1997 *Annual* RELIABILITY AND MAINTAINABILITY *Symposium*

New Results in Fault-Tree Analysis

Joanne Bechta Dugan and Stacy A. Doyle

Joanne Bechta Dugan, *PhD*
Dept. of Electrical Engineering
University of Virginia
Charlottesville, Virginia 22903-2442 USA
Internet (e-mail): jbd@virginia.edu

Summary & Purpose

After a brief review of traditional fault-tree solution methods, this tutorial presents several new & exciting approaches to fault-tree analysis:

- Binary Decision Diagrams (BDD) whose use improves solution-time dramatically for large models. The BDD can allow exact solution of very large systems which previously required approximate solution.
- Techniques for adapting the fault-tree model to the analysis of computer systems, *eg*, incorporation of coverage models into traditional cutset-based and BDD-based solutions. Coverage modeling is critically important for accurate analysis of fault-tolerant systems.
- Several new fault-tree gates which can model the dynamic behavior frequently associated with computer systems.

These new techniques allow the fault-tree model (long appreciated for its concise & unambiguous representational form) to be used to analyze complex fault-tolerant systems.

Joanne Bechta Dugan and Stacy A. Doyle

Joanne Bechta Dugan was awarded the BA (1980) in Mathematics and Computer Science from La Salle University, Philadelphia and the MS (1982) and PhD (1984) in Electrical Engineering from Duke University, Durham. Dr. Dugan is Associate Professor of Electrical Engineering at the University of Virginia, and was previously Associate Professor of Computer Science at Duke University and Visiting Scientist at the Research Triangle Institute. She has performed & directed research on the development & application of techniques for the analysis of computer systems which are designed to tolerate hardware & software faults. Her research interests thus include hardware & software reliability engineering, fault-tolerant computing, and mathematical modeling using dynamic fault trees, Markov models, Petri nets, and simulation. Dr. Dugan is the Senior Associate Editor of the *IEEE Transactions on Reliability*, a Senior Member of the IEEE, and a member of ACM, Eta Kappa Nu, and Phi Beta Kappa.

Stacy A. Doyle received a BA (1988) in Mathematics from The College of the Holy Cross and the MS (1992) and PhD (1995) in Computer Science from Duke University, Durham. She is a technical specialist at Lucent Technologies Advanced Software Construction Center. Her research interests include system-level reliability analysis of advanced fault-tolerant systems, as well as software reliability and testing.

Table of Contents

1 Introduction to Fault Trees

A fault tree model is a graphical representation of logical relationships between events (usually failure events). Fault trees were first developed in the 1960's to facilitate analysis of the Minuteman missile system, and have been supported by a rich body of research since their inception. Initially, a fault tree was defined as a tree (in the graph theoretic sense), but as fault tree analysis techniques evolved, more general connections were permitted. In the current usage, fault tree nodes (gates and basic events) can have more than one parent node, and thus a fault tree is no longer a tree.

Fault tree models have long been used for the qualitative and quantitative analysis of the failure modes of critical systems. A fault tree provides a mathematical and graphical representation of the combinations of events which can lead to system failure. The construction of a fault tree model can provide insight into the system by illuminating potential weaknesses with respect to reliability or safety. A fault tree can help with the diagnosis of failure symptoms by illustrating which combinations of events could lead to the observed failure symptoms. The quantitative analysis of a fault tree is used to determine the probability of system failure, given the probability of occurrence for failure events.

The construction of a fault tree, if performed manually, provides a systematic method for analyzing and documenting the potential causes of system failure. The analyst begins with the failure scenario being considered, and decomposes the failure symptom into its possible causes. Each possible cause is then investigated and further refined until the basic causes of the failure are understood. From a system design perspective, the fault tree analysis provides a logical framework for understanding the ways in which a system can fail, which is often as important as understanding how a system can succeed.

A fault tree consists of the undesired top event (system or subsystem failure) linked to more basic events by logic gates. The top event is resolved into its constituent causes, connected by AND, OR and M-out-of-N logic gates, which are then further resolved until basic events are identified. The basic events represent basic causes for the failure, and represent the limit of resolution of the fault tree. Fault trees do not generally use the *NOT* gate, because the inclusion of inversion may lead to a non-coherent fault tree [13], which complicates analysis. It is quite rare to have need for complementation in a fault tree, so this limitation is acceptable for the analysis of practical systems.

As an example, consider the fault tree shown in figure 1, which provides a simple analysis of a washing machine that overflows. The cause of the overflow can be attributed to one of two causes, either the shutoff valve is stuck open, or the machine stayed in "fill" mode too long. The first cause, failure of the shutoff valve is not considered further, as it is considered a basic event. When the

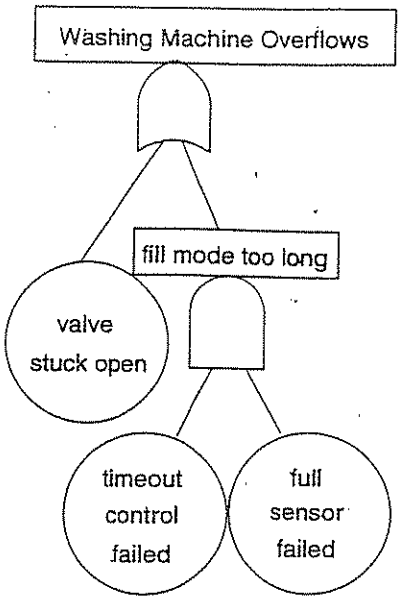


Figure 1: A simple fault tree model

washing machine is filling, either of two events can cause the filling to stop. First, there is a timer, which prevents the machine from filling indefinitely. This timer was designed into the system to help avoid a flood in case the tub leaks. Second, there is a sensor which determines when the tub is full. Both the timer and the sensor must fail for the machine to be unable to stop filling.

1.1 Cutset generation

Analysis of a fault tree begins with an enumeration of the *minimal cutsets*, the minimal sets of component failures which cause system failure. A cutset is a set of basic events whose occurrence causes the top event, system failure. A minimal cutset cannot be a subset of another cutset; that is, if any event is removed from a minimal cutset then it ceases to be a cutset.

A top-down algorithm for determining the cutsets of a fault tree starts at the top event of the tree and constructs the set of cutsets by considering the gates at each lower level. A set of cutsets is expanded at each lower level of the tree until the set of basic events is reached. If the gate being considered is an AND gate, then all the inputs must occur to enable the gate, so a gate is replaced at the lower level by a listing of all its inputs. If the gate being considered is an OR gate, then the cutset being built is split into several cutsets, one containing each input to the OR gate.

Figure 2 shows an example fault tree whose cutset generation is shown in figure 3. The undesired top event occurs when either subevents G2 or G3 occur, which are themselves AND combinations of other subevents or basic events. There are five basic events in the fault tree, labeled A1 through A5, which are all statistically independent. One basic event, A4 can contribute to system

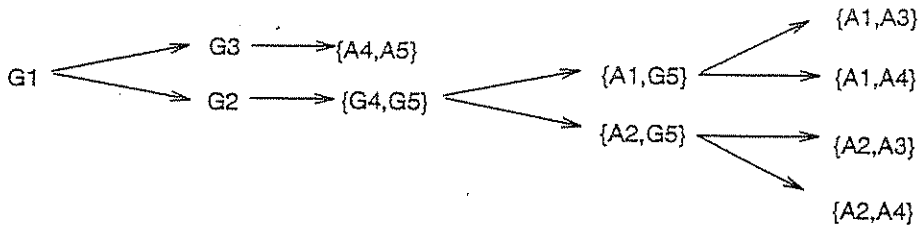


Figure 3: Cutset generation for a fault tree

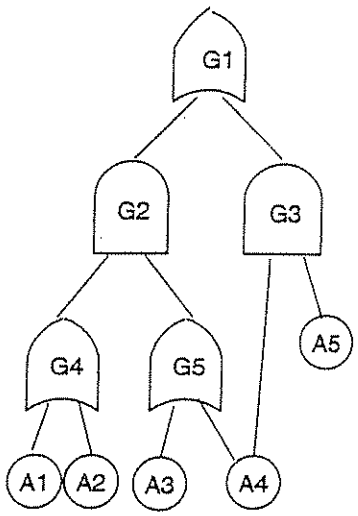


Figure 2: An example fault tree

lure along two paths.

The top-down algorithm starts with the top gate, G1. ıce G1 is an OR gate, it is replaced in the expansion by inputs, G2 and G3. G3 is an AND gate, and is replaced the expansion by the basic events {A4, A5}, a cutset this tree. G2 is expanded into { G4, G5 }, since both ıst occur to activate it. Expanding the G4 term splits : set into two, since it is a two-input OR gate: {A1, } and {A2, G5 }. Finally, the expansion of G5 splits :h sets in two, yielding {A1,A3}, {A1,A4}, {A2,A3}, l {A2,A4}, as the remaining minimal cutsets for the e.

f a gate being expanded is an M-out-of-N gate, then expansion is a combination of the OR and AND ex- ısions. The M-out-of-M gate is expanded into the $\binom{M}{N}$ nbinations of input events that can cause the gate to ur. For example, consider a cutset with a gate Gx t is a 3-out-of-4 gate, with inputs $I1$, $I2$, $I3$ and $I4$. te Gx gets split into four cutsets, replacing Gx with the r possibilities for selecting three of the inputs, namely, $, I2, I3\}, \{I1, I3, I4\}, \{I1, I2, I4\}$ and $\{I2, I3, I4\}$.

When using such an algorithm for generating cutsets, ıe reduction might be necessary. If a cutset contains same basic event more than once, then the redundant ies can be eliminated. If one cutset is a subset of ther, the latter can be removed from further consider- n, since it is not a minimal cutset. For example, the set of cutsets { {A1, A2, A1, A3 }, { A3, A4 }, { A2, A3, A4 } } can be reduced to { { A1, A2, A3 }, { A3, A4 } }.

