

Fault Tree Analysis of Computer-Based Systems

Joanne Bechta Dugan
Professor of Electrical & Computer Engineering
University of Virginia
(jbd@Virginia.edu)

Presentation Outline

- I. Introduction to fault trees
- II. Fault tree analysis of an example control system
- III. Fault trees as design aid for software systems
- IV. Adapting the fault tree to analysis of computer-based systems
- V. Dynamic fault trees for modeling sequential behavior
- VI. Modular approach to fault tree analysis
- VII. Sensitivity analysis
- VIII. Summary and Conclusions

Introduction to fault tree analysis

Fault trees provide a good framework for both qualitative and quantitative analysis because they have both a logical (boolean algebra) and probabilistic basis.

What is a fault tree?

- not a tree (in the graph-theoretic sense)
- a graphical representation of a logical function
- shows logical relationship between an event (failure) and its causes
- provides a logical framework for expressing combinations of component failures that can lead to system failure

Why use fault tree analysis?

A fault tree model provides a logical framework for analyzing the failure behavior of a system.

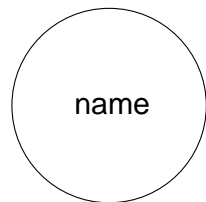
A fault tree model precisely documents which failure scenarios have been considered and which have not.

Fault tree analysis can be used to support engineering and management decisions, trade-off analysis and risk assessment.

The fault tree model has a well-defined boolean algebraic and probabilistic basis which relates probability calculations to boolean logic functions.

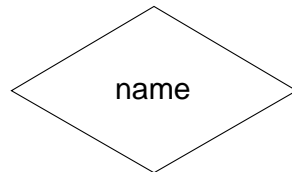
Basic Static Fault Tree Constructs

Basic Events



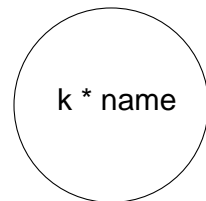
Basic Event: corresponds to a basic failure event (usually a component failure) in the system.

Characterized by failure rate or failure probability



Undeveloped Basic Event: A basic event that is not completely developed, usually because of unavailable information.

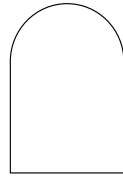
Characterized by failure rate or failure probability



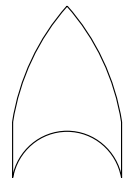
Replicated Basic Event; represents k statistically identical copies of a component

Characterized by failure rate or failure probability

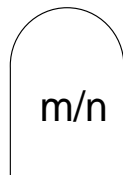
Static fault tree gates



AND gate - output event occurs only if ALL input events occur

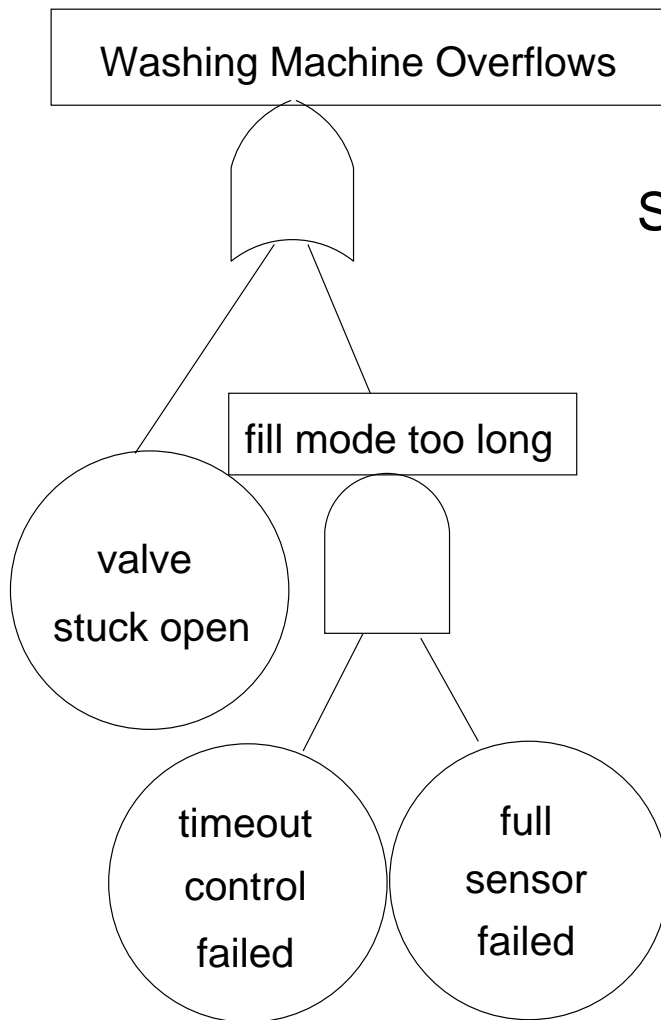


OR gate - output event occurs if one or more input events occur



m/n gate - output event occurs if m or more of the n inputs occur

Example Fault Tree



Structure Function:

Fail = valve_failed OR
(timer_failed AND sensor_failed)

$$F = A + BC$$

Probabilistic Fault Tree Analysis of Example

$$F = A + BC$$

$$\Pr[F] = \Pr[A + BC]$$

$$= \Pr[A] + \Pr[BC] - \Pr[ABC]$$

$$= 0.01 + 0.00375 - 0.0000375$$

$$= 0.0137125$$

Suppose

$$\Pr[A] = 0.01$$

$$\Pr[B] = 0.05$$

$$\Pr[C] = 0.075$$

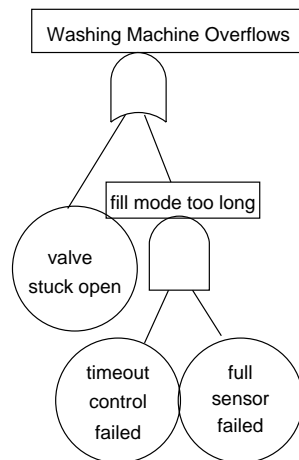
(all failures are independent)

Fault Tree Analysis - Cutsets

Most fault tree analysis techniques start with the generation of **cutsets**

A cutset is a set of basic events; if all the basic events in a cutset occur, then the top event (system failure) occurs.

A mincut (minimum cutset) is one that contains no redundant elements. If an element is removed from a mincut, it ceases to be a mincut.

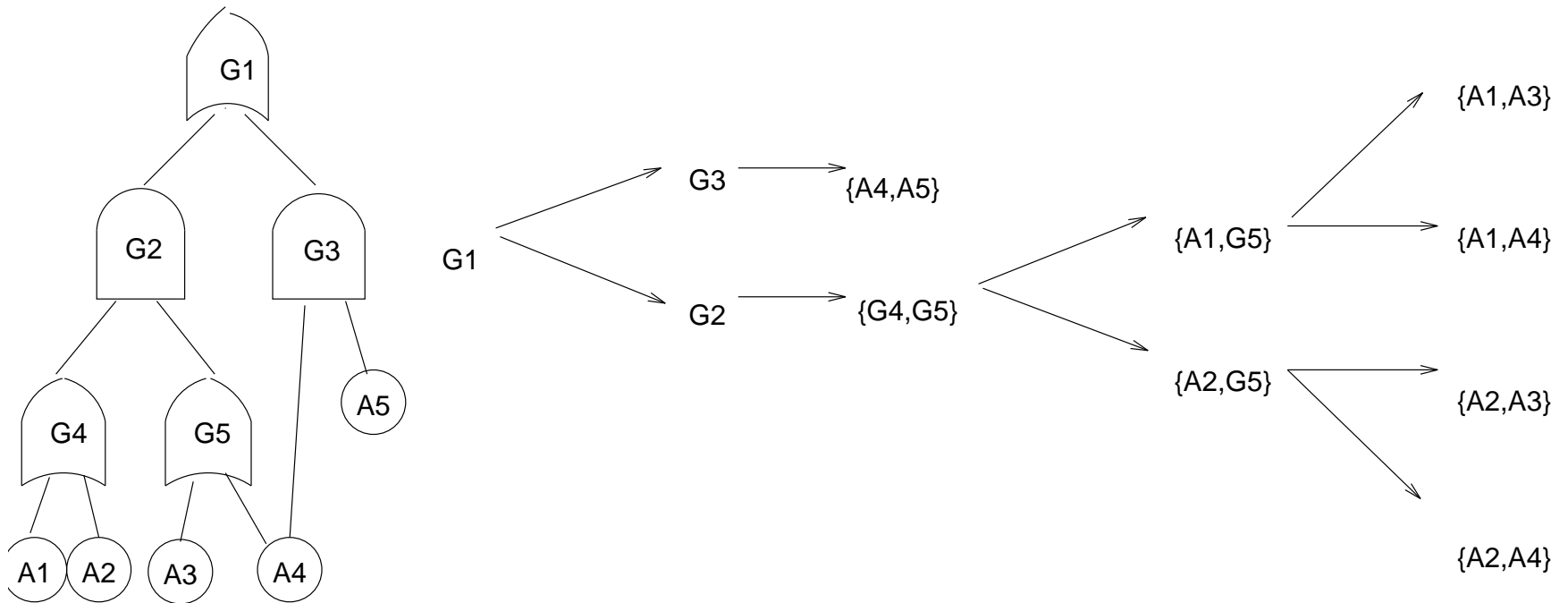


Cutsets:

{valve} (single point of failure)

{timer, sensor}

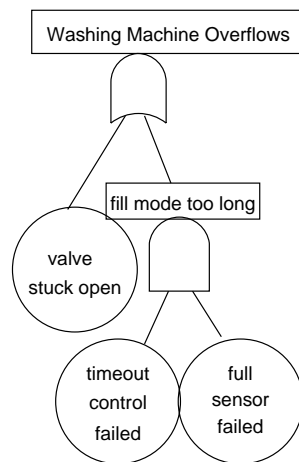
Cutset generation by example



Probabilistic Analysis using Cutsets

The probability of system failure is simply the probability that one or more of the cutsets occur.

But the cutsets are not disjoint so we cannot sum their individual probabilities. We must account for the overlap of the events.



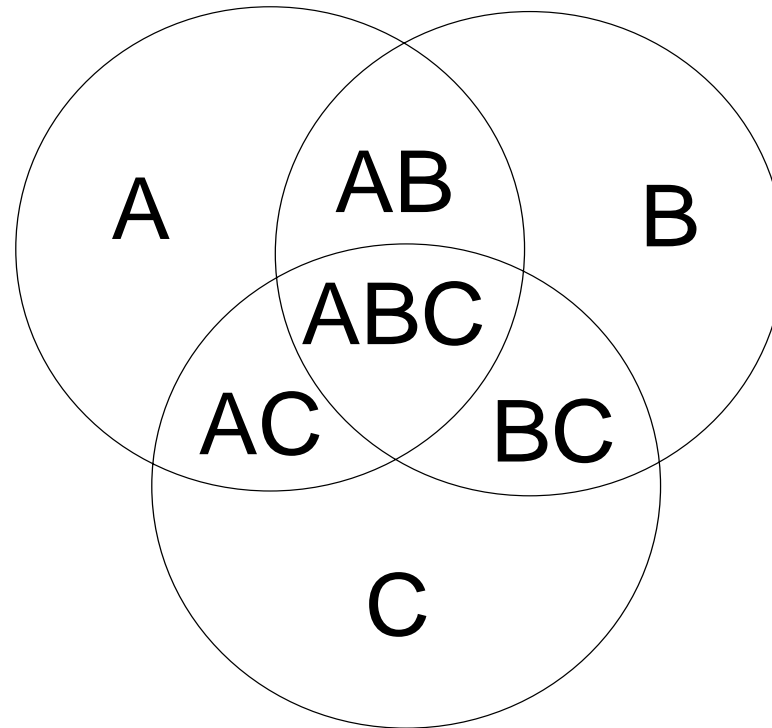
Cutsets:

{valve} (single point of failure)

{timer, sensor}

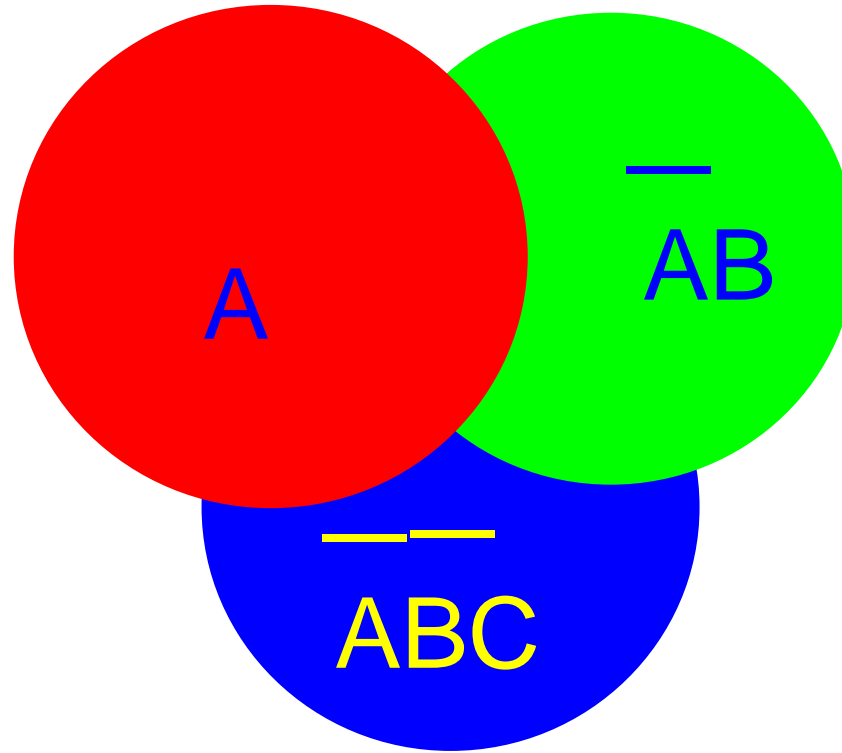
$$\text{Prob(failure)} = \text{Prob(valve failure)} + \text{Prob(both timer and sensor fail)} - \text{Prob(all three fail)}$$

Probabilistic Analysis using Inclusion-Exclusion



$$\begin{aligned} & \Pr(A) + \Pr(B) + \Pr(C) \\ & - \Pr(A \text{ and } B) - \Pr(B \text{ and } C) - \Pr(A \text{ and } C) \\ & + \Pr(A \text{ and } B \text{ and } C) \end{aligned}$$

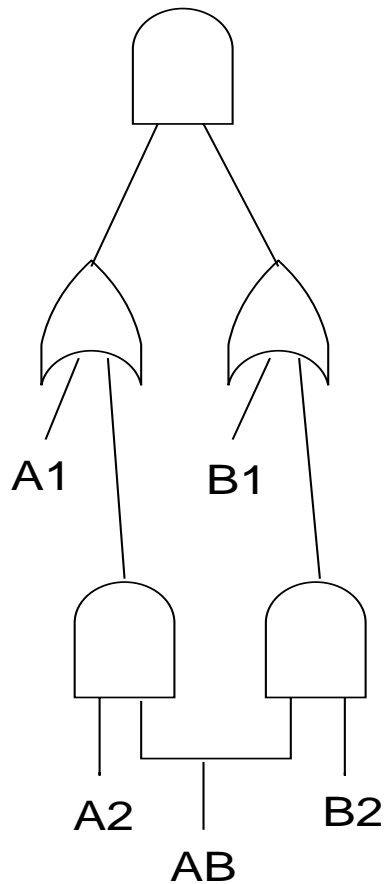
Probabilistic Analysis using Sum-of-Disjoint-Products



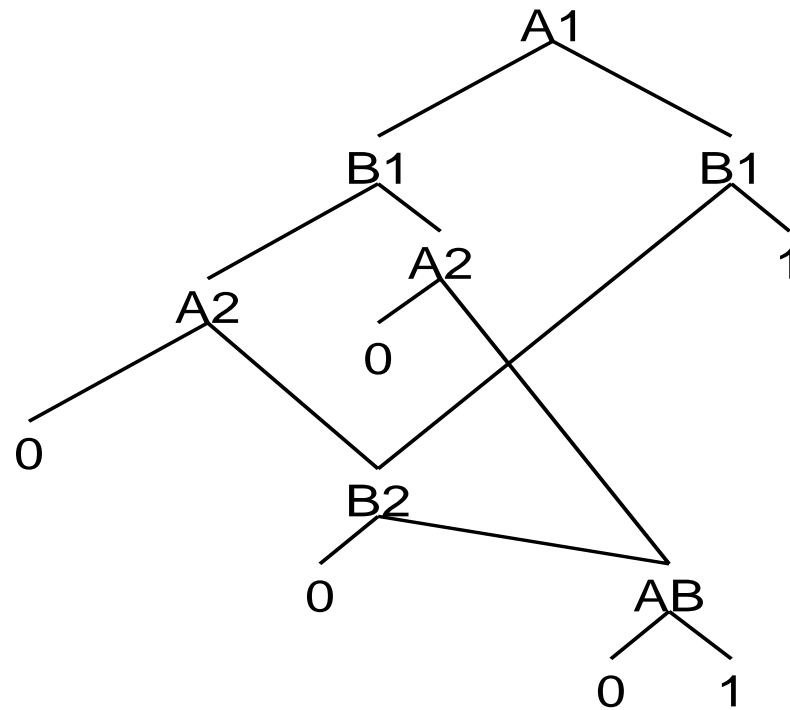
$$\Pr(A) + \Pr(\text{not } A \text{ and } B) + \Pr(\text{not } A \text{ and not } B \text{ and } C)$$

Probabilistic Analysis using Binary Decision Diagrams

Fault tree model



BDD representation



Presentation Outline

I. Introduction to fault trees

II. Fault tree analysis of an example control system

III. Fault trees as design aid for software systems

IV. Adapting the fault tree to analysis of computer-based systems

V. Dynamic fault trees for modeling sequential behavior

VI. Modular approach to fault tree analysis

VII. Sensitivity analysis

VIII. Summary and Conclusions

An example control system including software

Consider a simple tank level and flow control system.

