Computer-assisted Fault Tree Construction Using a Knowledge-based Approach

Margaret S. Elliott
melliott@ics.uci.edu

Technical Report 92-95

October 15, 1992

Contents

List of Figures

List of Tables

Computer-assisted Fault Tree Construction Using a Knowledge-based Approach

Margaret S. Elliott

Department of Information and Computer Science
University of California, Irvine, CA 92717-3425

Electronic Mail: melliott@ics.uci.edu
Phone: (714) 856-6226
Fax: (714) 856-4056

Keywords: Rule-based expert systems, knowledge-based systems, fault tree analysis, reliability analysis

October 13, 1992

Abstract

This paper presents a knowledge-based approach to performing a fault tree analysis of engineering systems. This knowledge-based approach can be used to build a fault tree from a graphical functional block diagram using reduction for replicated events, and to compute minimal cut sets. A methodology in predicate calculus for fault tree synthesis from a functional block diagram is formulated. An implementation of this methodology in a rule-based language is presented. Results are compared with manually prepared reliability reports on an electrical power system and a hydraulic system. The knowledge-based system outperforms the manual procedure in timeliness and accuracy.

1 Introduction

Fault tree analysis is a widely used technique for obtaining qualitative and quantitative reliability and system safety assessments of complex systems. This process consists of constructing a logical diagram (fault tree) that represents the causal relationships of system events contributing to a predetermined TOP event. Nodes connecting basic events in the tree are AND and OR gates for coherent trees. For more complex systems, fault trees known as noncoherent trees may also include NOT and XOR gates. Coherent fault trees represent systems with failure functions that have monotonic properties, whereas noncoherent fault trees do not[4, 9, 27]. In either case, TOP events for the tree are preselected from undesired system states that can occur because of subsystem failures.

The fault tree provides the derivation of the unique set of events that can cause the occurrence of the TOP event. For coherent fault trees, these sets are called minimal cut sets. They are minimal when the cut sets are reduced to those mutually exclusive sets not containing one set within another. In fault trees without replicated events, the cut sets *are* minimal. For noncoherent fault trees, the minimal cut sets are replaced by a set of prime implicants in Boolean algebra[46, 78].

Algorithms for determining the minimal cut sets include a Monte Carlo simulation technique and a deterministic approach. The Monte Carlo simulation method involves the use of probabilistic information about the tree's basic events to calculate minimal cut sets. The deterministic method involves the expansion of the top event of a fault tree into its Boolean algebra equivalent in a sum of products of basic events in the tree.

Computer programs have been developed to compute minimal cut sets from the fault tree using the Monte Carlo method[71] and using the deterministic approach [31, 33, 51, 56, 59, 61, 64, 70, 71, 77]. The SETS program[77], developed by Sandia Laboratories, finds minimal cut sets for coherent trees, and prime implicants for noncoherent trees. Several other computer codes exist for the computation of minimal cut sets but their descriptions are beyond the scope of this paper. References to these programs may be found in[50, 73].

Fault trees are also used for quantitative evaluations of system unavailability. One means of measuring system unavailability is to use the minimal cut sets of coherent trees by[5]:

1. Writing the structure function of the tree in Boolean algebra as a sum of products of basic event failure rates.

2. Applying statistical approximation techniques to basic events.

For noncoherent trees, similar methods apply to the analysis of prime implicants[19]. Direct evaluation codes, which quantify the system model for unavailability without the use of minimal cut sets are also available[28, 42, 55, 57]. More references for quantitative evaluation codes can be found in[50, 73].

The correctness of the fault tree construction directly affects the accuracy of the fault tree analysis. However, manual fault tree construction is laborious and prone to errors of omission and inaccuracies. If the fault tree is incorrect, then erroneous qualitative and quantitative assessments are calculated. Fault tree analysis results need to be precise to ensure reliable designs, especially where safety factors are essential to the elimination of catastrophic failures[21].

One means of reducing fault tree construction errors is to automate the process. Computer-aided techniques for producing a fault tree from the system design have been successful. Various data structures are used to represent the input of a system design. For example, Fussell[29] constructed a fault tree for an electrical system from the tabular input of the system design, component-transfer functions, mini-trees representing component failure modes, and system boundary conditions. DeVries[74] developed an Automated Fault-Tree Generation Methodology that generates a fault tree for electrical systems from the circuit representation of electrical components, fault models, and component transfer functions. Lapp and Powers[47, 48, 49] devised a Fault-Tree Synthsis program that constructs fault trees for chemical processing plants from complex digraphs allowing process feedforward and feedback loops. A method for on-line hazard aversion and fault tree diagnosis in chemical processes has been developed by Ulerich and Powers[69], in which digraphs are used for fault tree construction, diagnosis, and fault detection.

Disadvantages of the digraph techniques are pointed out in [2, 43] including the inability to correctly model all types of control loops. Kelly et al.[37, 38, 39, 40] report on an alternate digraph method of fault tree construction applied to a chemical processing plant in which they correct some of the pitfalls of previous digraph fault tree synthesis methods. Recently, Andrews and Brennan[3]; Bosche[11, 12, 13]; and Change and Hwang[17] have used digraphs as inputs to fault tree synthesis improving on the original Lapp and Powers method[47, 48, 49]. In the Computer Automated Tree (CAT) code[63, 79], decision tables are used as input to generate fault trees for electrical, hydraulic and mechanical systems.

In all the aforementioned codes, fault trees are generated from time-consuming tabular entries of the system configuration, nearly tantamount to the task of creating the fault tree itself. One method of producing a fault tree is to first build a functional block diagram, also referred to as a reliability block diagram, for the failure mode associated with the tree's TOP event. Fault trees can then be readily derived from this diagram as outlined in [73][1].

[1]See page V-6 to V-11 for a description of the "Immediate Cause" concept, which is analogous to the functional block diagram method used here.

Using a graphical functional block diagram for computer-assisted fault tree construction would provide a ready means of performing a fault tree analysis. Taylor[68] reports on a program that uses graphical input of system flow sheets and circuit diagrams to build fault trees for multi-state sequential fault trees. Taylor's algorithm is based on Fussell's approach[29] of storing mini-trees representing component interconnections for fault tree construction. Camarda et al.[1, 16] present a technique for constructing a fault tree by converting a reliability block diagram into a probabilistic graph, showing ways of correct system operation. From this graph, they derive the minimal tie-sets, a set of all possible paths of system success. These tie-sets are transformed into minimal cut sets by Boolean algebra laws.

This paper presents a fault tree construction methodology that creates a fault tree from a graphical functional block diagram using a knowledge-based approach. The manual process automated is described in [5] and [73]. The contributions of this paper are:

1. A fault tree synthesis methodology intended for 2-state coherent fault trees, adaptable by program developers to rule-based or logic-based programming languages.

2. A fault tree reduction algorithm incorporated into the methodology reducing a tree during its construction.

3. A recursive minimal cut set algorithm intended for list-processing languages using hash tables for efficient processing.

This methodology is demonstrated with first order predicate calculus, mapping blocks and their connections in a functional block diagram to fault tree nodes. The implementation of this methodology, in a Reliability Analysis Expert System (RAES)[23, 24, 25] is presented. The results of benchmark tests on an electrical[65] and hydraulic system are presented showing the system eliminating fault tree errors and completing the analysis in less time than the manual approach.

The errors in the manual approach found by RAES were verified by the engineer responsible for the manual report[65]. The amount of time saved was in the fault tree construction - computerized versus manual. In these experiments, the manually drawn functional block diagrams were entered graphically into RAES with careful attention given to the authenticity of the graphical diagram. Certainly, if one were to enter block diagrams into RAES with errors incurred during manual drawing or graphical entry, then the fault trees themselves would be invalid. Thus, correct functional block diagram entry is crucial to obtaining valid fault trees in RAES.

The RAES system was implemented while the author was employed at Northrop Research and Technology Center and hence, it is not available as public domain software. However, the author has recently completed a PROLOG implementation of the fault tree construction portion of the RAES system. This PROLOG code can be obtained by contacting the author.

Section 2 will present the nomenclature and assumptions related to this article. Section 3 presents the fault tree construction methodology, Section 4 gives the RAES overview with a discussion of the fault tree reduction and minimal cut set algorithm, Section 5 previews the RAES limitations, Section 6 presents RAES results, and, finally, conclusions are discussed in Section 7.

2 Nomenclature and Assumptions

2.1 Nomenclature

- $\cup$ - representing the logical OR.
- $\cap$ - representing the logical AND.
- $\sim$ - representing the logical NOT.
- $\Rightarrow$ - representing the implies logical relation between two statements.
- $\Pi_k(X)$ = Set of subsets of size k of a set X.
- $|X|$ = Cardinality of set X.
- Functional Block Diagram - A directed graph *FBD* consisting of two finite sets, a set B of vertices, also called functional blocks, and a set E of directed edges. We write $FBD = (B, E)$ in which each edge $e = (i, j)$ has a direction from its initial vertex i to its end vertex j.
- Fault Tree -

  Mathematically, a fault tree can be defined recursively as follows[41]: A fault tree is a finite set T of one or more nodes such that

  - a. There is a node existing as the root of the tree.
  - b. The remaining nodes are divided into $m \geq 0$ disjoint sets, $T_1, ..., T_m$, and each of these sets are also a tree. The trees $T_1, ..., T_m$ are referred to as subtrees of the root. Each root is a *parent* to the roots of its subtrees, also known as branches. The terminal nodes of a subtree are called leaves, and are referenced as *children* of the *parent* root.

A fault tree may also be defined as a boolean equation over the failures of X, for $X \in B$. Nodes in the tree are defined as:

  - Connector Nodes - Roots of the subtrees represented as AND and OR gates.
  - Event Nodes - Terminal nodes of subtrees representing the failures of blocks in a functional block diagram. (i.e. component or event failures).

- Minimal cut sets - Minimal cut sets are the sets of minimum blocks that can cause the top event of a fault tree to fail.

2.2 Assumptions

- The fault trees considered are coherent with two possible states for events, failed or functional.
- The failure rates assigned are for nonrepairable failures.

3 Fault Tree Construction Methodology

In this section, we present how a functional block diagram is mapped into its fault tree representation. First-order predicate calculus[36, 54, 76] provides a means of describing the semantics of failures and semantics of fault tree construction with defined predicates and functions. We define a set of formulas for deriving failures from functional block diagrams in Section 4.1. In Section 4.2, we define a set of rules for constructing fault trees from functional block diagrams. We show that these rules are complete (i.e. can derive all possible fault trees from a functional block diagram matching the description in Section 4.1) by examples of fault trees derived from all possible combinations of functional block diagrams.

3.1 Semantics of Failures

The semantics of failures in a functional block diagram are concerned with the ways in which block failures can cause a particular system failure. The system failure is identified by specifying, from the diagram, a block or set of blocks which could fail the system. To determine the ways that a failure could occur in these blocks, their connections to other blocks are traced. Each identified system failure constitutes a particular failure mode. For example, in an electrical power system of an aircraft, the system failure might constitute the ways that subsystems prevent power to a non-flight critical AC bus. The failure mode here is the lack of power from a non-flight critical AC bus.

We distinguish between the fundamental failure and the functional failure of a block with the following definitions:

- Fundamental failure - Block X has physically failed to perform its function such that any adjoining blocks Y_i with edges $\{X, Y_i\} \in E$ will functionally fail.
- Functional failure - Block X's functionality has failed such that either a fundamental failure has occurred at X or a functional failure has occurred at any adjoining blocks $\{Y_i\}$ with $\{Y_i, X\} \in E$.

To illustrate the semantics of failures, we define predicates on the domain of blocks in functional block diagrams. We then define formulas using those predicates to derive the cause of fundamental failures. These functions and predicates refer to blocks in the diagram using unique alphanumeric identifiers and specified types. The type of the block depicts the functionality of the connections as follows.