1.2 Fault tree analysis using cutsets

Qualitative analysis of the fault tree usually consists of studying the minimal cutsets, for example to determine if any single points of failure exist. A single point of failure is any component whose failure (by itself) can cause system failure. Single points of failure are identified by cutsets with only a single element. For the example fault tree of figure 2, all cutsets have cardinality two, so there are no single points of failure. Cutsets can help to identify system hazards which might lead to unsafe or failure states so that appropriate preventive measures can be taken or reactive measures planned. If the top event corresponds to an unsafe condition (rather than system failure), the cutsets can help determine which combinations of events lead to the unsafe condition, and can thus help to identify the need for interlocks. Vulnerabilities resulting from particular component failures can be identified by considering cutsets which contain the component of interest. For the example system, once A4 fails, the system is vulnerable to a failure of either A1, A2 or A5.

Quantitative analysis is used to determine the probability of occurrence of the top event, given the (estimated or measured) probability of occurrence for the basic events. The probability of occurrence for the top event of the tree can be determined from the set of minimal cutsets. The set of cutsets represents all the ways in which the system will fail, and so the probability of system failure is simply the probability that all of the basic events in one or more cutsets will occur.

$$Pr\{System\ \ Failure\} = Pr\{\bigcup_i C_i\}, \qquad (1)$$

where the C_i are the minimal cutsets for the system. Since the cutsets are not generally disjoint, the probability of the union is not equal to the sum of the probabilities of the individual cutsets. If the individual probabilities of the cutsets are simply added together, probability of system failure would be overestimated because the intersection of the events would be counted more than once.

1.2.1 Inclusion-Exclusion

Several methods exist for the evaluation of equation 1 [13, 15], the simplest of which is termed *inclusion-exclusion*. The inclusion-exclusion method is a generalization of the rule for calculating the probability of the union of two events:

$$Pr\{A \cup B\} = Pr\{A\} + Pr\{B\} - Pr\{A \cap B\} \quad (2)$$

and is given by

$$\begin{aligned} Pr\{\bigcup_{i=1}^{n} C_i\} &= \sum_{i=1}^{n} Pr\{C_i\} \\ &- \sum_{i<j} Pr\{C_i \bigcap C_j\} \\ &+ \sum_{i<j<k} Pr\{C_i \bigcap C_j \bigcap C_k\} \\ &\mp \cdots \\ &\pm Pr\{\bigcap_{i=1}^{n} C_i\} \end{aligned} \quad (3)$$

That is, the sum of the probabilities of the cutsets taken one at a time, minus the sum of the probabilities of the intersection of the cutsets taken two at a time, plus the sum of the probabilities of the intersection of cutsets taken three at a time, etc.

Equation 3 calculates the probability of system failure exactly. As each successive summation term is calculated and added to the running sum, the result alternatively overestimates (if the term is added) or underestimates (if the term is subtracted) the desired probability. Thus, bounds on the probability of system failure can be determined by using only a portion of the terms in Equation 3.

1.2.2 Sum of Disjoint Products

A computationally more efficient approach uses the properties of Boolean Algebra to evaluate equation 1. If the p minimal cut sets are labeled $C_i, i = 1, \ldots, p$, then system unreliability is $Prob[\bigcup_{i=1}^{p} C_i]$. A *sum-of-disjoint-products* (SDP) method for fault-tree evaluation uses the equation:

$$\begin{aligned} \bigcup_{i=1}^{p} C_i = (C_1) \bigcup (\overline{C}_1 C_2) \bigcup (\overline{C}_1 \overline{C}_2 C_3) \\ \bigcup \cdots \bigcup (\overline{C}_1 \overline{C}_2 \overline{C}_3 \cdots \overline{C}_{p-1} C_p), \end{aligned} \quad (4)$$

where $\overline{C_i}$ is that part of the universal set that is not in C_i, that is, the negation of C_i. Since the terms in the right-hand side of Equation 4 are mutually disjoint, the sum of the probabilities of the individual terms yields the exact unreliability of the system.

The basic approach to an SDP algorithm is to take each cut set and make it disjoint with each preceding cut set, using Boolean algebra. The best current SDP algorithm is called GKG-VT [16].

The SDP algorithm can be easily truncated. If the cut sets are sorted in decreasing order of their probability of occurrence, the larger contributions to system unreliability are associated with the cut sets with lower indices. Making the cut sets with the highest indices disjoint from the others that precede it can require a substantial time investment that does not contribute much in terms of accuracy. Suppose that the first ℓ ($\ell < p$) cut sets have been made disjoint from the preceding ones. That is, the $\overline{C}_1 C_2$, $\overline{C}_1 \overline{C}_2 C_3, \ldots, \overline{C}_1 \overline{C}_2 \overline{C}_3 \cdots \overline{C}_{\ell-1} C_\ell$ terms have all been determined. Then these terms, along with the remaining cut sets (indexed from $\ell + 1$ to p), can be used to provide bounds on the system unreliability:

$$\begin{aligned} Prob[C_1] + Prob[\overline{C}_1 C_2] + \cdots + Prob[\overline{C}_1 \overline{C}_2 \ldots \overline{C}_{\ell-1} C_\ell] \\ \leq Unreliability \leq \\ Prob[C_1] + Prob[\overline{C}_1 C_2] + \cdots + Prob[\overline{C}_1 \overline{C}_2 \ldots \overline{C}_{\ell-1} C_\ell] \\ + Prob[C_{\ell+1}] + Prob[C_{\ell+2}] + \cdots + Prob[C_p]. \end{aligned}$$

If the bounds are tight enough after making cut set ℓ disjoint, the process of making the remaining cut sets disjoint can be suspended. If the bounds from the ℓ disjoint cut sets are too loose, the procedure can continue with the $\ell + 1$st cut set. In fact, one can determine ℓ *a priori*, if the maximum acceptable width of the error interval is known (say, 10^{-d}). Determine a sum of the cut set probabilities that is less than $5 \times 10^{-(d+1)}$. For example, find the smallest ℓ such that

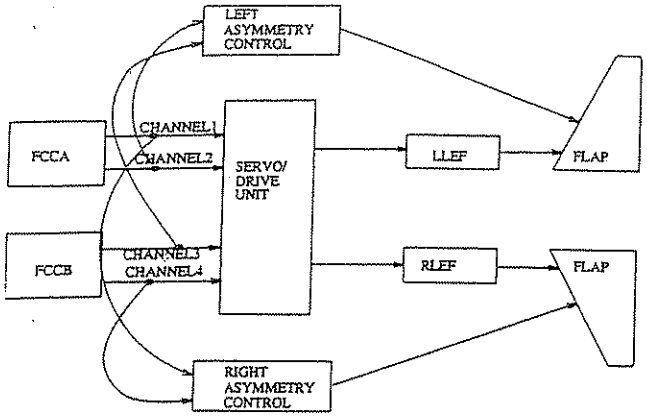
$$\left(\sum_{i=\ell+1}^{p} Prob[C_i] \right) \leq 5 \times 10^{-(d+1)} \quad (5)$$

The smallest ℓ value that satisfies equation 5 determines how many cut sets (the first ℓ) must be made disjoint from the preceding ones to achieve the desired accuracy.

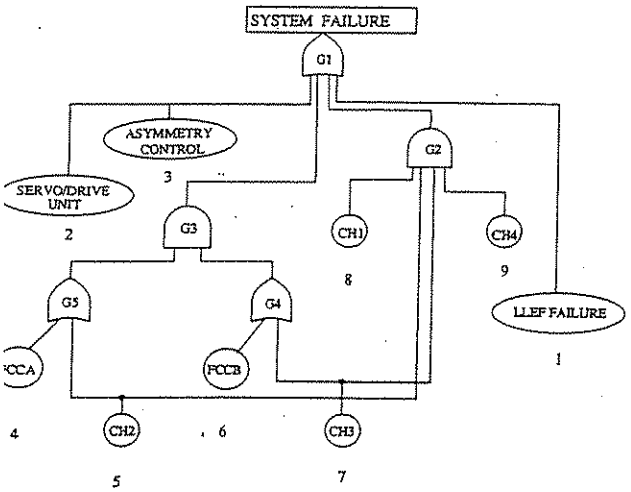
2 Fault tree analysis using Binary Decision Diagrams

A new alternative to the traditional cutset-based solution approach for combinatorial models uses the binary decision diagram (BDD) [2]. The BDD has primarily been used as a verification technique in circuit theory, but has recently been adapted to solve a fault tree model for both quantitative and qualitative reliability analysis [5]. Generally, a BDD is a "diagram" which tells the user how to determine the output value of the function by examining the values of the inputs [1]. The BDD as constructed by Bryant[2, 3] has no duplicate subtrees or redundant vertices. The algorithms developed by Bryant for manipulating BDDs have time complexity proportional to the sizes of the graphs being operated on, and hence are quite efficient as long as the graphs do not grow too large.

By traversing a BDD representation of the structure function, it is possible to quickly calculate the system unreliability without explicitly using cutsets. The biggest



igure 4: Schematic diagram of F18 flight control system EF



ure 5: Fault tree model of F18 flight control system EF

wback of BDDs is that the size of the graph depends vily on the input variable ordering used to build the D. A bad ordering can result in the same exponen- problem as existed originally with the combinatorial del. A good ordering will result in fewer indices in the D. Heuristics can be used to find a reasonable variable ering [2].

Determining the BDD representation of a fault tree

ile it is possible to derive the BDD for a system from combinatorial model, this work focuses only on the t tree to BDD conversion. In order to demonstrate conversion process, we present a small example sys-, the leading edge flap (LEF) for the F18 flight control em (FCS).

he leading edge flaps of the F18 flight control system in takeoff and landing. Figure 4 contains the LEF matic diagram for both leading edge flaps.

fault tree model of the left LEF (LLEF) is seen in Figure 5. In this simple example, it is easy to see the combinations of events which lead to system failure, the top node of the tree. The fault tree has a unique number associated with each of the system's components. These numbers are the indices assigned to the components and determine the ordering of the vertices representing the components in the BDD.

To construct a BDD from the fault tree, a depth-first traversal of the tree is done and the BDD is constructed from the bottom up [4]. The BDD of the LLEF is shown in Figure 6. Sub-BDDs are created (they are actually BDDs themselves) and combined based on the logic operation of the current gate. In other words, once BDDs are created for all the inputs of a given gate of the fault tree, the logic operation of the gate (AND/OR) is applied to all the input BDDs using Bryant's APPLY operation [3].

