SOFTWARE ENGINEERING 90
RICIS SYMPOSIUM

JOHNSON GRANT
IN-61-CR
OVERRIDE
332280
P.110

SOFTWARE SAFETY

Dr. Nancy Leveson

Information and Computer Science Dept.
University of California, Irvine
Irvine, California 92717
(714) 856-5517
e-mail: nancy@ics.uci.edu

July, 1987

(NASA-CR-187920) SOFTWARE SAFETY (California Univ.) 110 p CSCL 09B

N91-17623

Unclas
G3/61 0332280

OUTLINE

INTRODUCTION TO SOFTWARE SAFETY

- What is Software Safety?
- What is its relationship to other software qualities?

Real-Time Safety-Critical Systems

When computers are used to control complex, time-critical mechanical devices or physical processes such as:

Air Traffic
Nuclear Fission
Hospital Patient Monitoring
Defense and Aerospace Systems

where a run-time error or failure can result in death, injury, loss of property, environmental harm.

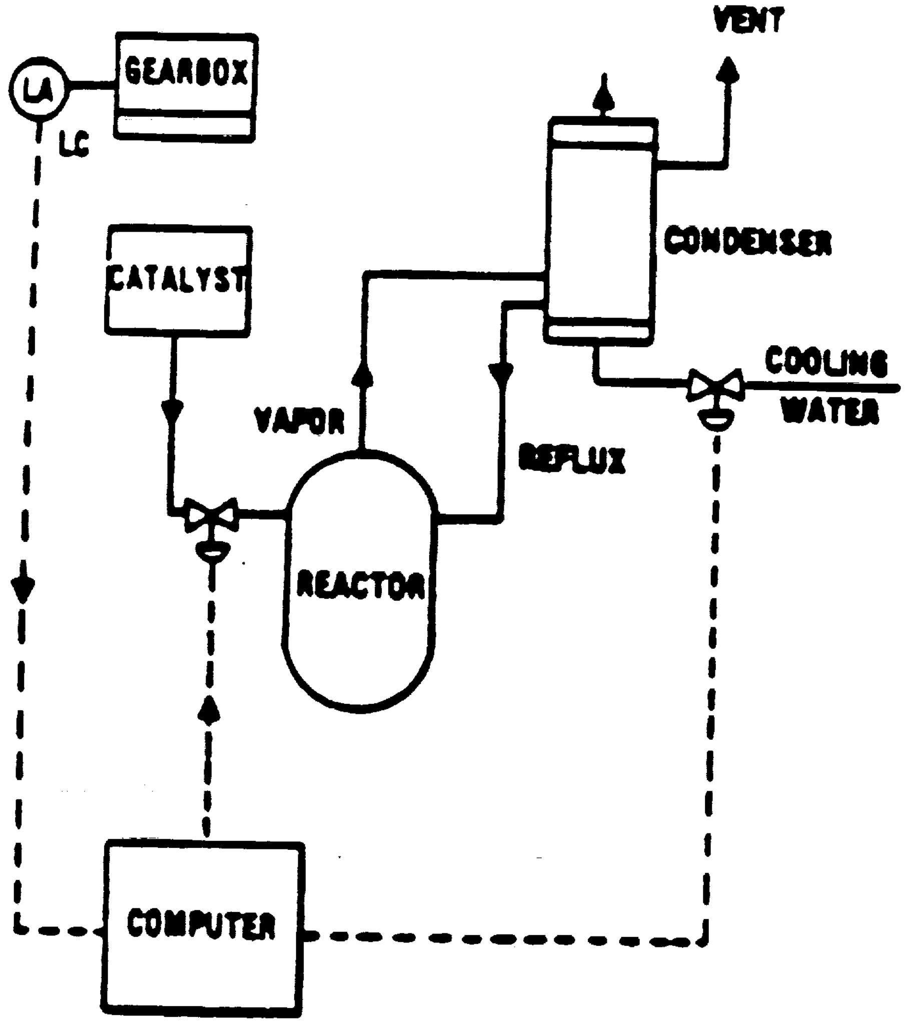


Figure 1. A computer-controlled batch reactor

Facts about accidents:

- Most accidents originate in system interfaces. Caused by complex, unplanned interactions between components of the system.
- Accidents often involve multiple failures of different components.
- System accidents intimately intertwined with complexity and coupling.

COMPUTER-INVOLVED ACCIDENTS:

Almost all due to inadequate design foresight and requirements specification

- incomplete or wrong assumptions about about operation of controlled system or required operation of computer
- unhandled controlled-system states and environmental conditions

Goals in building complex systems:

- attaining mission (functional correctness)
- preventing undesired events while attempting to achieve the mission (safety, security)
- minimizing cost

Focusing on safety separately from other goals:

- Allows for conflict resolution and decisions about tradeoffs
- Allows differential handling of erroneous states
- Provides discipline and procedures for looking for errors
- Focuses attention and assigns responsibility
- Allows measuring and ensuring safety separately from other goals

Implications and Challenges for Software Engineering

- Requirements for software safety analysis and verification being included in contracts and by government licensing agencies.
- New standards for safety-critical software.
- National and international working groups
- Safety involves multiple areas of traditional software research along with safety engineering.

  reliability

  security

SYSTEM RELIABILITY: considers problems concerned with ensuring system will perform a required task or mission for a specified time in a specified environment.

SYSTEM SAFETY: considers problems of not causing an accident in the process.