- *other* - a typical block representing a component or event.
- *series* - a block existing as a member of a series, a set Y_i of blocks of type *series*. The *series* blocks between the start *other* block Z and the end *other* block X are in two parallel lines. Block Z is connected to the first two blocks of the series, Y_1 and Y_2, such that $(Y_1, Z), (Y_2, Z) \in E$. Block X is connected to the last two *series* blocks such that $(X, Y_{n-1}), (X, Y_n) \in E$.
- *switch* - a block representing a component of type *switch*. A switch X may have several input blocks Y_i such that $(Y_i, X) \in E$, but these input blocks can only be of type *other*. A switch transfers control (power, fluid, etc) from one input block to another to avoid a failure condition (e.g. hydraulic fluid level too low).
- *k-out-of-n* - a block X with n inputs that fails if at least k of its inputs fail (i.e. k-out-of-n:F type failure). The input blocks to a *k-out-of-n* block are of type *other* or *switch*.

Definitions
We define the following functions on blocks:

- $After(X) = \{Y_1, ..., Y_n | (X, Y_i) \in E\}$

- $Before(X) = \{Y_i, ..., Y_n | (Y_i, X) \in E\}$
- $Knum(X) = k$ where $K\_out\_of\_n(X)$ is true.
- $Multiple\_before(X) = \{Y_1, ..., Y_n\} \in Before(X)$ where $|Before(X)| > 1$.
- $Set\_of\_K\_out\_of\_N(X) = \Pi_k(Before(X))$ where $k = Knum(X)$.

We define the following predicates on functional block diagrams:

- $Other(X)$ = True if X is a block of type *other.*
- $Series(X)$ = True if X is a block of type *series.*
- $Switch(X)$ = True if X is a block of type *switch.*
- $Fail(X)$ = True if X is a block with a fundamental failure.
- $K\_out\_of\_N(X)$ = True if X is a block of type *k-out-of-n.*
- $Detect\_fail(X)$ = True if X has been selected as a block where a detected failure implies system failure, i.e. a functional failure has occurred at X.
- $Single\_after(X, Y)$ = true iff $|After(X)| = 1$ and $Y \in After(X)$.
- $Single\_before(X, Y)$ = true iff $|Before(X)| = 1$ and $Y \in Before(X)$.
- $None\_before(X)$ = true iff $|Before(X)| = 0$.

We define the following formulas to derive the cause of failures from a functional block diagram:

1. $Detect\_fail(X) \cap Other(X) \bigcap_{Y_i \in Multiple\_Before(X)} Other(Y_i) \cup Switch(Y_i) \cap \exists Z \exists Y_j \in Y[Single\_after(Y_j, Z) \cap Other(Y_j) \cap Switch(Z)$
   $\Rightarrow Fail(X) \cup ((Detect\_fail(Y_j) \cup Fail(Z))$
   $(\bigcap_{V_i \in (Y - Y_j) \in Multiple\_Before(X)} Detect\_fail(V_i)))]$.
2. $Detect\_fail(X) \cap \exists Y[Single\_before(X, Y)$
   $\Rightarrow Fail(X) \cup Detect\_fail(Y)]$
3. $Detect\_fail(X) \cap Other(X) \bigcap_{Y_i \in Multiple\_Before(X)} Series(Y_i)$
   $\Rightarrow Fail(X) \bigcup_{Y_i \in Multiple\_Before(X)} Detect\_Fail(Y_i)$

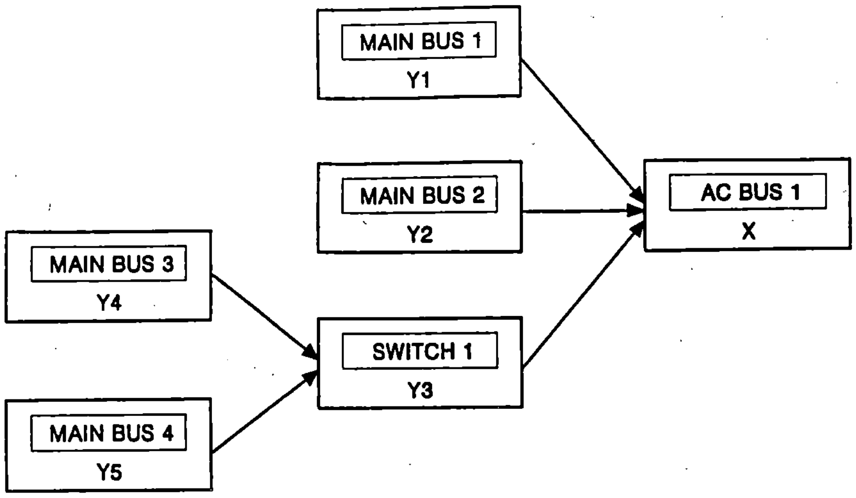

Figure 1: Functional Block Diagram for Multiple Failures

4. $Detect\_fail(X) \cap (Other(X) \cup Switch(X))$
   $\bigcap_{Y_i \in Multiple\_Before(X)} (Other(Y_i) \cup Switch(Y_i))$
   $\Rightarrow Fail(X) \cup (\bigcap_{Y_i \in Multiple\_Before(X)} Detect\_Fail(Y_i))$

5. $Detect\_fail(X) \cap K\_out\_of\_n(X)$
   $\Rightarrow Fail(X)(\bigcup_{Y_i \in Set\_of\_K\_out\_of\_n(X)} (\bigcap_{Z_j \in Y_i} Detect\_fail(Z_j)))$

6. $Detect\_fail(X) \cap None\_before(X)$
   $\Rightarrow Fail(X)$

In the following examples, these formulas are applied in numerical order.

3.1.1 Multiple Failures

Figure 1 exemplifies the relationship modeled by Formula 4. In this functional block diagram, X is connected to blocks *Y1* and *Y2* of type *other*, and to block *Y3* of type *switch*. Formula 4 shows the ways in which a functional failure can occur at X. Since $Multiple\_Before(X) = \{Y1, Y2, Y3\}$, the conclusion shows that either X fundamentally fails or blocks $Y1, Y2$, and $Y3$ have a functional failure.

According to Formula 6, since $None\_before(Y1)$ is true, $Detect\_fail(Y1)$ implies $Fail(Y1)$. Similarly, $Detect\_fail(Y2)$ implies $Fail(Y2)$. Formula 4 applies to $Detect\_fail(Y3)$, since $Y3$, of type *switch*, is connected to blocks $Y4$ and $Y5$, of type *other*. These three blocks expand into: $Fail(Y3) \cup (Detect\_fail(Y4) \cap Detect\_fail(Y5))$. Applying Formula 6 yields

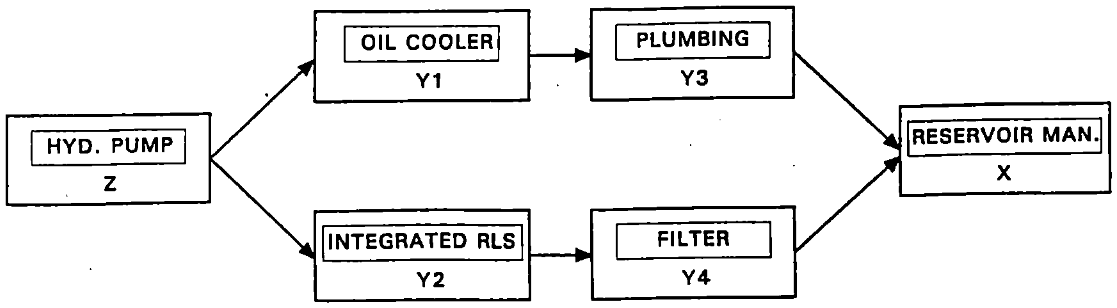

Figure 2: Functional Block Diagram for Series Failures

the transformation of *Detect_fail*($Y4$) and *Detect_fail*($Y5$) into *Fail*($Y4$) and *Fail*($Y5$). Thus, a functional failure occurs at X by one of two ways: a fundamental failure of X, or failure of one of these combinations: $\{Y1, Y2, Y3\}$ or $\{Y1, Y2, Y4, Y5\}$.

3.1.2 Series Failures

Figure 2 shows the *series* relationship. If any of the components between the beginning block X, and the ending block Z fail, then X will have a functional failure. In this diagram, X is of type *other*, the blocks in $\{Y1, ..., Y4\}$ are of type *series*, and Z is of type *other*. Consider the functional failure of the reservoir manifold X. Formula 3 applies since the members of *Multiple_Before*(X) $= \{Y3, Y4\}$ are of type *series*. The conclusion of Formula 3 implies that either a fundamental failure occurs at X, or functional failures occur at either $Y3$ or $Y4$. Formula 2 applies to *Single_before*($Y3, Y1$) and *Single_before*($Y4, Y2$).

Next, Formula 2 applies to *Detect_fail*($Y1$) with *Single_before*($Y1, Z$) implying *Fail*($Y1$) $\cup$ *Detect_fail*(Z). Similarly, Formula 2 implies *Fail*($Y2$) $\cup$ *Detect_fail*(Z). However, the first application of Formula 2 shows that a functional failure of Z can occur so we do not consider it again. We see that *None_before*(Z) is satisfying Formula 6, implying the fundamental failure of Z. Thus, the functional failure of X in the *series* relationship in Figure 2 suggests that a fundamental failure could occur at any one of these blocks: $X, Y1, Y2, Y3, Y4, Z$.

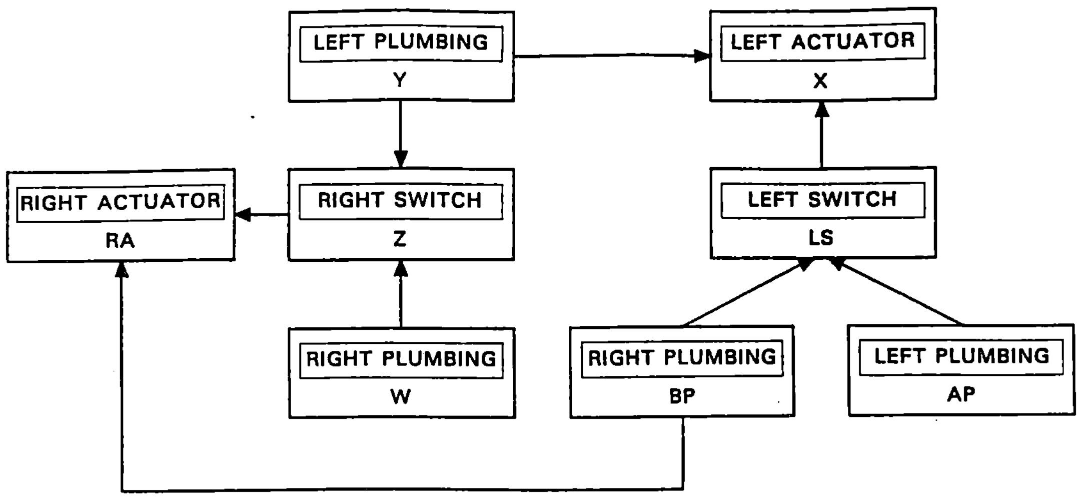

Figure 3: Functional Block Diagram for Switch Failure

3.1.3 Switch Failures

Figure 3 shows a portion of a functional block diagram modeling the *switch* relationship in a hydraulic system. There are three circuits controlling the flow of hydraulic fluid into each actuator. Switches, Z and LS, transfer fluid between lines if the hydraulic oil level drops to a low level. Consider the lack of fluid flow from the left actuator X. We first apply Formula 1 $Detect\_fail(X)$ where $Multiple\_before(X) = \{Y, LS\}$. $Single\_after(Y, Z)$ is true for the left plumbing Y and the right switch Z. The conclusion is that if fluid flow from X fails, then either the fundamental failure of X occurs, or the functional failure of LS occurs along with the fundamental failures of either Z or Y.

Next, we apply Formula 4 for $Detect\_fail(LS)$ with $Multiple\_before(LS)$ returning $\{AP, BP\}$. This implies that if the power from LS fails, then either switch LS fundamentally fails, or both AP and BP have functional failures. The failure relationship derived at this point is:

- $Fail(X) \cup ((Detect\_fail(Y) \cup Fail(Z)) \cap (Fail(LS)$
  $\cup(Detect\_fail(AP) \cap Detect\_fail(BP))))$

Formula 6 applies to the remaining $Detect\_fail$ relations converting all into $Fail$ relations. The conclusion is that the functional failure of X can occur with the fundamental failures of any of these combinations: X; Y and LS; Y, AP and BP; Z and LS; Z, AP and BP.