The key features of this system are:

- a water tank, fed by a water pump on the inflow and regulated by control and stop valves on the inflow and outflow pipes.
- a tank and level control system with three sensors (level, inflow and outflow) implemented in software
- a tank bypass to prevent overflow, controlled by the three stop valves
- valve actuation and control implemented in software

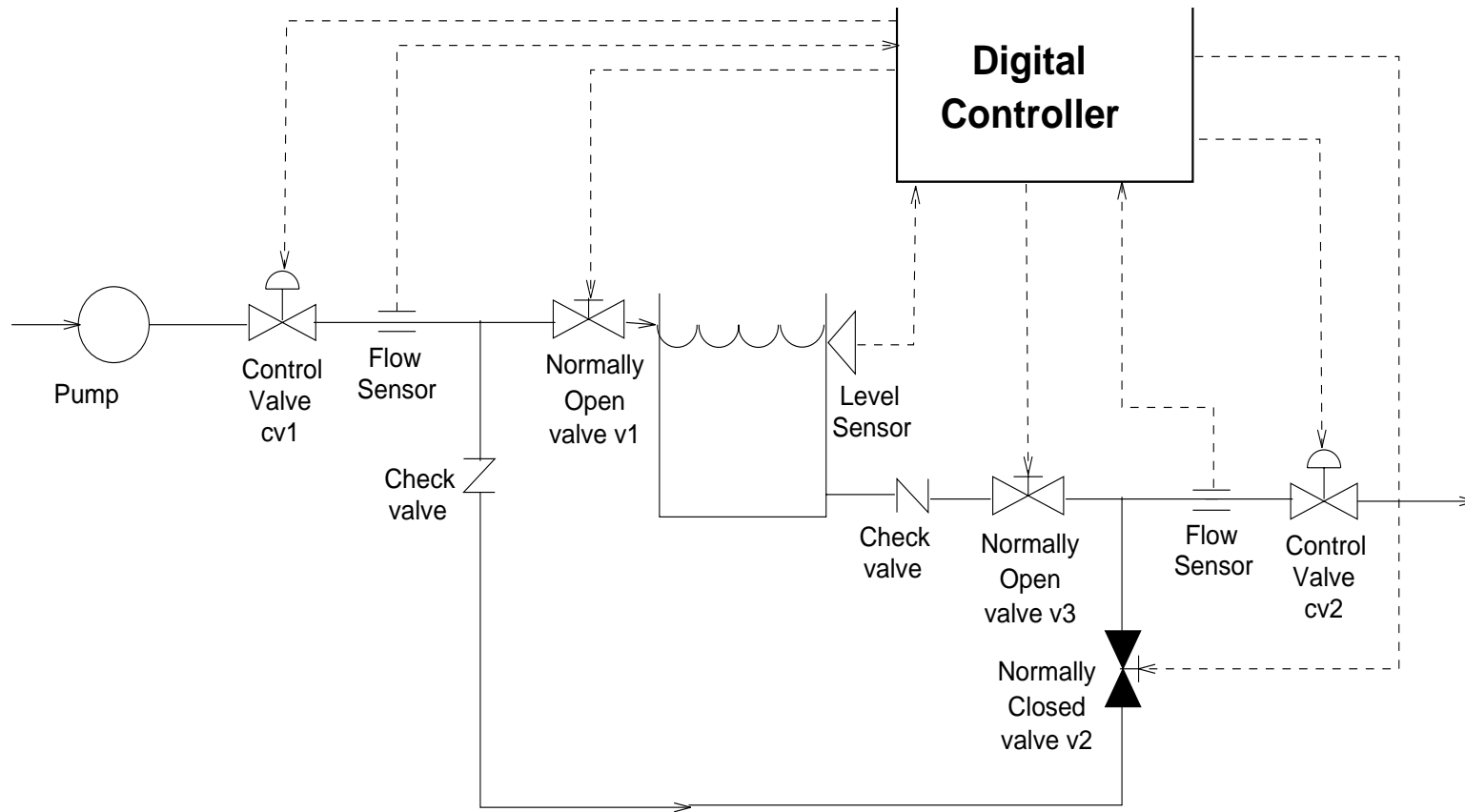
Function of example system

The function is to maintain the water level and downstream flow rate at particular values by opening and closing control valves cv1 and cv2.

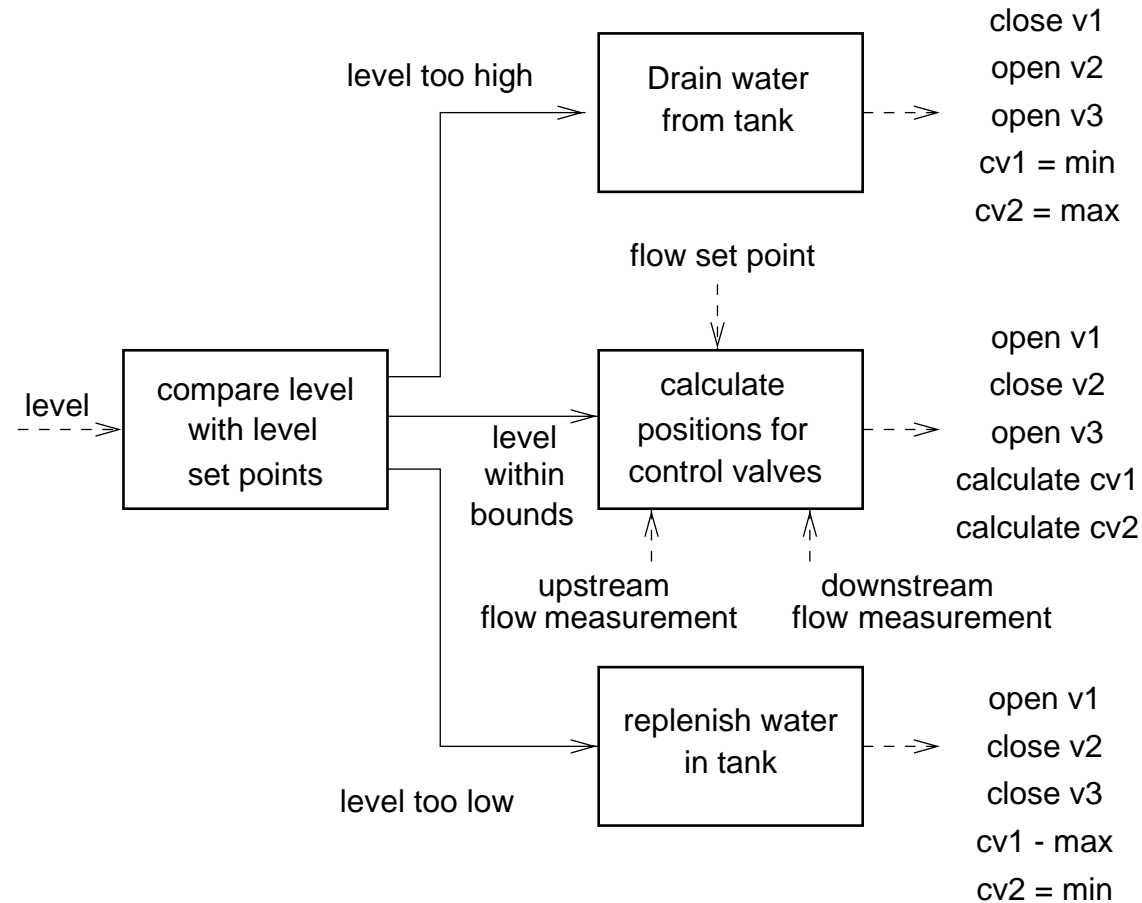
The controller receives inputs from the three sensors, implements the control logic and then gives commands to the two control valves and the two stop valves.

(This example is adapted from an example in: S. Guarro, M. Yau and M. Motamed, "Development of tools for safety analysis of control software in advanced reactors", NUREG/CR-6465, April 1996.)

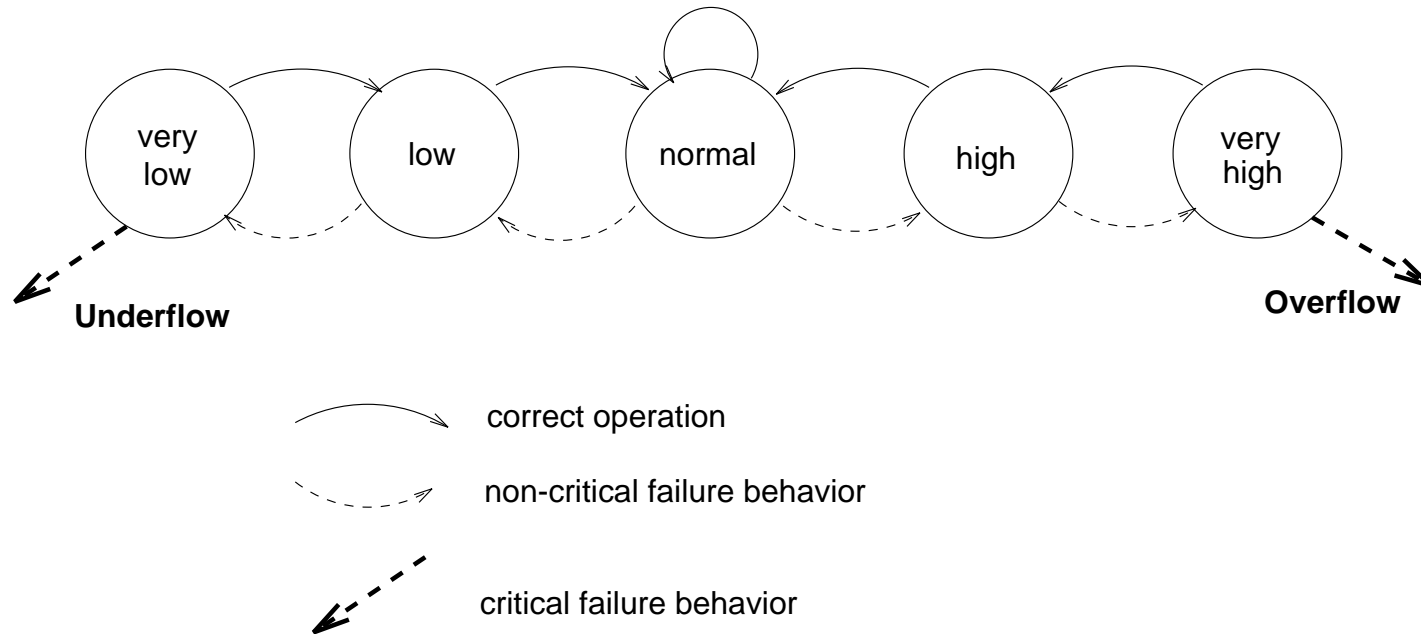
Diagram of example system



Control Flow



Software Operational modes



System Failure modes

The control system is not designed to be fault tolerant, so that most hardware failures present either an overflow or underflow hazard. These are the two system-level failure modes being considered. If either of the control valves or any of the three sensors fail, the system fails, as the software will be unable to control the system. A tank leak, pipe leak or pump failure are also considered single points of failure.

Further, an underflow can occur if valve v1 or v2 fails, thus preventing proper inflow, unless valve v3 can be closed. Therefore, if v3 and either v1 or v2 fail, the system fails. Overflow can occur if valve v3 fails, thus preventing outflow, unless v1 can be closed. Thus the failure of both v1 and v3 leads to system failure.

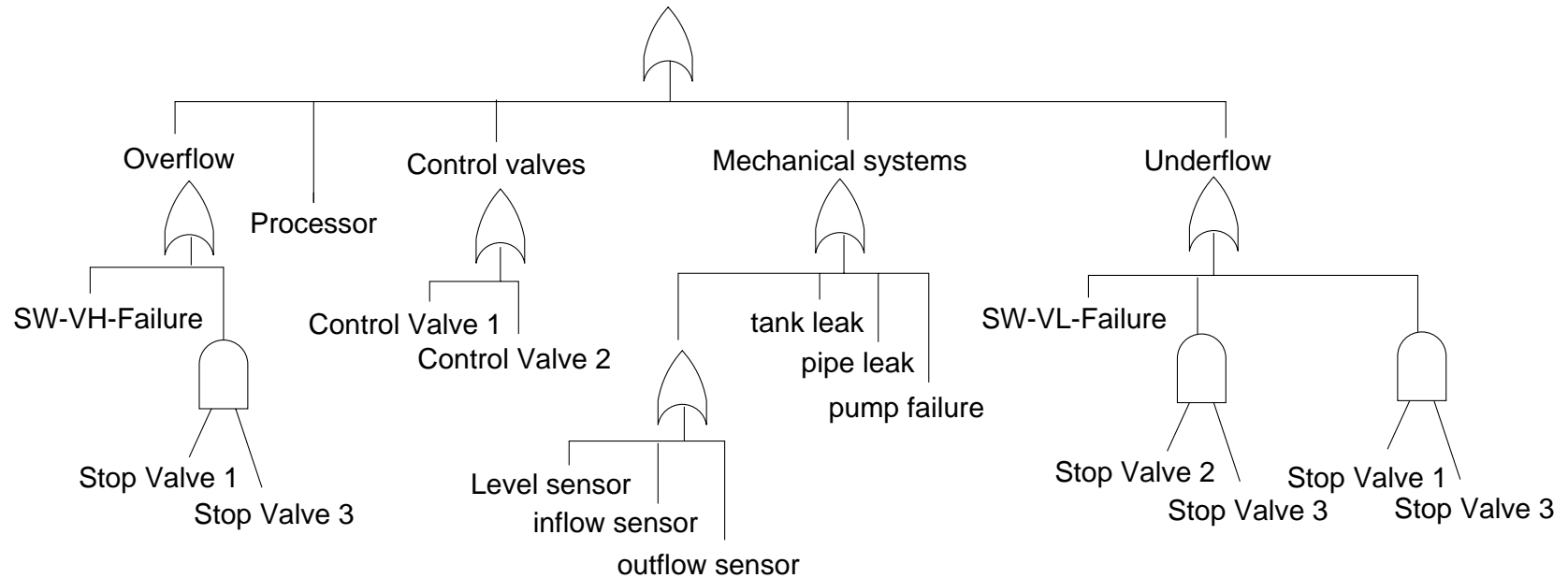
Software Failure Modes

Software failures have been characterized as an improper change of state. Software failures are further classified as critical and non-critical.

Non-critical software failures lead to less than optimal performance but do not lead to system failure.

A software failure when the tank water level is very low or very high can lead to system failure.

Fault Tree Model for Example System



Presentation Outline

- I. Introduction to fault trees
- II. Fault tree analysis of an example control system
- III. Fault trees as design aid for software systems**
- IV. Adapting the fault tree to analysis of computer-based systems
- V. Dynamic fault trees for modeling sequential behavior
- VI. Modular approach to fault tree analysis
- VII. Sensitivity analysis
- VIII. Summary and Conclusions

Fault trees as a design aid for software systems

Fault tree analysis can help to insure that the software system *does not do* what it is *not* supposed to do. (As contrasted with a formal design review which helps insure that the software *does* what it *is* supposed to do.)

For robust software systems, fault trees can help identify high-risk areas (either quantitatively or qualitatively).

Can manage risk by preventive or protective measures applied to identified high-risk areas.

- exhaustive testing
- formal methods
- exception handling
- acceptance tests
- interlocks
- redesign

Using fault trees to manage risk

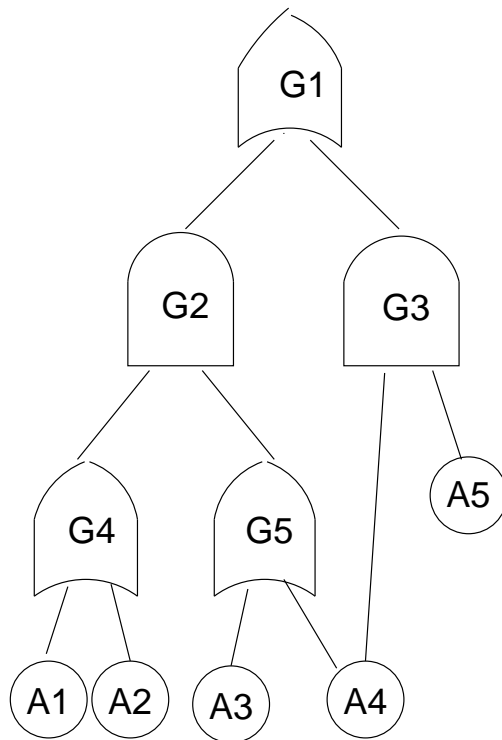
AND gates can be protected by disallowing one of the inputs[1]

- exhaustive testing or formal proof to show module cannot fail
- test for failure condition and provide recovery routine

OR gate can be protected by disallowing *all* inputs or by providing detection and recovery point. (The detection and recovery routines must be simple enough to be certifiably correct.)

