information
4) Combine dependent events
5) Modularisation
6) Update the dependency information
7) Re-modularise for each type of dependency

The following sections will provide a detailed exploration of each of the seven steps using the example fault tree displayed in figure 5.

4.1 Re-organize The Dependency Information

When there exist more than one type of dependency, there might be overlap between different dependency groups. That is, the dependency groups must be considered together in order that an independent sub-tree can be found. By identifying the overlap and merging corresponding dependency groups, this step is aimed to provide a precise picture of the dependency relationship between basic events. For example, if there is secondary failure dependency existing between primary failure event a and secondary failure event b, and also a maintenance dependency exists between basic event a and c, although they are contained in different dependency groups, basic events a, b and c should bear the same dependency serial number to ensure they will end up in the same module for the later Markov analysis.

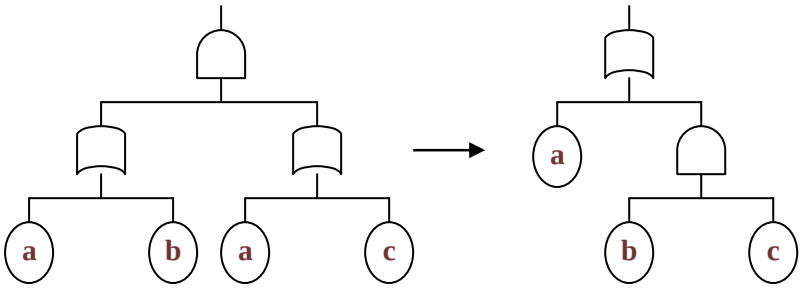
Considering the dependency information of the example fault tree provided in table 1, dependency serial number information is listed in table 2 after the re-organization and will be referred to during the course of the following steps.

Dependency serial number	Number of basic events	Basic event list
1	2	5 and 8
2	3	1, 2 and 7

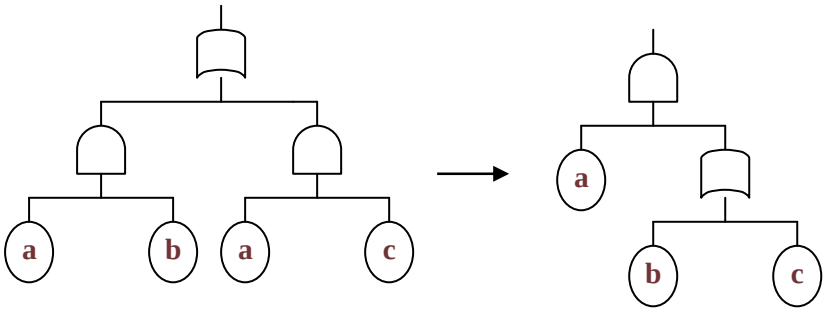
Table 2. Dependency serial information

4.2 Fault tree simplification

Simplification of the fault tree structure is aimed at reducing the fault tree to its most concise form without changing the logic function it represents. The simplification is achieved by applying to the fault tree the framework as follows which is composed of 4 reduction techniques, of which the first 3 are used in the Faunet code [8]:
Contraction: subsequent gates of the same type are contracted to form a single gate. This structures the fault tree as an alternating sequence of AND gates and OR gates.
Extraction: this looks for structures of the type illustrated in figure 6 and converts them as illustrated. The effect is to identify the common factor, i.e. the repeated basic event.