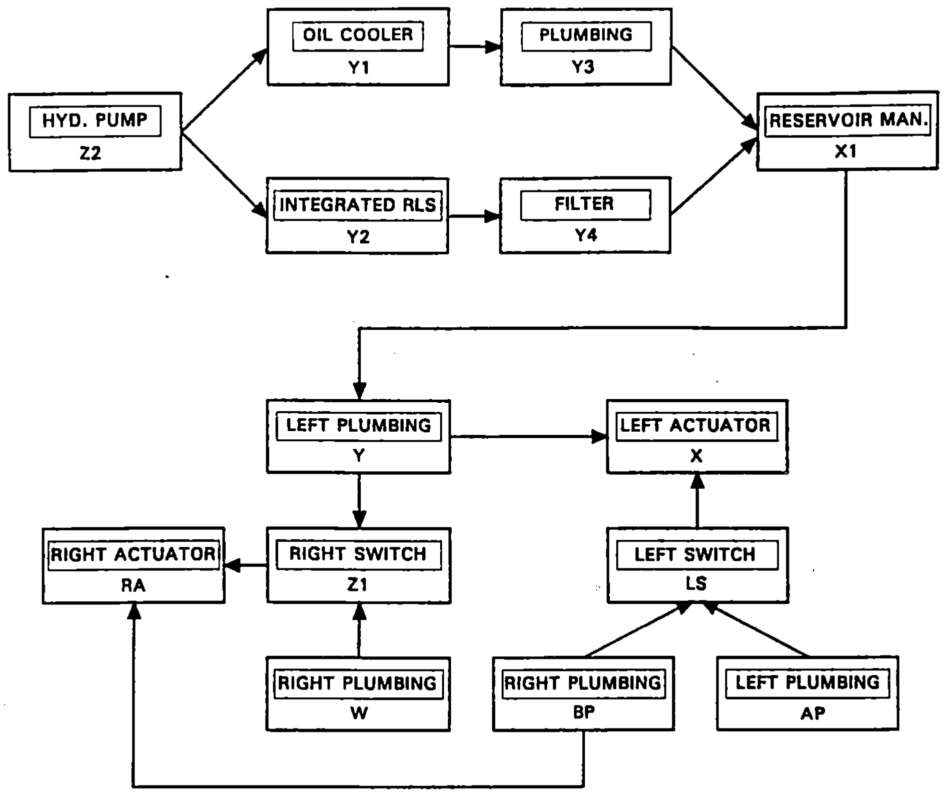

Figure 4: Functional Block Diagram for Series and Switch Combination

3.1.4 Combination of Switch and Series Failures

Figure 4 illustrates the combination of the *series* and *switch* relationships shown in Figure 2 and Figure 3 respectively. Consider the functional failure of the left actuator X in Figure 4.

After application of Formula 1 to $Detect\_fail(X)$ where $Multiple\_Before(X) = \{Y, LS\}$, the conclusion is that the functional failure of X is caused by the fundamental failure of X, or the functional failure of LS accompanied by the fundamental failure of Y or $Z1$. Next, we apply Formulas 2 and 3 to the *series* relationship, and Formula 4 to the *switch* connection deriving the following equation:

- $Fail(X) \cup ((Fail(Y) \cup Fail(X1) \cup Fail(Y1)$
  $\cup Fail(Y2) \cup Fail(Y3) \cup Fail(Y4) \cup Fail(Z1) \cup Fail(Z2))$
  $\cap (Fail(LS) \cup (Fail(AP) \cap Fail(BP))))$

This equation implies that the functional failure of left actuator X is caused by the fundamental failures of one of the following: X; or one of $\{Y, Y1, Y2, Y3, Y4, Z1, Z2\}$ along with the failure of either LS, or both AP and BP.

3.1.5 k-out-of-n:F Failures

Consider the 2-out-of-3:F relationship for the functional failure of X in the block diagram in Figure 1. We first apply Formula 5 with $Knum(X) = 2$. The resultant equation yields all subsets of size 2 of the set $Before(X)$ as follows:

- $Fail(X) \cup (Detect\_fail(Y1) \cap Detect\_fail(Y2))$
  $\cup(Detect\_fail(Y2) \cap Detect\_fail(Y3))$
  $\cup(Detect\_fail(Y1) \cap Detect\_fail(Y3))$

Formula 4 is applied to both $Detect\_fail(Y3)$ predicates. Then, Formula 6 is applied to the $Detect\_fail$ relation for $Y1, Y2, Y4$ and $Y5$. The final equation is:

- $Fail(X) \cup (Fail(Y1) \cap Fail(Y2))$
  $\cup(Fail(Y2) \cap (Fail(Y3) \cup (Fail(Y4) \cap Fail(Y5))))$
  $\cup(Fail(Y1) \cap (Fail(Y3) \cup (Fail(Y4) \cap Fail(Y5))))$

This equation implies that the 2-out-of-3:F type of functional failure for X in Figure 1 is caused by one of: X; $Y1$ and $Y2$; $Y2$ and $Y3$; $Y2$, $Y4$ and $Y5$; $Y1$ and $Y3$; or $Y1$, $Y4$ and $Y5$.

Suppose we were interested in applying these axioms for the 1-out-of-3:F relationship. The resultant equation is:

- $Fail(X) \cup (Fail(Y1) \cap Fail(Y2) \cap (Fail(Y3) \cup (Fail(Y4) \cap Fail(Y5))))$

This equation represents the same set of failures derived in Section 4.1.1 for the multiple failures example.

3.2 Semantics of Fault Tree Construction

The semantics of fault tree construction involves the relationship between the nodes in the fault tree with the failures of blocks in the functional block diagram. A fault tree is derived from the functional block diagram by selecting the block or blocks to be the TOP event in the tree. The remaining nodes in the tree are formed by recursively tracing the causes of functional failures of the TOP block or blocks in the diagram. The fault tree presents a measure of the qualitative and quantitative analysis of the functional failure of the TOP event.

To illustrate the semantics of the fault tree construction, we define predicates and functions used to generate fault trees from functional block diagrams. These are combined with previously defined predicates and functions on blocks in functional block diagrams to form fault tree construction rules. We use the term *rule* here instead of *formula* to distinguish semantics of fault tree construction from semantics of failures. Predicate calculus quantifiers such as $\forall$ and $\exists$ are sometimes eliminated for simplification. The existence of a top level connector node (OR for one top level block and AND for multiple blocks) is assumed.

Definitions

We define the following functions on fault trees:

- $Not\_in\_tree(X,Y) = \{Z_1,...,Z_n\} | Z_i \in Multiple\_before(X)$ and $Tree\_fail(Z_i)$ is false for $Z_i \in T$. $\forall Z_i$ where $Tree\_fail(Z_i)$ is true, each Z_i has a parent with a higher level in T than Y, which is parent of X.
- $Add\_and\_to\_or(X,Y,W) = \{Z_1,...,Z_n\} | \forall Z_i \in T$ there exists a corresponding $X_i \in B$. W, a connector of type *and*, is created as the parent of Z and the child of Y, a connector of type *or*.

We define the following predicates on functional block diagrams:

- $Tree\_detect\_fail(X)$ = True iff $X \in T$ and X represents the block $X \in B$ for which $Detect\_fail(X)$ is true.
- $Tree\_fail(X)$ = True iff X is $\in T$ and X represents block $X \in B$ for which $Fail(X)$ is true.
- $Parent\_node(X,Y)$ = True iff Y is parent of $X \in T$.
- $Or\_parent(X,Y,Z)$ = True iff X,of type *or*, is the parent of Z and Y is set equal to X; or X,of type *and*, is parent of Z while Y, of type *or*, is created as a new child of X with Z moved as a child of X to a child of Y.
- $Length(Z) = |Z|$.
- $Add\_child\_to\_or(X,Y)$ = true iff $X \in T$ is created as the child of connector node Y, of type *or*, where X represents $X \in FBD$.

We define the following rules to generate a fault tree from a functional block diagram:

1. $Tree\_detect\_fail(X) \cap Single\_before(X, Y) \cap Parent\_node(X, Par))$
   $\Rightarrow Or\_parent(Par, Or, X) \cap Add\_child\_to\_or(Y, Or) \cap$
   $Tree\_fail(X) \cap Tree\_detect\_fail(Y)$
2. $Tree\_detect\_fail(X) \cap Other(X) \cap Parent\_node(X, Par)$
   $\bigcap_{Y_i \in Not\_in\_tree(X,Par)} Length(Y) > 1 \cap \exists Y_j \in Y[Single\_After(Y_j, Z)$
   $\cap Switch(Z) \cap Other(Y_j)$
   $\Rightarrow Or\_parent(Par, Or, X) \bigcap_{W_i \in Add\_and\_to\_or(Y,Or,And)} Tree\_detect\_fail(W_i) \cap Or\_parent(And, Or2,$
   $Add\_child\_to\_or(Z, Or2) \cap Tree\_fail(X) \cap Tree\_fail(Z).$
3. $Tree\_detect\_fail(X) \cap (Other(X) \cup Switch(X)) \cap Parent\_node(X, Par)$
   $\bigcap_{Y_i \in Not\_in\_tree(X,Par)} \cap Length(Y) = 1 \cap (Other(Y_i) \cup Switch(Y_i))$
   $\Rightarrow Or\_parent(Par, Or, X) \cap Add\_child\_to\_or(Y_1, Or) \cap$
   $Tree\_detect\_fail(Y_1) \cap Tree\_fail(X)$
4. $Tree\_detect\_fail(X) \cap (Other(X) \cup Switch(X)) \cap Parent\_node(X, Par)$
   $\bigcap_{Y_i \in Not\_in\_tree(X,Par)} Length(Y) = 0$
   $\Rightarrow Tree\_fail(X)$
5. $Tree\_detect\_fail(X) \cap Other(X) \cap Parent\_node(X, Par)$
   $\bigcap_{Y_i \in Multiple\_Before(X)} Series(Y_i)$
   $\Rightarrow Or\_parent(Par, Or, X) \cap Tree\_fail(X)$
   $\bigcap_{Y_i \in Y} (Add\_child\_to\_or(Y_i, Or) \cap Tree\_detect\_fail(Y_i))$
6. $Tree\_detect\_fail(X) \cap (Other(X) \cup Switch(X)) \cap Parent\_node(X, Par)$
   $\bigcap_{Y_i \in Not\_in\_tree(X,Par)} Length(Y) > 1 \cap (Other(Y_i) \cup Switch(Y_i))$
   $\Rightarrow Or\_parent(Par, Or, X) \cap Tree\_fail(X)$
   $\bigcap_{Z_i \in Add\_and\_to\_or(Y,Or,And)} Tree\_detect\_fail(Z_i)$
7. $Tree\_detect\_fail(X) \cap K\_out\_of\_n(X) \cap Parent\_node(X, Par)$
   $\Rightarrow Tree\_fail(X) \cap Or\_parent(Par, Or, X)$
   $\bigcap_{Y_i \in Set\_of\_K\_out\_of\_n(X) Z_j \in Add\_and\_to\_or(Y_i,Or,And)} Tree\_detect\_fail(Z_j)$
8. $Tree\_detect\_fail(X) \cap None\_before(X)$
   $\Rightarrow Tree\_fail(X)$

In the following subsections, we apply the fault tree construction rules to build fault trees for the functional block diagrams presented in Section 4.