[1] Herbert Hecht and Myron Hecht, "Fault Tolerant Software." In D.K. Pradhan, editor, *Fault-Tolerant Computing: Theory and Techniques*, volume 2, pages 658-696. Prentice-Hall, 1986.

Example Risk Mitigation



Suppose basic events represent software modules.

Can protect G3 by preventing failure of module A4

- exhaustive testing
- proof of correctness

Then can protect G2 by

- preventing failure of A3
- preventing failure of both A1 and A2
- or by providing detection and recovery handler for G4

Presentation Outline

I. Introduction to fault trees

II. Fault tree analysis of an example control system

III. Fault trees as design aid for software systems

IV. Adapting the fault tree to analysis of computer-based systems

V. Dynamic fault trees for modeling sequential behavior

VI. Modular approach to fault tree analysis

VII. Sensitivity analysis

VIII. Summary and Conclusions

Modeling Fault-Tolerant Computer Systems

Fault Tolerant Computer (FTC) systems can actively handle many faults and errors that may occur.

Because FTC are adaptive and flexible, faulty components can be switched out automatically, and spares switched in.

However, adaptability and flexibility often result in increased complexity. Increased complexity can mean decreased reliability.

If the fault tolerance mechanisms (error detection, recovery, reconfiguration) fail, this failure could lead to overall system failure, *even if adequate functioning resources remain.*

A coverage model is used to analyze the behavior of the computer system in the presence of a fault. The results of the coverage model are then incorporated into the overall system model.

Covered vs. Uncovered Faults

A *covered* fault is one from which the system can automatically recover.

- Recovery from transient does not change system state.
- Recovery from a permanent fault discards faulty component.

An *uncovered* fault is one which leads to immediate system failure, regardless of the state of the system.

How to estimate coverage?

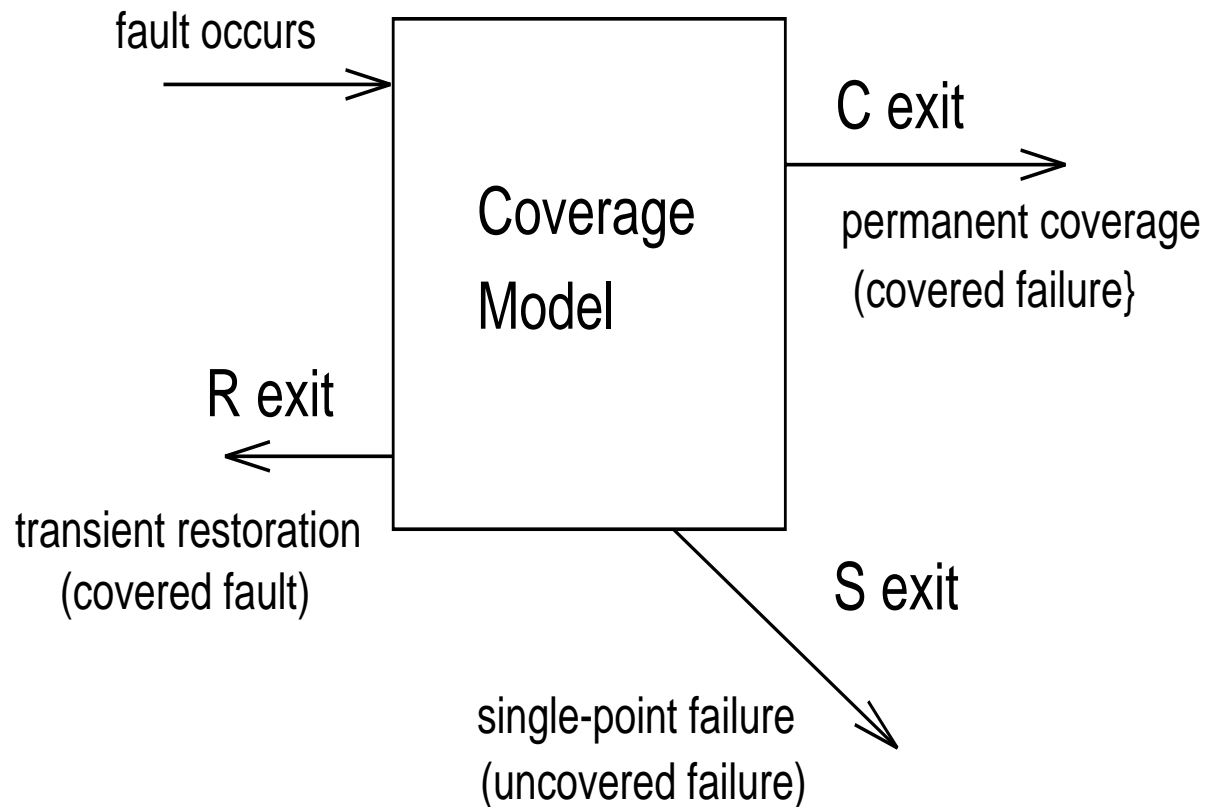
If a working or prototype version of the system exists, or if enough information is available about a system being designed, then coverage probabilities can be estimated.

A model of the recovery process can be developed. The parameters for the model can be measured from a fault injection on the working prototype and or estimated from data collected in the field.

A detailed simulation model of the system recovery process can be developed.

If the details of the recovery process are not known, reasonable parameters can be deduced from other, similar systems.

General structure of a coverage model



The entry point to the model is the occurrence of the fault, and the three exits (R,C, and S) are the three possible outcomes.

R exit: Transient Restoration

Correct recognition of and recovery from a transient fault. A transient is usually caused by external or environmental factors, such as excessive heat or a glitch in the power line.

The vast majority of faults are transient.

Successful recovery from a transient fault restores the system to an operational state without discarding any components - for example by masking the error, retrying an instruction, or rolling back to a previous checkpoint.

Reaching this exit successfully requires:

- timely detection of an error produced by the fault;
- performance of an effective recovery procedure; and
- swift disappearance of the fault (the cause of the error).

C exit: Permanent coverage

Determination of the permanent nature of the fault, and the successful isolation and removal of the faulty component.

S exit: Single Point failure

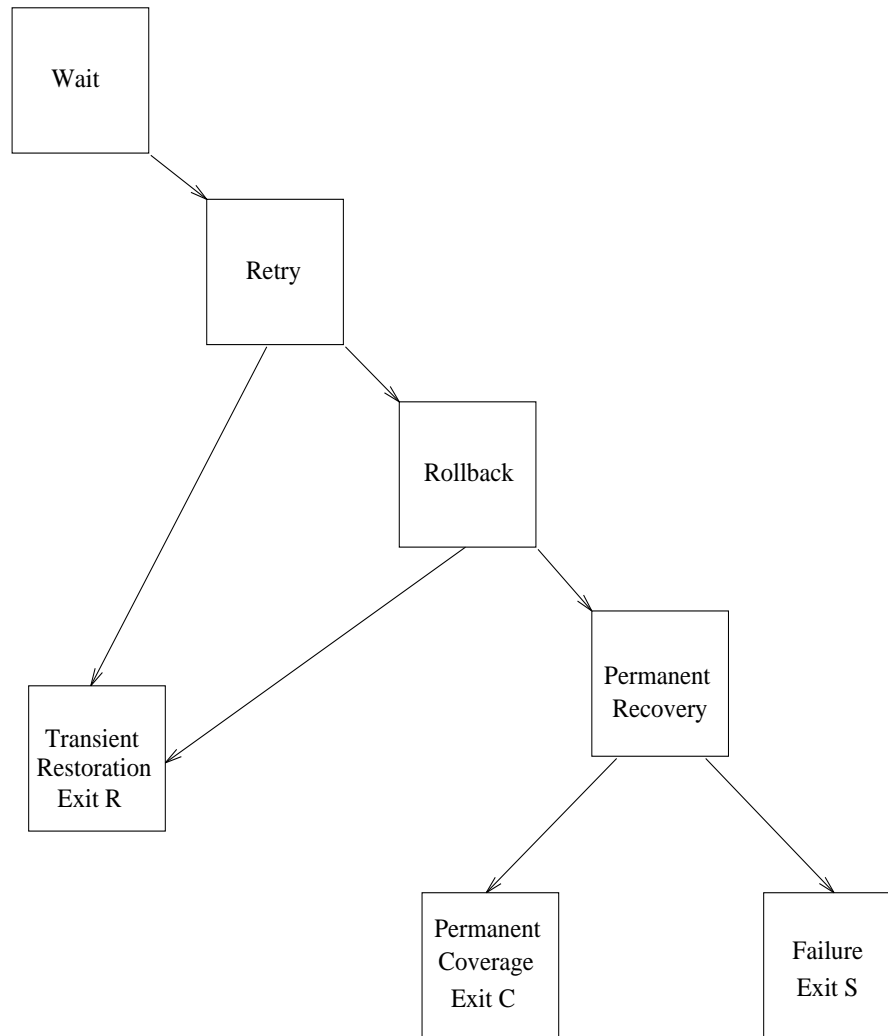
A single fault causes the system to fail, generally when an undetected error propagates through the system, or if the faulty unit cannot be isolated and the system cannot be reconfigured.

Typical fault recovery for a processor

A processor contains built-in test circuitry so that error checking occurs concurrently with instruction execution. If an error is detected, the instruction 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.

Example coverage model for processors



Example of transient restoration

Transient Restoration attempt: Assume that the fault is transient, and begin a multi-step recovery procedure that continues as long as an error is detected. If an error persists after all three steps have been performed, then a permanent recovery procedure must be invoked.

- Step 1: Wait for 0.1 second and do nothing. If the fault is transient it may disappear during this time, allowing rollback to succeed.
- Step 2: Retry the current instruction several times, for as long as a half-second. The probability that the retry will be successful (i.e., no error is detected) is 0.5.
- Step 3: If an error persists, perform a rollback to a previous checkpoint, followed by recomputation, taking 2 sec. total. The rollback succeeds in removing the error 80% of the time.

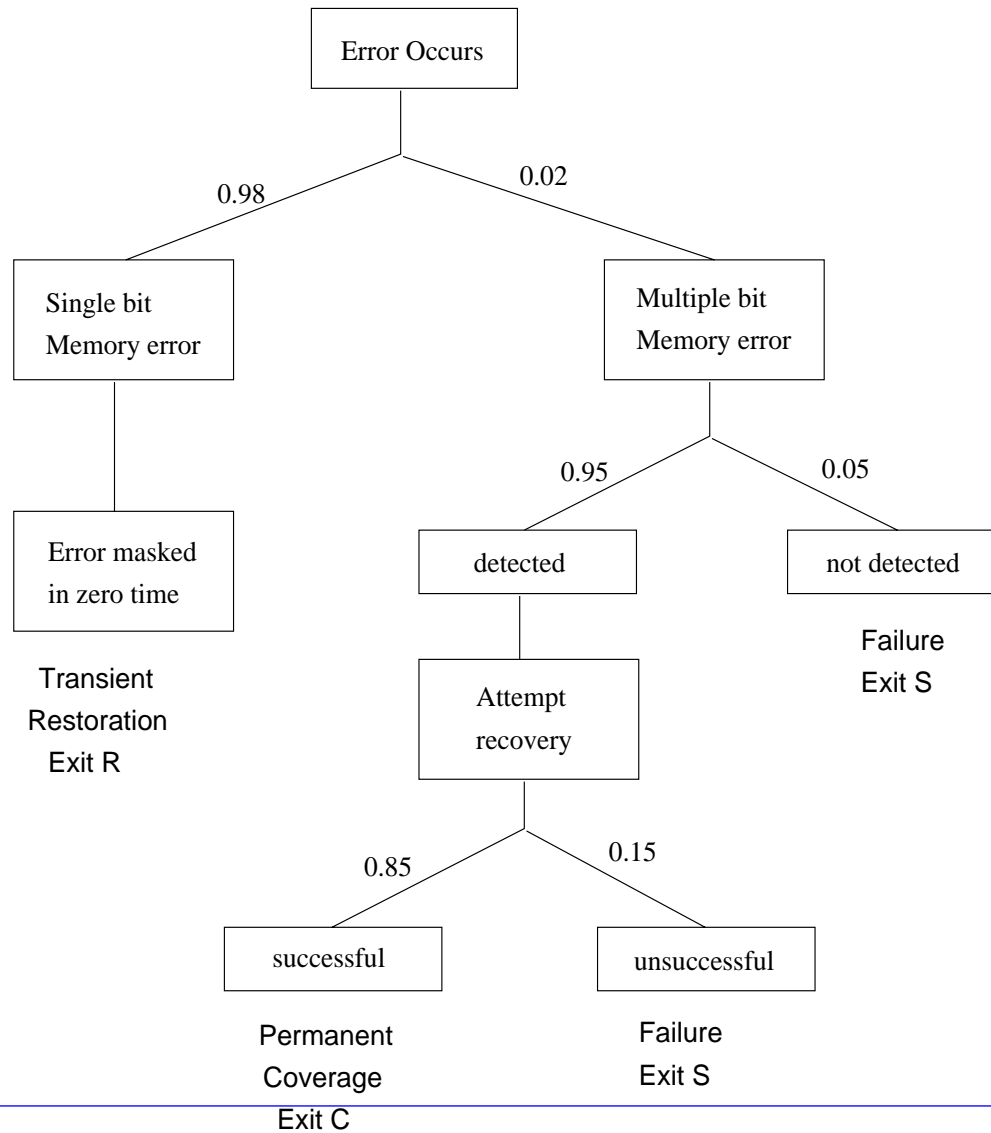
Example of permanent coverage and single-point failure

If an error still persists after the rollback, it is assumed to be caused by a permanent fault, and a system level permanent fault recovery process is begun, to remove the offending processor from the set of active units and to reconfigure the system to continue without it. The permanent fault recovery process succeeds with probability 0.875.

The permanent coverage procedure is invoked against a a persistent transient fault as well as against a permanent fault.

If the permanent fault recovery process fails, then a single-point failure is said to occur.

Coverage model for memories



Example of recovery process for memory faults

The memory uses an error correcting code, so a single-bit error is always detectable and correctable, and no reconfiguration is required. If 98% of all memory faults affect only a single bit, then the probability of reaching the R exit is 0.98.

The 2% of faults that affect more than one memory bit are 95% detectable. 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.

Experimentation on a prototype system revealed that this recovery from the detected multiple memory errors works 85% of the time. Thus, the probability of reaching the C exit is the probability that a multiple fault occurs, is detected, and is recovered from is:

$$c = 0.02 \times (0.95 \times 0.85) = 0.01615$$

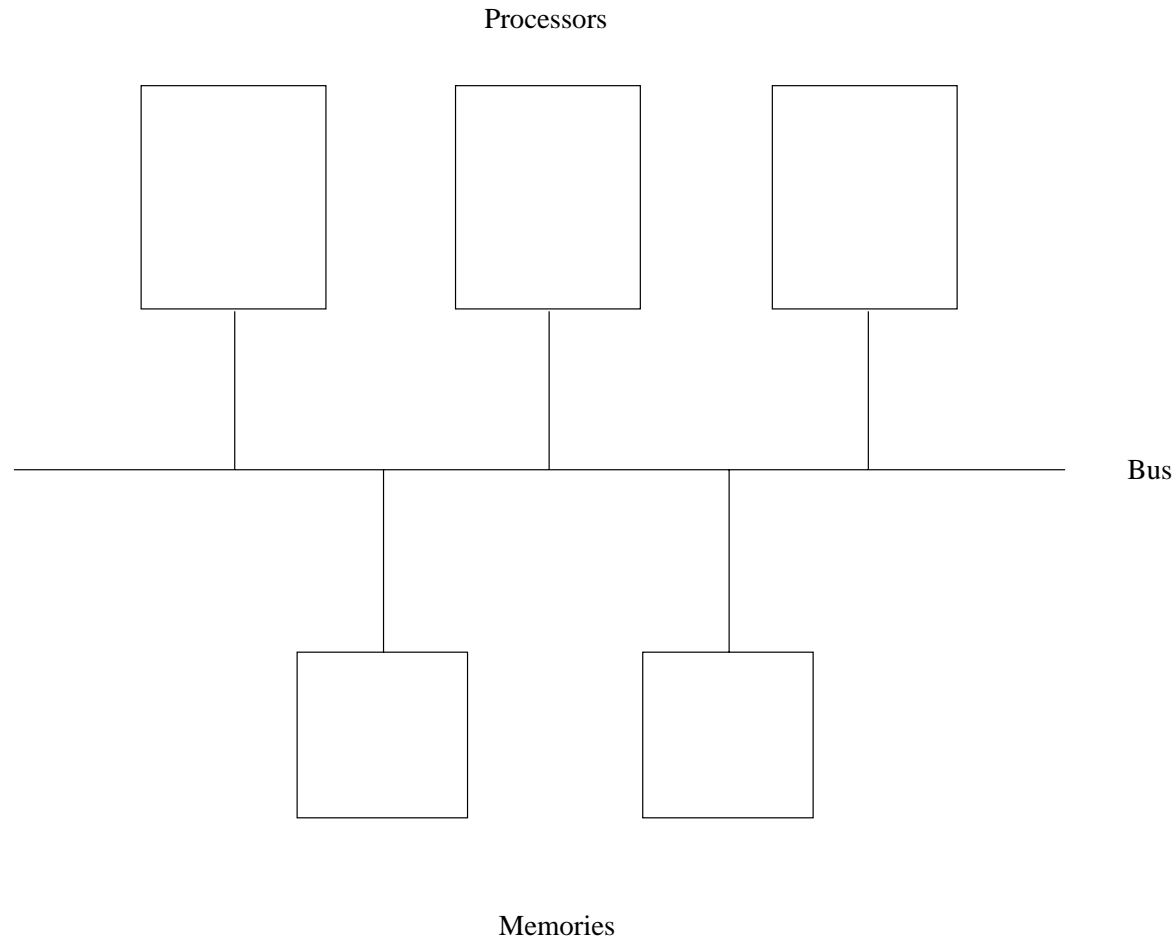
Single point failure for memory faults

There are two paths to the single point failure exit.