Case 6a



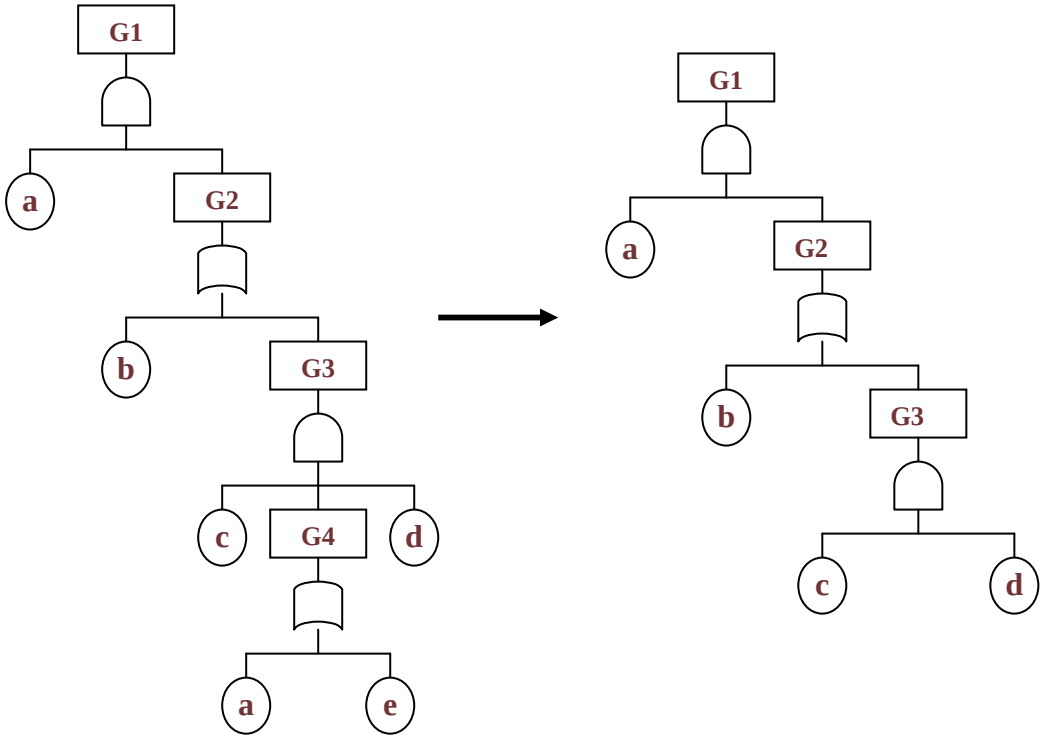
Case 6b

Figure 6. Extraction

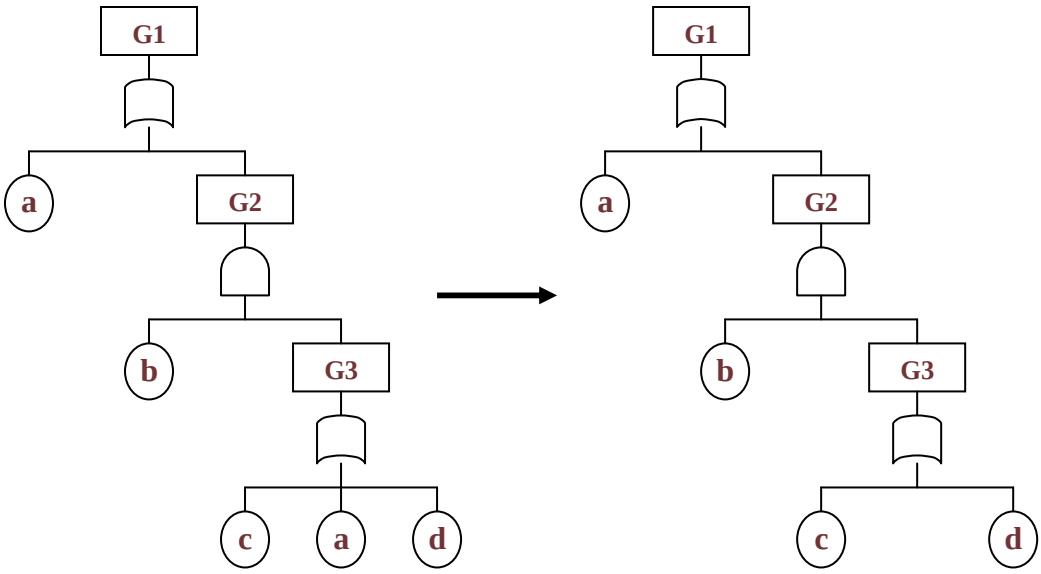
Factorisation: pairs of (independent) events that always occur together in the same gate type are identified and combined to form a single complex event. Events which occur in dependency groups are not included when defining factors.
Elimination: this process employs the Boolean laws of absorption displayed as follows:

$$a+(a.b)=a$$
$$a.(a+b)=a$$

By extending the absorption law, the elimination technique can reduce events that are repeated over any number of levels of a fault tree branch. This can be illustrated by the two examples shown in Figure 7:



Case 7a



Case 7b

Figure 7. Elimination

Primary and secondary gates are defined to describe the algorithm of this reduction technique. A primary gate is the highest gate at which the repeated variable is first encountered as an input. A secondary gate is a gate, below the primary gate, at which a second occurrence of the event appears as an input [9]. The algorithm is that when the primary and secondary gates are of different types, the secondary gate can be eliminated from the fault tree, while if the primary and secondary gates are of the same type, the repeated event under the secondary gate can be eliminated. For example case 7a in figure 7, the repeated variable is event a, the primary gate is G1, the secondary gate is G4. since G1 and G4 are different gate type, G4 is eliminated as shown. For case 7b the repeated event is a and is an input to primary gate G1 and secondary gate G3. In this case G1 and G3 are both OR gates and so event 1 is deleted from the secondary gate G3.

When the elimination results in gates that have only one input, these gates are replaced in the fault tree structure by their single input.

Applying the simplification to the example fault tree in figure 5, the reduced fault tree will be obtained as shown in figure 8. The reductions include the contractions between G8 and G9, eliminations of event 2 under G12, and finally factorisation of event 3 OR 4. To distinguish factors from other elements in the fault tree, these factors are named from 3001 onwards.