3.2.1 Multiple Failures Fault Tree

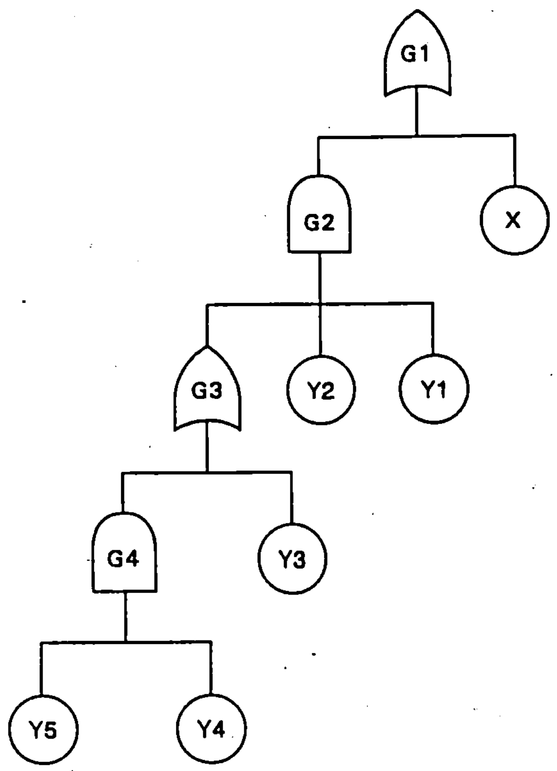

Figure 5: Fault Tree for Multiple Failures

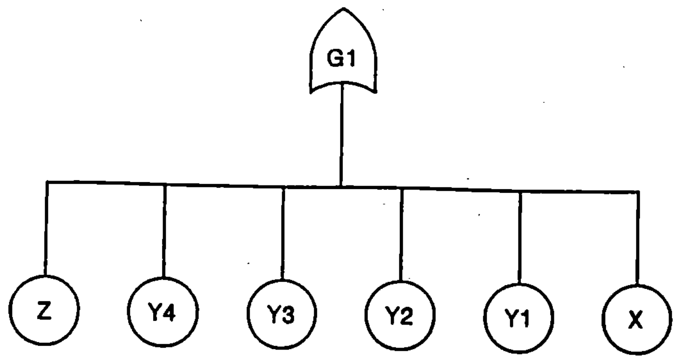

Figure 6: Fault Tree for Series Failures

To represent multiple failures in a fault tree, we expand block X in Figure 1 into the fault tree in Figure 5. We first apply Rule 6, which results in the formation of AND node G2 as a child of OR node G1. Each block returned by the function $Not\_in\_tree(X, G1)$, $\{Y1, Y2, Y3\}$, is added as a child of AND node G2. The fault tree at this stage is represented by the following boolean equation:

- $Tree\_fail(X) + ((Tree\_detect\_fail(Y1))(Tree\_detect\_fail(Y2))$ $(Tree\_detect\_fail(Y3))))$

The *Tree_detect_fail* relations for $Y1$ and $Y2$ can be resolved into *Tree_Fail* relations using Rule 8. To complete the tree, Rule 6 is applied to $Tree\_detect\_fail(Y3)$, resulting in the creation of OR node G3. $Y3$ is moved as a child of AND node G2 to a child of OR node G3. This application also results in the creation of AND node G4, as a child of OR node G3, with children: $Tree\_detect\_fail(Y4)$ and $Tree\_detect\_fail(Y5)$. Finally, Rule 8 is applied to create *Tree_Fail* relations for $Y4$ and $Y5$.

3.2.2 Series Failure Fault Tree

To generate a *series* failure fault tree, we expand block X in Figure 2 into the fault tree in Figure 6. We first apply Rule 5, forming two nodes, $Tree\_detect\_fail(Y3)$ and $Tree\_detect\_fail(Y4)$, as children of OR node G1. Rule 1 results in the formation of two more children to OR node G1: $Y1$ and $Y2$. Rule 1 applies again to the $Tree\_Detect\_fail(Y1)$ and $Tree\_detect\_fail(Y2)$ with $Single\_before(Y1, Z)$ and $Single\_before(Y2, Z)$ both true. However, as stated in Section 3.1.2, we only consider the functional failure of block Z once in the *series* relationship. Thus, Z is added once as a child of OR node G1.

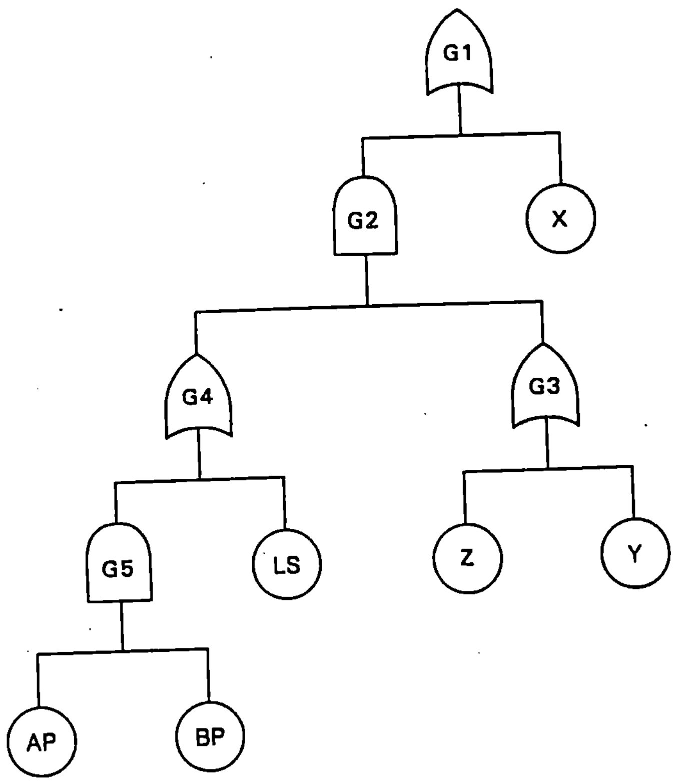

Figure 7: Fault Tree for Switch Failure

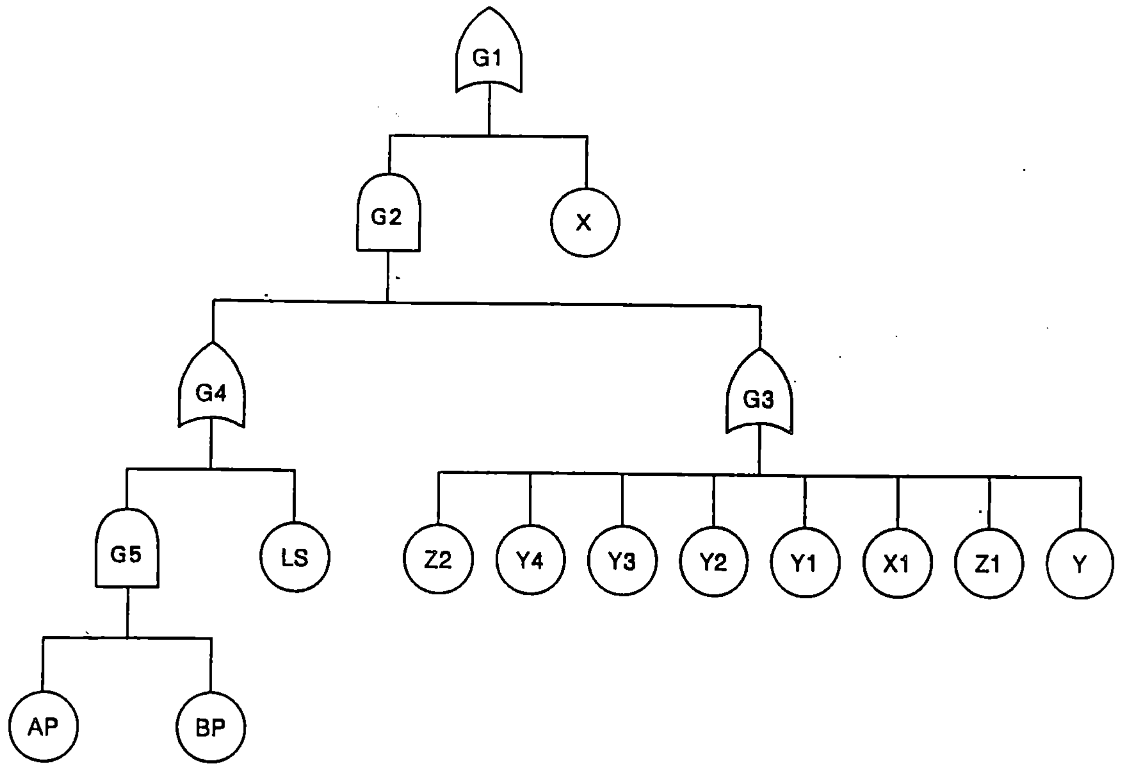

Figure 8: Fault Tree for Series and Switch Combination Failures

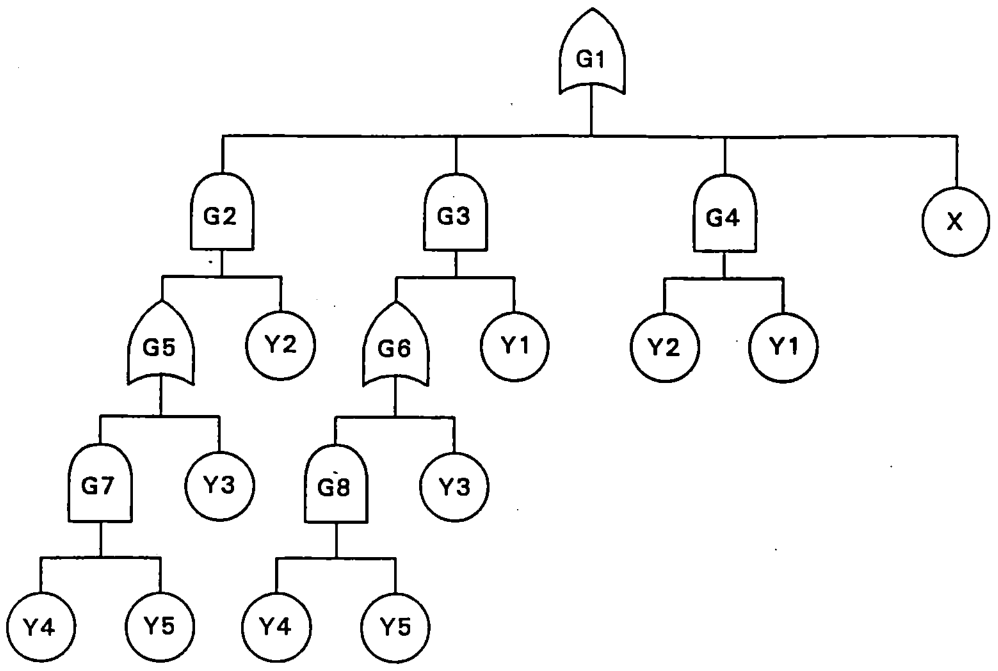

Figure 9: Fault Tree for k-out-of-n:F Failure

has $X1$ supplying it power, Rule 1 applies resulting in the addition of $X1$ as a child of OR node G3. Rule 5 is applied next to the expansion of the *series* beginning with $X1$, adding as children of OR node G3: $Y1$,$Y2$,$Y3$,$Y4$, and $Z2$. To expand $Tree\_detect\_fail(LS)$, as in the previous *switch* fault tree in Figure 7, Rule 6 is applied. Finally, Rule 8 transforms the remaining $Tree\_detect\_fail$ relations into $Tree\_fail$ relations.

3.2.5 k_out_of_n:F Failure Fault Tree

To show the construction of the k-out-of-n:F fault tree in Figure 9, we select X in Figure 1 with attribute $Knum(2)$ as the top event for the 2-out-of-3:F failures fault tree. Initially, Rule 7 applies resulting in a series of AND nodes under OR node G1, one for each 2-out-of-3:F combination. The interim fault tree at this stage is represented by the following boolean equation:

- $Tree\_fail(X)$
  $+(Tree\_detect\_fail(Y1)Tree\_detect\_fail(Y2))$
  $+(Tree\_detect\_fail(Y1)Tree\_detect\_fail(Y3))$

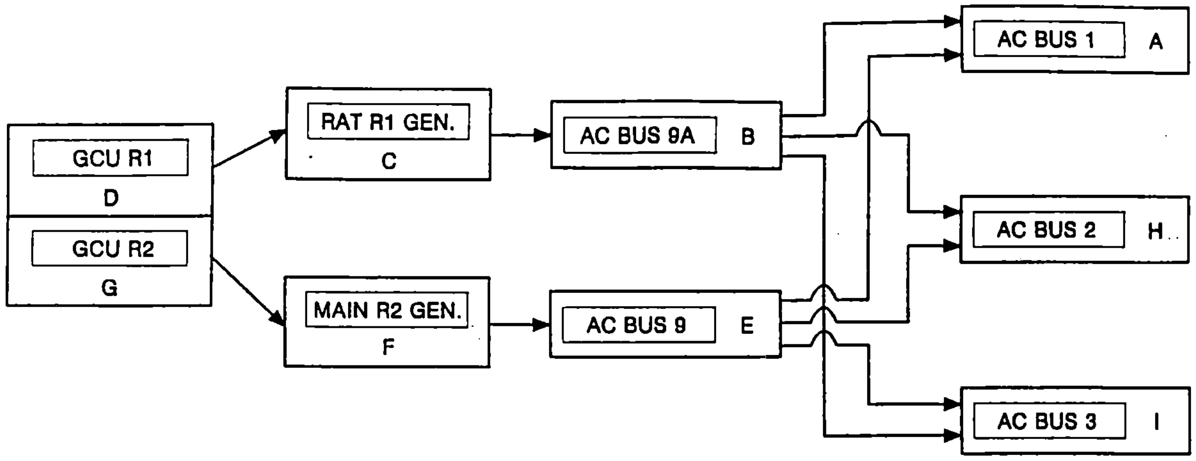

Figure 10: Functional Block Diagram for Non-flight Critical AC Mode

$+(Tree\_detect\_fail(Y2)Tree\_detect\_fail(Y3))$

To resolve each *Tree_detect_fail*$(Y3)$ relation, Rule 6 is applied resulting in the creation of AND nodes G7 and G8 with the formation of their children, *Tree_detect_fail*$(Y4)$ and *Tree_detect_fail*$(Y5)$. Finally, all remaining *Tree_detect_fail* relations satisfy *None_before*, and become *Tree_fail* relations by application of Rule 8.