- The memory fault causes a single-point failure if a multiple-bit error is not detected (with probability 0.02×0.05)
- A multiple-bit memory error is detected, but the attempted recovery is not successful, with probability $0.02 \times 0.95 \times 0.15$

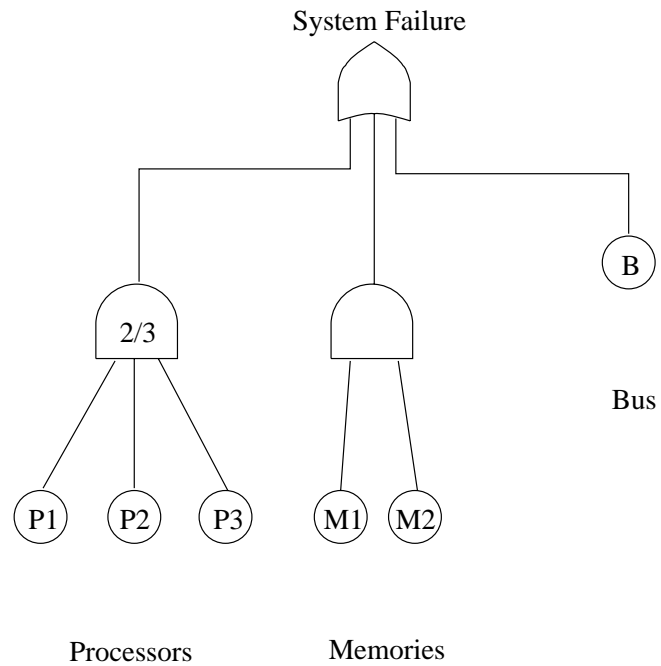
Thus the probability of single point failure is the sum of these two cases, or 0.00385.

Example - 3P2M system

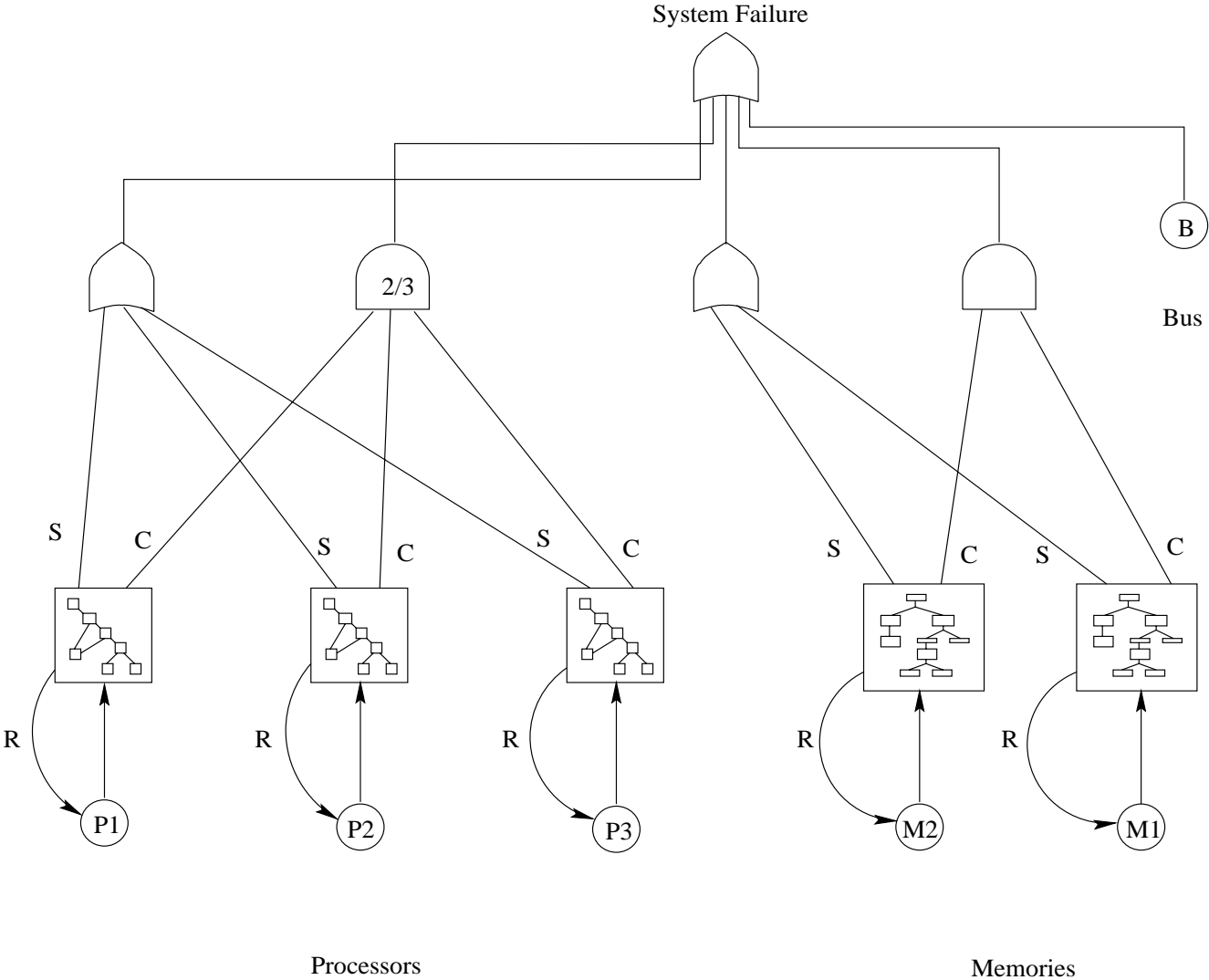


3 Processors and 2 memories connected via a bus. 2 Processors, one memory and the bus are needed for correct operation.

3P2M fault tree



Adding coverage to fault tree



Probabilities for covered and uncovered basic events

No fault or transient restoration	covered fault
	uncovered fault

$$\Pr[\text{component fault}] = p$$

$$\Pr[\text{component operational}] = q$$

$$\Pr[\text{covered failure}] = cp$$

$$\Pr[\text{Uncovered failure}] = sp$$

$$\Pr[\text{No fault or transient restoration}] = q + rp$$

Note that the events “fail covered” and “fail uncovered” are mutually exclusive (i.e. not independent).

Presentation Outline

- I. Introduction to fault trees
- II. Fault tree analysis of an example control system
- III. Fault trees as design aid for software systems
- IV. Adapting the fault tree to analysis of computer-based systems
- V. Dynamic fault trees for modeling sequential behavior**
- VI. Modular approach to fault tree analysis
- VII. Sensitivity analysis
- VIII. Summary and Conclusions

Sequence dependencies

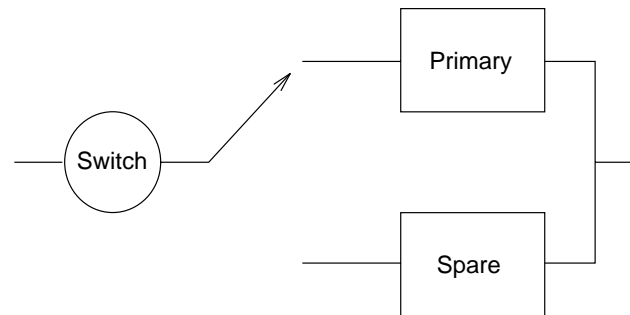
Traditional fault trees cannot model sequence dependent failures, in which the *order* that events occur is important.

We define special purpose gates for modeling sequence dependencies, and solve the resulting fault tree as a Markov chain.

The development of a correct Markov model for a complex system can be difficult. Our approach is to use the fault tree for model development and automatically convert the fault tree to the equivalent Markov chain. The dynamic fault tree model is considerably simpler than the equivalent Markov chain.

Coverage models are automatically added to the resulting Markov chain which is solved via a numerical differential equation solver.

Example of sequence dependency



If switch fails after primary fails (and after spare is activated) then the system is still operational.

If the switch fails before the primary fails, then the spare cannot be activated and the system fails, even though the spare is operational.

Failure criteria depends on *order* in which failures occur. This system can be solved correctly via a Markov model.

Sequence dependency gates

Several special purpose gates have been added to the traditional fault tree gates. These special *dynamic* gates capture sequence dependencies which frequently arise when modeling fault tolerant computer systems. If a dynamic gate is part of a fault tree then it is solved via a Markov chain, rather than by using traditional methods.

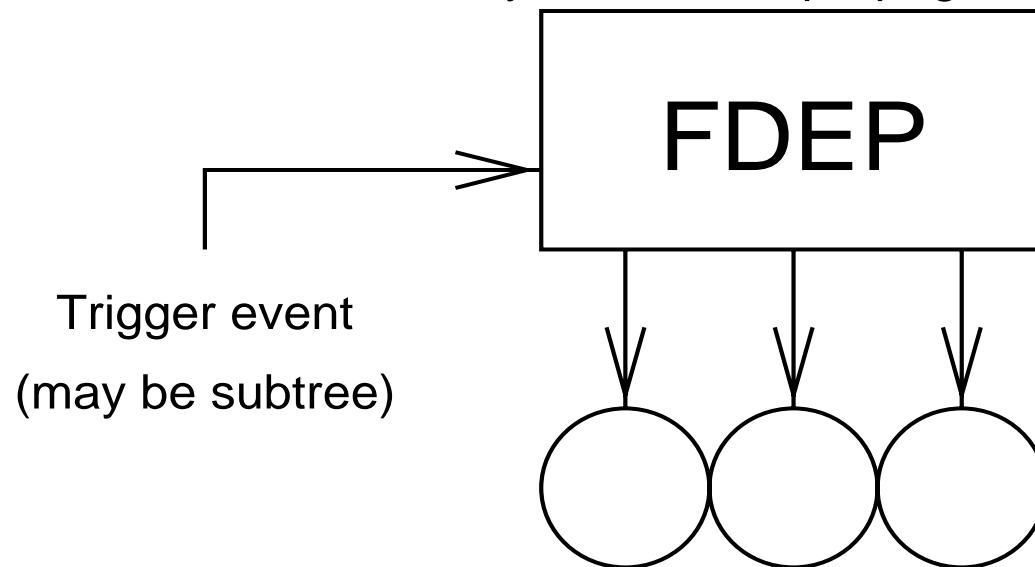
The special dynamic gates include:

- *Functional dependency* gate for modeling situations where one component's correct operation is dependent upon the correct operation of some other component
- *Spare* gate for modeling cold, warm and hot pooled spares
- *Priority-AND* gate for modeling ordered ANDing of events. Note that many traditional fault trees include the Priority AND gate; most simply approximate with an AND gate

Functional Dependency gate

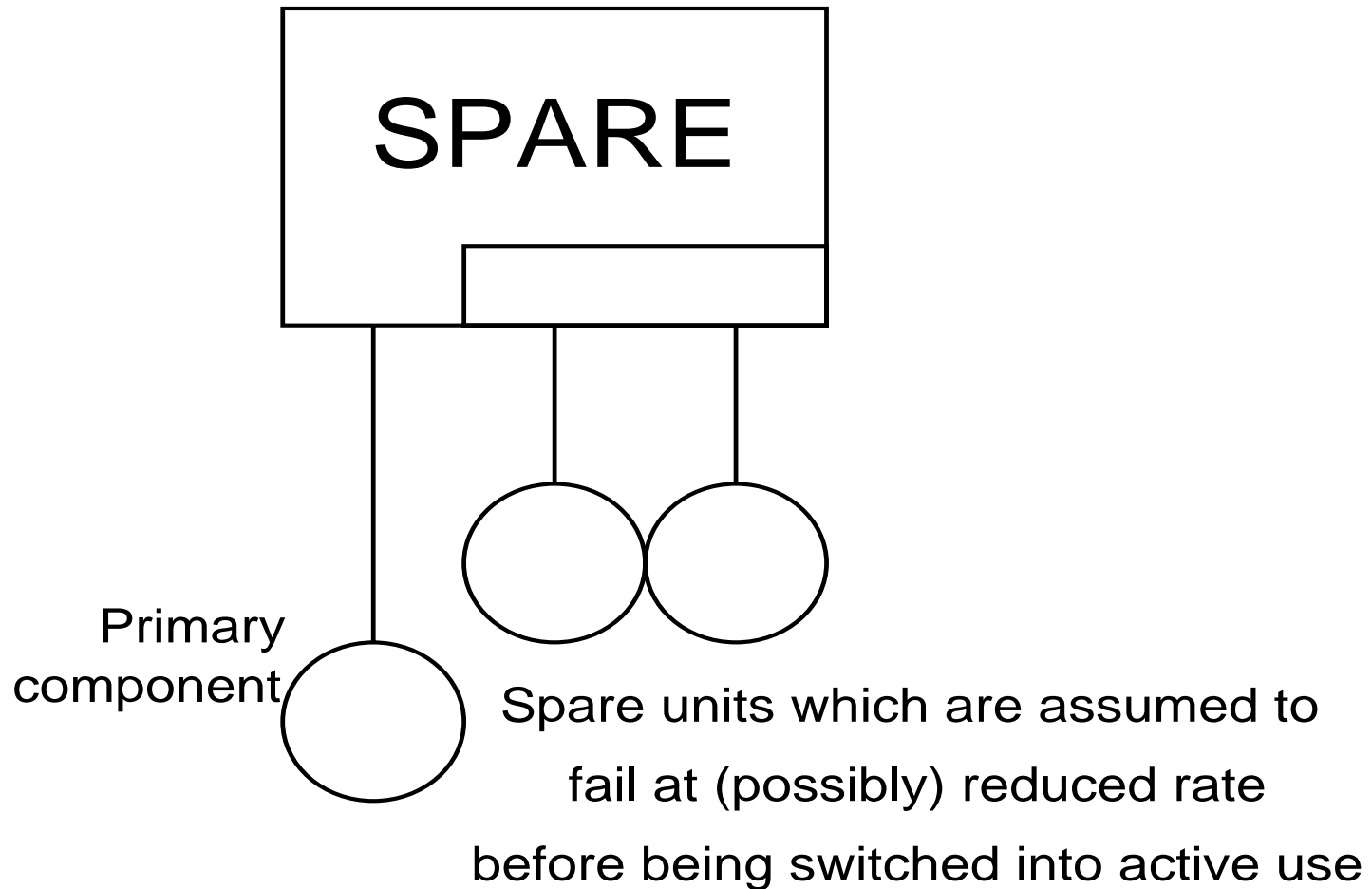
FDEP produces no logical output.

Its only effect is on propagating failures.