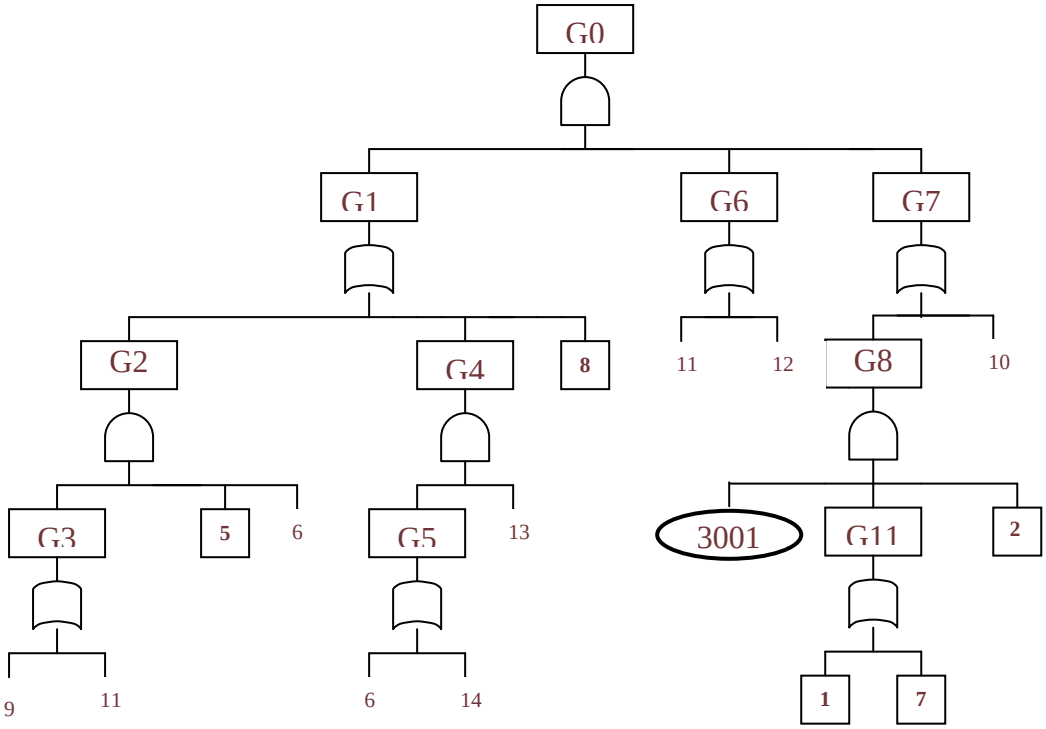


Figure 8. Simplified fault tree

4.3 Form the dependency information

This step is to decide the dependency serial numbers which each gate in the fault tree is dependent upon. The information forms the basis for the implementation of the next combination step. The dependency of each gate is defined by a list of all the dependency serial numbers to which basic events below it in the fault tree structure belong.

This step is conducted by traversing the fault tree to decide the dependency serial number that each gate features. This process is illustrated with a simple example (see Figure 9).

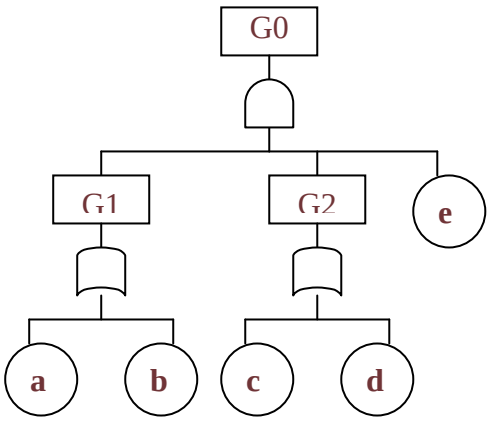


Figure 9. Form dependency information

Assume that after the re-organisation of dependency group information, events a and b bear dependency serial number 1, and events c, d and e belong to dependency serial number 2.

Therefore the dependency of G1 is the dependency serial of its immediate descendants, i.e., serial 1, since both of its input events a and b feature in dependency serial 1. Similarly, the dependency of G2 is serial 2. And finally the dependencies of G0 is identified as serial 1 and 2 because G1 has dependency serial 1 and both G2 and event e are characterized by dependency serial 2.

Therefore, according to this algorithm, the dependency serial information of each gate in the simplified fault tree in figure 8 is summarized in table 3.

Gate	G0	G1	G2	G3	G4
Dependency serials	1, 2	1	1	-	-
Gate	G5	G6	G7	G8	G11
Dependency serials	-	-	2	2	2

Table 3. Gate dependency serials

4.4 Combine dependent events

The purpose of this step is to restructure the fault tree in a way which will separate those events with the same dependency serial into separate branches. Using the information generated in the previous phase, each gate will be examined in turn, additional gates of the same logic type as the gate being investigated, are added where necessary to group the input events (immediate descendants) of the same dependency serial. The reason for implementing the 'combination' phase is that the resulting new gates (numbered from 20001 upwards) are leading to a fault tree structure with the smallest independent sub-trees for each dependency.