3.3 Fault Tree Construction for Redundant Systems

Fault tree construction for redundant systems requires special consideration for the redundancy among the blocks in the functional block diagram. Figure 10 portrays a subset of a functional block diagram representing an aircraft electrical power system with redundancy. Figure 11 shows the fault tree for the selection of multiple top events from the diagram in Figure 10. Before the application of the fault tree rules, the initial tree contains top level AND node G1 with children: A and H. We trace the construction of the subtree under OR node G3 coinciding with the expansion of *Tree_detect_fail*(A). Rule 6 applies resulting in the creation of AND node G5 with children: *Tree_detect_fail*(B) and *Tree_detect_fail*(E).

Due to redundancy occuring between blocks B and E, the function *Not_in_tree* is essential for proper fault tree construction. For example, *Tree_detect_fail*(B) expands into OR node G9 with children: *Tree_detect_fail*(B) and AND node G13. G13 has two children: *Tree_detect_fail*(E) and *Tree_detect_fail*(C). Then, Rule 3 resolves *Tree_detect_fail*(E)

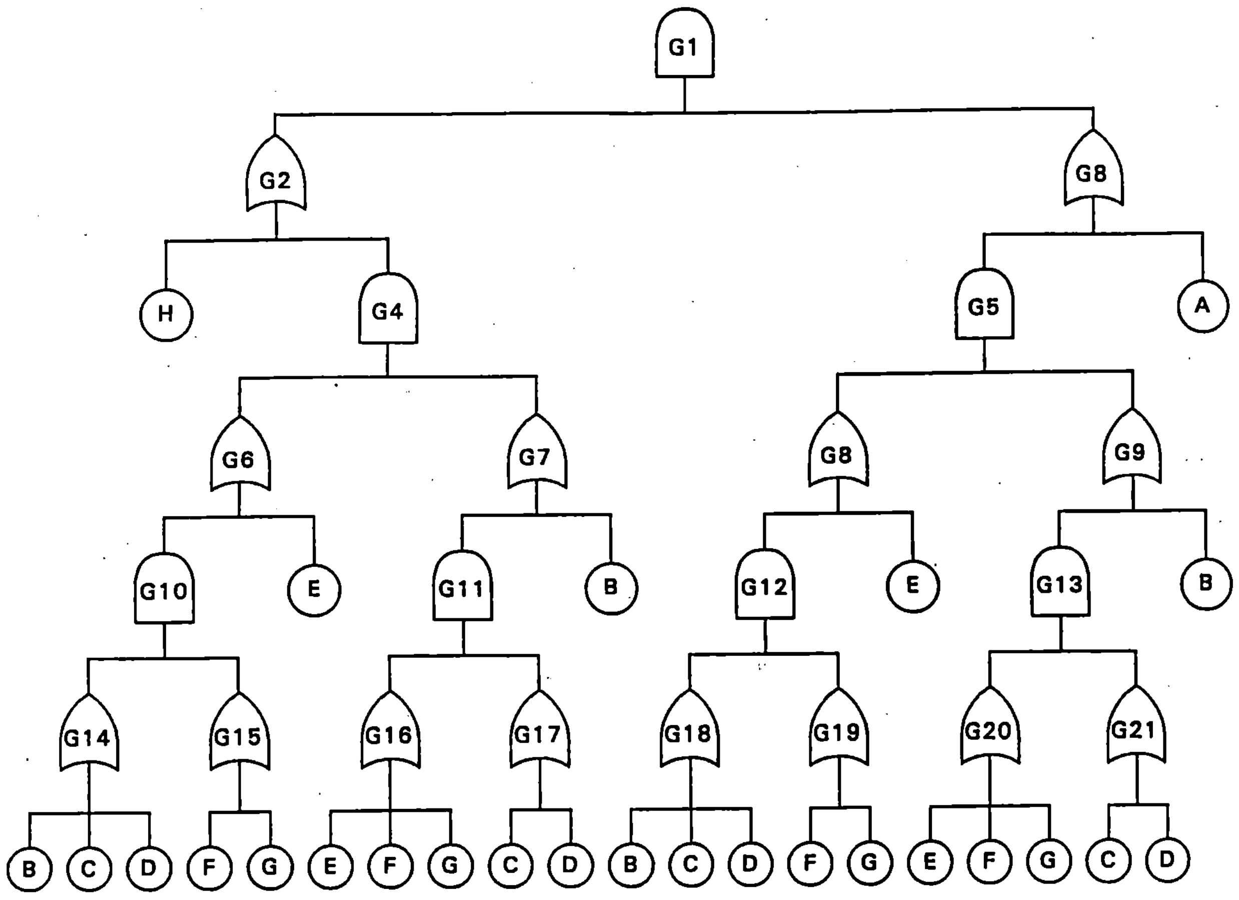

Figure 11: Fault Tree for Multiple Selected Tops

into $Tree\_fail(E)$ and adds event node F as a child of OR node G20. The list returned from Not_in_tree for the expansion of E includes only F - not B and F - because the redundant relationship between E and B has already been resolved in the tree. Rule 1 resolves the $Tree\_detect\_fail$ relations for F and C. Finally, under AND node G5, $Tree\_detect\_fail(E)$ is resolved into $Tree\_fail(E)$ and AND node G12 with children: B and F. These remaining $Tree\_detect\_fail$ relations for B and F are resolved into OR nodes G18 and G19, in the same fashion as OR nodes G20 and G21. The expansion of the remaining OR node G2 proceeds similarly as $Tree\_detect\_fail(H)$ is identical to $Tree\_detect\_fail(A)$.

4 System Overview

A reliability analysis expert system (RAES) was developed using KEE[35] and Common Lisp on a Symbolics 3650 computer. Figure 12 shows the system architecture for RAES. The fault tree construction methodology presented in Section 3 could be applied using any computer language. However, the rules presented are conveniently mapped into knowledge-based system building tools where pseudo-English rules can be used to apply the fault tree methodology. A subset of RAES including the fault tree construction rules was implemented in the logic programming language PROLOG[14].[2]

Designed in a generic style, RAES accepts input of a graphical functional block diagram for each failure mode of a system design, redundancy requirements for each failure mode, and the mission time for the fault tree analysis. An optional fault tree reduction algorithm provides minimization of the fault tree during its creation, while maintaining the boolean equivalent of the original fault tree. The k-out-of-n:F failure modeling, shown in fault tree construction Rule 7, is intended for a future version of the PROLOG code. The output of the system includes a graphical fault tree, a boolean equation, a statistical report listing the minimal cut sets of failure paths for each fault tree with associated failure probabilities, and a report verifying the system redundancy requirements. For electrical system designs, recommended design improvements to increase system reliability are produced in a report. The redundancy and rewiring rules are discussed in [24].

4.1 Fault Tree Reduction

Assumption

[2]The graphics, reduction algorithm, and minimal cut set algorithm are absent from the current version but will be added in the future.

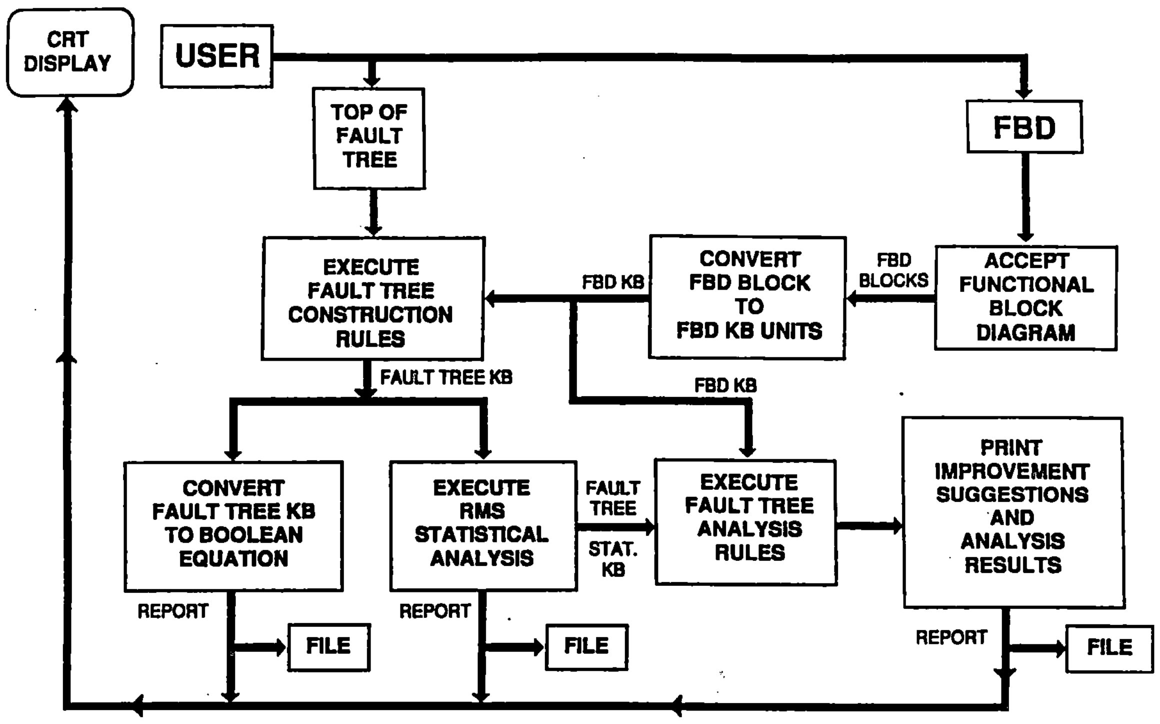
CRT DISPLAY
USER
TOP OF FAULT TREE
FBD
EXECUTE FAULT TREE CONSTRUCTION RULES
FBD KB
CONVERT FBD BLOCK TO FBD KB UNITS
FBD BLOCKS
ACCEPT FUNCTIONAL BLOCK DIAGRAM
FAULT TREE KB
FBD KB
CONVERT FAULT TREE KB TO BOOLEAN EQUATION
EXECUTE RMS STATISTICAL ANALYSIS
FAULT TREE
STAT. KB
EXECUTE FAULT TREE ANALYSIS RULES
PRINT IMPROVEMENT SUGGESTIONS AND ANALYSIS RESULTS
REPORT
FILE
REPORT
FILE
REPORT
FILE

Figure 12: RAES Architecture

- The fault tree, before its reduction, is an AND/OR tree such that AND nodes have either basic events or OR nodes as children, and OR nodes have either basic events or AND nodes as children.