- Usually many system failures which can occur without causing a mishap

- Sometimes even conflicts between functional and safety requirements

Relationship between Safety and Security

- Both involve threats
- Both are negative requirements
- Both are system qualities
- Both may require high levels of assurance

Are they the same?

advertent vs. inadvertent actions

BASIC SYSTEM SAFETY PRINCIPLES

- What is System Safety?
- System Safety Analysis Techniques

SYSTEM SAFETY ENGINEERING

The application of scientific, management, and engineering principles to ensure adequate safety within the constraints of operational effectiveness, time, and cost throughout the system life cycle.

HAZARD ANALYSIS

- Identify hazards
- Assess risk

HAZARD CONTROL

- Eliminate hazards
- Minimize hazard occurrence or effects
- Document and track hazards and progress made toward resolution of associated risk.

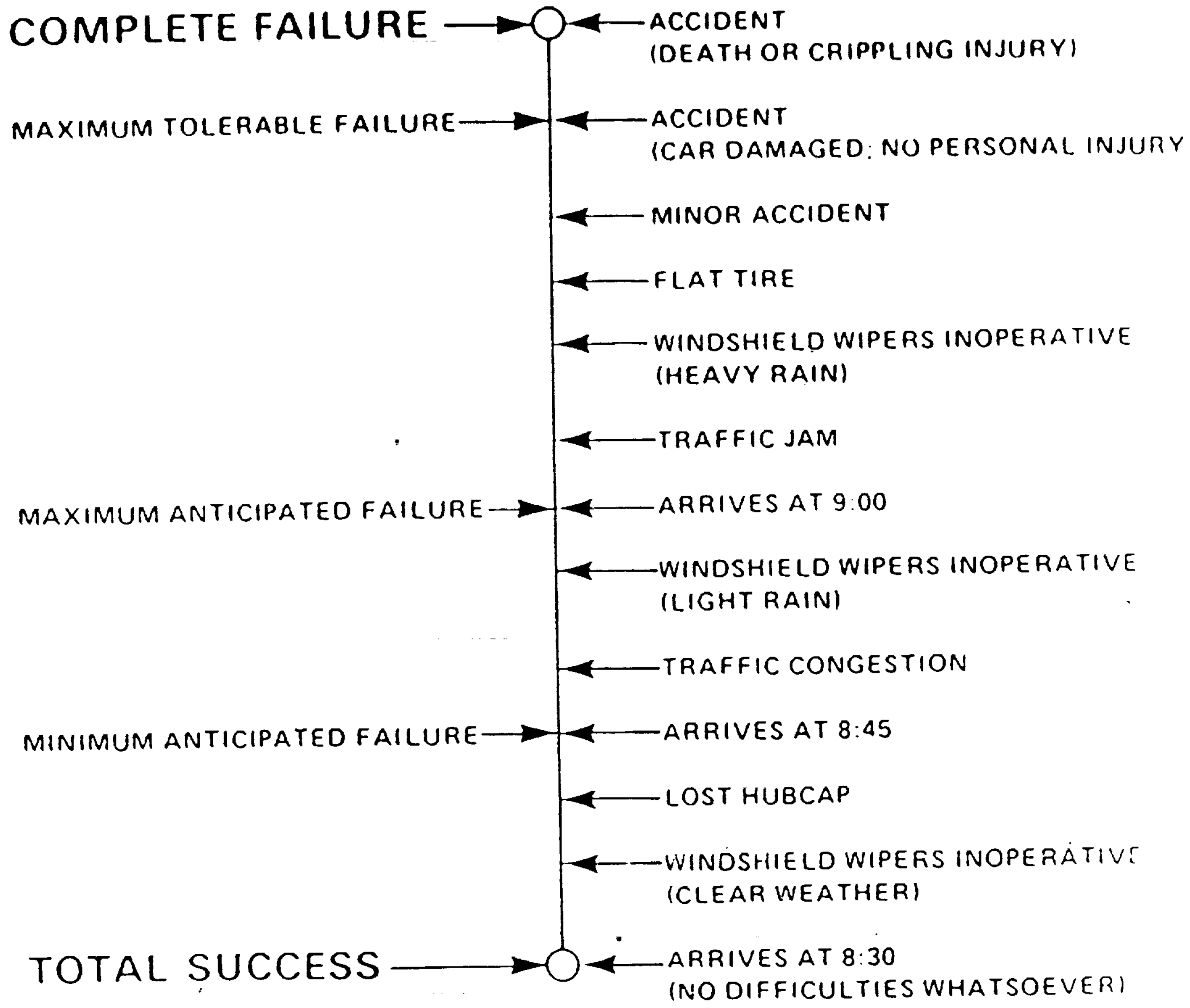
Preliminary Hazard Analysis (PHA):

- identify safety-critical areas and functions
- identify and evaluate hazards in terms of severity and likelihood
- identify safety design criteria to be used

Results used in:

developing system safety requirements

preparing performance and design specifications


COMPLETE FAILURE
ACCIDENT (DEATH OR CRIPPLING INJURY)
MAXIMUM TOLERABLE FAILURE
ACCIDENT (CAR DAMAGED; NO PERSONAL INJURY
MINOR ACCIDENT
FLAT TIRE
WINDSHIELD WIPERS INOPERATIVE (HEAVY RAIN)
TRAFFIC JAM
MAXIMUM ANTICIPATED FAILURE
ARRIVES AT 9:00
WINDSHIELD WIPERS INOPERATIVE (LIGHT RAIN)
TRAFFIC CONGESTION
MINIMUM ANTICIPATED FAILURE
ARRIVES AT 8:45
LOST HUBCAP
WINDSHIELD WIPERS INOPERATIVE (CLEAR WEATHER)
TOTAL SUCCESS
ARRIVES AT 8:30 (NO DIFFICULTIES WHATSOEVER)

HAZARD SEVERITY

MIL-STD-882B. System Safety Program Requirements:

Category I. Catastrophic: may cause death or system loss.