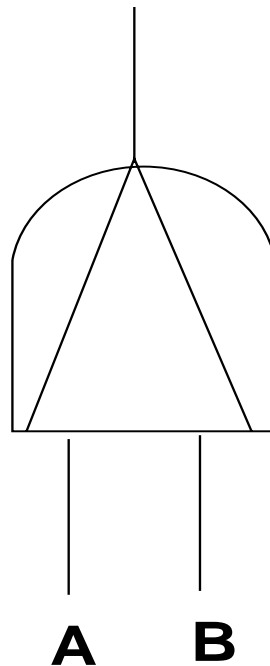
dependent basic events
are forced to occur when trigger
event occurs

Spare gate



Priority-AND gate

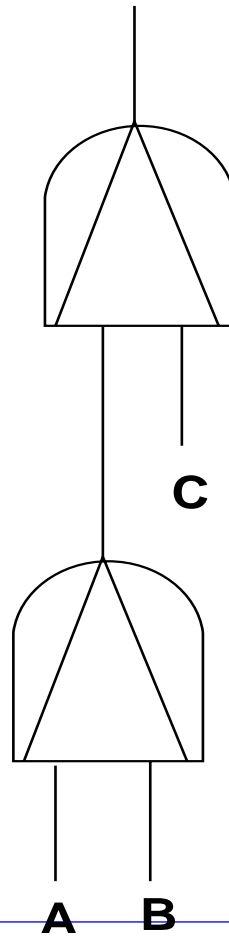
Output occurs if A and B occur, and A occurs before B



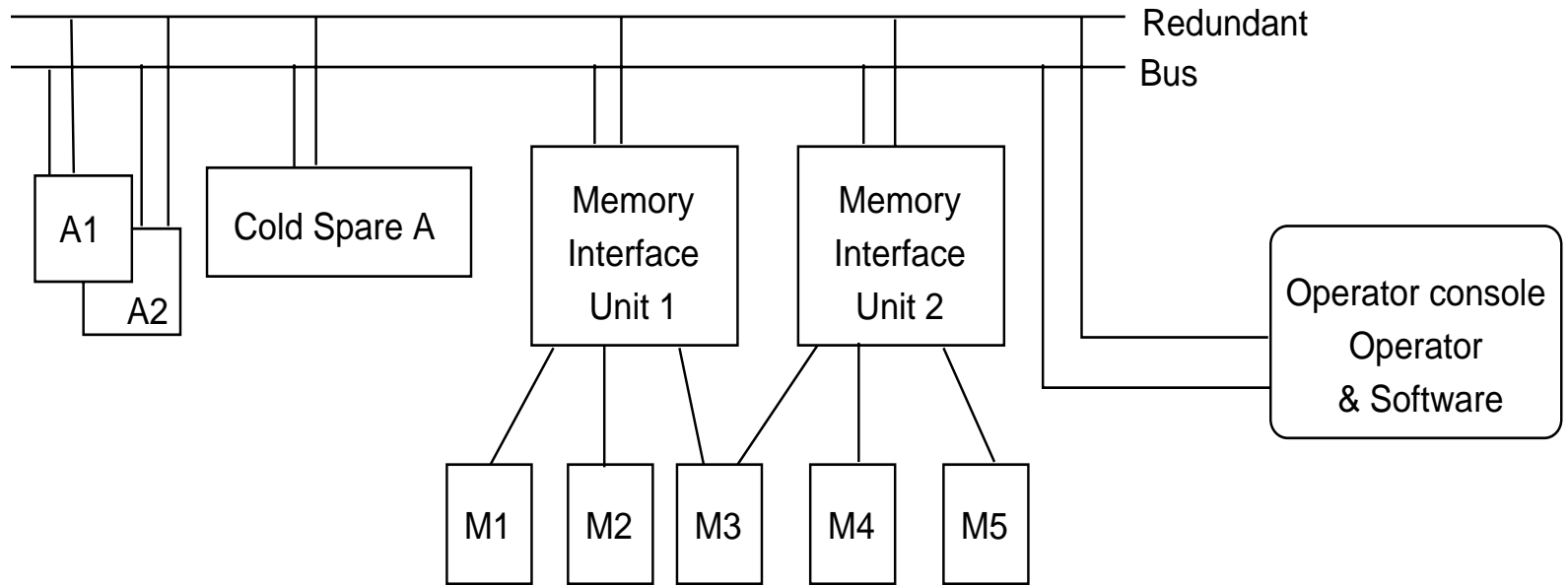
Inputs (may be subtrees)
which may occur in any order

Cascading Priority-AND gates

A before B before C



HECS: Hypothetical Example Computer System



HECS system description

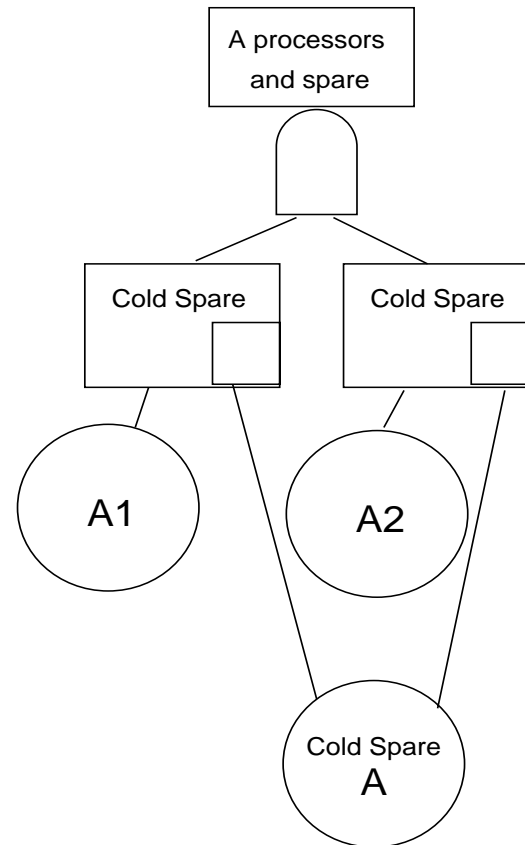
HECS consists of dual-redundant processors A1 and A2 and a cold spare which can replace either upon failure. A cold spare is one which is assumed not to fail before being used.

HECS has 5 memory units; three are required. These memory units are connected to the bus via two memory interface units. If the memory interface unit fails, the memory units connected to it are unusable. Memory unit 3 (M3) is connected to both interfaces for redundancy; thus M3 is accessible as long as either interface unit is operational.

There is also a human operator who interfaces with the system via a console, and runs some software application.

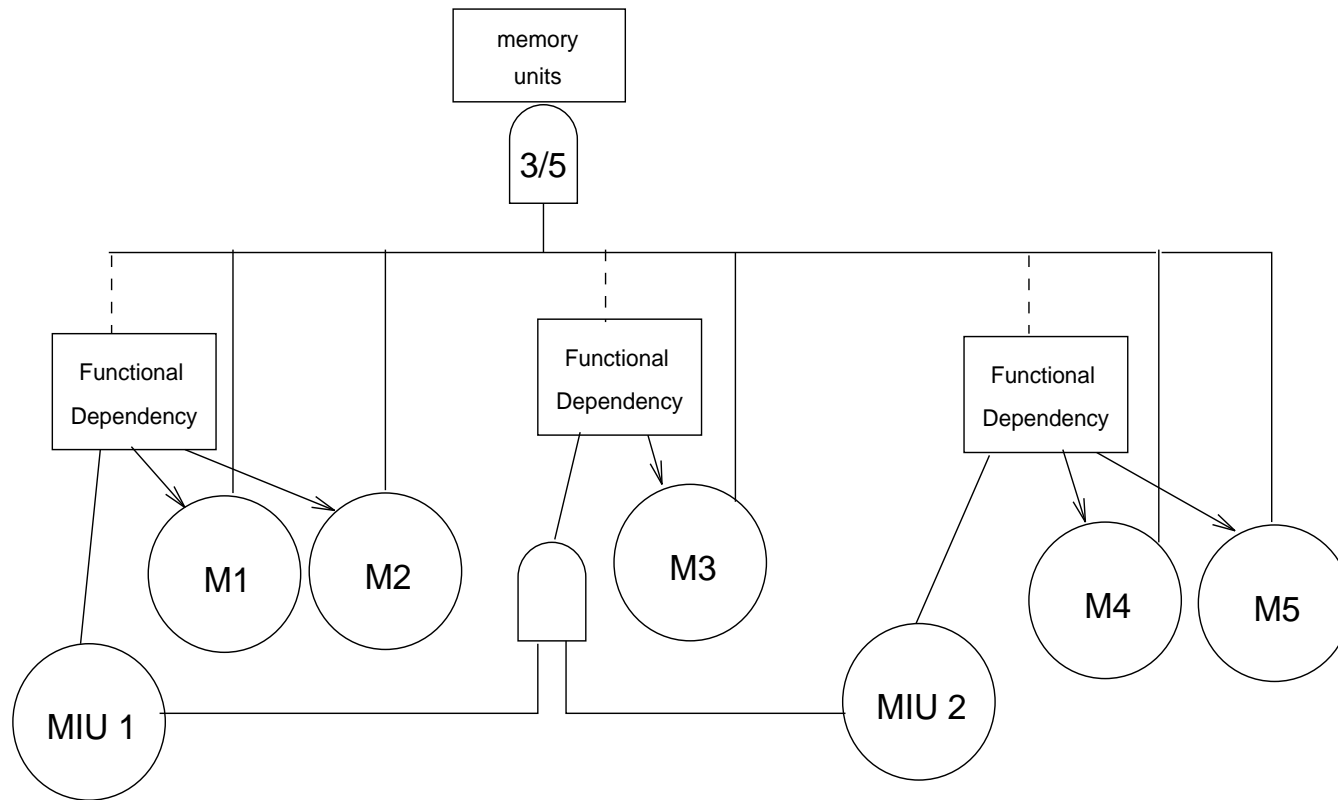
HECS requires at least one of the three A processors, at least 3 of the memory units, at least one of the redundant busses, and the operator, console and software to be operating correctly.

Modeling the cold spares

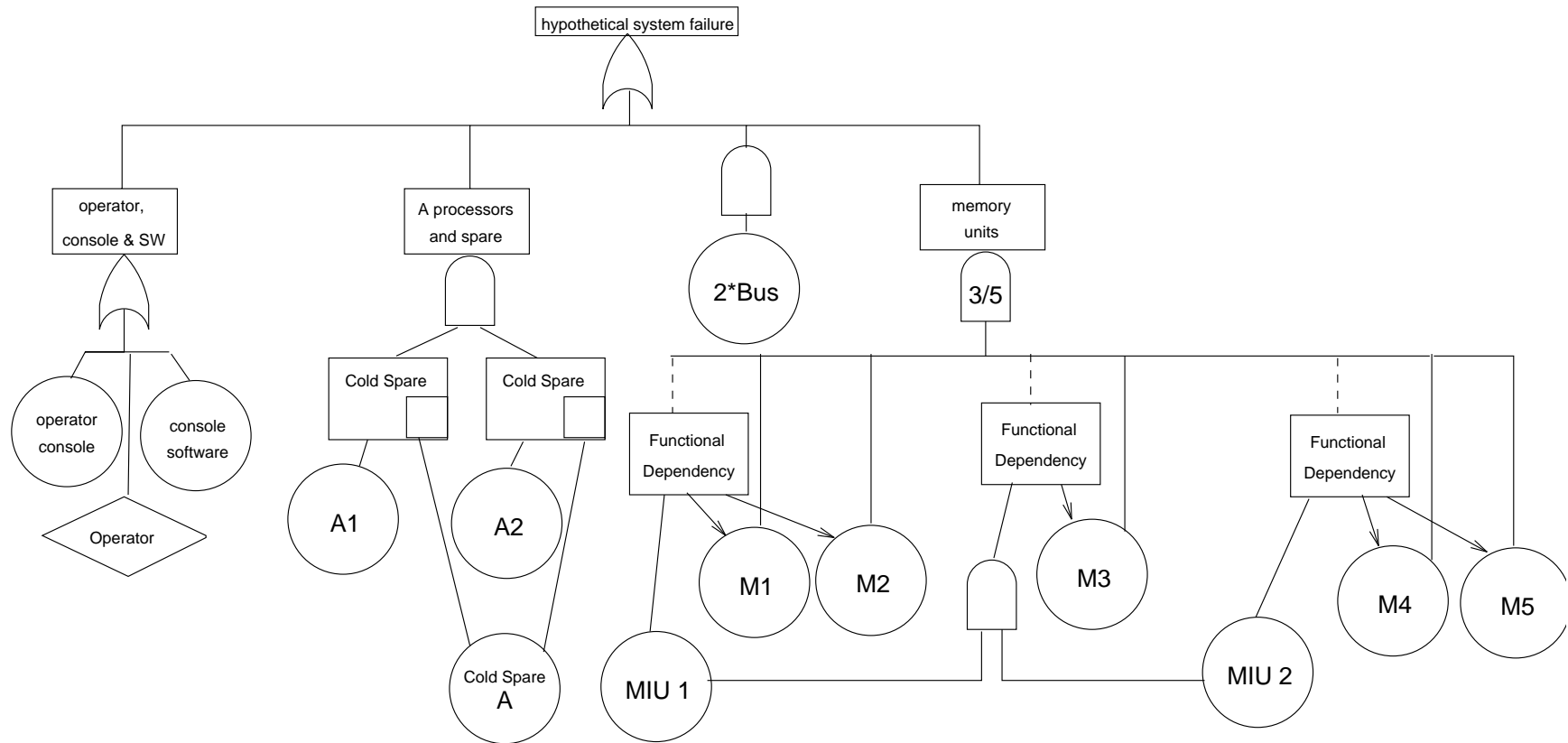


Notice that the cold spare is shared between the two processors. First to fail is replaced with the spare; the spare is then unavailable if the other fails.