For the example in figure 9, the application of combination will result in a new fault tree structure as shown in figure 10. G2 and basic event e, both of which feature dependency serial 2, is grouped under the new gate 20001, which consequently also bears dependency serial 2.

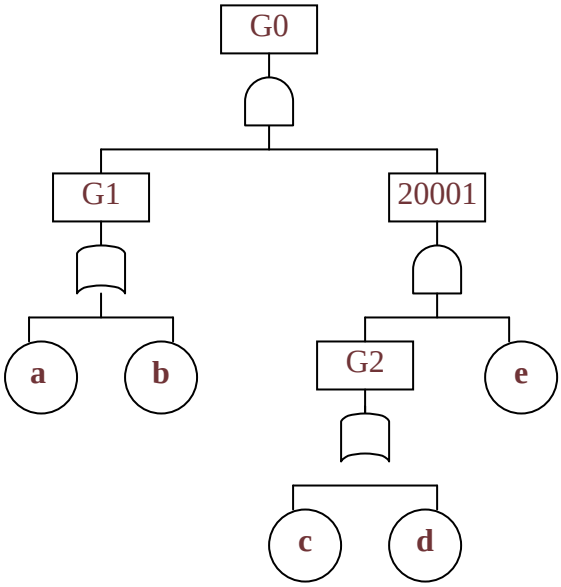


Figure 10. Restructured fault tree

Similarly, when applied to the fault tree in figure 8, the ‘combination’ step will produce the restructured fault tree shown in figure 11 with new gates 20001 and 20002.

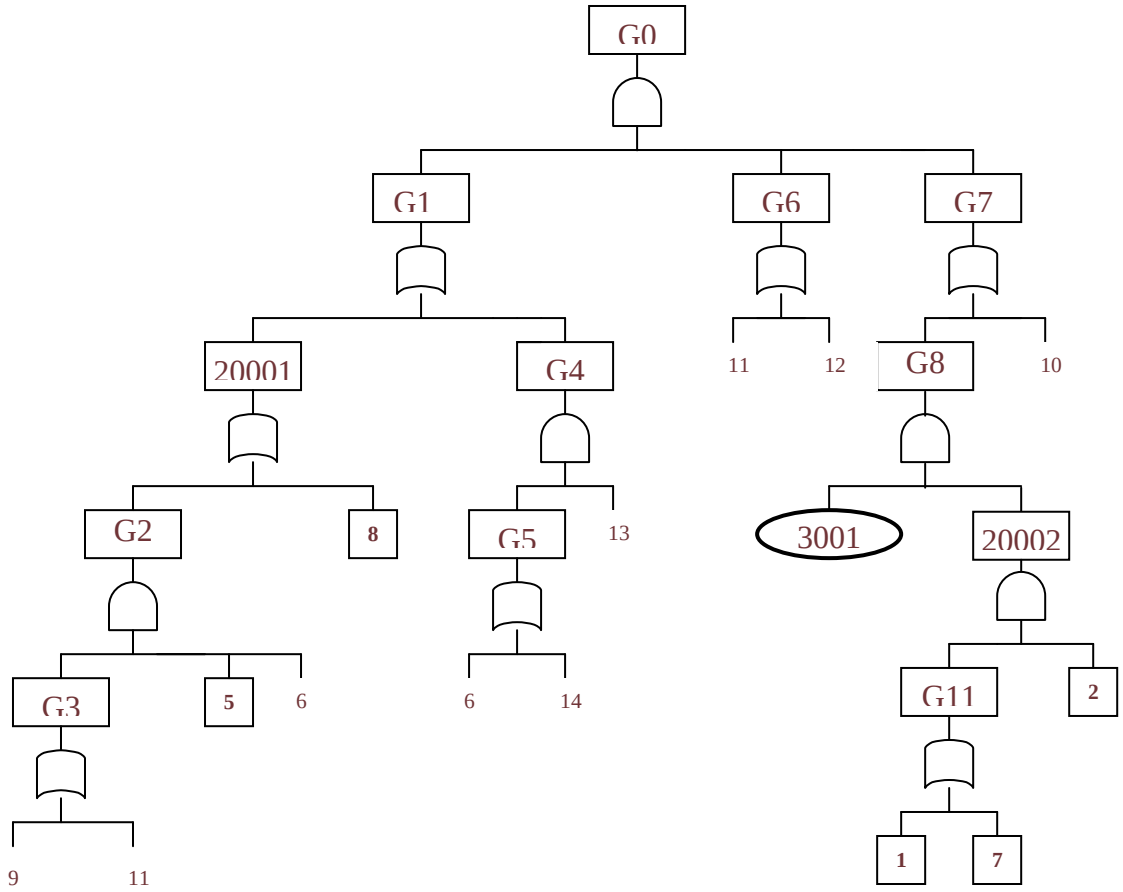


Figure 11. Combination

When there exist more than one way to carry out the combination process due to the overlap of dependency serial numbers between the sibling gates, each of the combination approaches will lead to the same final result.