Category II. Critical: may cause severe injury, severe occupational illness, or major system damage.

Category III. Marginal: may cause minor injury, minor occupational illness, or minor system damage.

Category IV. Negligible: will not result in injury, occupational illness, or system damage.

NHB 5300.4 (1.D.1), a NASA document:

Category 1. Loss of life or vehicle (includes loss or injury to public).

Category 2. Loss of mission (includes both postlaunch abort and launch delay sufficient to cause mission scrub).

Category 3. All others.

DOE 5481.1 (nuclear):

Low. Hazards that present minor on-site and negligible off-site impacts to people or the environment.

Moderate. Hazards that present considerable potential on-site impacts to people or environment, but at most only minor off-site impacts.

High. Hazards with potential for major on-site or off-site impacts to people or the enviromment.

HAZARD PROBABILITY

- Described in terms of occurrences per unit of time, events, population, items, or activity.
- Derive from modelling or from historical safety data from similar systems.

Subsystem Hazard Analysis (SSHA):

- Identify hazards associated with design of subsystems including:

  component failure modes

  critical erroneous human inputs

  hazards resulting from functional relationships between components of the subsystem

- Determine how operating or failure modes of components affects safety of the system.

- Identify necessary actions to determine how to eliminate or reduce risk of identified ~~actions~~ hazards.

- Evaluate design with respect to safety requirements of subsystem specification.

System Hazard Analysis (SHA):

- Identify hazards created by interfaces between subsystems or by system operating as a whole including human errors.

- Examines all subsystem interfaces for

  (a) compliance with safety criteria in system requirements specification.

  (b) possible combinations of independent, dependent, and simultaneous hazardous events or failures, including failures of controls and safety devices, that could cause hazards.

  (c) degradation of safety of system from normal operation of system and subsystems.

Operating and Support Hazard Analysis (OSHA):

- identify hazards and risk reduction procedures during all phases of system use and maintenance.

- especially examines hazards created by man/machine interface.

ANALYSIS TECHNIQUES

Design reviews and walkthroughs

Checklists

Fault Tree Analysis (FTA)

Event Tree Analysis (or Incident Sequence Analysis)

Hazard and Operability Studies (HAZOP)

Random Number Simulation Analysis (RNSA)

Failure Modes and Effects Analysis (FMEA)

INDUCTIVE APPROACHES	DEDUCTIVE APPROACHES
• reasoning from individual to general	• reasoning from general to specific
• determine <u>what</u> system states possible	• determine <u>how</u> given system state can occur
• postulate particular fault or initiating condition + attempt to ascertain effect on system operation	• postulate system has failed in certain way + try to find out what behavior could cause or contribute to it
e.g. how will loss of some particular control surface affect flight of aircraft	e.g. plane crashes, what could have caused it?
• for complex systems, impossible to identify all possible component failure modes	• difficult to find all causal sequences

• when subsystems put together, new failure modes may appear

SYSTEM SAFETY DESIGN PROCEDURES

GOAL: Eliminate identified hazards or, if not possible, reduce associated risk to an acceptable level.

Order of precedence for applying safety design techniques:

(1) Intrinsically safe design

(2) Prevent or minimize occurrence of hazards.

e.g. monitoring
automatic control (automatic pressure relief valves, speed governors, limit-level sensing controls)
lockouts
lockins
interlocks

(3) Control hazard if it occurs using automatic safety devices.

detection of hazards
fail-safe designs
damage control
containment
isolation of hazards

(4) Provide warning devices, procedures, and training to help personnel react to hazard.

Interlock - ensures a sequence of operations occurs in correct order

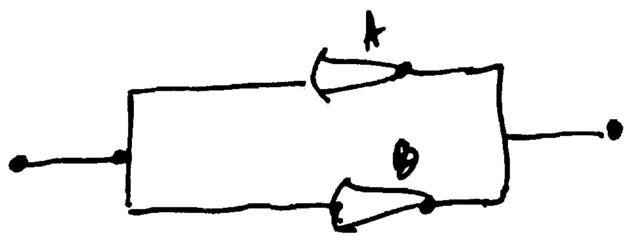
1) ensure event A does not occur inadvertently

2) ensure event A does not occur while condition C exists

3) ensure event A occurs before event D

FMEA example

1	2	3	4	5	
Component	Failure Probability	Failure Mode	% Failures by Mode	Effects Critical	Non-Critical
A	$1x10^{-3}$	Open	90		X
		Short	5	X ($5x10^{-5}$)	
		Other	5	X ($5x10^{-5}$)	
B	$1x10^{-3}$	Open	90		X
		Short	5	X ($5x10^{-5}$)	
		Other	5	X ($5x10^{-5}$)	