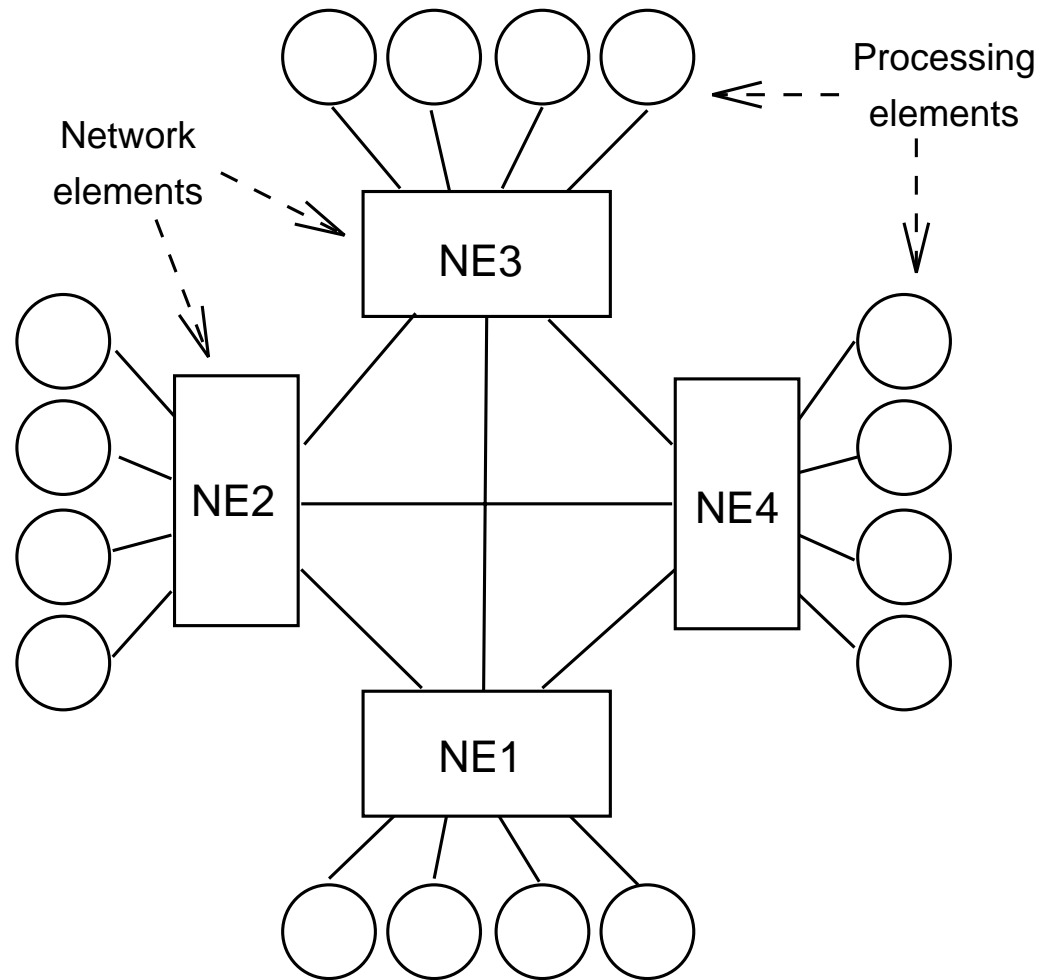
Modeling the memory units



HECS system-level fault tree model

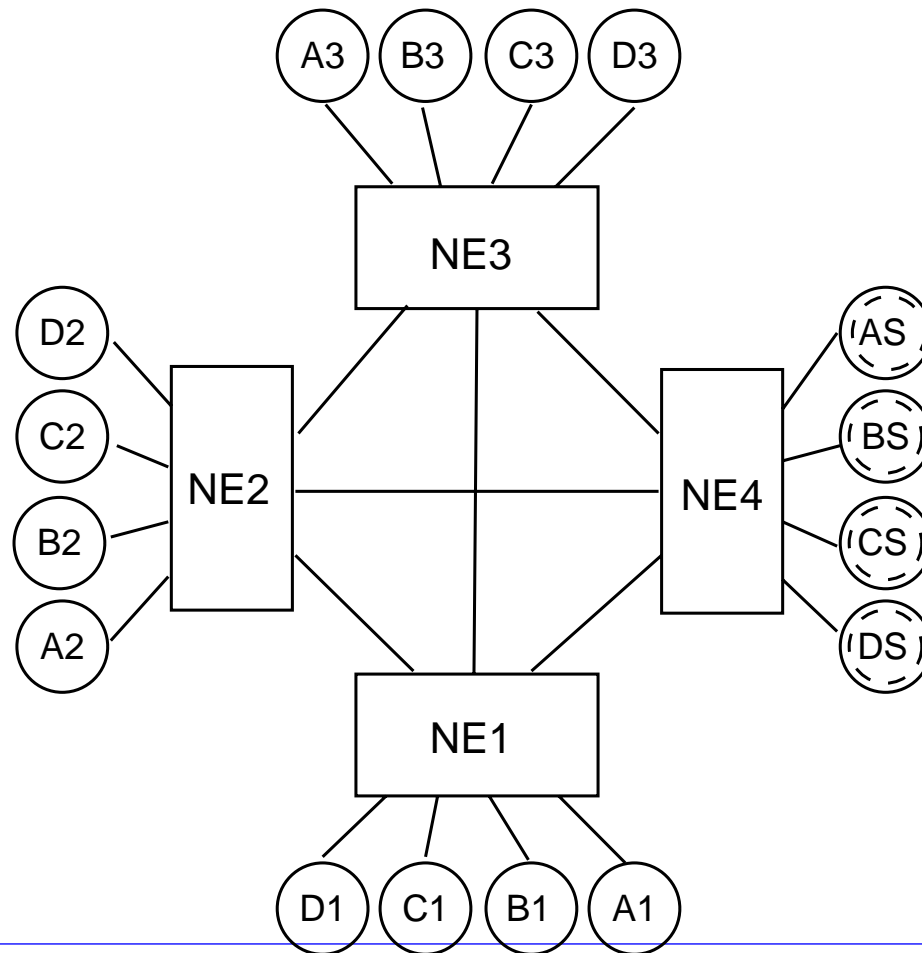


Example: Fault Tolerant Parallel Processor

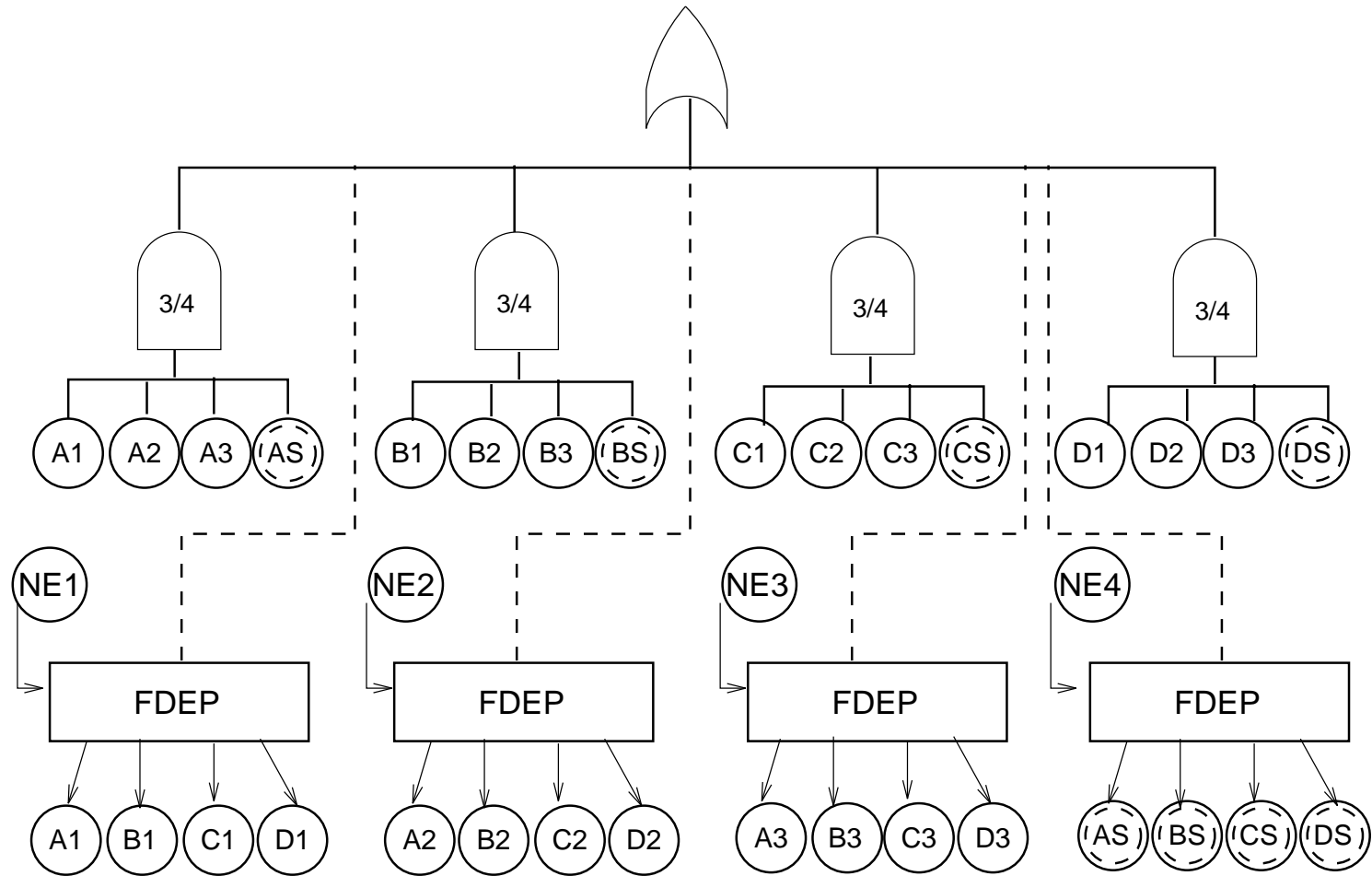


FTPP configuration #1

One spare per triad

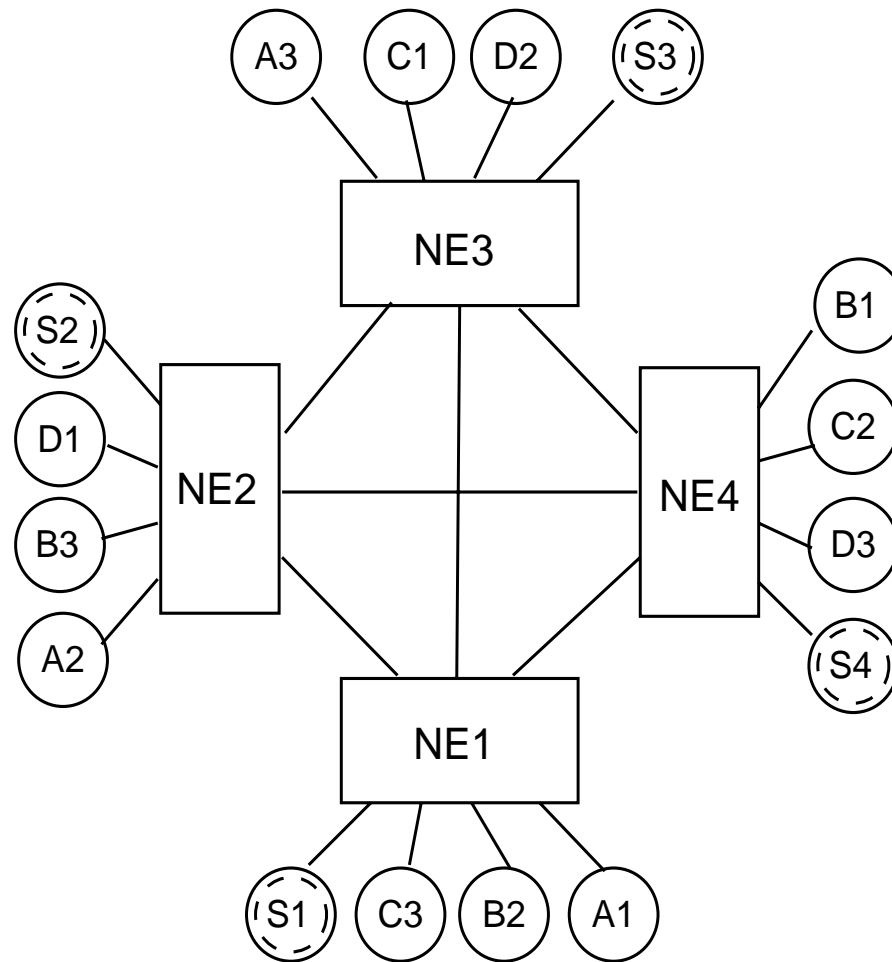


Fault tree for FTPP configuration #1



FTPP configuration #2

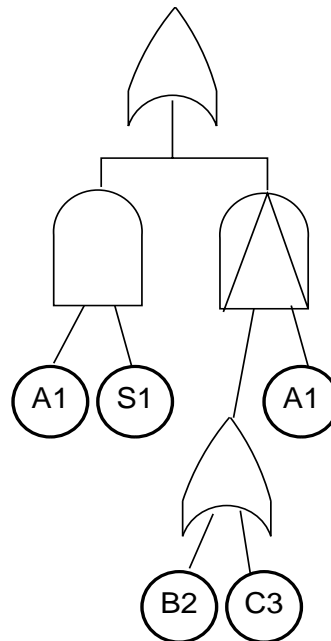
One spare per NE



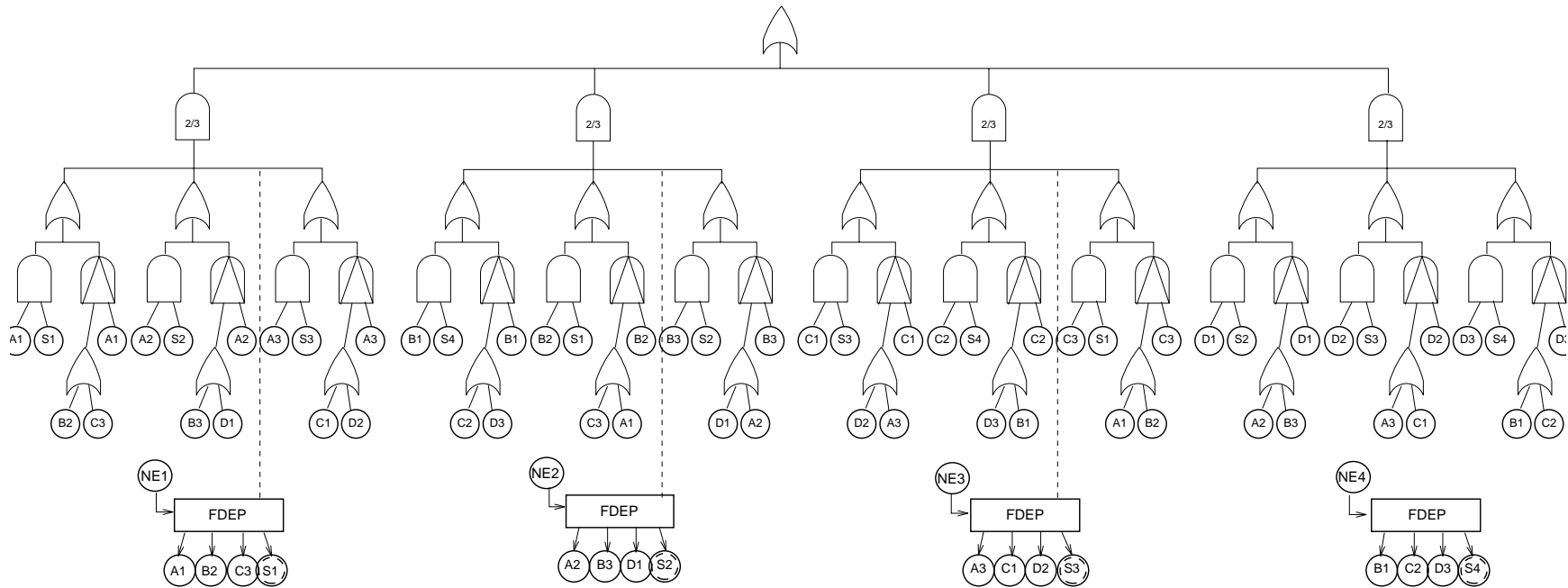
Failure conditions for FTPP configuration #2

Consider as an example, the first member of the A triad, specifically component A1. Now A1 will fail if

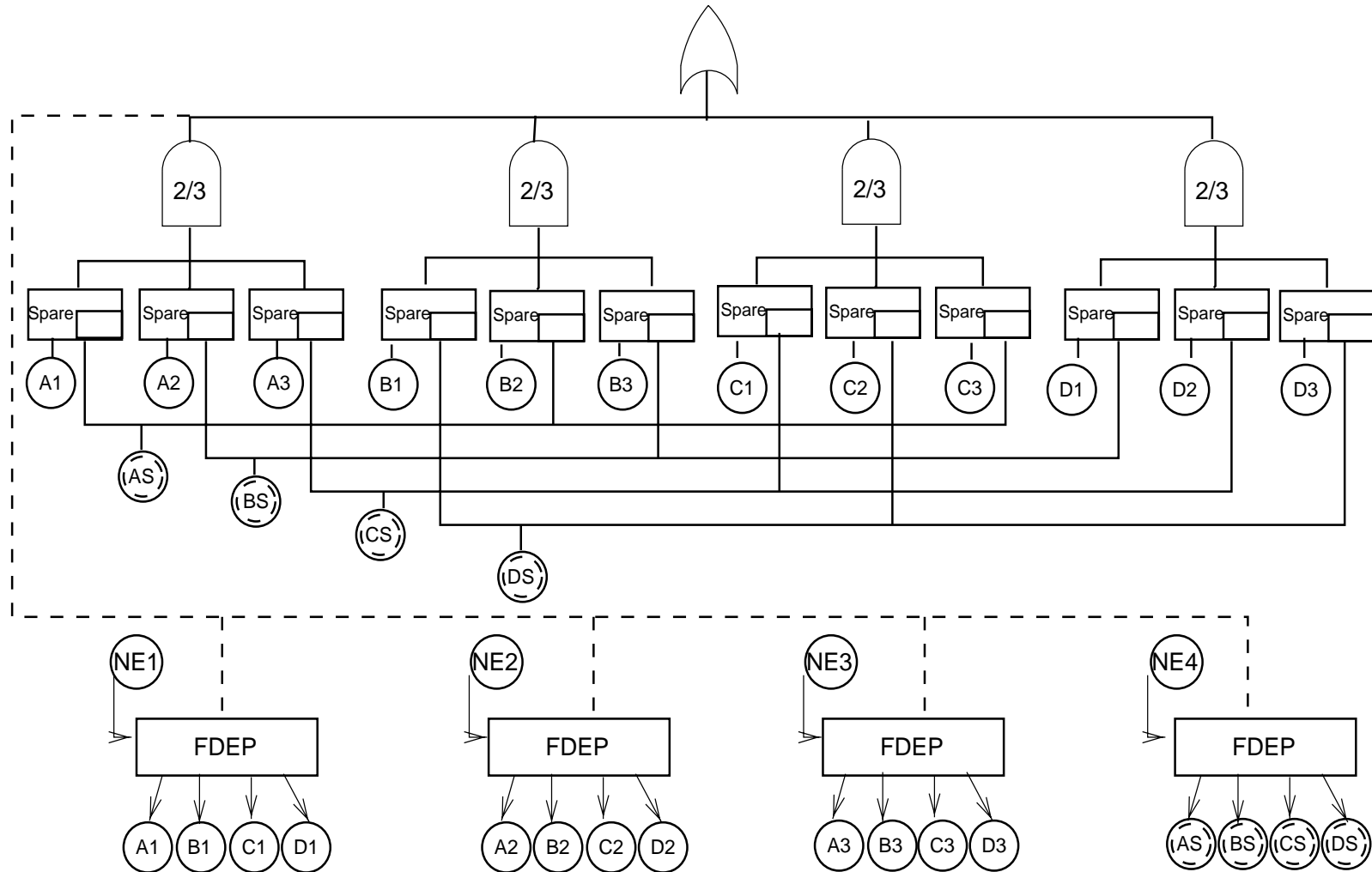
- both A1 and its spare (S1) fails OR
- if either of the other processors on the same NE fail before A1 does, thus using the spare first. In this case there will be no spare available when A1 fails.