Definitions

- module - Subtree in the fault tree for which some $X \in T$ has been expanded into its $Tree\_detect\_fail(X)$.
- ANDP - Node in fault tree linked to a duplicate module with identical expansion of $Tree\_detect\_fail(X)$ for $X \in T$.
- node set - Set of $\{X_0\}U\{X_1, ..., X_i\}U\{Y_1, ..., Y_n\} \in B|$ $Tree\_detect\_fail(X_0)$ or $Tree\_fail(X_0)$ is true; $\forall X_k \in \{X_1, ..., X_i\}|Single\_before(X_j, X_k)$ is true, for $j = 0, ..., i-1$ and $k = 1, ..., i$; $Single\_after(X_0, Y_1)$ is true and $\forall Y_k \in \{Y_2, ..., Y_n\}|$ $Single\_after(Y_j, Y_k)$ is true, for $j = 1, ..., n-1$ and $k = 2, ..., n$, and $Tree\_fail(Y_k)$ is true $\forall Y_k$ as children of an OR node higher in the tree than X_0.
- level - The hierarchical location of a node in the fault tree beginning with the value of 1 for the top level node, 2 for the top level's children, etc.
- node set level - Level of node at the highest position(lowest number) in the node set.
- superior node - Node with a node set level higher in position (lower in number) than the potential node set level.
- inferior node - Node with a node set level lower in position (higher in number) than the potential node set level.

A fault tree reduction algorithm is includes in the fault tree construction methodology to reduce the computational time for fault tree construction and for minimal cut set calculation. Two methods are used in the reduction:

1. Pointers linking modules to duplicate subtrees in the top eight levels of the tree.
2. Reduction of nodes in subtrees by applying Boolean algebra laws of absorption: $X + XY = X$ and $AA = A$ for all subtrees.

The identification of modules with replicated events has proven beneficial for fault tree analysis[18, 44, 52, 62, 75]. The concept of a module for coherent systems was discussed by Birnbaum and Esary[9] applied to reliability analysis. They defined *binary coherent* systems as those whose performance is enhanced by the improved performance of their components.

They defined modules as subsets of the basic components of a system organized into some substructure (or assembly of components) that can be treated as independent of the system. Further discussion of coherent systems is given in [6]. Chatterjee[18] decomposes a fault tree with replicated events into statistically independent modules eliminating duplicated events or gates.

The concept of modules for noncoherent systems was presented by Locks[52] along with techniques for minimizing noncoherent fault trees. Wilson[75] presents a computer technique that modularizes noncoherent systems using a Boolean sum of logical products of basic events. Modules are identified by Kohda, Henley and Inoue in [44] for both coherent and noncoherent trees as subtrees consisting of at least two events. These events have no inputs from the rest of the tree and no outputs except from the module's output event. Their module definition for coherent trees is similar to our definition. The identification of ANDP nodes in RAES coincides with the labeling of OR nodes with duplicate children in [44].

Unlike the abovementioned reduction techniques that reduce the fault tree *after* its construction, RAES reduces the fault tree *during* its generation. We combine a module identification technique for the first eight levels of the fault tree with Boolean algebra reduction for subtrees at all levels. The limitation of module-identification and pointers to the top eight levels enabled lower level subtrees to benefit from Boolean algebra reduction. In this way, the modules used as pointer links are reduced to a manageable size for quantification. Execution time for fault tree construction was optimum with pointers on the top eight levels for the fault trees reduced in this study. Experimentation with varying limits on pointer level versus boolean reductions will continue in future work.

4.1.1 Example of Module Identification

Module identification occurs during a $Tree\_detect\_fail(X)$ expansion. Instead of adding an AND node to the tree, the reduction algorithm identifies a $Tree\_fail$ relation with an equivalent expansion. A pointer node labeled ANDP is formed linking the new $Tree\_detect\_fail$ to its duplicate $Tree\_fail$ node. Figure 13 shows the reduced version of Figure 11. Since $Tree\_Detect\_fail(H)$ has a subtree expansion which matches the subtree of AND node G5 (representing expansion of $Tree\_Detect\_fail(A)$), an ANDP pointer is formed for H linking it to AND node G5. The $Tree\_fail(A)$ relation is first reduced using Boolean algebra. Not only do these pointers save computer storage, they also reduce computation time for the minimal cut set algorithm described in Section 4.2.

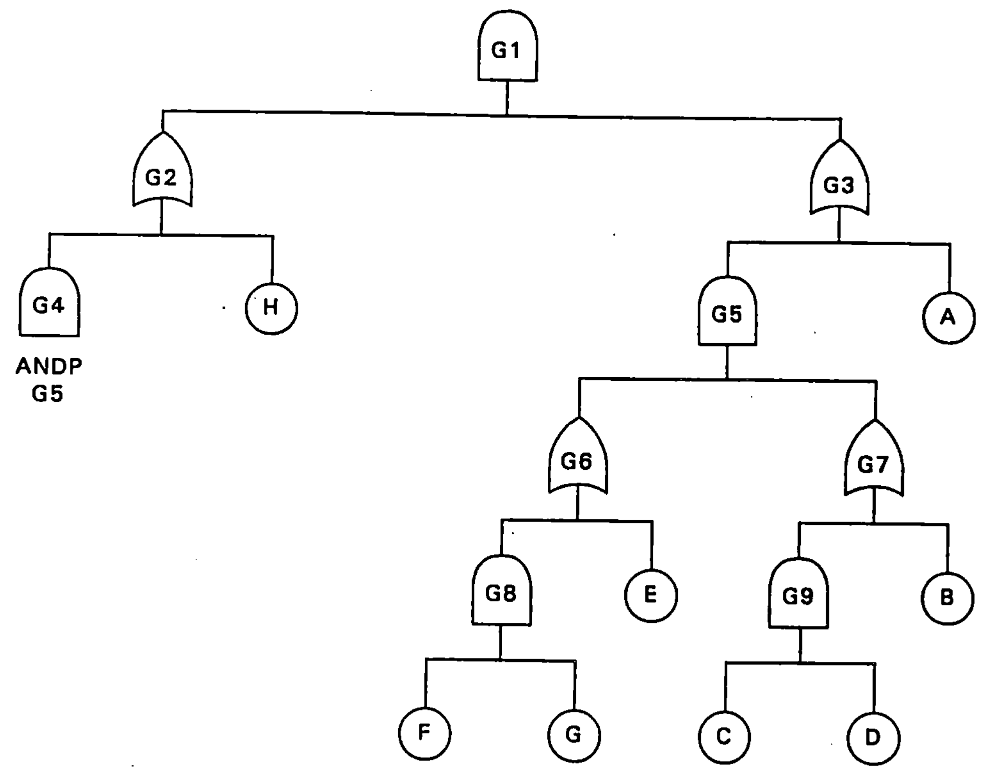

Figure 13: Reduced Fault Tree with Pointer Nodes

4.1.2 Analysis of Boolean Reduction

Lemma 1 *In the Boolean algebra absorption law, $XX = X$, if X is replaced by a sum of n products, the absorption law holds for that sum of n products.*

Proof: If X is replaced by a sum of products, $(Y + Z)$, where Y or Z may also be a sum of products, such that the total products in X is n, then using the distributive postulate of Boolean algebra, $X(Y+Z) = XY+XZ$, we show the following reduction when X is replaced with a sum of 2 products:

- $(Y+Z)(Y+Z) = Y(Y+Z)+Z(Y+Z) = YY+YZ+ZY+ZZ = Y+ZY+Z = Y+Z.$

Theorem 1 *In the AND/OR tree T represented by Boolean equation Z, suppose subtree S has an AND parent node with children:$\{N_1, ..., N_i\} | N_j \in \{N_2, ..., N_i\}$ is a basic event or an OR node, and N_1 is a basic event. If a duplicate node D matching N_1 is in T as a child of an AND node $2n(n \geq 0)$ levels below one of N_j, then node D can be deleted from T while maintaining Boolean equivalence in Z.*

Proof, by induction on n:

Basis step: We show that for $n = 0$ and for Boolean equation Z representing T with duplicate nodes, D and N_1, if D is an inferior node to N_1 by a level difference of $2n(= 0)$, D can be deleted from T maintaining Boolean equivalence in Z.

Proof of Basis step: The Z satisfying $n = 0$ is equivalent to AA, which by the absorption law is equivalent to A. Hence, we have shown that,for $n = 0$, one element can be deleted without loss of Boolean equivalence.

Inductive Step: By the inductive hypothesis, we assume that for $n \geq 1$, in Boolean equation Z representing T with duplicates, D and N_1, if D is lower than N_1 at a level difference of $2n$, node D can be deleted from T maintaining Boolean equivalence in Z. We show that this also holds for $2(n + 1)$.

Consider this Boolean equation:

$$Z = (X_1(X_2 + X_3(X_4+, ..., +X_m(X_{m+1}+, ..., +(X_{l-1}X_l))))) \qquad (1)$$

where $X_1 = X_m$ and l is the maximum level (depth) of the corresponding fault tree T.

In fault tree T depicting this Boolean equation, X_m at level m, is the inferior duplicate of X_1 at level 2. Thus, by our inductive hypothesis, we assume that X_m can be deleted from Z. We show that if X_m is moved to level $m + 2$ as a child of a deeper AND node, then X_{m+2} can also be deleted. The level difference between X_1 and X_m is $m - 2 = 2n(n \geq 1)$.

Solving for m, we get $m = 2n + 2$. After moving X_m to level $m + 2$, the new level difference is $(m+2) - 2 = m$. By the previous equation, $m = 2n + 2 = 2(n+1)$. Thus, we have shown that the new level difference is $2(n+1)$.

We use the distibutive postulate $X(Y + Z) = XY + XZ$, and the absorption law, $XX = X$, of Boolean algebra to show this deletion process with Boolean equivalence. Using the distributive postulate, we expand Boolean expression Z, representing tree T, into a sum of products. Since the AND/OR nodes are nested, as in Equation 1, the higher level event(X_1) will be distributed over all products in the second sum. If any of those products contains a duplicate term (X_m), then the absorption law eliminates the duplicate term. If we move X_m to level $m + 2$ under a new AND/OR expression, we use the absorption principle again to conclude that at a level difference of $2(n + 1)$, we can delete X_{m+2} and maintain a boolean equivalence.

Using Lemma 1, we can replace N_1 and D in this theorem with a sum of products expression, and apply the absorption prinicple to delete D. Similarly, if N_1 was of the form: $X + YZ$, using the distributive property, we can expand N_1 to: $(X + Y)(X + Z)$. Then if D is located at a level difference of $2n$, and it is of the form $(X + Y)$ or $(X + Z)$, we can delete D.

4.1.3 Boolean Reduction Algorithm

The general algorithm used to reduce the fault trees was added to RAES by modification of Rule 6 in the methodology to do the following:

1. Search for any existing OR node whose $Tree\_fail(Y)$ node set is identical to the node set of $Tree\_detect\_fail(X)$. If a match is found, form a pointer node for the expansion of $Tree\_detect\_fail(X)$ labeled ANDP containing the address of the $Tree\_fail$ AND node.
2. If no pointer node is found in Step 1, then do the following:
   - Search for any superior OR node whose node set level is higher in the tree than the potential OR node's level with identical node sets. Do not add the potential node if a superior node is found.
   - If no superior nodes are found, search for all inferior OR nodes whose node set level is lower than the potential node's level. Delete all inferior nodes found. Add the $Tree\_detect\_fail(X)$ expansion to the tree.

Figure 13 shows the reduced fault tree for the tree in Figure 11. The reduction algorithm was applied to the subtree under AND node G5. Referring to nodes in Figure 11, we explain how the reduction works.

The subtrees below OR node G9 are expanded as shown with no reduction occurring yet. Then, *Tree_detect_fail*(E) under AND node G5 is expanded using reduction. As Rule 6 is expanding *Tree_detect_fail*(E), the search for a superior node begins with *Tree_detect_fail*(F) under OR node G19. The node set for F is $\{E, F, G\}$ and its accompanying node set level is that of E, level 5. Since no superior node is found, a search for inferior nodes ensues. A match is found under OR node G20 whose node set is $\{E, F, G\}$, with node set level 7. According to Theorem 1, the node set level difference is $2n(n = 1)$, so we delete OR node G20 and its children. Since AND node G13 now has only one child, OR node G21, the children of OR node G21 become the children of OR node G9. The two nodes, G13 and G21, are deleted.

The final reduction occurs when *Tree_detect_fail*(B) under AND node G12 is being expanded with node set $\{B, C, D\}$ and node set level 7. During the search for superior nodes, OR node G21 is found with matching node set $\{B, C, D\}$ at level 5. Applying Theorem 1, we see that the node set level difference is $2n(n = 1)$, so OR node G18 is not added. The remaining nodes are adjusted to form the reduced subtree in Figure 13.