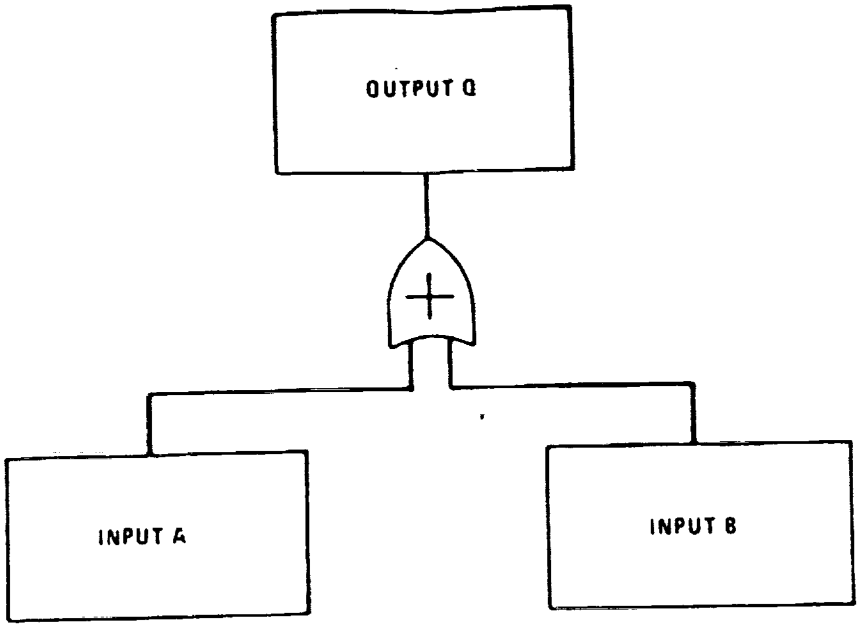


Figure IV-2. The OR-Gate

It is important to understand that causality never passes through an OR-gate. That is, for an OR-gate, the input faults are never the causes of the output fault. Inputs to an OR-gate are identical to the output but are more specifically defined as to cause. Figure IV-3 helps to clarify this point.

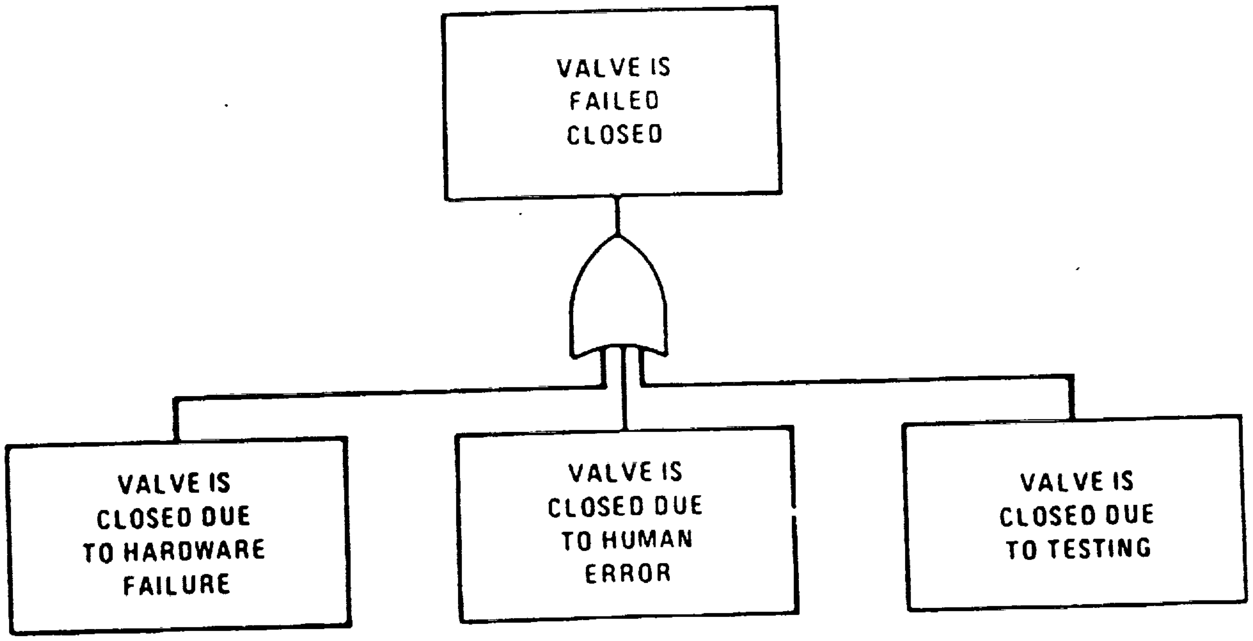

VALVE IS FAILED CLOSED
VALVE IS CLOSED DUE TO HARDWARE FAILURE
VALVE IS CLOSED DUE TO HUMAN ERROR
VALVE IS CLOSED DUE TO TESTING