Fault tree for FTPP configuration #2



Alternative Fault Tree for configuration #2



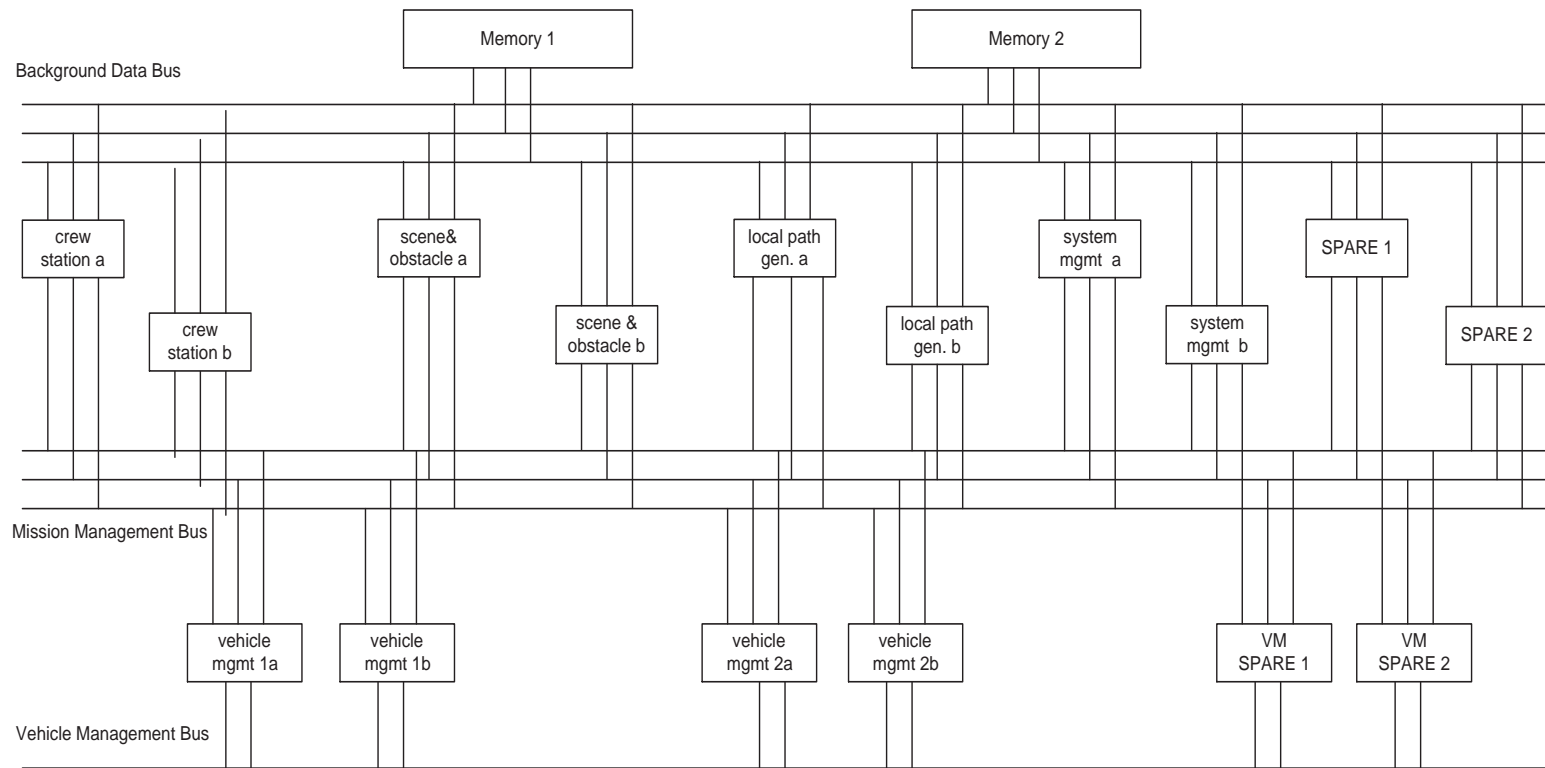
Mission Avionics System Example

The success/failure of the system is driven by the need to provide certain software functionality

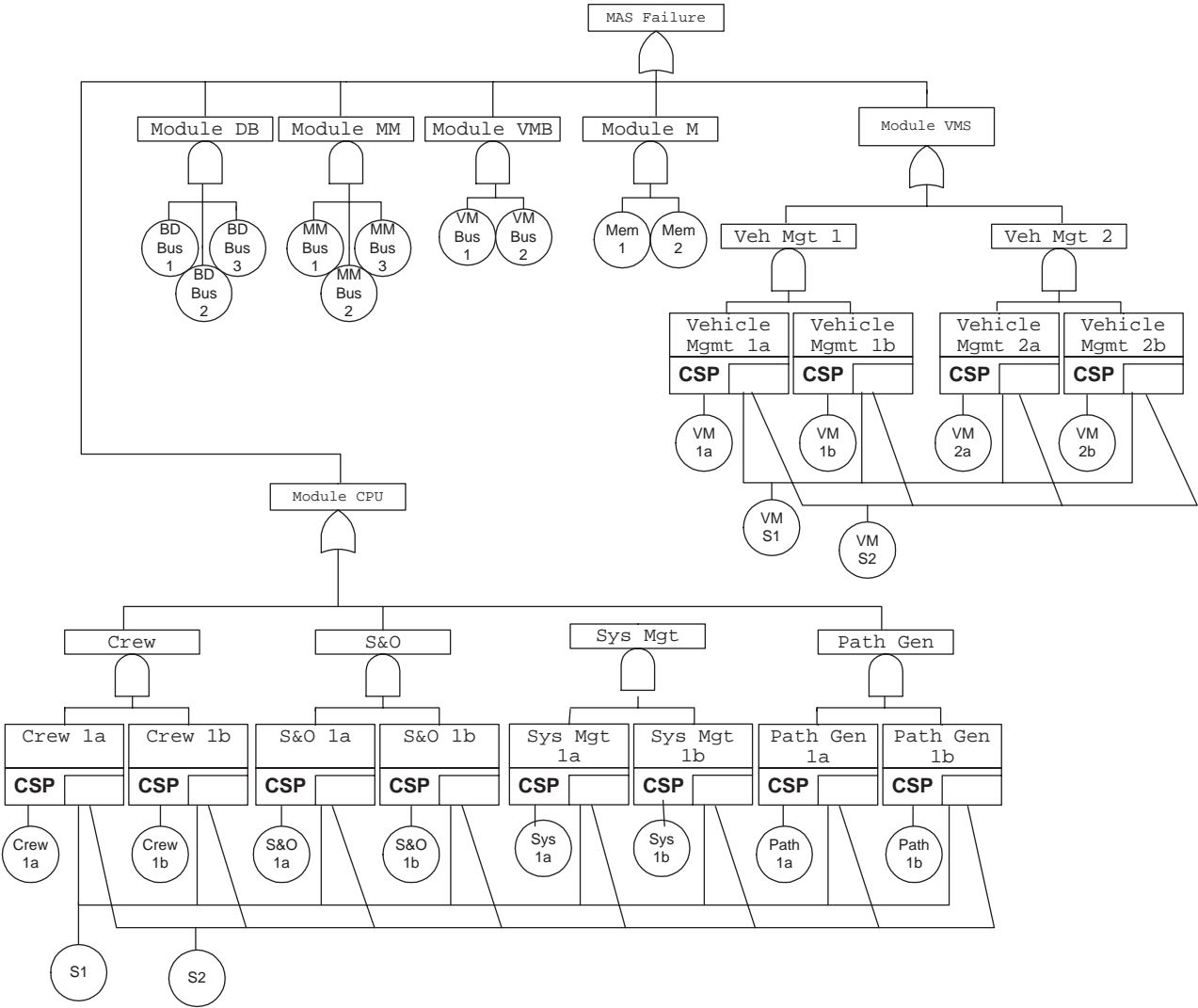
- crew station management
- scene & obstacle processing
- local path generation
- system management functions
- vehicle management

Fault tolerance is achieved via redundant processors (hot spares), pools of cold spares and redundant buses.

MAS system architecture



Fault Tree model for MAS



Redundant Software Architecture

Next Consider the path generation (PathGen) and scene&obstacle (S&O) functions:

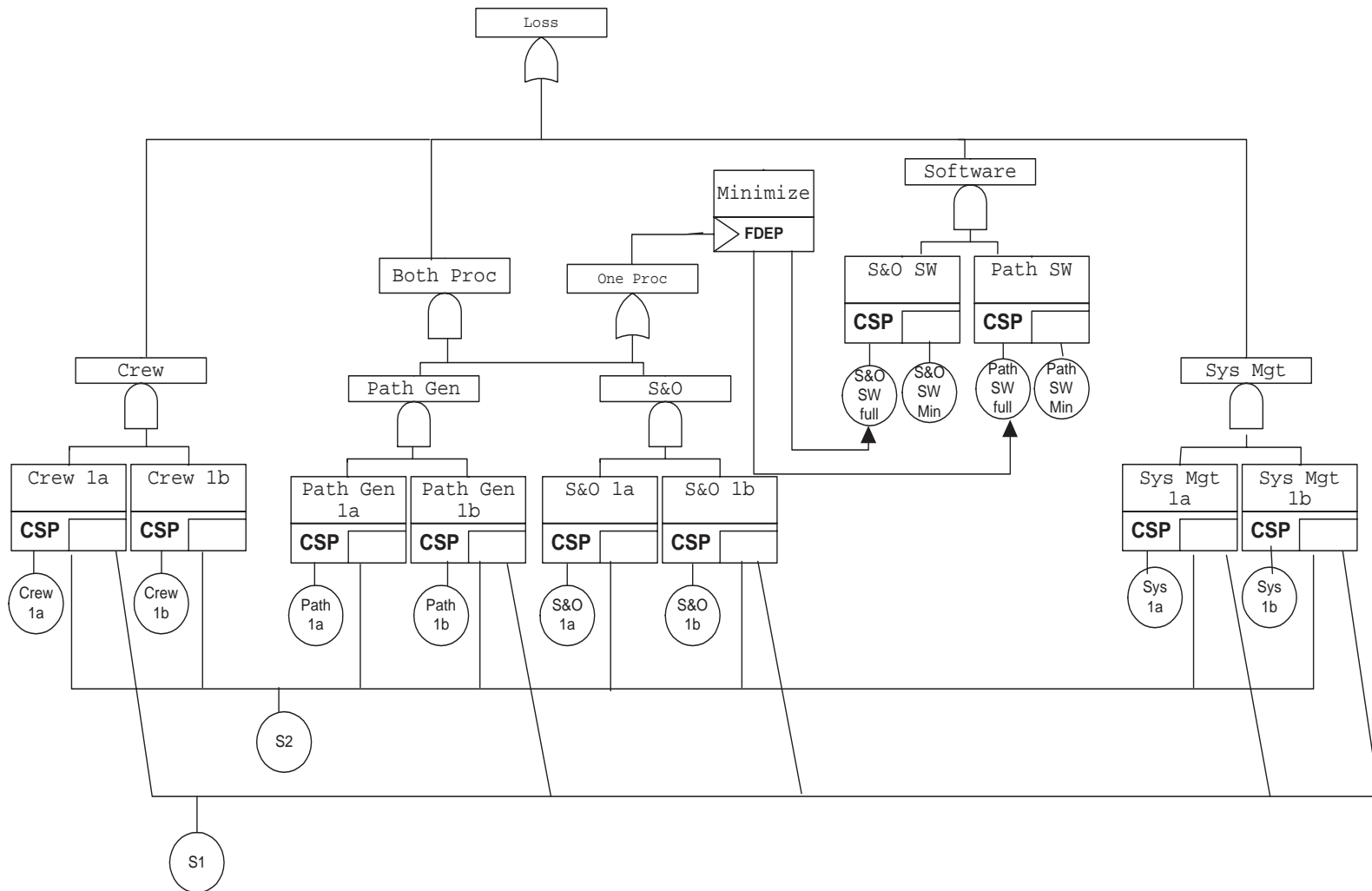
Each function needs a single processor to provide full functionality.

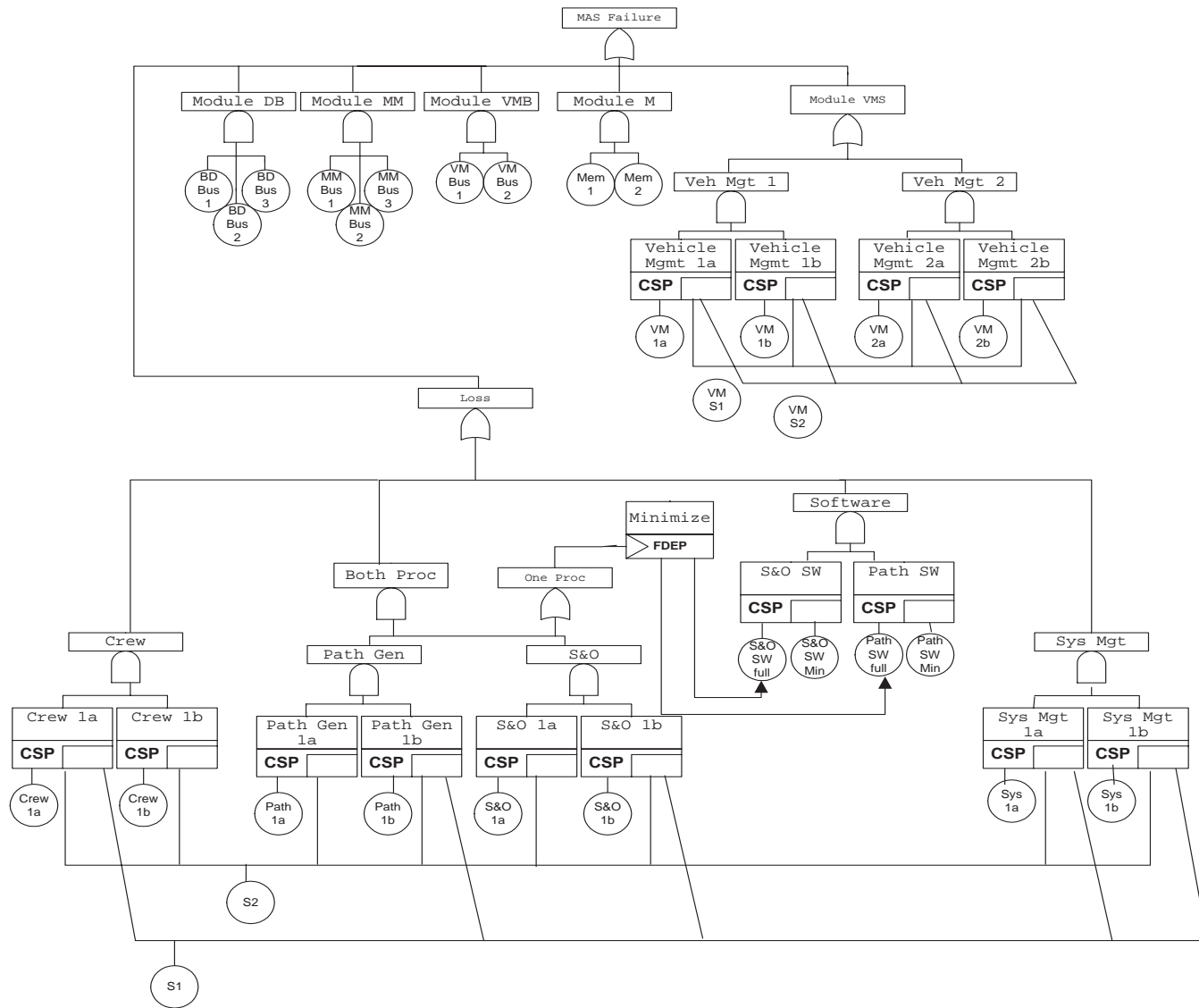
There is also a reduced version of each function that can provide minimum functionality (PathGenMin and S&OMin).

In the event of a detected software fault in PathGen the system can switch to PathGenMin (similarly for S&O)

Further, if there are no longer 2 full processors available, the system will switch to PathGenMin and S&OMin running on a single processor.

Redundant Software MAS model

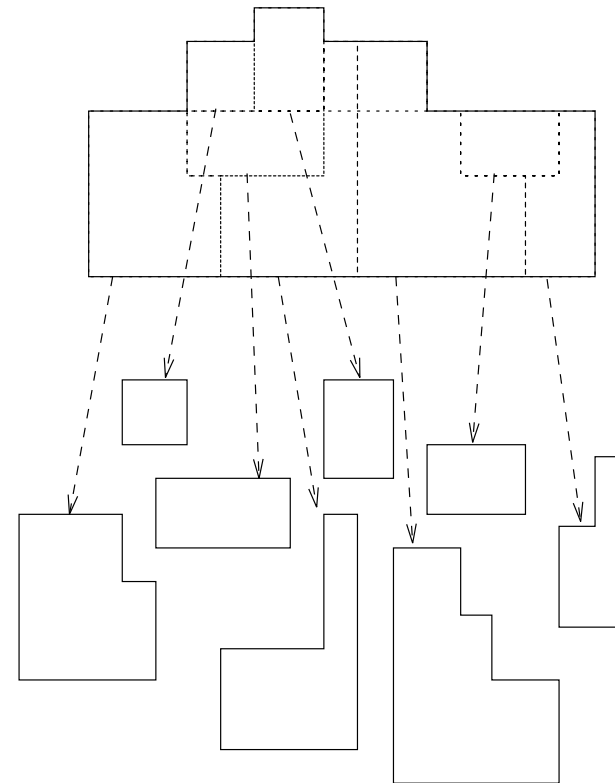
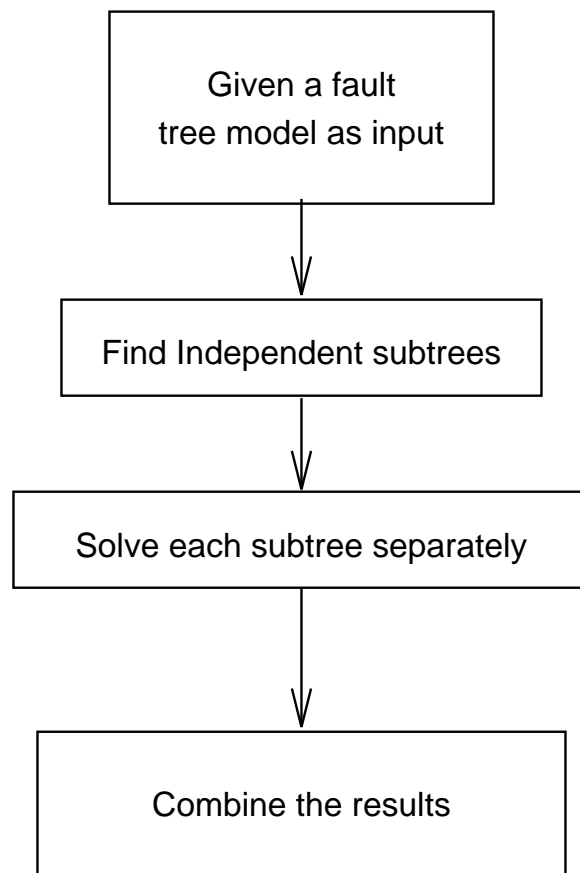




Presentation Outline

- I. Introduction to fault trees
- II. Fault tree analysis of an example control system
- III. Fault trees as design aid for software systems
- IV. Adapting the fault tree to analysis of computer-based systems
- V. Dynamic fault trees for modeling sequential behavior
- VI. Modular approach to fault tree analysis**
- VII. Sensitivity analysis
- VIII. Summary and Conclusions

Modular Approach to fault tree analysis



The best of both worlds

Divide-and-conquer helps produce models that may not be too large to solve.