To APPLY a logic operation to two BDDs, the BDDs are traversed in a depth-first fashion and a new BDD, representing the combination of the 2 original BDDs, is produced. Assume there are two BDDs, f and g, and the logic operation to be applied is $< op >$. The new BDD represents the result of $f < op > g$. Let f and g represent the current vertices of the respective BDDs. If f and g have the same index, the current vertex, initially the root, of the new BDD will get the value of f (or g since they are the same). The value pointed to by the left branch will be the result of $left(f) < op > left(g)$; the value pointed to by the right branch will be the result of $right(f) < op > right(g)$. This continues until terminal vertices are reached, or the indices of the two vertices do not match. Alternatively, consider the case where the index of f is less than the index of g. The current vertex of the new BDD will get the value of f, the left branch will have the result of $left(f) < op > g$, and the right branch will have the result of $right(f) < op > g$. The reverse would be true if g's index was less than that of f.

There are certain simplifying conditions which hold for the case of terminal vertices. For example, suppose the logic operation is OR and the current vertex of one of the BDDs is a terminal vertex. By the rules of Boolean algebra, $1 + x = 1$ and $0 + x = x$. In other words, if the terminal vertex is 1, the vertex of the new BDD gets a value of 1. If the terminal vertex is 0, the vertex of the new BDD gets the value of the subtree rooted at x. In either case, the traversal of the subtree of one BDD is unnecessary, since the value of the other BDD will not change. Similar conditions hold for the AND logic operation. Here, $0 \cdot x = 0$ and $1 \cdot x = x$.

A portion of the LLEF BDD construction (figure 6) will be presented in detail. The remainder is built using the same process. To aid in the understanding of the construction, the indices are included with the vertices in the BDDs.

Consider the subtree rooted at the AND gate G3; the first path traversed leads to node FCCA. This means OR gate G5 will be applied once BDDs are built for all G5

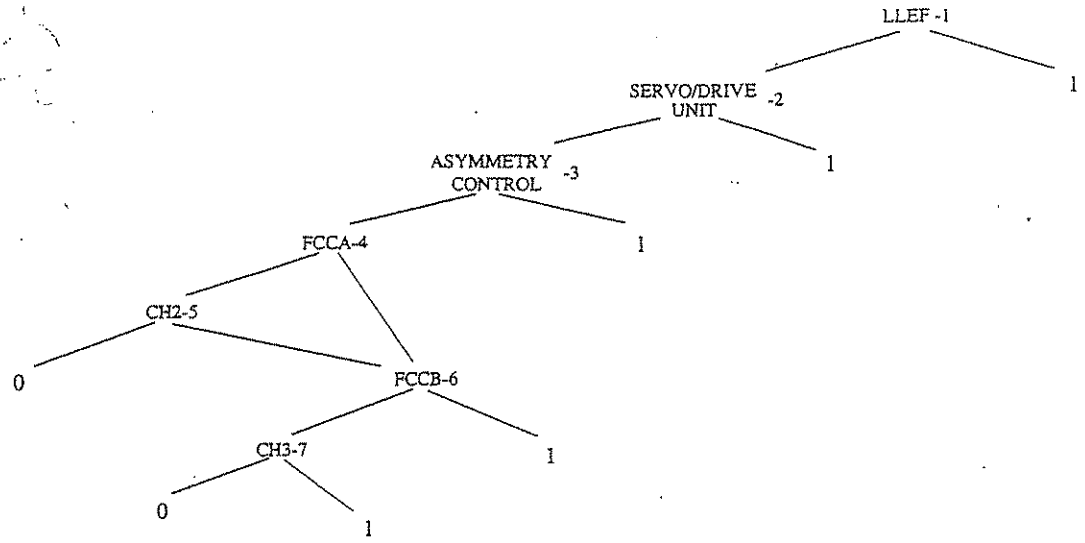


Figure 6: BDD of the LLEF

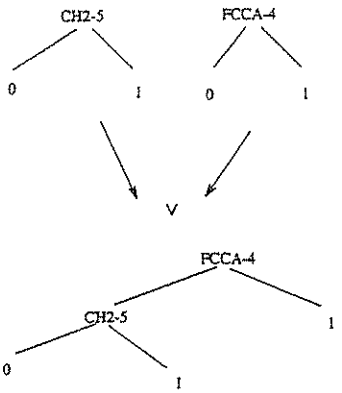


Figure 7: BDD construction up to gate G5 of LLEF fault tree

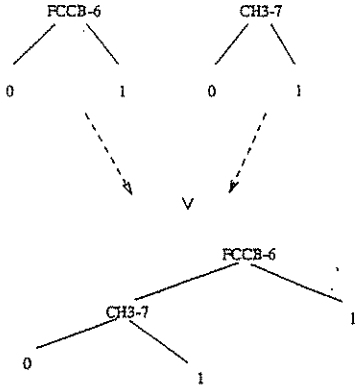


Figure 8: BDD construction up to gate G4 of LLEF fault tree

inputs. Figure 7 shows the initial BDDs for basic events FCCA and CH2. The BDD representing an individual basic event consists of a vertex and two branches, left and right, pointing to terminal vertices 0 and 1, respectively. The left branch represents the event in which the component (represented by the vertex) does not fail. The right branch represents the failure of the component represented by the vertex. If terminal vertex 1 is reached, the system is in a state of failure. If terminal vertex 0 is reached, the system is still operational; i.e., the events which have occurred have not caused the system to fail.

The BDD resulting from the application of OR gate G5 is also shown in Figure 7. Node FCCA is the root of the BDD since it has a lower index than CH2. This graph demonstrates that if FCCA fails (the right branch is taken), the "system" fails. System operation in this case requires that both FCCA and CH2 function correctly. If FCCA does not fail (the left branch is taken), the system state depends on the state of CH2. If CH2 fails (right branch taken), the system fails; if CH2 does not fail (left branch taken), the system does not fail. The state of the system is determined from the value of the terminal vertex which is reached by taking a given path.

Figure 8 shows the BDDs for components FCCB and CH3 and the BDD resulting from the application of gate G4. This is virtually identical to the application of gate G5.

Figure 9 shows the BDD resulting from the application of AND gate G3 on the BDDs which represent G3 inputs. The boxed section of the BDD is redundant, so it is removed and the branch which points to it is redirected to the other instance of the same subtree (Figure 10).

The conversion of the fault tree to an equivalent BDD continues in a similar manner. Figure 6 shows the full BDD representing the LLEF fault tree, based on the given ordering of the components.

2.2 Alternative variable ordering

To demonstrate the negative impact a bad ordering can potentially have on the BDD construction, assume the in-