Figure IV-3. Specific Example of the OR-Gate

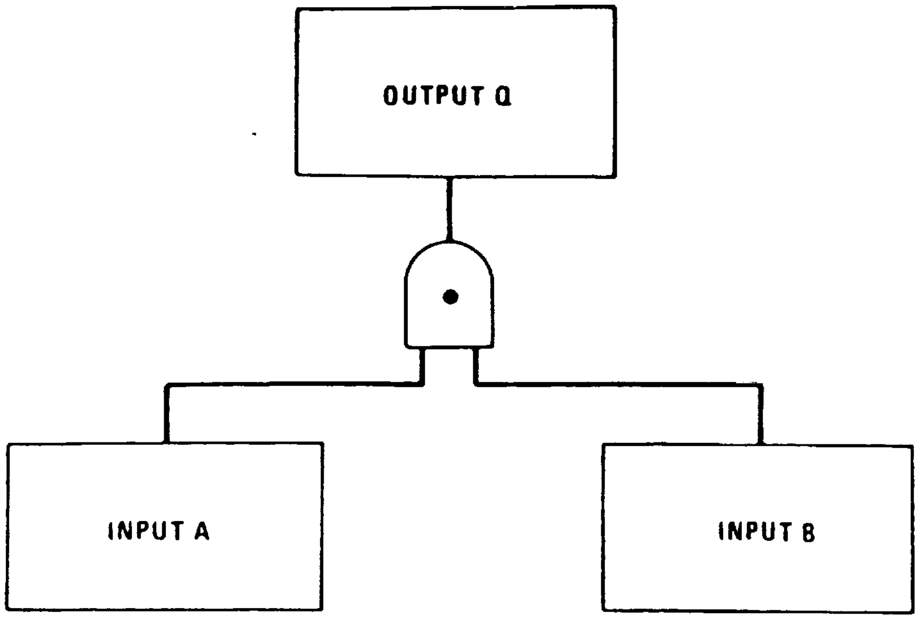


Figure IV-5. The AND-Gate

In contrast to the OR-gate the AND-gate does specify a causal relationship between the inputs and the output, i.e., the input faults collectively represent the cause of the output fault. The AND-gate implies nothing whatsoever about the 'ecedents of the input faults. An example of an AND-gate is shown in Figure IV-6. A failure of both diesel generators and of the battery will result in a failure of all onsite DC power.

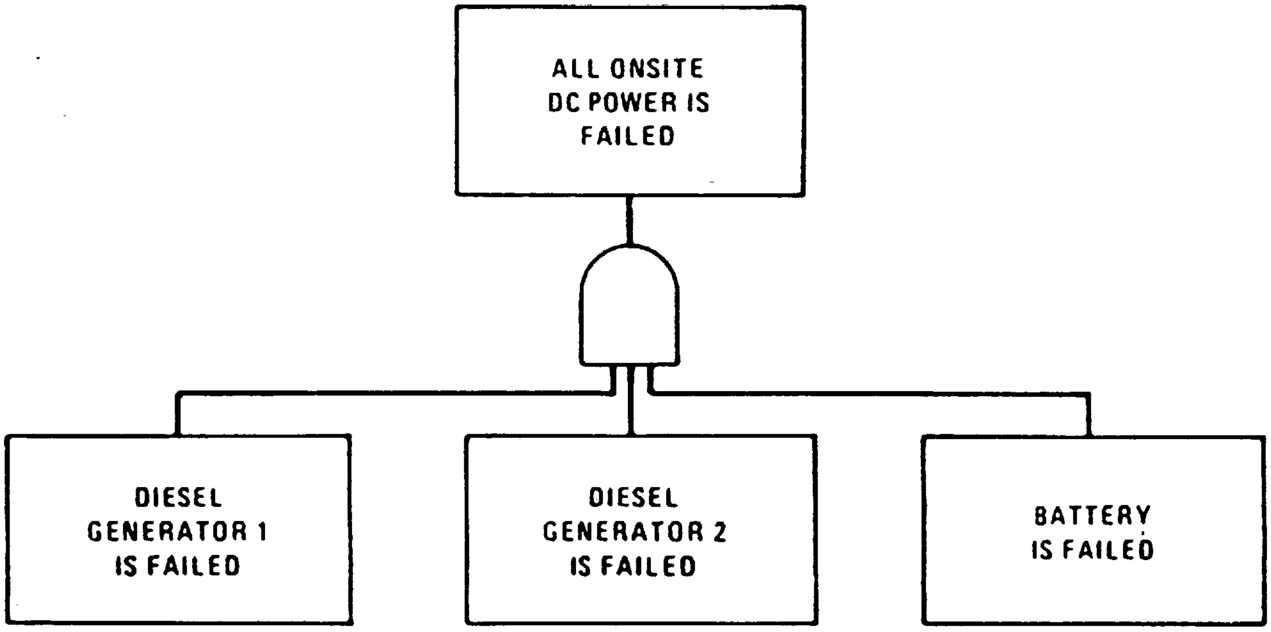

ALL ONSITE DC POWER IS FAILED
DIESEL GENERATOR 1 IS FAILED
DIESEL GENERATOR 2 IS FAILED
BATTERY IS FAILED

Figure IV-6. Specific Example of an AND-Gate

Quantitative modelling using fault trees:

- Attach probabilities to nodes of tree.
- Use boolean algebra to calculate minimal cut sets.

  minimal cut set: All unique combinations of events that can cause the top-level event.

A Sample Software Safety Program

- Software Development Management Responsibilities
- Software Hazard Analysis
- Establishing Software Safety Requirements
- Software Safety Requirements Review
- Software Safety Design Concepts
- Software Design and Recovery Analysis
- Software Safety Design Review
- Code Verification and Validation
- Assessment of Risk

Software Safety: involves ensuring that the software will execute within a system context without resulting in unacceptable *risk*.

Risk is defined in terms of *hazards* -- states of the system that when combined with certain environmental conditions *could* lead to a mishap.

Risk = f (Pr [hazard occurs], Pr [hazard leads to mishap],
Severity of worst potential mishap)

Safety critical software: software which can directly or indirectly cause or allow a hazardous system state to exist.

GENERAL COMMENTS

- Safety must be specified and designed into software from the beginning.