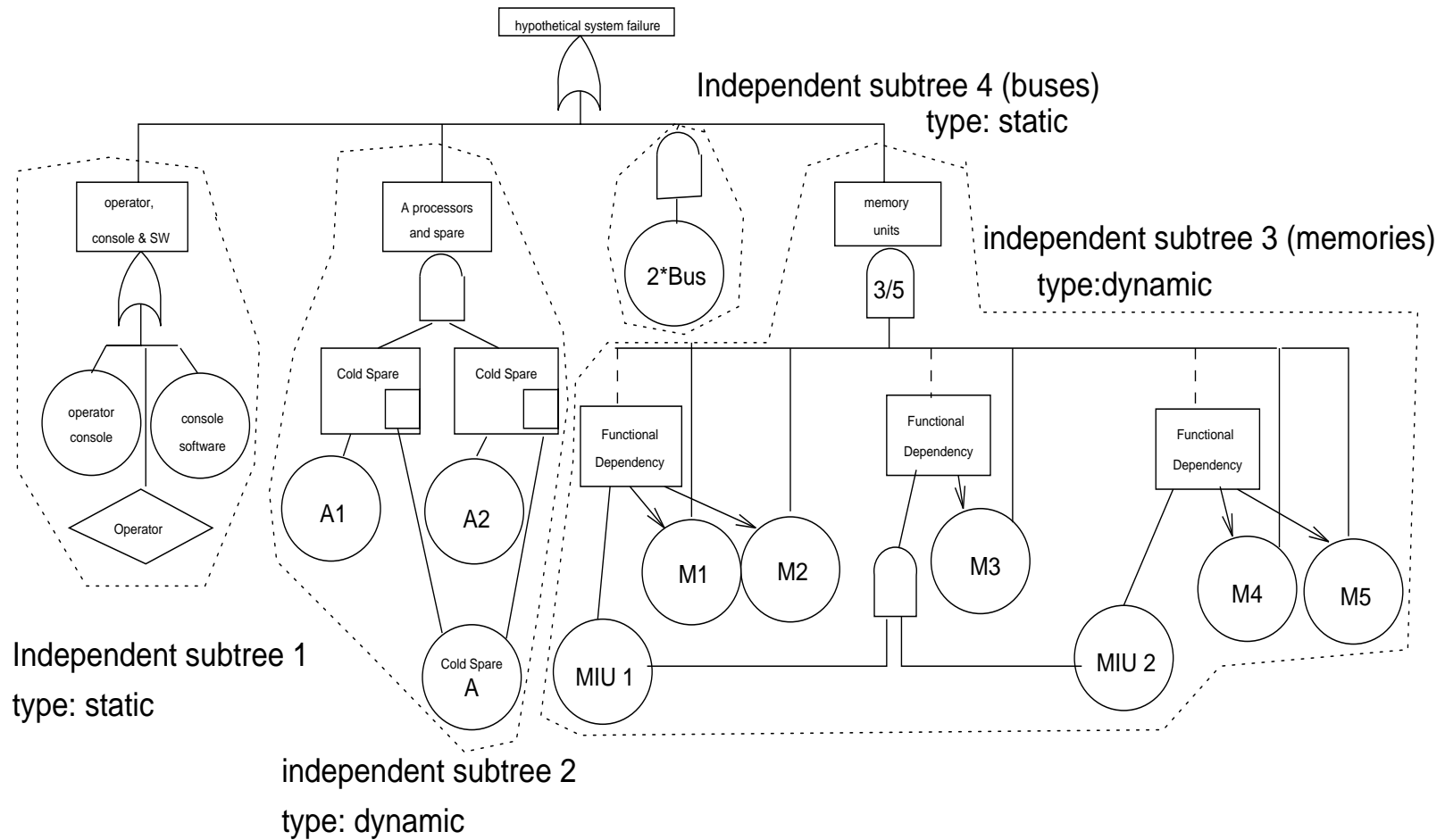
For static modules (containing AND, OR, K-of-M) gates, use the fast and efficient BDD (Binary Decision Diagram) approach.

For dynamic modules, convert to equivalent Markov model for solution.

Different solution methods can aid in validation and testing.

Modularization allows consideration of different solution methods (i.e. simulation).

Modular Solution of HECS



Presentation Outline

- I. Introduction to fault trees
- II. Fault tree analysis of an example control system
- III. Fault trees as design aid for software systems
- IV. Adapting the fault tree to analysis of computer-based systems
- V. Dynamic fault trees for modeling sequential behavior
- VI. Modular approach to fault tree analysis
- VII. Sensitivity analysis**
- VIII. Summary and Conclusions

Sensitivity analysis

Reliability Analysis tells only part of the story.

“What are the weak points in the system?”

“How do my results change with changing input parameters?”

“What is the most cost-effective way to improve reliability?”

These questions require sensitivity analysis of reliability analysis results.

Modular approach to sensitivity analysis

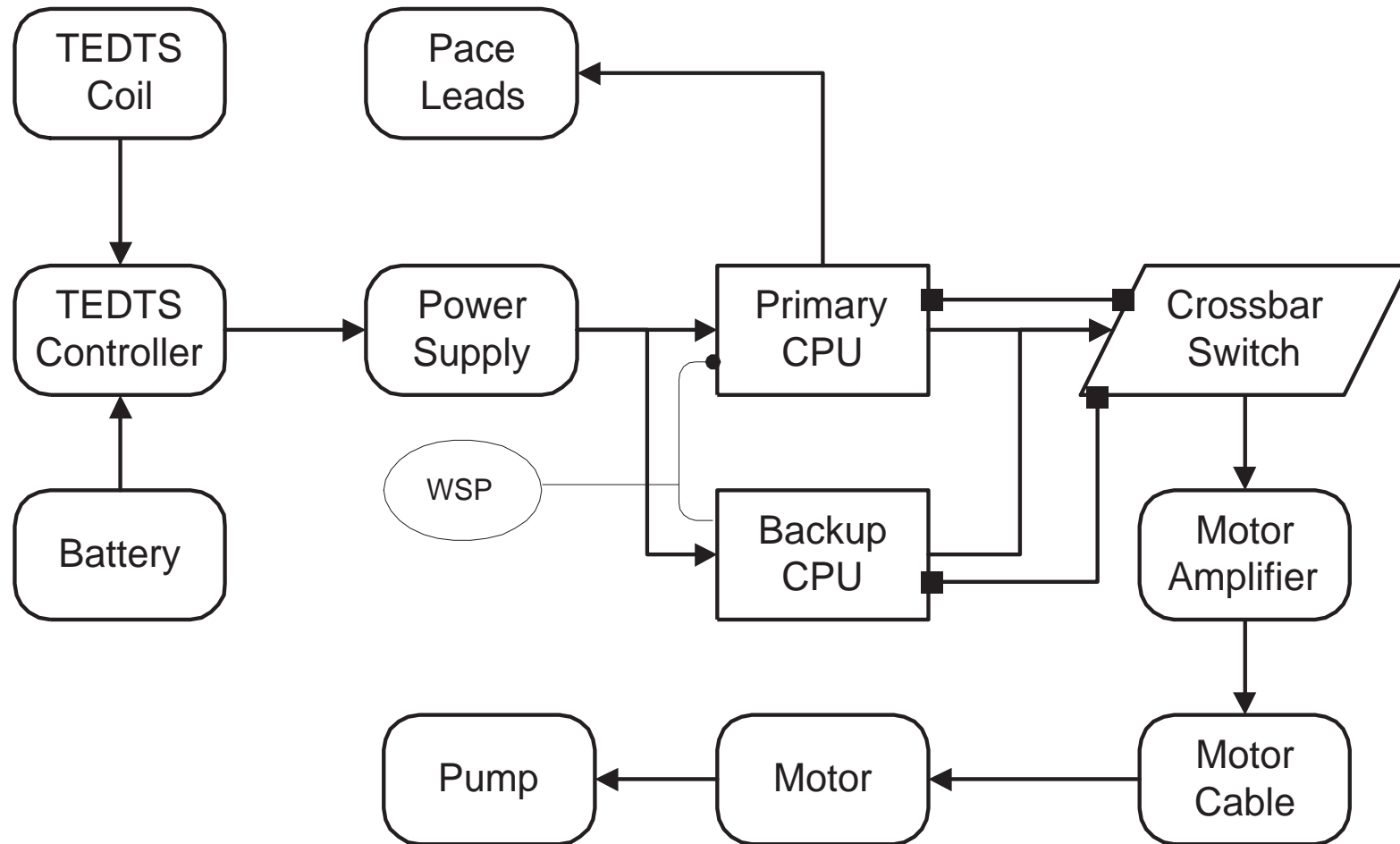
Sensitivity analysis (also called importance analysis) can use partial derivative.

Sensitivity results from different submodels can be easily combined using “chain-rule”.

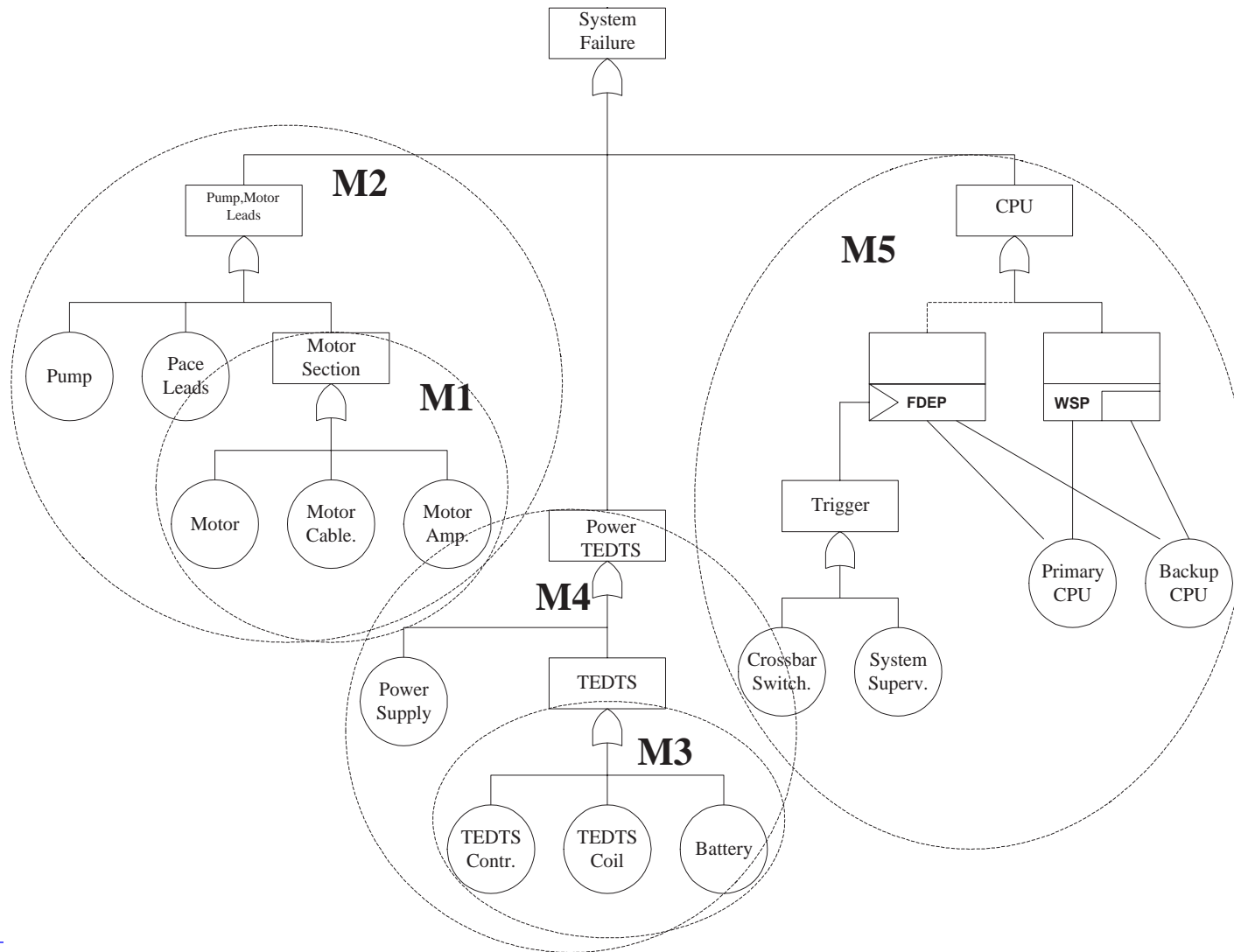
Sensitivity analysis for BDD is almost free while calculating reliability.

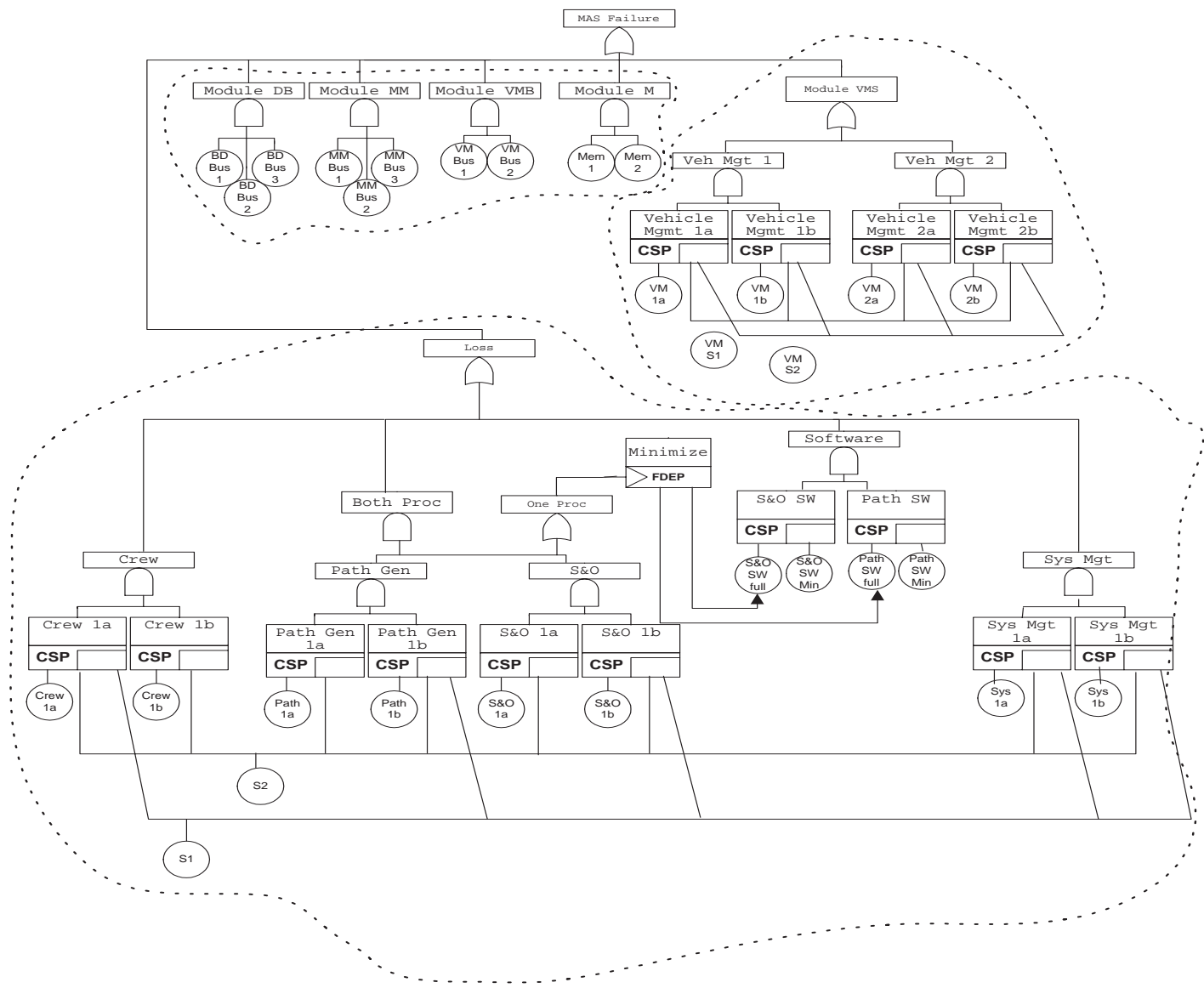
Sensitivity analysis for Markov chain is more troublesome but we have developed an interesting, efficient approximation.

Example: Cardiac Assist System



Cardiac Assist System Fault Tree





Summary

The *DFT* (dynamic fault tree) methodology is ideally suited for the analysis of computer-based systems.

DFT uses a modular approach to FTA, detecting modules using a fast and efficient algorithm.

Modules are classified as static or dynamic, depending on the types of gates included.

Static modules are solved using the BDD approach; dynamic modules are solved using Markov chain methods.

Coverage models can assess the effect of complex recovery mechanisms.

Dynamic gates can allow modeling of sequence dependencies that arise from complex redundancy management.

Software for Dynamic Fault Tree Analysis

Galileo/ASSAP is a software package for fault tree analysis which embodies the DFT approach.

(Being developed for NASA Langley Research Center, expected completion Nov. 2001. Beta version available for evaluation.)