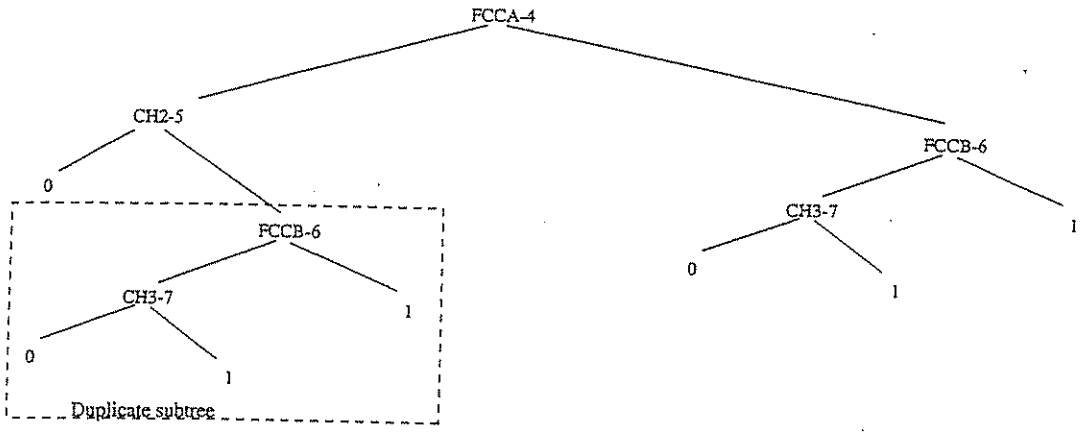


Figure 9: BDD construction up to gate G3 of LLEF fault tree

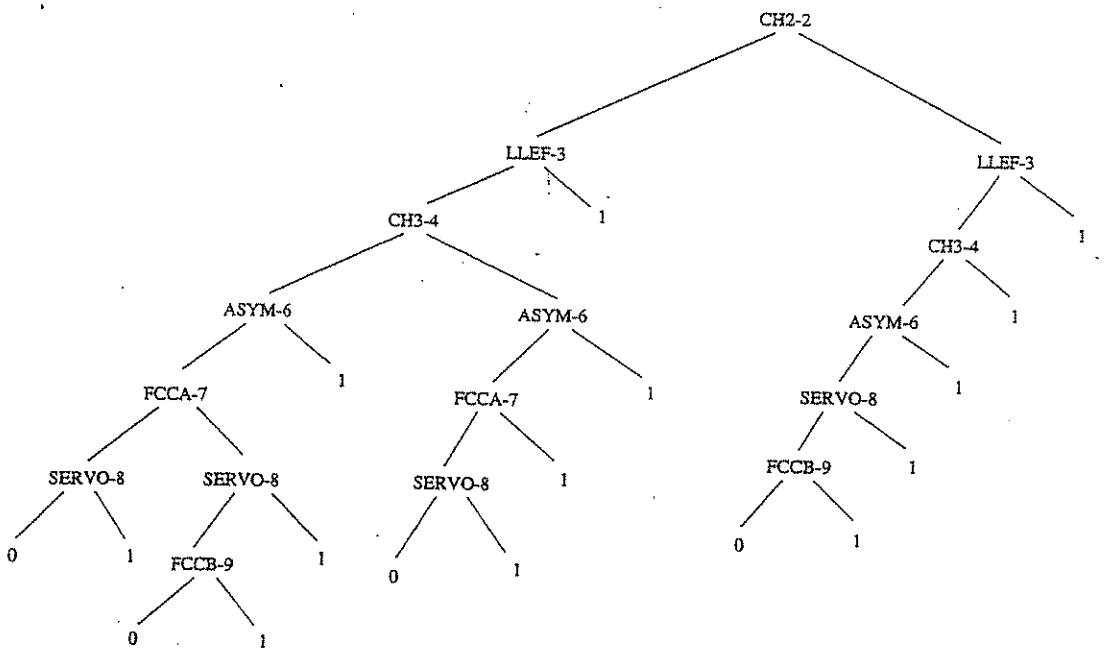


Figure 11: LLEF BDD with alternate input variable ordering

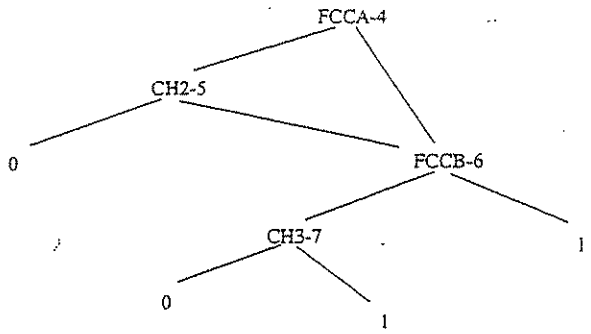


Figure 10: Reduced BDD representing subtree rooted at gate G3

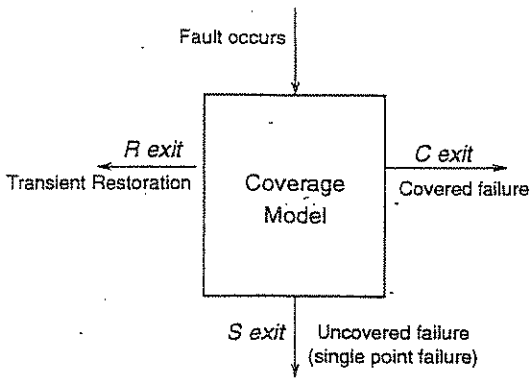


Figure 12: General structure of a coverage model

put variable for the LLEF system is as follows: $CH1-1$, $CH2-2$, $LEF-3$, $CH3-4$, $CH4-5$, $ASYM-6$, $FCCA-7$, $SERVO-8$, and $FCCB-9$. The BDD result from this ordering is shown in Figure 11. This BDD contains 16 non-leaf nodes as opposed to the original BDD (Figure 6) which contains 7 non-leaf nodes. Because the LLEF system is small, the bad ordering will not significantly affect the reliability analysis solution time. If a bad input variable ordering is used for a larger system, the analysis time could be drastically affected.

2.3 Calculating unreliability from a BDD

Each non-leaf node in the BDD represents a component which can fail, and each path from the root to a leaf node represents a disjoint combination of component failures and non-failures. If a path leads from a node to its left branch, then the non-failure of the component is considered for that path. If the path leads from a node to its right branch, then the failure of the component is considered for that path. If the leaf node for a path is labeled with a "1" then the path leads to system failure; if the leaf is labeled with a "0" then the path represents an operational system configuration. The probabilities associated with the arcs on each path are either P_{node} (the probability of node failure) for the right branch or $Q_{node} = (1 - P_{node})$ for the left branch. In Figure 6, the probability of taking the path $LLEF \rightarrow S \rightarrow 1$ is $Q_{LLEF}P_S$. The unreliability of the system is given by the sum of the probabilities for all the paths from the root to a leaf node labeled "1". Thus the symbolic representation of the probability of system failure for the LLEF example is given by

$$
\begin{aligned}
&P_{LLEF} + Q_{LLEF}P_S \\
&+Q_{LLEF}Q_SP_A \\
&+Q_{LLEF}Q_SQ_AP_{FCCA}P_{FCCB} \\
&+Q_{LLEF}Q_SQ_AP_{FCCA}Q_{FCCB}P_{CH3} \\
&+Q_{LLEF}Q_SQ_AQ_{FCCA}P_{CH2}P_{FCCB} \\
&+Q_{LLEF}Q_SQ_AQ_{FCCA}P_{CH2}Q_{FCCB}P_{CH3}
\end{aligned}
$$

3 Modeling fault-tolerant computer systems

Because computer systems are easily programmed and can thus be designed to actively handle a variety of complex scenarios, they are finding increased usage in embedded control systems. In addition to providing an increased level of flexibility for reactive systems, an embedded computer can also provide a measure of adaptive fault tolerance, switching out failed components and switching in spares, for example. This increased functionality does not come without cost, however, as the failure of the computer system can lead to the failure of the entire application, even if adequate functioning resources remain. The introduction of computer systems into embedded systems introduces additional failure modes to consider when analyzing reliability. That is, the analysis must allow multiple failure modes, rather than the binary designation of "operational" or "failed." (Some mechanical and electrical systems also exhibit multiple failure modes.) In addition, the effects of one of the failure modes (a *covered* failure) are local to the affected component, while the effects of the other failure mode (an *uncovered* failure) are globally malicious, causing immediate system failure.

A coverage model is used to analyze the behavior of the computer system in the presence of faults, and provides an estimate of the relative probability of an uncovered *vs.* a covered component failure (given that a fault has occurred). A fault does not necessarily lead to a component failure, as the computer system may recognize and handle a transient fault and continue operating as if the fault had not occurred. In this section we discuss methods for including coverage modeling into a fault tree analysis of a system. More detail on coverage modeling and incorporating coverage models into Markov models appears in [9, 10].

3.1 General structure of coverage model

The general structure of a model that represents the recovery process which is initiated when a fault occurs is

own in figure 12. The entry point to the model signi-
s the occurrence of the fault, and the three exits sig-
fy three possible outcomes. The transient restoration
it (labeled R) represents the correct recognition of and
covery from a transient fault. A transient is usually
used by external or environmental factors, such as ex-
ssive heat or a "glitch" in the power line. It is gener-
y believed that the vast majority of faults are transient.
ccessful recovery from a transient fault restores the sys-
n to a consistent state without discarding any compo-
nts, for example by retrying an instruction or rolling
ck to a previous checkpoint. Reaching this exit suc-
sfully requires timely detection of an error produced
the fault, performance of an effective recovery proce-
re, and the swift disappearance of the fault (the cause
the error).

The permanent coverage exit (labeled C) denotes the
ermination of the permanent nature of the fault, and
successful isolation and removal of the faulty compo-
it. The single point failure exit (labeled S) is reached
en a single fault causes the system to crash. This gen-
lly occurs if an undetected error propagates through
system, or if the faulty unit cannot be isolated and
s the system cannot be reconfigured.

t is here that we can differentiate between a fault and
ailure in a single component. If a fault affects a com-
ient, the component does not necessarily fail, since the
lt may be transient. A component failure occurs if the
nsient restoration is unsuccessful, and thus the tran-
it restoration exit is not reached. If the permanent
erage exit is reached instead, then a *covered compo-
t failure* is said to occur. If the single-point failure
is reached, then an *uncovered component failure* oc-
s. A covered component failure may or may not lead
ystem failure depending on the remaining redundancy
he system.

Example coverage modeling

rder to illustrate the inclusion of coverage in the anal-
of fault tolerant systems, consider a simple computer
em (Called 3P2M) consisting of three processors and
shared memories communicating over a shared bus
ure 13). The 3P2M system is operational as long as
east two processors can communicate with at least one
he memories.

he fault tree model of the 3P2M system is shown in
ure 14. It shows that the 3P2M system fails when
of the three processors fail, when both memories fail
when the bus fails. This fault tree model does not
ct coverage failures, which are analyzed separately.
numbers which appear under the nodes of the fault
represent the variable ordering which will be used to
struct the BDD.

he processors in the 3P2M system contain built-in
circuitry so that error checking occurs concurrently
instruction execution. If an error is detected, the in-