- Effective safety programs require changes throughout entire software life cycle.

- Enhancing reliability is not enough.

- The success of any software safety effort hinges on the ability of software, system, and safety engineers to cooperate and work together.

Basic approach adapted from system safety engineering

1) Identify potential software-related hazards

   (hazard : A condition or state of the system with the potential for leading to an accident)

2) Control hazards

   - Analysis

     Start from hazard and work backward to see if and how could occur.

   - Design

     Passive Control

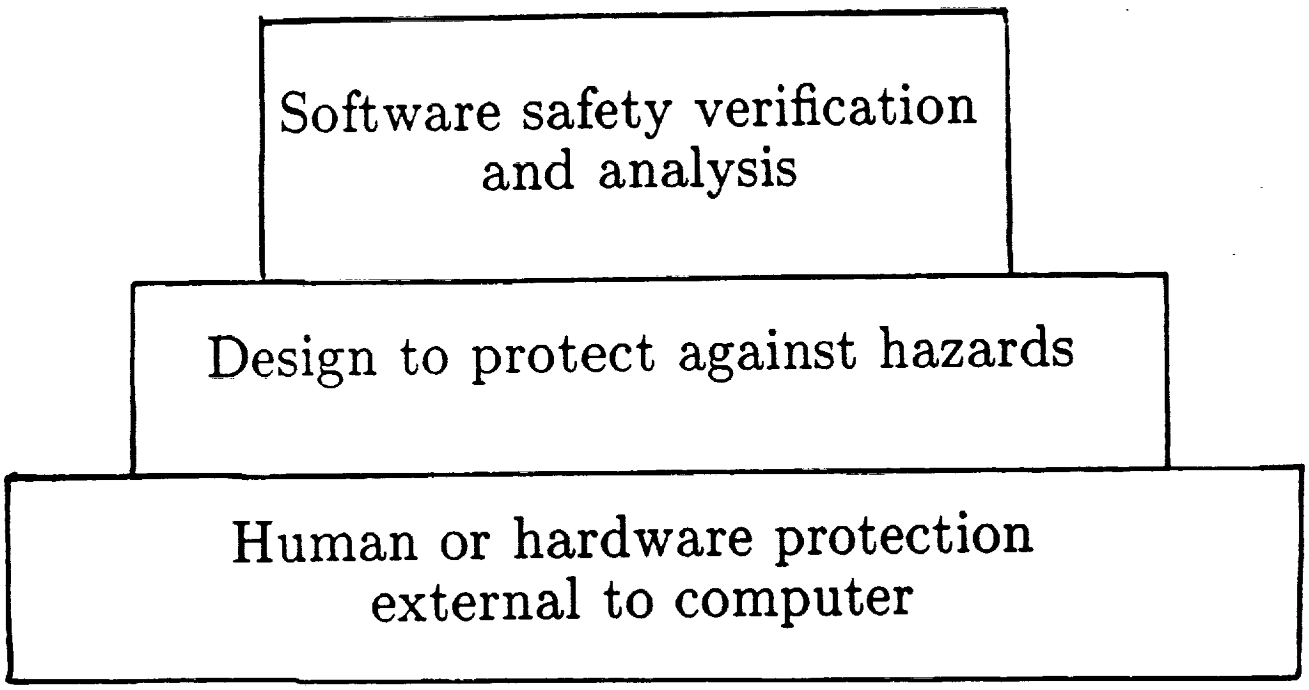
     Active Control

Philosophy and Goals

Building a bridge between software engineering and system engineering.

- Use modeling and analysis for understanding and prediction
- A systems approach — well-defined interfaces with hardware engineers, systems engineers, and quality assurance groups.
- Focus on failures with the most serious consequences
- Layers of protection

Layers of Protection


Software safety verification and analysis
Design to protect against hazards
Human or hardware protection external to computer

- Integrate into usual software development process

  Activities span the life cycle.

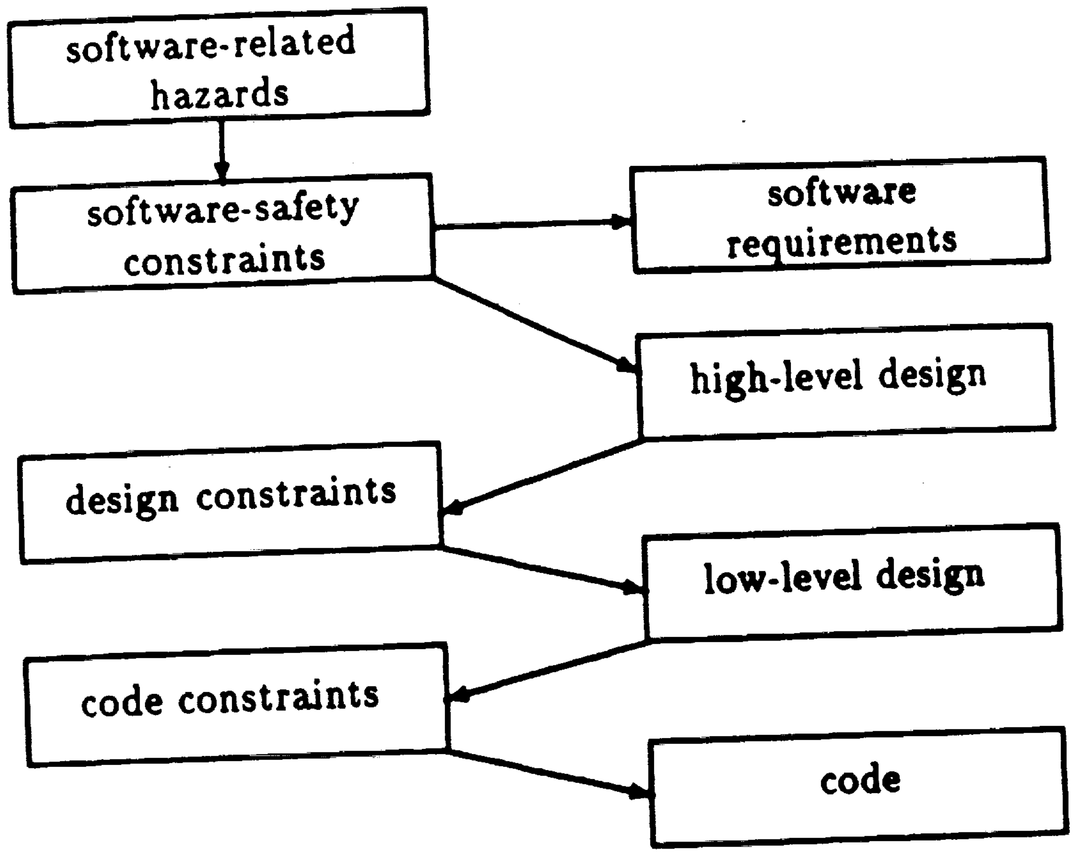
  Catch errors early — verification distributed throughout development.