4.5 Modularisation

The task of this phase is defined as to identify modules in the fault tree. A module of a fault tree is a sub-tree that is completely independent from the rest of the tree. After the modularisation, each module will be replaced with a super-event in the original fault tree structure. The super-event has the same reliability characteristics as the fault tree section which it has replaced and is determined using Markov theory or fault tree theory depending on whether the corresponding module contains dependent basic event or not.

The algorithm developed by Rauzy and Dutuit [10] provides an efficient means to identify the modules, which mainly requires two depth-first traversals of the fault tree. The first performs a step-by-step traversal recording for each gate and event, the step number at which the first, second, and final visits to that node were made. It also records the number of appearances in the traversal which will be used in a later stage. In this first traversal, it must be noted that the graph under a vertex is never traversed twice [11]. Therefore when gates

appear more than once in the tree, only its first appearance will be traversed completely, after this, its appearances elsewhere in the tree will be treated like a basic event.

In order to ensure that dependent basic events featuring the same dependency serial will end up in the same module, they are treated as a single basic event with the same label during the first traversal. All events in the same dependency group will be replaced with an id that characterizes the particular dependency serial. For example, in the fault tree in figure 9, both a and b will be replaced by label 10001, and c, d and e by 10002 (dependency event numbering starts at 10001).

The principal of the algorithm for modularisation is that if any descendant of a gate has a first visit step number smaller than the first visit step number of the gate, then it must also occur beneath another gate. Similarly, if any descendant has a last visit number greater than the second visit number of the gate, then again it must occur elsewhere in the tree. Therefore a gate can be identified as heading a module only if:

- the first visit to each descendant is after the first visit to the gate and
- the last visit to each descendant is before the second visit to the gate

Then the second pass through the fault tree assesses these conditions. The maximum (Max) of the last visits and the minimum (Min) of the first visits of all the descendants (any gates and events appearing below that gate in the tree) for each gate will be obtained based on the result of the first traversal.

Therefore, based on the fault tree in figure 11, the two traversals will provide the information given in tables 4 – 7.

Gates	**Visit number**	**Gates**	**Visit number**	**Gates**	**Visit number**
G0	1	G4	14	G8	27
G1	2	G5	15	3001	28
20001	3	6	16	20002	29
G2	4	14	17	G11	30
G3	5	G5	18	10002	31
9	6	13	19	10002	32
11	7	G4	20	G11	33
G3	8	G1	21	10002	34
10001	9	G6	22	20002	35
6	10	11	23	G8	36
G2	11	12	24	10	37
10001	12	G6	25	G7	38
20001	13	G7	26	G0	39

Table 4. 1st traversal – event visit

Gates	G0	G1	20001	G2	G3	G4
1st visit	1	2	3	4	5	14
2nd visit	39	21	13	11	8	20
Last visit	39	21	13	11	8	20
Number of appearances	1	1	1	1	1	1

Gates	G5	G6	G7	G8	20002	G11
1st visit	15	22	26	27	29	30
2nd visit	18	25	38	36	35	33
Last visit	18	25	38	36	35	33
No. of appearances	1	1	1	1	1	1

Table 5. 1st traversal - gates

Basic events	9	11	10001	6	14
1st visit	6	7	9	10	17
2nd visit	6	23	12	16	17
Last visit	6	23	12	16	17
Number of appearances	1	2	2	2	1
Basic events	13	12	3001	10002	10
1st visit	19	24	28	31	37
2nd visit	19	24	28	32	37
Last visit	19	24	28	34	37
No. of appearances	1	1	1	3	1

Table 6. 1st traversal – basic events

Gates	G0	G1	20001	G2	G3	G4
Min	2	3	4	5	6	10
Max	38	23	23	23	23	19
Gates	G5	G6	G7	G8	20002	G11
Min	10	7	27	28	30	31
Max	17	24	37	35	34	34

Table 7. 2nd traversal

Therefore, according to the conditions for a module, G0, G7, G8, and 20002 are identified as heading the modules. To distinguish themselves from other events in the fault tree, these modules are assigned a unique id starting from 6001 onwards, i.e., G0 – 6001, G7 – 6002, G8 – 6003, and 20002 – 6004. their structure are shown in figure 12.