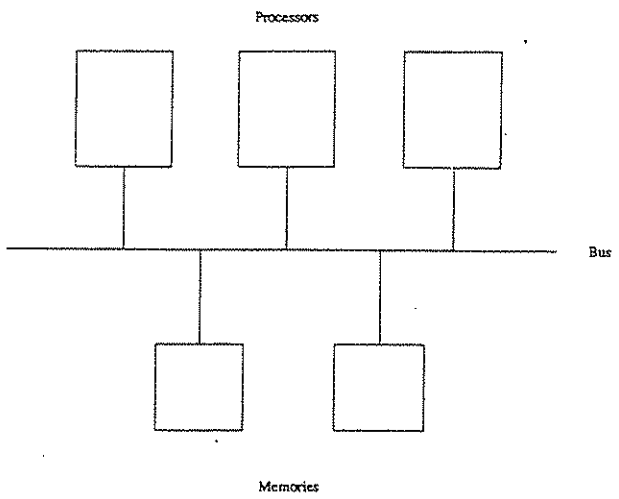


Figure 13: The 3P2M example system

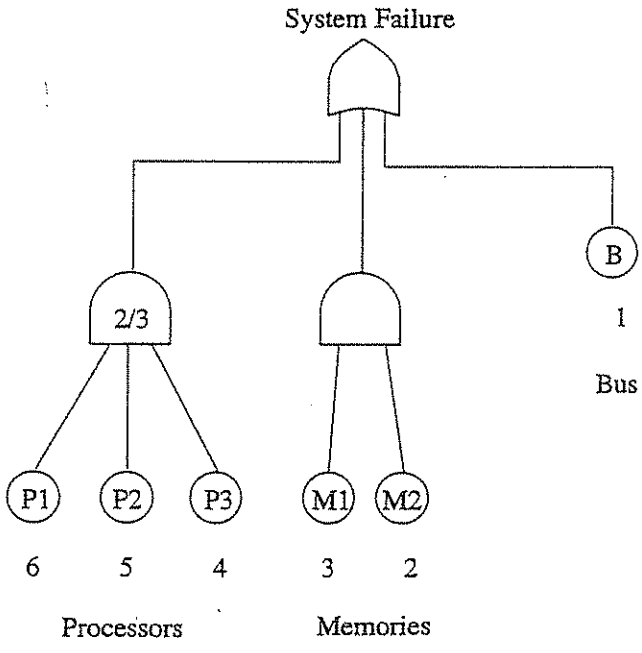


Figure 14: Fault tree model of 3P2M system

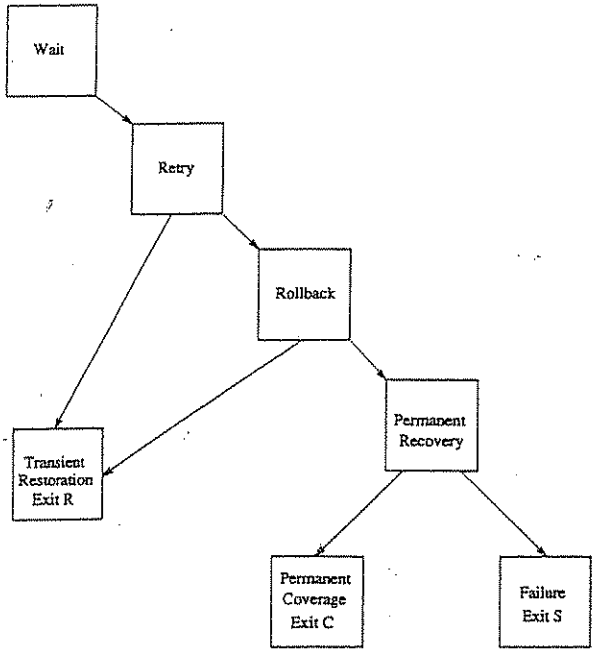


Figure 15: Coverage model for processor subsystem

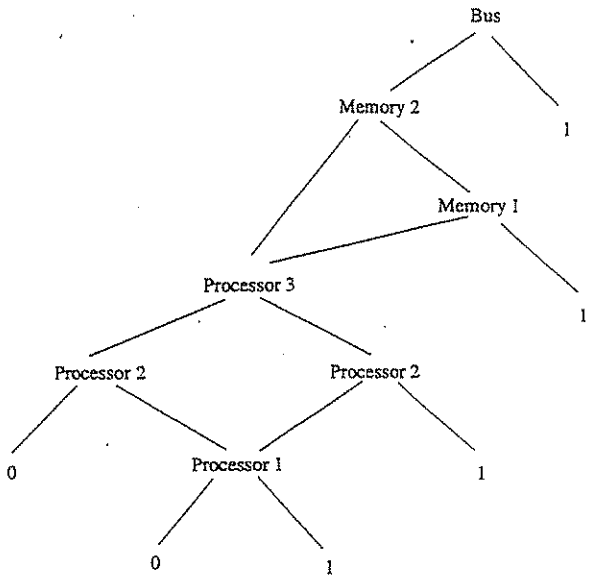


Figure 16: BDD of 3P2M fault tree

struction is retried immediately. Partial results are stored in case the retry is unsuccessful, so that the computation can be continued from some intermediate point (called a checkpoint). The process of continuing a computation from a previously saved checkpoint is called a rollback. In some cases the fault is such that the rollback is not successful, so the computation must start over after a system-level recovery procedure is invoked. An example of a processor fault coverage model is shown in Figure 15.

A BDD representation of the 3P2M fault tree is shown in Figure 16, using the variable ordering noted in Figure 14.

A recovery procedure for the memory subsystem of the 3P2M system is shown in Figure 17. The memory uses an error correcting code, so a single-bit error is always detectable and correctable, and no reconfiguration is required. Thus the R exit of the coverage model will be reached.

When a multiple memory error is detected, the affected portion of memory is discarded, the memory mapping function is updated, and the needed information is reloaded from a previous checkpoint and updated to represent the current state of the system. When the system successfully recovers from the detected multiple memory error, the C exit is reached.

There are two paths to the single point failure exit. A memory fault causes a single-point failure if a multiple-bit error is not detected. If a multiple-bit memory error is detected, but the attempted recovery is not successful, the S exit is also taken.

More details on the solution of these coverage models is in [14], chapter 9.

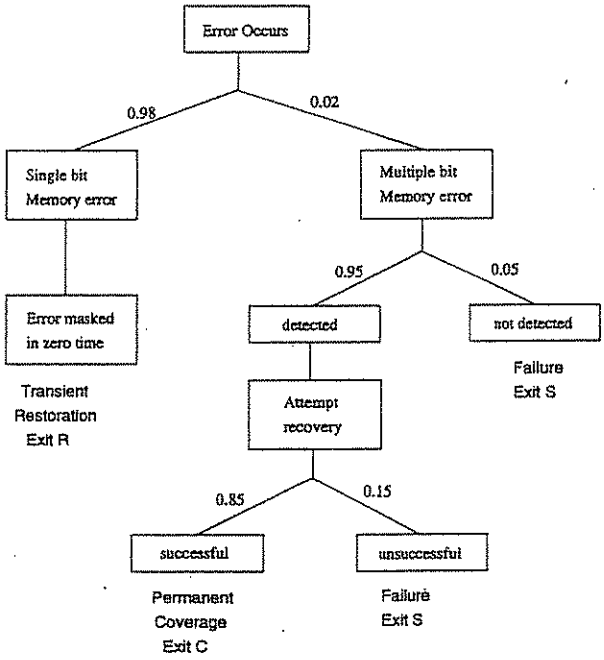


Figure 17: Coverage model for memory subsystem

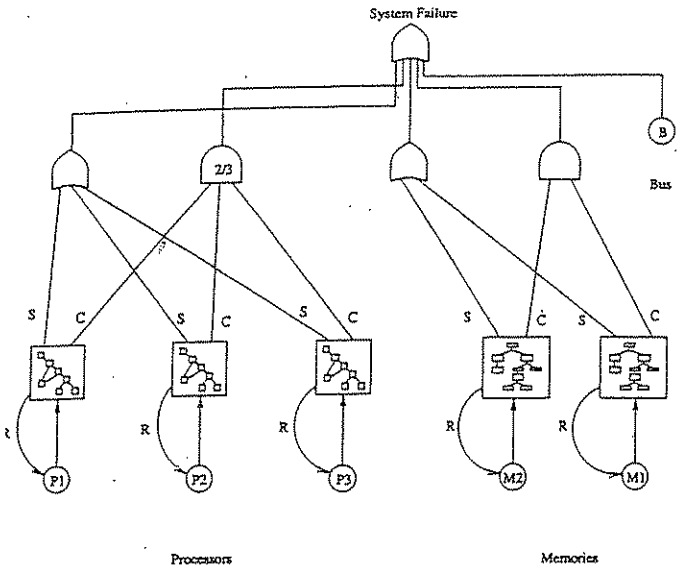


Figure 18: Adding coverage models to fault tree model of 3P2M system

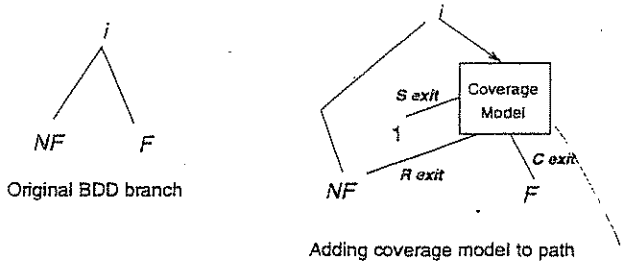


Figure 19: Adding coverage to BDD for node i

3.3 Combining system structure and coverage models

3.3.1 Cutset-based solution with coverage

The fault tree shown in Figure 14 does not consider coverage. Rather, the associated coverage models are automatically included in the fault tree solution. In this section we will describe pictorially how imperfect coverage impacts a cutset-based solution of a fault tree model.

In figure 18 we explicitly show the incorporation of coverage models into the fault tree. For each component which can experience coverage failures (processors and memories), there is a coverage model associated with the basic event. When the basic event occurs (i.e. the component experiences a fault), the coverage model is entered. The solution of the coverage model tells (probabilistically) which exit (R, C or S) is reached. If the R exit is reached, then the component has not failed, so the R exit points back to the basic event. What this really means is that it behaves as if the basic event has not occurred.

If, on the other hand, the C exit is taken, then we say that a covered failure has occurred, and this covered failure combines with other covered failures to possibly lead to system failure. The covered failures are the failures which are normally included in a fault tree model if coverage is ignored.

An uncovered failure leads to immediate system failure, regardless of the state of other components in the system. Thus each of the S exits lead directly to the top event in the fault tree.

Coverage models are included in the fault tree in a systematic manner, as can be seen by comparing figures 14 and 18. All S exits connect through OR gates to the top event. All R exits lead back to the basic event, all C exits maintain the original connection. Adding the coverage models to the fault tree diagram manually can needlessly complicate the diagram, so we add the coverage models automatically to the solution and do not require them to be shown in the fault tree itself.

The coverage models are added to the fault tree solution by augmenting the set of minimal cutsets with a set of singleton cutsets. Each singleton cutset represents the uncovered failure of a component which can experience such a failure. The rest of the solution uses a normal cutset-based approach, with the addition of considering the mutual exclusivity of covered and uncovered failures. This algorithm is described in detail in [8].

3.3.2 BDD solution with coverage

Coverage models can also be automatically included in a BDD solution of a fault tree. Given the BDD representation of a fault tree which does not include coverage modeling, coverage models are added during the traversal of the BDD to calculate unreliability. Details of this calculation appear in [6, 7].

As the BDD is traversed, when a node i is reached that can experience an uncovered failure, a coverage model is inserted on the path. Call NF the node to which i points (in the original BDD) for non-failure of i, and call F the node to which node i points for failure. (See figure 19.) When the coverage model is added for node i, both the original "not failed" branch and the R (transient restoration) exit from the coverage model point to NF. The "failed" branch is split into "failed covered" (pointing to F) and "failed uncovered". The "failed uncovered" branch points to terminal node "1" because an uncovered failure leads to immediate system failure.