This reduction algorithm was tested by computing minimal cut sets using both the reduced and non-reduced rules for the electrical fault trees derived in this study. For the flight critical and mission critical failure modes the execution of the non-reduced rules was beyond the scope of the computing power of the Symbolics computer. However, for the non-flight critical AC and DC failure modes, the reduction data is shown in Table 1. This table reflects the greater than 50% reduction in fault tree nodes and in time of fault tree construction using the reduction algorithm.

4.2 Minimal Cut Set Algorithm

Minimal cut set algorithms have been successfully implemented in computer programs. Some programs such as the PREP code[71, 72] use Monte Carlo techniques, while others use deterministic algorithms, such as MOCUS[31, 32], ELRAFT[64], MICSUP[56], SETS[77], and PREPE[71]. Detailed discussions about these and many more computer codes can be found in [50] and [73]. Most computer codes developed using these algorithms analyze coherent fault trees. Worrell's SETS[77] code handles both coherent and noncoherent fault trees. The widely used MOCUS[31, 32] algorithm stores the Boolean Indicated Cut Sets(BICS) in a matrix and then uses a pattern matching technique to determine the minimal cut sets.

Concern for reducing computational time and storage necessary to compute the minimal

FAULT TREE DATA	AC MODE	DC MODE
NON-REDUCED NODES	80	330
NON-REDUCED TIME	15 min.	30 min.
REDUCED NODES	39	146
REDUCED TIME	3 min.	10 min.

Table 1: Reduced Versus Non-reduced Fault Trees

cut sets resulted in the development of more efficient algorithms[33, 51, 61]. The DICOMICS algorithm[33] is based on segmentation of the tree into a fully equivalent forest of subtrees for expansion into minimal cut sets. The Fatram[61] algorithm uses dynamic storage to increase the MOCUS algorithm's efficiency. Limnios and Ziani[51] developed an improvement to the top-down MOCUS algorithm computing the BICS faster; however, MOCUS is still used for minimal cut sets. More recently, Vatn[70] developed the CARA algorithm that builds a virtual cut set tree structure used to obtain minimal cut sets. Vatn's algorithm requires less storage and is faster than MOCUS.

Of the previously mentioned minimal cut set algorithms, the one closest to the RAES algorithm is MICSUP[56]. Both use the bottom-up approach to compute minimal cut sets, using depth-first search to calculate minimal BICS for each intermediate gate. Unlike MICSUP, RAES uses recursion with a list-processing language and a hash table for intermediate cut set storage. By manually applying the Fatram[61] algorithm and the one in [51] to the non-flight critical AC fault tree used in this study, it was determined that RAES makes more comparisons than these algorithms. However, storage requirements are no greater using RAES than Fatram and are less than required in [51].

Fault tree reduction before the minimal cut set computation reduces computer time and space needed by the minimal cut set algorithm. Furthermore, by using a hash table for storage of minimal cut sets, the amount of dynamic storage needed is for minimal cut sets only. This is an improvement over MICSUP which needs an array dimensioned to a maximum expected minimal cut set size.

The recursive algorithm used in RAES to compute the minimal cut sets is defined with two functions: *Min_cutset* and *Compute_min*. These functions are defined as:

Function Min_cutset(N)

1. Retrieve the branches of the node N as a list B.
2. If N is an OR node, for each branch B_i do:
    - a) If B_i is an AND or OR node, execute $L = Min\_cutset(B_i)$. Append L to list C.
    - b) If B_i is an ANDP node, retrieve its minimal cut sets from the knowledge base and append to C.
    - c) If B_i is an EVENT node, append it to C.
3. If N is an AND node, for each branch B_i do:
    - a) If B_i is an AND or OR node, execute $L = Min\_cutset(B_i)$. Add list L to C as a new member of C.
    - b) If B_i is EVENT node, append it to C.

    At end of AND node processing, execute $C = Compute\_mins(C)$.
4. Return list C as minimal cut sets for N.

Function Compute_mins(C)

1. Clear hash table H and add contents of C to hash table H.
2. For first two lists in C do :
    - a) Combine each member of $C1$ with each of $C2$ into list $C3$.
    - b) Compare each $C3_i$ with each member of H and do:
        - Add $C3_i$ to H if 1) No supersets of $C3_i$ are in H, 2) No subsets of $C3_i$ are in H, or 3)No duplicates of $C3_i$ are in H.
        - Add $C3_i$ to H if supersets of $C3_i$ are in H but delete supersets from H.
        - Do not add $C3_i$ to H if subsets of $C3_i$ are in H.
    - If there are any remaining lists in C, append list of nodes in H to C and execute $Compute\_mins(C)$ recursively.
    - After all lists in C have been processed, return contents of H as a list.

These are the steps taken by the minimal cut set algorithm:

1. Execute *Min_cutset*(AND) for each AND node with ANDP pointers linked to it. Store the minimal cut sets for these AND nodes in their knowledge base units.

2. Execute $M = Min\_cutset(N)$ for the tree top level connector N. The list of nodes returned in M contains the minimal cut sets for the failure of N's top event.

This algorithm is implemented in the Lisp language. Future work includes its addition to the PROLOG code available from the author. During the experiments of RAES, a hash table improved computational speed over use of a list to store minimal cut sets. The key used for hashing is the first element of a sorted cut set list. The cut sets are stored in a subclass in the RAES knowledge base for further statistical application. The exponential distribution provides the probabilities of failure for the blocks in each minimal cut set where

λ = failure rate is represented as $N * 10^{-6}$, where N is an integer, and where t = mission time, the probability of failure, F_n for each minimal cut set n is the following:
$F_n(t) = \prod_{i=1}^{k}(1 - e^{-\lambda_i t})$ where
λ_i is the ith block failure rate.
The total failure probability is:
$F_t = \sum_{i=1}^{n} F_n$.

5 Limitations

The fault tree construction methodology presented in this paper is limited to 2-state coherent fault trees. Therefore, complex systems with control loops as found in chemical processing[47], cannot be modeled using this system. The RAES methodology would be enhanced by inclusion of construction of noncoherent fault trees with control loop structures. To derive noncoherent fault trees from graphical block diagrams, we need to include additional block types and rules to transform functional block diagrams into noncoherent trees. The addition of noncoherent fault trees is being considered for future work.

Currently, the knowledge-based system is constrained to the input of a graphical or tabular functional block diagram. A more flexible approach would be to also allow input of a fault tree for minimal cut set calculation. Another limitation is that all minimal cut sets are calculated without consideration given to reducing cut sets via the specification of a maximum cut set size or of limits on probabilities. In addition, allowing functional block diagram interconnections to be derived directly from computer-aided design(CAD)

FAILURE MODE	MANUAL	RAES	ERRORS IN MANUAL
NON-FLIGHT AC	1 DAY	3 MINS.	0 OUT OF 74 MIN CUT SETS
NON-FLIGHT DC	4 DAYS	10 MINS.	1 OUT OF 116 MIN CUT SETS
MISSION CRITICAL DC	5 DAYS	45 MINS.	14 OUT OF 140 MIN CUT SETS
FLIGHT CRITICAL DC	7 DAYS	2.5 HRS.	2 OUT OF 480 MIN CUT SETS

Table 2: RAES Electrical Analysis Times

data would enhance the system's utility. Previous experiments by the author with the use of CAD data as input to RAES have been attempted with preliminary success[26].

6 RAES Results

RAES was tested with the reliability analysis of two aircraft subsystems: a fault tolerant electrical power system and a hydraulic system. For the electrical system, four failure modes were considered - non-flight critical AC mode, non-flight critical DC mode, mission critical DC mode, and flight critical DC mode. The hydraulic system used one failure mode - the loss of power to the F-18 aircraft's hydraulic system. The functional block diagrams used to test RAES were obtained from manual reliability analysis reports on a fault tolerant electrical power system[65] and hydraulic system designs for an aircraft. The minimal cut sets obtained using RAES were compared to these manual results showing that RAES performed the same manual task in less time and without the errors revealed in the manual approach. Table 2 illustrates these findings for the electrical power system and Table 3 shows similar data for the hydraulic system.

The meaning of the manual column as shown in Tables 2 and 3 is the amount of time

FAILURE MODE	MANUAL	RAES	ERRORS IN MANUAL
HYDRAULIC POWER	2 DAYS	2 HOURS	30 OUT OF 350 MIN CUT SETS

Table 3: RAES Hydraulic Analysis Times

needed by an expert reliability engineer to draw the fault tree, graphically enter the tree into a computer system for minimal cut set calculation, and obtain the statistical analysis. The LTG[7] was used in the manual approach for this study. This program uses the MOCUS[31, 32] algorithm and KITT[71] code. Because of limitations on input tree size of the MOCUS algorithm, sections of the larger fault trees, such as the one for Flight Critical DC mode, required minimal cut sets to be combined manually from subtrees.

The times in the RAES column in Tables 2 and 3 show the amount of computer time used to compute the fault tree with reduction on the Symbolics computer. The time needed for functional block diagram entry was eliminated from the analysis since the point of interest in this study was improved efficiency in fault tree construction. The RAES time for graphical entry of a block diagram with 18 blocks was about 30 to 45 minutes.

In Table 2, one can see that there is little difference between the days to perform this task manually for non-flight DC and mission critical DC, even though the computer time for RAES to complete this task increased from 10 minutes to 45. This small differential in the manual approach is due to the functional block diagram and fault tree for non-flight DC mode containing similar connections to the mission critical DC mode. Hence, moving manually from one failure mode to the other enabled the engineer to duplicate some of the tree's nodes. However, RAES does not use historical data from other fault trees for its computations and consequently, the computation times increase more dramatically.

Many manual errors were found for the hydraulic system. This is because the manual fault tree contained a major error resulting in 30 minimal cut sets never being calculated.

Acknowledgements

The author also wishes to thank Abas Ghasemkhani of Northrop's Aircraft Division for his help in establishing the reliability engineering requirements for the system, and for his feedback on the user interface.

References

[1] K. K. Aggarwal. Comments on an efficient simple algorithm for fault tree automatic synthesis from the reliability graph. *IEEE Transactions on Reliability*, Vol. 28(No. 4):309, October 1979.

[2] P. K. Andow. Difficulties in fault-tree synthesis for process plant. *IEEE Transactions on Reliability*, Vol. 29(No. 1):2–9, April 1980.

[3] J. Andrews and G. Brennan. Application of the digraph method of fault tree construction to a complex control configuration. *Reliability Engineering and System Safety*, Vol. 28(No. 3):357–384, 1990.

[4] G. Apostolakis, S. Garribba, and G. Volta, editors. *Synthesis and Analysis Methods for Safety and Reliability Studies.* Plenum Press, New York, N.Y., 1980.

[5] R. E. Barlow and H. E. Lambert. Introduction to fault tree analysis. In *Reliability and Fault Tree Analysis*, pages 7–35. Society for Industrial and Applied Mathematics, September 1975.

[6] R. E. Barlow and F. Proschan. *Statistical Theory of Reliability and Life Testing.* Holt, Rinehart and Winston,Inc., New York, N.Y., 1975.

[7] L. Bass, H. W. Wynholds, and W. R. Porterfield. Fault tree graphics. In *Reliability and Fault Tree Analysis.* Society for Industrial and Applied Mathematics, Philadelphia, Penna., 1975.

[8] N. N. Bengiamin, B. A. Bowen, and K. F. Schenk. An efficient algorithm for reducing the complexity of computation in fault tree analysis. *IEEE Transactions on Nuclear Science*, Vol. NS-23:1442–1446, October 1976.

[9] Z. W. Birnbaum and J. D. Esary. Modules of coherent binary systems. *Journal of Society for Industrial and Applied Mathematics*, Vol. 13(No. 2):444–462, June 1965.

[10] Z. W. Birnbaum, J. D. Esary, and S.C. Saunders. Multi-component systems and structures and their reliability. *Technometrics*, Vol. 3(No. 1):55–77, February 1961.

[11] A. Bossche. Computer-aided fault tree synthesis 1. (System modeling and causal trees). *Reliability Engineering and System Safety*, Vol. 32(No. 3):217–241, 1991.

[12] A. Bossche. Computer-aided fault tree synthesis 2. Fault tree construction. *Reliability Engineering and System Safety*, Vol. 33(No. 1):1–21, 1991.

[13] A. Bossche. Computer-aided fault tree synthesis 3. Real-time fault location. *Reliability Engineering and System Safety*, Vol. 33(No. 2):161–176, 1991.