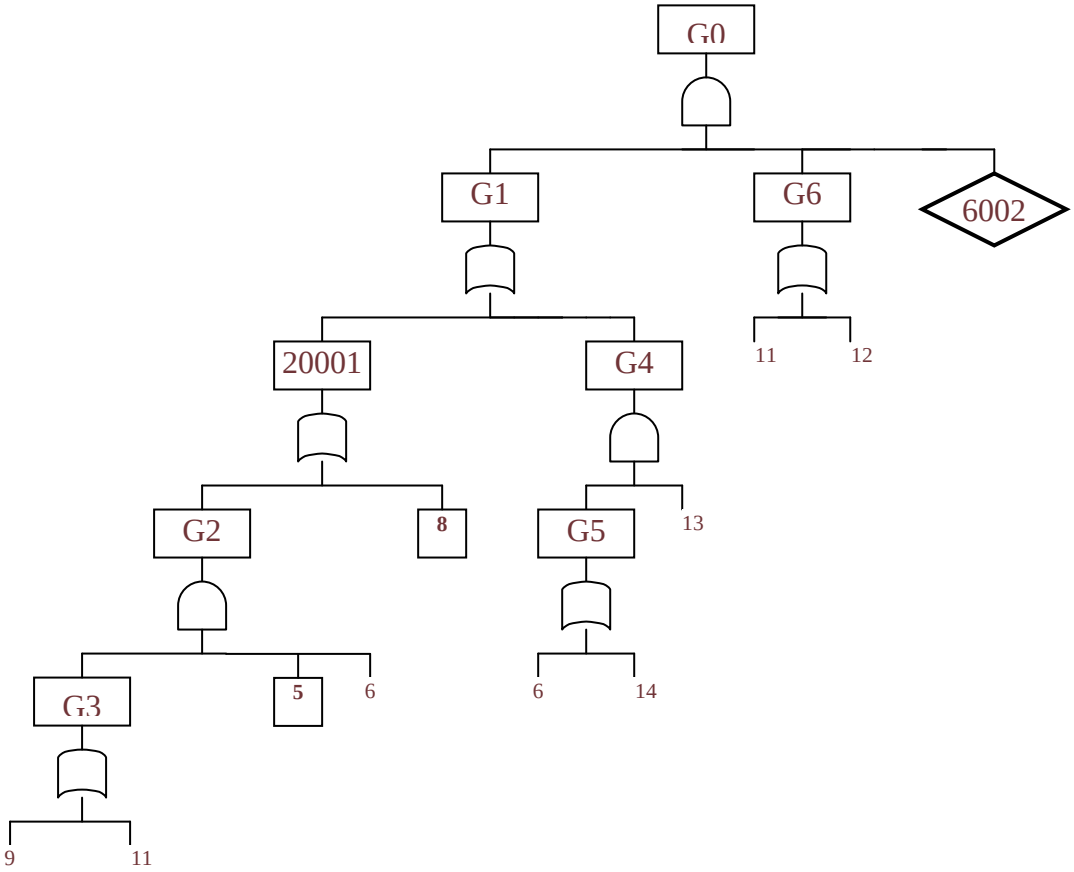
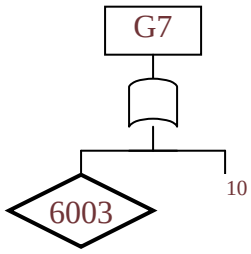
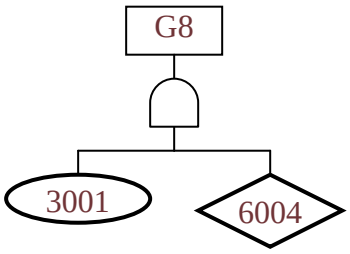


Figure 12a. Module 6001



Fi Figure 12c. ıle 6002
Module 6003



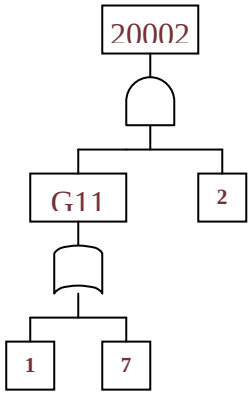


Figure 12d. Module 6004

Figure 12. Modules identified

4.6 Update the dependency information

By this stage, independent sub-trees have been identified. However, with the aim to find out the smallest modules which contain dependent basic events, the task has not been

accomplished yet. To attain the aim, two points must be made clear: the first is which modules contain which dependency serial; and the second is whether these modules are the smallest one. This step is designed to provide the information required to answer these two questions.

Slightly different from step 3, dependency information is updated establishing not only which dependency serials each gate contains but also its mutual dependency serials. The mutual dependency serial of a gate is a list of dependency serials which all of its immediate descendants feature.

Take module 6001 in figure 12 for example, it can be determined that gate G2 contains dependency serial 1 and since only one of its three input events features dependency serial 1, it has no mutual dependency serial. Gate 20001 is slightly different: since both of its input events, gate G2 and event 8 features dependency serial 1, gate 20001 bears dependency serial number 1 as its mutual dependency serial. When it comes to gate G0, it is a similar case with G2 with the exception that one of its input events is a module which is treated as an independent basic event in this process.