In addition to adding coverage models for nodes on a path, additional nodes may need to be added to the BDD. When a BDD is constructed, the variables (nodes, basic events) are ordered. When the BDD is traversed, the nodes on a path appear in increasing order. We use this ordering to determine when additional nodes need to be inserted into the BDD. For each path, whenever a node is skipped (in relation to the ordering) and that node can experience an uncovered failure, the skipped node, and its corresponding coverage model, is inserted into the BDD. Suppose for example that a path leads from $node(i)$ to

$node(i+2)$, and that $node(i+1)$ can experience an uncovered failure. $Node(i+1)$was skipped in this path of the BDD because its (covered) failure did not contribute to system unreliability for that path. $Node(i+1)$ is added to the BDD such that the $node(i)$-branch that originally lead to $node(i+2)$now leads to $node(i+1)$. The newly-inserted $node(i+1)$ has branches for non-failure and failure. The non-failure branch points to $node(i+2)$. The failure branch points to the coverage model for $node(i+1)$. The coverage model has three exits, for uncovered failure, covered failure and transient restoration. The S exit (uncovered failure) leads to a leaf node labeled with "1"; the C exit and R exit both point to $node(i+2)$.

Thus, coverage can be added to the BDD by adding a coverage model for each node that can produce an uncovered failure, and adding extra nodes to the BDD when an uncovered node is skipped. (Including coverage does not actually alter the BDD but rather alters the unreliability calculation based on the original BDD. For this example we alter the BDD for illustrative purposes.) For the 3P2M example, the resulting BDD (including coverage) is shown in Figure 20. The lightly shaded regions show the nodes which needed to be added to the given paths in order to account for their uncovered failures.

4 Modeling dynamic system behavior

4.1 Dynamic fault-tree gates

A major disadvantage of traditional fault tree analysis is the inability of standard fault tree models to capture sequence dependencies in the system and still allow an analytic solution. As an example of a sequence dependent failure, consider a system with one active component and one standby spare connected with a switch controller [13]. If the switch controller fails after the active unit fails (and thus the standby is already in use), then the system can continue operation. However, if the switch controller fails before the active unit fails, then the standby unit cannot be switched into active operation and the system fails when the active unit fails. Thus, the failure criteria depend not only on the combinations of events, but also on the sequence in which events occur.

Systems with various sequence dependencies are usually modeled with Markov models. If, instead of using standard fault tree solution methods, the fault tree is converted to a Markov chain for solution, the expressive power of a fault tree can be expanded by allowing certain kinds of sequence dependencies to be modeled[9]. There are several different kinds of sequence dependencies in fault tolerant systems. This section identifies several different types of sequence dependencies, and defines specific gates to express these behaviors in fault tree models. Subsequent sections demonstrate the use of these gate types in several examples.

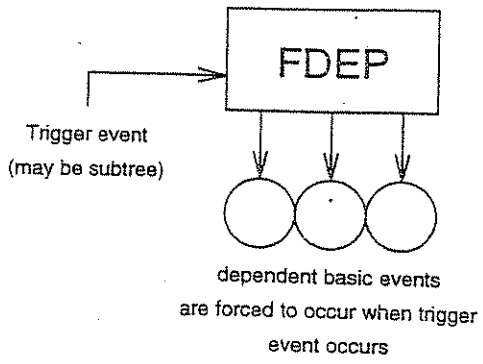


Figure 21: Functional dependency gate

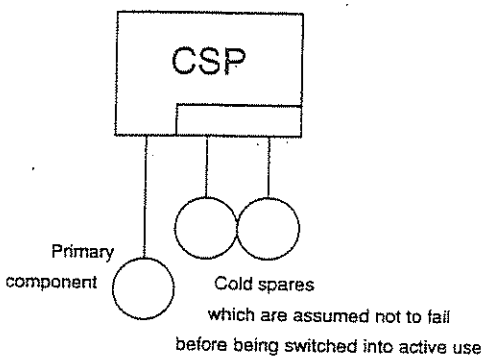


Figure 22: Cold spare gate

4.1.1 Functional dependency gate

Suppose that a system is configured such that the occurrence of some event (call it a *trigger* event) causes other dependent components to become inaccessible or unusable. A *functional dependency* gate (see figure 21) has a single trigger input (either a basic event or the output of another gate in the tree), a non-dependent output (reflecting the status of the trigger event) and one or more dependent basic events. The dependent basic events are functionally dependent on the trigger event. When the trigger event occurs, the dependent basic events are forced to occur. In the Markov chain generation, when a state is generated in which the trigger event is satisfied, all the associated dependent events are marked as having occurred. The separate occurrence of any of the dependent basic events has no effect on the trigger event.

The functional dependency gate is useful, for example, when communication is achieved through some network interface elements, where the failure of the network element isolates the connected components. In this case, the failure of the network element is the trigger event and the connected components are the dependent events.

4.1.2 Cold spare gate

Consider a system that utilizes cold spares (spare components that are unpowered, and thus do not fail before being used). Such systems cannot be modeled exactly using standard fault tree techniques because the system

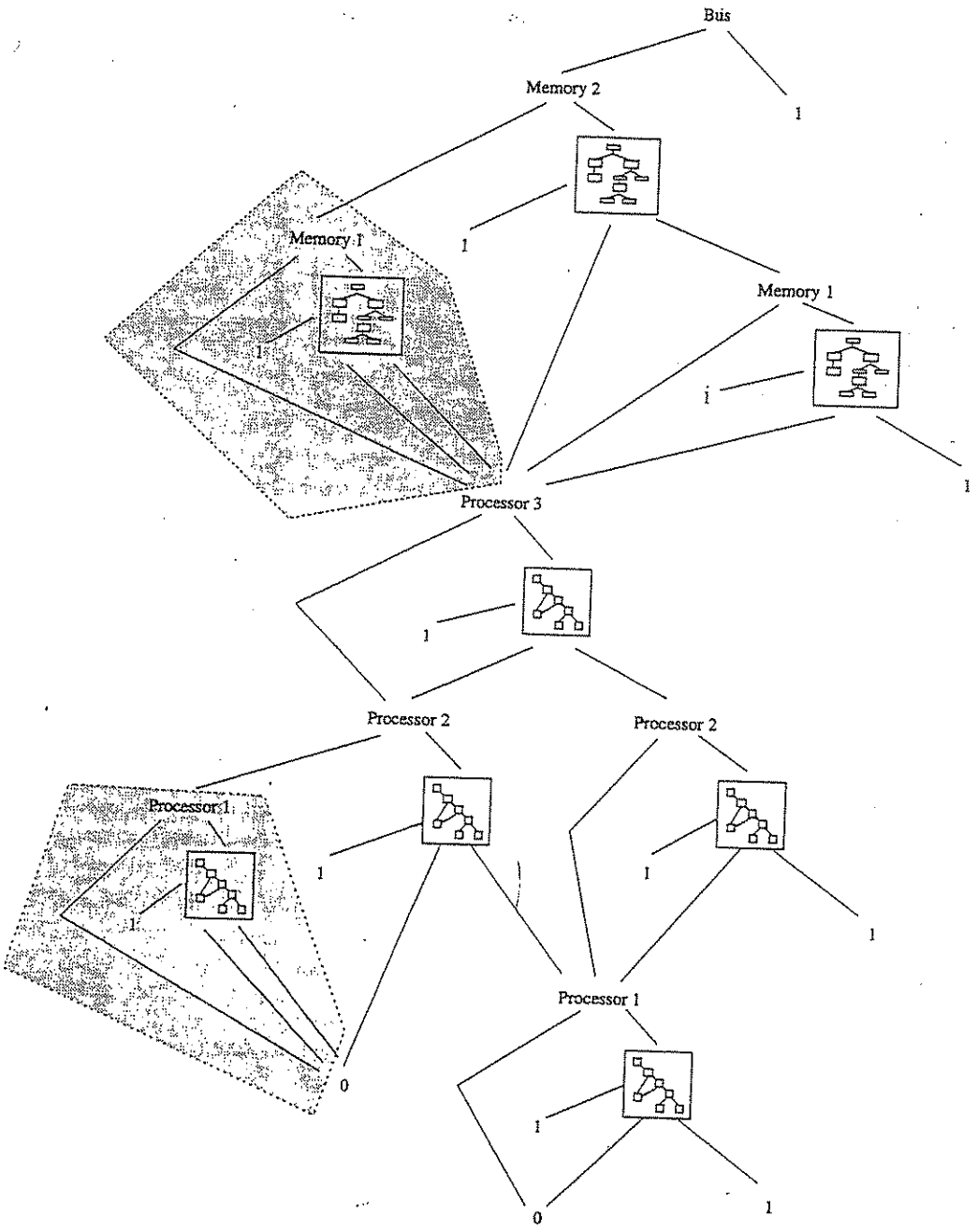


Figure 20: Adding coverage models to BDD for 3P2M example. The shaded areas represent added nodes.

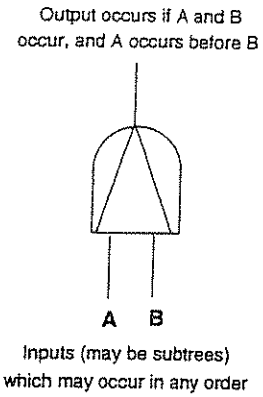


Figure 23: Priority-AND gate

failure criteria cannot be expressed in terms of combinations of basic events, all using the same time frame.

We address this fault tree deficiency by introducing a *cold spare* (CSP) gate (see figure 22), with one primary input and one or more alternate inputs. All inputs are basic events. The primary input is the one that is initially powered on, and the alternate input(s) specify the (initially unpowered) components that are used as replacements for the primary unit. The cold spare gate has one output which becomes true after all the input events occur.

The cold spare gate can be used if spare units are shared between active processors, in which case the basic event representing the cold spare inputs to more than one CSP gate. In this situation, the spare is only available to one of the CSP gates, depending on which of the primary units fails first. The FTPP example will illustrate the use of the CSP gate when one spare unit is available for three active processors.

The conversion of the fault tree to a Markov chain makes the consideration of cold spares possible. In a state where the primary unit is operational, the cold spares are not permitted to fail. However, once the primary unit has failed, then the first alternate unit can fail. After the first alternate fails, the remaining alternates are allowed to fail, one at a time in the order specified, until the spares are exhausted. The possibility of being unable to reconfigure correctly the spare unit into operation is captured in the (separately specified) coverage model.