  Information derived from early activities (modeling and analysis) is used to drive the design and coding.

- Combine formal and informal approaches

  Static analysis using formal proofs and structured walkthroughs

  Dynamic analysis to provide confidence in the models and assumptions used in the static analysis.


software-related hazards
software-safety constraints
software requirements
high-level design
design constraints
low-level design
code constraints
code

CAVEATS

- No magic potions
- Nothing is absolutely safe
- No techniques are perfect
- Risk elimination vs. risk displacement
- Nothing is safe under all conditions

MANAGEMENT

Management commitment to safety is the most crucial requirement for achieving it.

- degree of safety achieved is directly dependent upon emphasis given to it.
- goals of safety can be accomplished only with the support of management.

Need to:

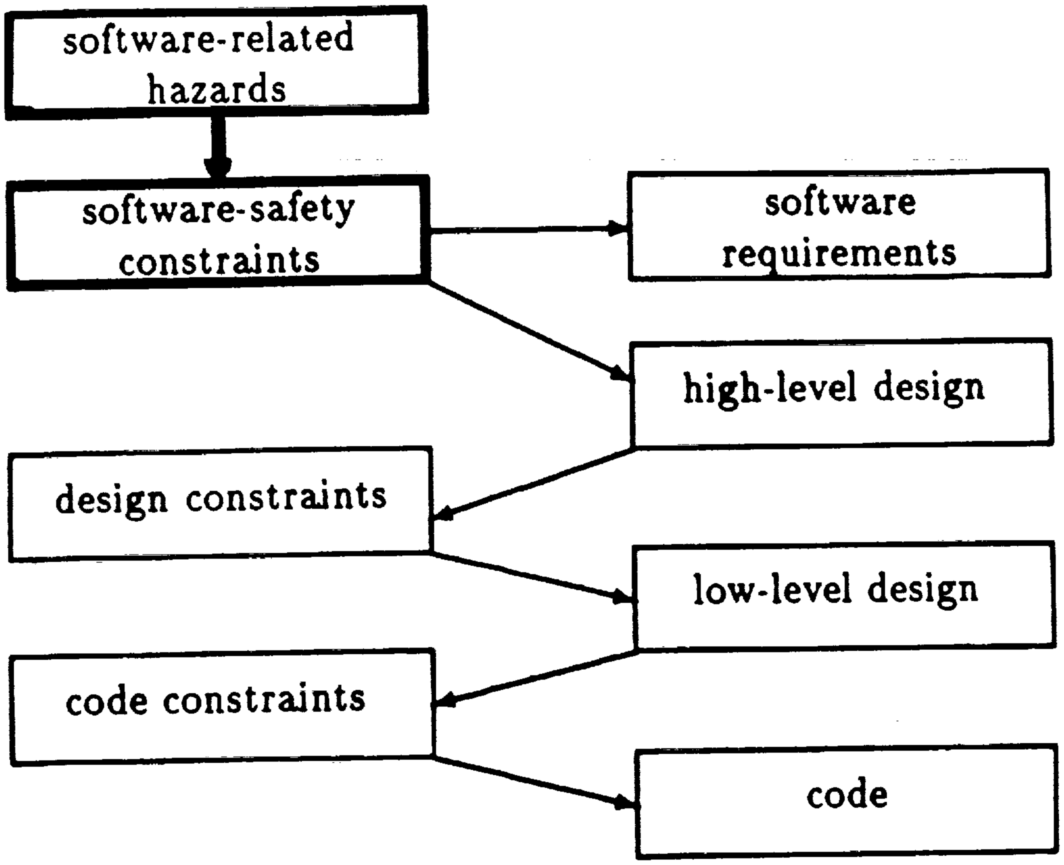
- set policy and define goals for software safety
- delegate responsibility
- grant authority
- fix accountability
- clearly delineate lines of authority, cooperation, and administration

General responsibilities of software safety management and personnel:

- Participation in early planning of the safety program

- Continual close interaction with system safety group during the life of the program

- Participation in all aspects of the software development activities to ensure that software hazards are eliminated or controlled to an acceptable level.