Accordingly, table 8 below gives the dependency serial information of each of the modules shown in figure 12.

Gates	G0	G1	20001	G2	G3	G4
Dependency serial contained	1	1	1	1	-	-
Mutual dependency serial	-	-	1	-	-	-
Gates	G5	G6	G7	G8	20002	G11
Dependency serial contained	-	-	-	-	2	2
Mutual dependency serial	-	-	-	-	2	2

Table 8. Updated gate dependency serials

4.7 Re-modularise for each dependency serial

In this last phase, re-modularisation will be carried out for each of the modules identified which contain dependencies. This process will identify the smallest independent section for each dependency serial. Table 8 indicates that module 6001 led by gate G0 and module 6004 led by gate 20002 contains dependency serials 1 and 2 respectively, on which re-modularisation will be conducted.

The Re-modularisation consists of two steps: firstly, it has to be determined whether a certain module is already the smallest one for the given dependency serial. If is, the re-modularisation process is completed then. Otherwise, the second step needs to be carried out.

To answer the question posed in the first step, one has to refer to the information generated by table 8. The solution states that for an existing module to be the smallest one in terms of a given dependency serial, the mutual dependency serials of the gate which leads the module

must include the given dependency serial. The underlying algorithm is when this condition is fulfilled, there is no way to further break down the module so that a smaller independent section will be obtained which contains the dependency serial in question.

Accordingly, by referring to table 8, it can be concluded that module 6004 led by gate 20002 is already the smallest independent section for dependency serial 2 which includes basic events 1, 2 and 7. Whilst, module 6001 with top gate G0 does not fulfil the condition. accordingly the second stage of the re-modularisation will be carried out.

The second step can be generalized by the following steps.

a) Traverse the module from the top event, always following the gate which contains the given dependency serial and recording the downward path, until the gate is encountered whose mutual dependency serials also include the given dependency serial.

For example, regarding module 6001 in figure 12, in terms of dependency serial 1, the downward path will be: G0, G1, 20001.

b) The last gate appearing in the Path established in step (a) as having the correct mutual dependency serial would be leading the smallest independent sub-tree if it had been identified as a module. The fact that it is not a module indicates that some of its descendants must have occurred elsewhere in the module, which are defined as preventing elements. In this step, those preventing elements and their appearances outside the fault tree section headed by this gate will be identified.

One solution is to see whether the number of appearances of any descendant under this gate is the same as its number of appearances in the whole module. If it is different, the descendant turns out to be the preventing element.

For example, basic events 6 and 11 are identified as preventing elements because both of them occur only once under gate 20001 but occur twice in module 6001 (see table 6).

c) After the preventing elements have been detected, the next thing is to identify an independent sub-tree from the existing module to include those preventing elements. The approach can be illustrated by the specific example of module 6001.

First some information should be listed:
The downward Path is: G0, G1, 20001
The potential module: led by gate 20001
Preventing elements: basic event 6 with another occurrence at location 16 and basic event 11 at location 23 in table 4.

Traverse upward from preventing elements and record their antecedents:
Event 6, G5, G4, G1
Event 11, G6, G0
For each preventing element, the traversal stops at the gate which also appears in the down path established in step(a). After the upward traversal for each preventing element is finished, pick one of the last gates in the traversals which appears at the highest level of the module structure. Therefore, in terms of event 6, the traversal stops at G1 and for event 11, it stops at G0. And G0 will be singled out as it lies in the first level of the module. Now the new

potential module will be formed by combining fault tree sections led by G1 and G6 as G1 and G6 are both immediate input events to G0 and include preventing elements 6 and 11.

In this new potential module, no preventing element has been detected, the combination of G1 and G6 becomes the new module labelled 6005 which is smallest for dependency serial 1. See figure 13 and 14.