The functional dependency gate and the cold spare gate can interact in an interesting way. Suppose that the spare units are functionally dependent on some other (otherwise unrelated) component. The occurrence of the trigger event can render one or more of the spares unusable, even if they have not been switched into active operation yet. Then, if the primary unit fails, the spares are unavailable to replace it. This is the only case where a spare can "fail" even while it is unpowered.

4.1.3 Priority-AND gate

The Priority-AND gate is logically equivalent to an AND gate, with the added condition that the events must occur

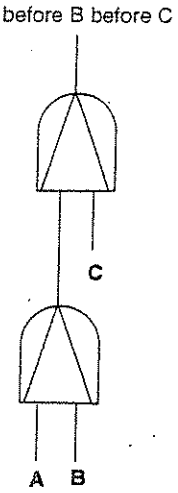


Figure 24: Cascading priority-AND gates

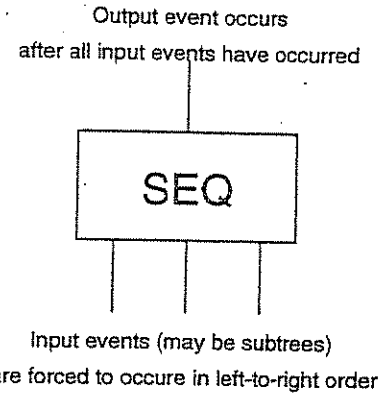


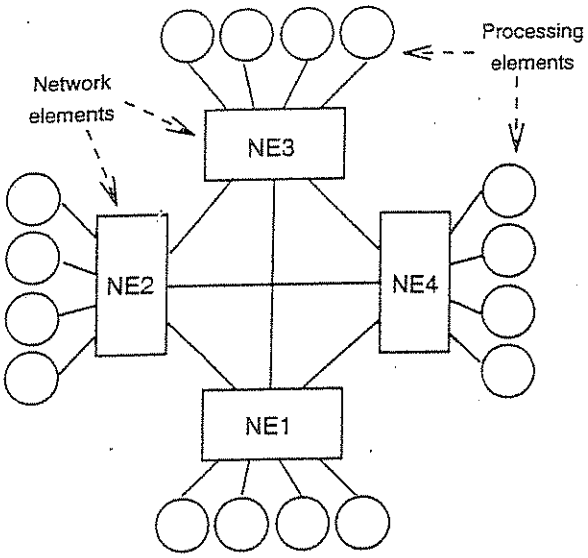
Figure 25: Sequence enforcing gate

in a specific order. The Priority-AND gate (as shown in figure 23) has two inputs, A and B. The output of the gate is true if both A and B have occurred, and if A occurred *before* B. If both events have not occurred, or if B occurred before A then the gate does not fire. To represent the behavior that A occurs before B which occurs before C, the Priority-AND gates can be cascaded as shown in figure 24.

4.1.4 Sequence enforcing gate

The *sequence enforcing gate* forces events to occur in a particular order. The input events are constrained to occur in the left-to-right order in which they appear under the gate (i.e., the leftmost event must occur before the event on its immediate right which must occur before the event on its immediate right is allowed to occur, etc.). The sequence enforcing gate can be contrasted with the Priority-AND gate in that the Priority-AND gate *detects* whether events occur in a particular order (the events can occur in any order) where the sequence enforcing gate will only *allow* the events to occur in a specified order.

In the generation of a Markov chain from a fault tree



ure 26: An instance of the fault tolerant parallel pro-
ɔor

taining a sequence enforcing gate, states that repre-
t any other ordering than that permitted by the se-
nce enforcing gate are never generated.

Examples using dynamic fault tree gates

this section, we analyze several configurations of a
lt-tolerant parallel processor, to illustrate the use of
sequence dependency gates. More example analyses
ıg the dynamic gates can be found in [9].
Ve consider several models of the FTPP (Fault Toler-
Parallel Processor) [11, 12] cluster, to compare vari-
configurations of triads with spares. An instance of
FTPP cluster is shown in figure 26, and consists of 16
cessing elements (PE), with 4 connected to each of 4
work elements (NE). The network elements are fully
nected. In the clusters modeled here, the 16 proces-
s are logically connected to form 4 triads, each with
spare. We investigate three triad/spare configura-
ıs, the first two with hot spares and the third with
l spares:

utilizes hot spares; there is one spare for each triad and all spares are attached to the same network element.

2 also uses hot spares; there is one spare on each network element and the spare PE can substitute for any failed PE attached to the same network element.

3 is the same as #1, with all spares on the same NE, but in configuration #3 the spares are cold.

The processing elements in all three configurations
ctionally depend on the network element to which they

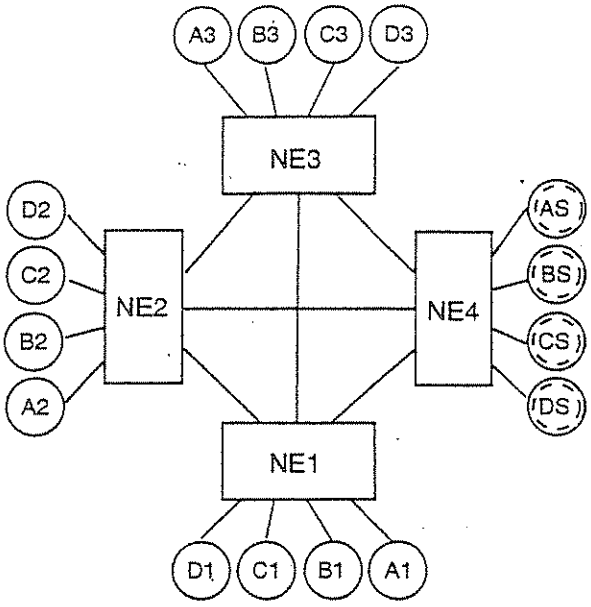


Figure 27: Configuration #1 with one spare per triad

are connected. If a network element experiences a permanent failure, the processing elements connected to it are then considered failed.

For all models, a triad fails when it has fewer than 2 active components; the system fails if any triad fails.

4.2.1 FTPP Configuration #1

Configuration #1 (shown in figure 27) divides the active elements of a triad among *NE1*, *NE2* and *NE3*, and uses the PE's on *NE4* as spares. The PE's that are in the same relative position on the first three network elements form a triad, and the PE in the same relative position on *NE4* serves as a hot (active) spare for the triad. The spares are indicated in the figure by a dashed circle.

The fault tree model for configuration #1, shown in figure 28, uses four functional dependency gates (FDEP) to reflect the dependence of the processing elements on the network elements. Where there are two basic events with the same label, they both represent the same basic event. The FDEP gates are not explicitly connected to the other gates in the tree, since the reliability requirements (all 4 triads must be operational) do not explicitly mention the network elements. Figure 28 shows four 3/4 gates connected to the top OR gate, one 3/4 gate for each triad. A triad fails when only one element remains (3 of the 4 elements have failed).

4.2.2 FTPP Configuration #2

Configuration #2 is an FTPP cluster with hot spares distributed across the network elements instead of grouped on the same network element (see figure 29). The spare element (marked with a dashed circle) on each network element can substitute for any failed PE connected to the

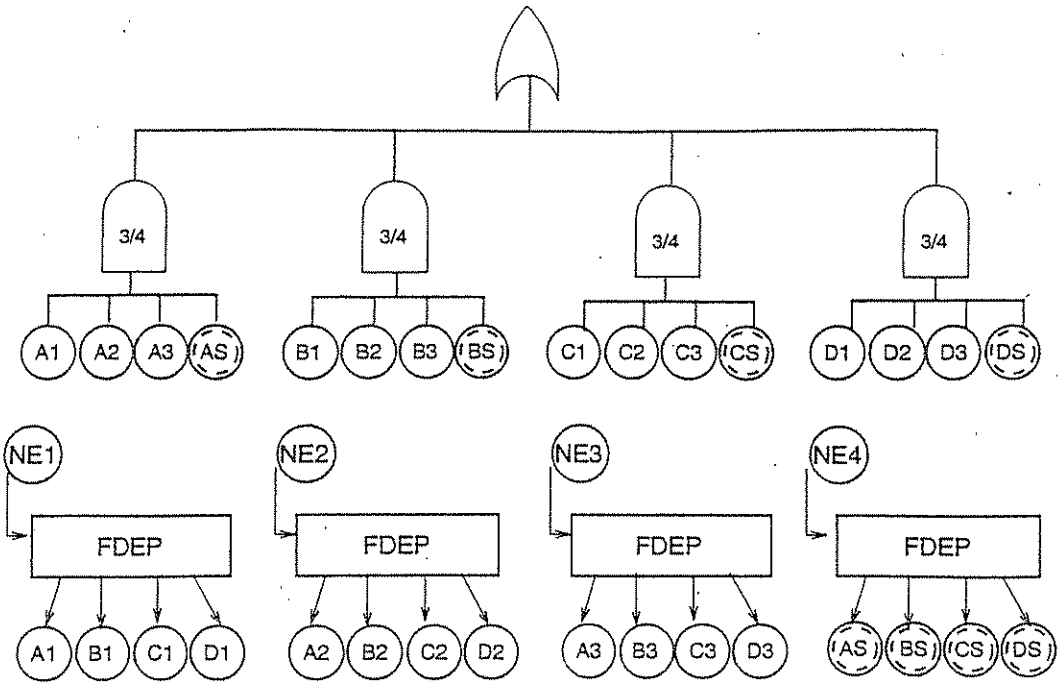


Figure 28: Fault tree model for configuration #1

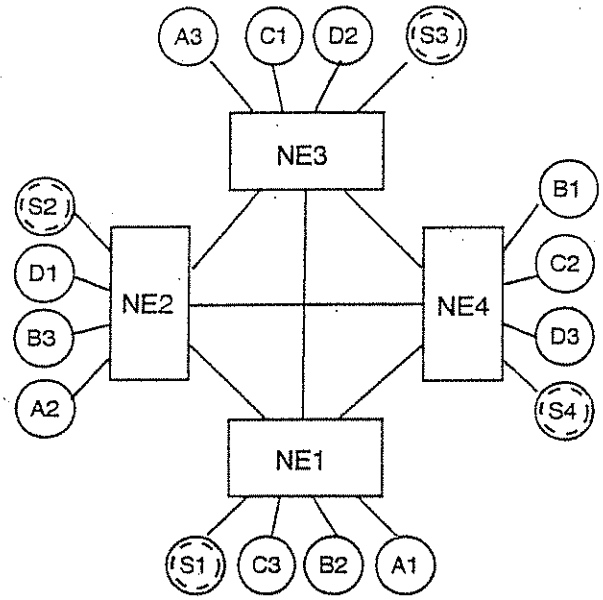


Figure 29: Configuration #2 with one spare per NE

same NE. That is, processing element S_1 can substitute for a failed PE connected to *NE1*.