[14] I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley Publishing Company, Inc., 1990.

[15] K. S. Brown. Evaluating fault trees (and & or gates only) with repeated events. *IEEE Transactions on Reliability*, Vol. 39(No. 2):226–235, June 1990.

[16] P. Camarda, F. Corsi, and A. Trentadue. An efficient simple algorithm for fault tree automatic synthesis from the reliability graph. *IEEE Transactions on Reliability*, Vol. R-27(No. 3):215–221, August 1978.

[17] C. Chang and H. Hwang. New developments of the digraph-based techniques for fault-tree synthesis. *Industrial & Engineering Chemical Research*, Vol. 31(No. 6):1490–1502, June 1992.

[18] P. Chatterjee. Modularization of fault trees: A method to reduce the cost of analysis. In *Reliability and Fault Tree Analysis*, pages 101–126. Society for Industrial and Applied Mathematics, September 1975.

[19] T. L. Chu and G. Apostolakis. Methods for probabilistic analysis of noncoherent fault trees. *IEEE Transactions on Reliability*, Vol. R-29:354–360, Dec. 1980.

[20] Y. Chunning and S. Dinghua. Classification of fault trees and algorithms of fault tree analysis. *Microelectronics Reliability*, Vol. 30(No. 5):891–895, 1990.

[21] U.S. Nuclear Regulatory Commission. Reactor safety study - an assessment of accident risk in U.S. commercial nuclear power plants. Technical Report WASH-1400, Washington, DC, October 1975.

[22] D. L. Cummings, S. A. Lapp, and G. J. Powers. Fault tree synthesis from a directed graph model for a power distribution network. *IEEE Transactions on Reliability*, Vol. R-32(No. 2):140–149, June 1983.

[23] M. S. Elliott. Knowledge-based systems for reliability analysis in concurrent design. In *IJCAI89 Concurrent Engineering Design Workshop Proceedings.* American Association for Artificial Intelligence, August 1989.

[24] M. S. Elliott. Knowledge-based systems for reliability analysis. In *1990 Reliability and Maintainability Symposium*, pages 481–489. IEEE, January 1990.

[25] M. S. Elliott. Reliability analysis expert system in concurrent engineering. In *Proceedings of the 2nd National Symposium on Concurrent Engineering*. Concurrent Engineering Research Center, February 1990.

[26] M.S. Elliott. Knowledge-based systems in logistics. Independent Research and Development Project Description 90-R-500, Northrop Research and Technology Center, 1989.

[27] J. D. Esary and F. Proschan. Coherent structures of non-identical components. *Technometrics*, Vol. 5(No. 2):191–209, May 1963.

[28] T. Feo. Paft77, program for the analysis of fault trees. *IEEE Transactions on Reliability*, Vol. R-35(No. 1):48–50, April 1986.

[29] J. B. Fussell. Computer aided fault tree construction for electrical systems. In *Reliability and Fault Tree Analysis*, pages 37–56. Society for Industrial and Applied Mathematics, September 1975.

[30] J. B. Fussell. How to hand-calculate system reliability and safety characteristics. *IEEE Transactions on Reliability*, Vol. R-24(No. 3):169–174, August 1975.

[31] J. B. Fussell, E. B. Henry, and N. H. Marshall. Mocus - a computer program to obtain minimal sets from fault trees. Technical Report ANCR-1156, Aerojet Nuclear Company, Idaho Falls, Idaho, March 1974.

[32] J. B. Fussell and W. E. Vesely. A new methodology for obtaining cut sets for fault trees. *Transactions of American Nuclear Society*, Vol. 15(No. 1):262, April 1972.

[33] S. Garribba, R. Mussio, F. Naldi, G. Reina, and G. Volta. Efficient construction of minimal cut sets from fault trees. *IEEE Transactions on Reliability*, Vol. R-26(No. 2):88–93, June 1977.

[34] E. J. Henley and H. Kumamoto. *Probabilistic Risk Assessment Reliability Engineering, Design, and Analysis.* IEEE Press, New York, N. Y., 1992.

[35] Intellicorp Corp. *KEE Software Development System User's Manual*, 3.0-v-1 edition, 1986. KEE is a trademark of Intellicorp.

[36] P. C. Jackson. *Introduction to Artificial Intelligence.* Petrocelli Books., 1974.

[37] B. E. Kelly and F. P. Lees. The propagation of faults in process plants: 1. Modelling of fault propagation. *Reliability Engineering*, Vol. 16:3–38, 1986.

[38] B. E. Kelly and F. P. Lees. The propagation of faults in process plants: 2. Fault tree synthesis. *Reliability Engineering*, Vol. 16:39–62, 1986.

[39] B. E. Kelly and F. P. Lees. The propagation of faults in process plants: 3. An interactive, computer based facility. *Reliability Engineering*, Vol. 16:63–86, 1986.

[40] B. E. Kelly and F. P. Lees. The propagation of faults in process plants: 4. Fault tree synthesis of a pump system changeover sequence. *Reliability Engineering*, Vol. 16:87–108, 1986.

[41] D. H. Knuth. *The Art of Computer Programming, Vol. 1.* Addison-Wesley., 1968.

[42] B. V. Koen and A. Carnino. Reliability calculations with a list processing technique. *IEEE Transactions on Reliability*, Vol. R-23(No. 1):43–50, April 1974.

[43] T. Kohda and E. J. Henley. On digraphs, fault trees and cut sets. *Reliability Engineering and System Safety*, Vol. R-23:35–61, 1988.

[44] T. Kohda, E. J. Henley, and K. Inoue. Finding modules in fault trees. *IEEE Transactions on Reliability*, Vol. R-38(No. 2):165–176, June 1989.

[45] M. A. Kramer and Jr. B. L. Palowitch. A rule-based approach to fault diagnosis using the signed directed graph. *AIChE Journal*, Vol. 33(No. 7):1067–1078, July 1987.

[46] H. Kumamoto and E. J. Henley. Top-down algorithm for obtaining non-coherent fault trees. *IEEE Transactions on Reliability*, Vol. R-27(No. 4):242–249, October 1978.

[47] S. A. Lapp and G. J. Powers. Computer-aided synthesis of fault trees. *IEEE Transactions on Reliability*, pages 2–13, April 1977.

[48] S. A. Lapp and G. J. Powers. The synthesis of fault trees. In *Nuclear Systems Reliability Engineering and Risk Assessment*, pages 778–799. Society for Industrial and Applied Mathematics, 1977.

[49] S. A. Lapp and G. J. Powers. Update of Lapp-Powers fault tree synthesis algorithm. *IEEE Transactions on Reliability*, Vol. R-28(No. 1):12–15, 1979.

[50] W. S. Lee, D. L. Grosh, F. A. Tillman, and C. H. Lie. Fault tree analysis, methods, and applications - a review. *IEEE Transactions on Reliability*, Vol. R-34(No. 3):194–203, August 1985.

[51] N. Limnios and R. Ziani. An algorithm for reducing cut sets in fault tree analysis. *IEEE Transactions on Reliability*, Vol. R-35(No. 5):559–562, December 1986.

[52] M. O. Locks. Synthesis of fault trees; an example of noncoherence. *IEEE Transactions on Reliability*, Vol. R-28(No. 1):2–5, April 1979.

[53] J. S. Mullhi, M. L. Ang, and F. P. Lees. The propagation of faults in process plants: 5. Fault tree synthesis for a butane vaporiser systems. *Reliability Engineering and System Safety*, Vol. R-23:31–49, 1988.

[54] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company., 1980.

[55] L. B. Page and J. E. Perry. An algorithm for exact fault-tree probabilities without cut sets. *IEEE Transactions on Reliability*, Vol. R-35(No. 5):544–558, December 1986.

[56] P.K. Pande, M. E. Spector, and P. Chatterjee. Computerized fault tree analysis TREEL and MICCSUP. Technical Report ORC 75-3, Operation Research Center, University of California, Berkeley, April 1975.

[57] F. A. Patterson and B. V. Koen. Direct evaluation of fault trees using object-oriented programming techniques. *IEEE Transactions on Reliability*, Vol. R-38(No. 2):186–192, June 1989.

[58] G. J. Powers, F. C. Tompkins, and S. A. Lapp. A safety simulation language for chemical processes: A procedure for fault tree synthesis. In *Reliability and Fault Tree Analysis*, pages 57–75. Society for Industrial and Applied Mathematics, September 1975.

[59] R. A. Pullen. AFTP fault tree analysis program. *IEEE Transactions on Reliability*, Vol. R-33(No. 2):171, June 1984.

[60] Jr. R. T. Hessian, B. B. Salter, and E. F. Goodwin. Fault-tree analysis for system design, development, modification, and verification. *IEEE Transactions on Reliability*, Vol. 39(No. 1):87–91, April 1990.

[61] D. M. Rasmuson and N. H. Marshall. FATRAM - a core efficient cut-set algorithm. *IEEE Transactions on Reliability*, Vol. R-27(No. 4):250–253, October 1978.

[62] A. Rosenthal. Decomposition methods for fault tree analysis. *IEEE Transactions on Reliability*, Vol. R-29(No. 2):136–138, June 1980.

[63] S. L. Salem, G. E. Apostolakis, and D. Okrent. A new methodology for the computer-aided construction of fault trees. *Journal of Nuclear Energy*, Vol. 4:417–433, 1977.

[64] S. N. Semanderes. ELRAFT a computer program for the Efficient Logic Reduction Analysis of Fault Trees. *IEEE Transactions on Nuclear Science*, Vol. NS-18:481–487, February 1971.

[65] M. Shah and A. Ghasemkhani. Preliminary Redundancy/Reliability Analysis Fault Tolerant Electrical Power System (FTEPS). Technical Report Northrop FSCM No. 76823, Northrop Aircraft Division, 1987.

[66] M. L. Shooman. The equivalence of reliability diagrams and fault-tree analysis. *IEEE Transactions on Reliability*, Vol. R-19(No. 2):74–75, May 1970.

[67] K. Stecher. Evaluation of large fault-trees with repeated events using an efficient bottom-up algorithm. *IEEE Transactions on Reliability*, Vol. R-35(No. 1):51–58, April 1986.

[68] J. R. Taylor. An algorithm for fault-tree construction. *IEEE Transactions on Reliability*, Vol. R-31(No. 2):137–146, June 1982.

[69] N. H. Ulerich and G. J. Powers. On-line hazard aversion and fault diagnosis in chemical processes: The digraph and fault tree method. *IEEE Transactions on Reliability*, Vol. R-37(No. 2):171–177, June 1988.

[70] J. Vatn. Finding minimal cut sets in a fault tree. *Reliability Engineering and System Safety*, Vol. 36(No. 1):59–62, 1992.

[71] W. E. Vesely. Reliability and fault tree applications at the NRTS. In *Proceedings 1970 Reliability and Maintainability Conference*, volume Vol. 9, pages 472–480, 1970.

[72] W. E. Vesely and R. E. Narum. PREP and KITT computer code for the automatic evaluation of a fault tree. Technical Report IN-1349, Idaho Nuclear Corporation, Idaho Falls, Idaho, 1970.

[73] W.E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, 1981.

[74] R.C. De Vries. An automated methodology for generating a fault tree. *IEEE Transactions on Reliability*, Vol. 39(No. 1):76–86, April 1990.

[75] J. Wilson. Modularizing and minimizing fault trees. *IEEE Transactions on Reliability*, Vol. R-34(No. 4):320–322, October 1985.

[76] P. H. Winston. *Artificial Intelligence*. Addison-Wesley Publishing Company,Inc., 1984.

[77] R. B. Worrell. Using the set equation transformation system in fault tree analysis. In *Reliability and Fault Tree Analysis*, pages 165–185. Society for Industrial and Applied Mathematics, September 1975.

[78] R. B. Worrell, D. W. Stack, and B. L. Hume. Prime implicants of non-coherent fault trees. *IEEE Transactions on Reliability*, Vol. R-30(No. 2):98–100, June 1981.

[79] J. S. Wu, S. L. Salem, and G. E. Apostolakis. The use of decision tables in the systematic construction of fault trees. In *1977 Nuclear Systems Reliability Engineering and Risk Assessment*, pages 800–824. Society for Industrial and Applied Mathematics, 1977.