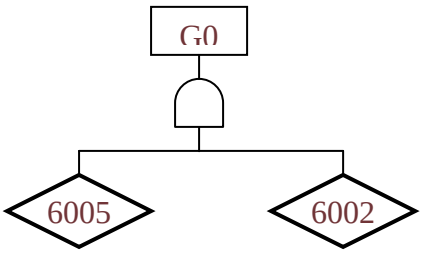


Figure 13. Module 6001

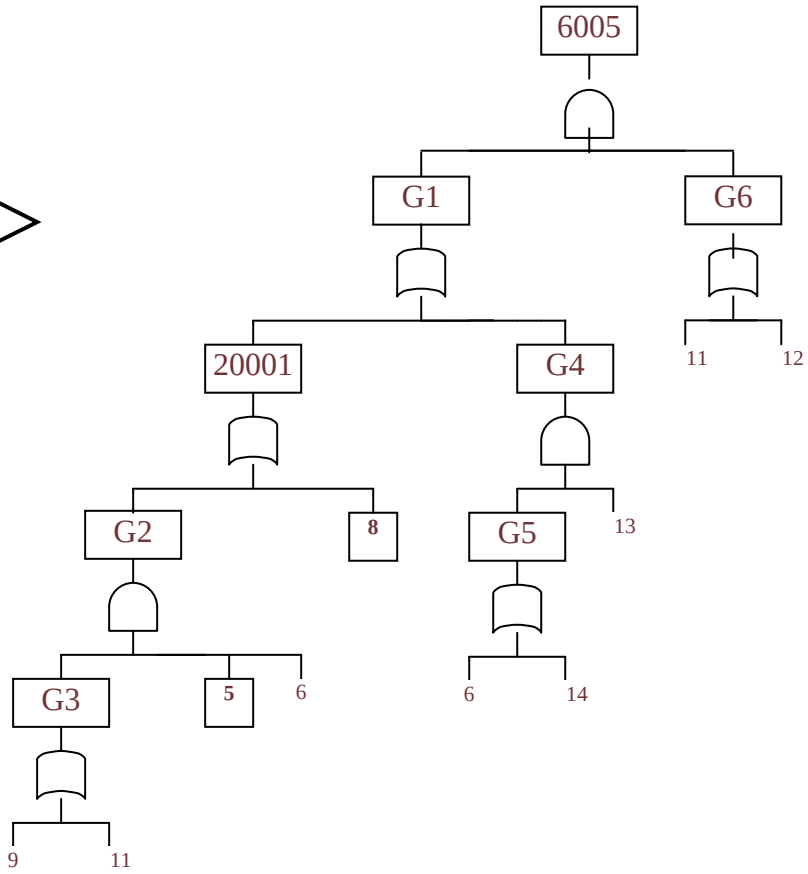


Figure 14. Module 6005

If the new potential module contains any preventing elements, the procedures are repeatedly applied until a new module is identified.

d) If a module includes more than one dependency serial, they shall be dealt with one after another in the same way.

So far, having progressed through the seven stages, modules 6005 and 6004 are identified as the smallest modules for dependency serial 1 and 2 respectively. These modules will be handled with the Markov method in the quantitative reliability analysis.

5. Conclusion

The algorithm presented in this paper provides a method which enables efficient analysis of fault trees which contain dependent basic events. The fault tree is structured in order to identify the smallest modules containing dependent basic events. For these modules Markov analysis is used to determine the failure probability and failure frequency. These predictions are in turn used to quantify higher level modules until the top event characteristics are obtained. For those sections or modules which do not contain any dependencies the conventional fault tree analysis method or Binary Decision Diagram method are employed.

References

1. Andrews J.D. and Moss T.R., "Reliability and Risk Assessment", 2nd edition, Professional Engineering Publishing, 2002.
2. Kumamoto & Henley E.J. “Reliability Engineering and Risk Assessment”, Prentice Hall, 1981.
3. Vesely W.E., “A Time Dependent Methodology for Fault Tree Evaluation”, Nuclear Design and Engineering, Vol. 13, 1970, pp337-360
4. Meshkat L, Dugan J.B. and Andrews J.D., “Dependability Analysis of Systems with On-demand and Active Failure Modes Using Dynamic Fault Trees”, IEEE Transactions on Reliability, Vol 51 No 2, June 2002, pp 240-251
5. Hassl D.F., Roberts N.H., Vesely W.E. and Goldberg F.F., ‘Fault Tree Handbook”, US Nuclear Regulatory Commission, 1981, NUREG-0492
6. Andrews J.D. and Dunnett S.J., “Analysis Methods for Fault Trees That Contain Secondary Failures”, Proc. Instn Mech. Engrs, Vol. 218 Part E: J. Process Mechanical Engineering, pp93-102
7. Pullum L. and Dugan J.B., “Fault Tree Models for the Analysis of Complex Computer-based Systems”, Proceedings of the Annual RAM Symposium, Las Vegas, Jan 22-25 1996, pp 200-207.
8. Platz O., Olsen J.V., “FAUNET: A Program Package for the Evaluation of Fault Trees and Networks”, Riso report No 348, DK-4000, Roskilde Denmark, Sept 1976.
9. Sun H. and Andrews J.D., “Identification of Independent Modules in Fault Trees Which Contain Dependent Basic Events”, Reliability Engineering and System Safety, Vol. 86, Iss. 3, pp285-296, December 2004
10. Dutuit Y and Rauzy A, “A linear Time Algorithm to Find Modules in Fault Trees”, IEEE Trans Reliability, 45, No 3, 1996.
11. Anand A and Somani A.K., “Hierarchical Analysis of Fault Trees with Dependencies, using Decomposition”, PROCEEDINGS Annual RELIABILITY and MAINTAINABILITY Symposium, Anaheim, 19 – 22 January 1998, pp 69-75