The fault tree model for configuration #2 is shown in figure 30. The functional dependency gates FDEP again reflect the dependence of the processing elements on the network elements. A triad failure is again attributed to losing the majority of operational elements, but it is more difficult to describe the failure of a member of the triad. A member of the triad is failed if it and its spare fail or if its spare is not available when needed. The spare is not available if some other PE on the same NE fails and uses the spare before it is needed by the original PE. For example, in figure 30, the leftmost OR gate that inputs into the leftmost 2/3 gate represents the failure of the first member of the first triad. This member fails if both $A1$ (the first member of the first triad) and its spare ($S1$) fail, or if the spare is being used because another failure has already occurred when $A1$ fails. The spare will already be in use when $A1$ fails if either $B2$ or $C3$ (the other two active components on the same NE) have failed before $A1$ does. This condition is reflected in the Priority-AND gate that inputs to the same OR gate. There is a similar structure of AND and Priority-AND gates to represent the failure of the other members of the triads.

4.2.3 FTPP Configuration #3

The third configuration is used to investigate the effect on reliability of keeping the spares unpowered until needed. The FTPP configuration modeled in this section is the same as configuration #1 (figure 27) except that the spares are cold rather than hot. There is one spare for each triad, and all spares are connected to the same net-

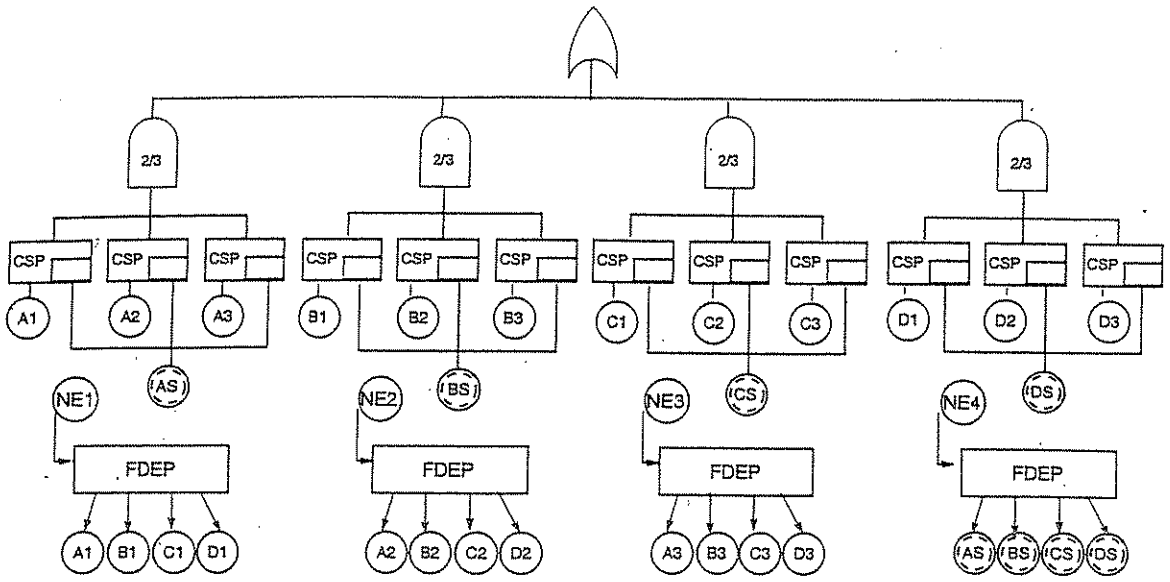


Figure 31: Fault tree model for configuration #3 with one COLD spare per triad

rk element. The fault tree model for this system, shown figure 31, uses the cold spare gate. There is one cold are gate for each member of each triad, where the inilly active members of the triad are used as the primary uts. The basic event representing the cold spare PE is nected to all three cold spare gates since it can subute for any of the elements.

References

] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, c-27(6):509–516, June 1978.

] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, c-35(8):677–691, August 1986.

] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

] Olivier Coudert and Jean Christophe Madre. Metaprime, an interactive fault-tree analyser. *IEEE Transactions on Reliability*, 43(1):121–127, March 1994.

] Olivier Coudert and Jean Christophe Madre. Fault tree analysis: 10^{20} prime implicants and beyond. In *Proceedings of the Reliability and Maintainability Symposium*, pages 240–245, January, 1993.

] Stacy A. Doyle and Joanne Bechta Dugan. Dependability assessment using binary decision diagrams. In *Proc. IEEE Int. Symp. on Fault-Tolerant Computing, FTCS-25*, June 1995.

[7] Stacy A. Doyle, Joanne Bechta Dugan, and Mark A. Boyd. Combinatorial-models and coverage: A binary decision diagram (BDD) approach. In *Proceedings of the Reliability and Maintainability Symposium*, January, 1995.

[8] Stacy A. Doyle, Joanne Bechta Dugan, and Ann Patterson-Hine. A combinatorial approach to modeling imperfect coverage. *IEEE Transactions on Reliability*, pages 87–94, March 1995.

[9] Joanne Bechta Dugan, Salvatore J. Bavuso, and Mark A. Boyd. Fault trees and markov models for reliability analysis of fault tolerant systems. *Reliability Engineering and System Safety*, 39:291–307, 1993.

[10] Joanne Bechta Dugan and K. S. Trivedi. Coverage modeling for dependability analysis of fault-tolerant systems. *IEEE Transactions on Computers*, 38(6):775–787, 1989.

[11] Richard E. Harper. Reliability analysis of parallel processing systems. In *Proceedings of the 8th Digital Avionics Systems Conference*, pages 213–219, 1988.

[12] Richard E. Harper, Jaynarayan H. Lala, and John J. Deyst. Fault tolerant parallel processor architecture overview. In *Proceedings of the 18th Symposium on Fault Tolerant Computing*, pages 252–257, 1988.

[13] E. J. Henley and H. Kumamoto. *Probabilistic Risk Assessment*. IEEE Press, 1992.

[14] Michael Pecht, editor. *Product Reliability, Maintainability and Supportability Handbook*. CRC Press, 1995.

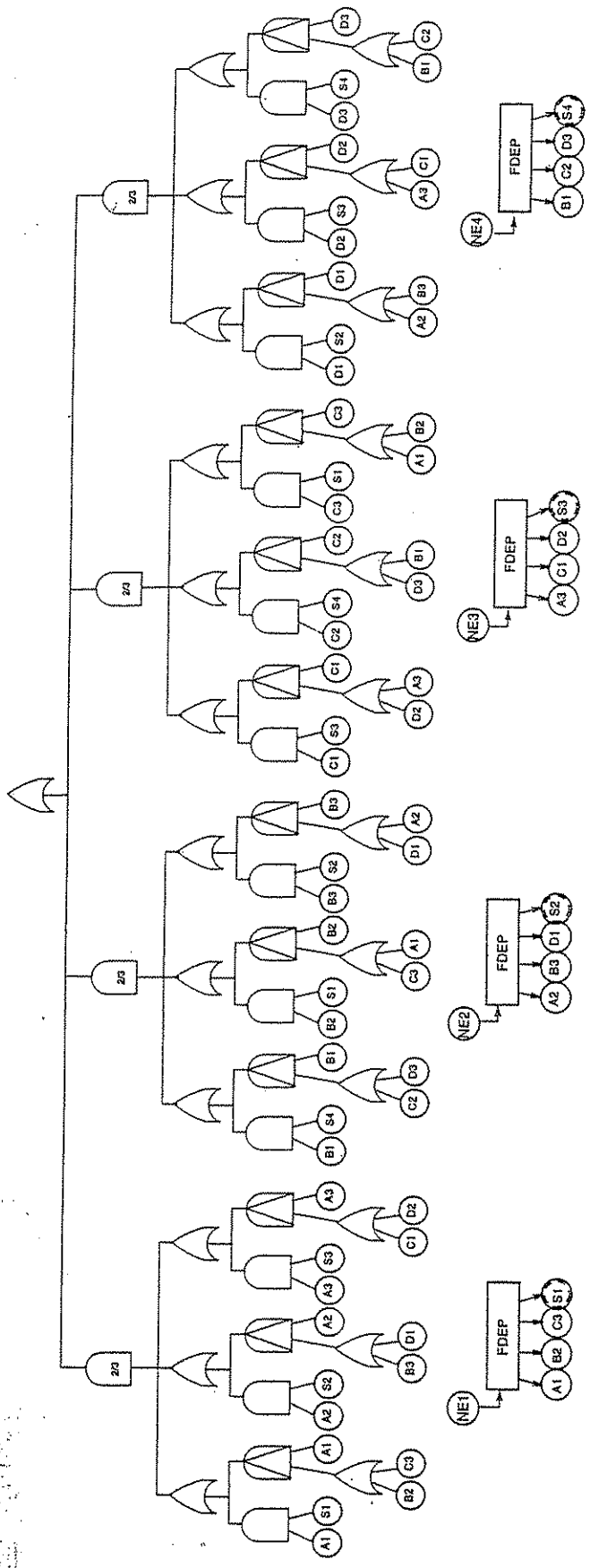


Figure 30: Fault tree model for configuration #2

[15] M. L. Shooman. *Probabilistic Reliability: An Engineering Approach.* McGraw-Hill, New York, 2nd edition, 1990.

[16] Malathi Veeraraghavan and Kishor S. Trivedi. An improved algorithm for symbolic reliability analysis. *IEEE Transactions on Reliability*, 40(3):347–358, August 1991.