Software Hazard Analysis

- Model the software/system interface
- Analysis to identify software-related hazards
- Integrate with system safety analysis and system engineering models and analysis.

SOFTWARE HAZARD ANALYSIS

1) If operates "correctly," will any hazardous states result?

2) If there are failures, will hazards result?

   Single failures?

   Multiple failures?

Fault Tree Analysis

- A graphic model of the various parallel and sequential combinations of faults (or system states) that will result in the occurrence of a predefined undesired event.

- Events can involve hardware failures, human mistakes, software design faults, computer hardware failures, etc.

- Start with list of system hazards (PHA). Assume hazard has occurred, and work backward to determine set of possible causes. Preconditions described with either AND or OR relationships.

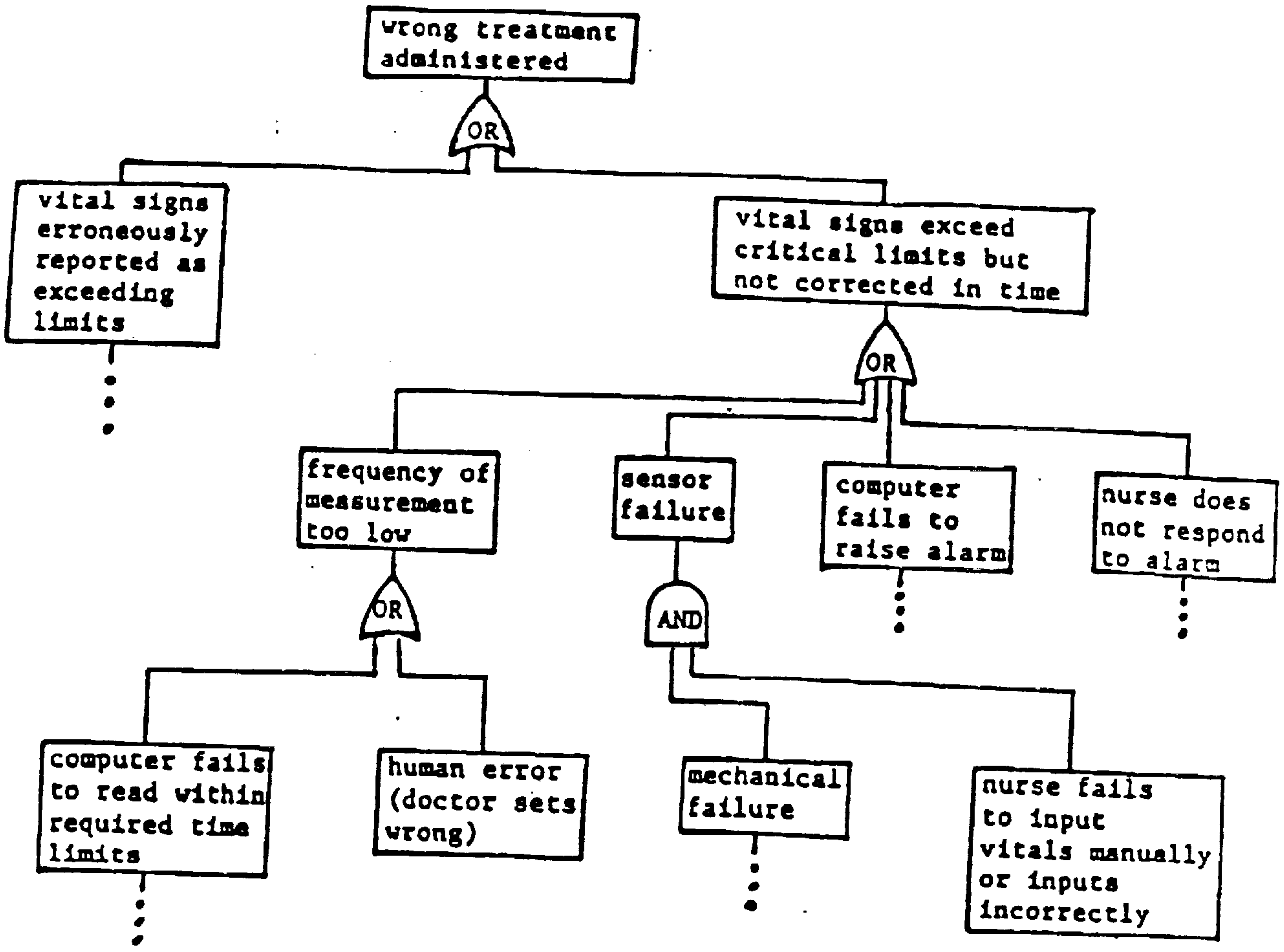

wrong treatment administered
OR
vital signs erroneously reported as exceeding limits
vital signs exceed critical limits but not corrected in time
OR
frequency of measurement too low
sensor failure
computer fails to raise alarm
nurse does not respond to alarm
OR
AND
computer fails to read within required time limits
human error (doctor sets wrong)
mechanical failure
nurse fails to input vitals manually or inputs incorrectly

Figure 1. Top Levels of Patient Monitoring System Fault Tree

PETRI NET MODELS

Have developed analysis procedures to:

- identify hazards and safety-critical single and multiple failure sequences
- determine software safety requirements including timing requirements
- analyze the design for safety and fault tolerance
- guide in the use of failure detection and recovery procedures

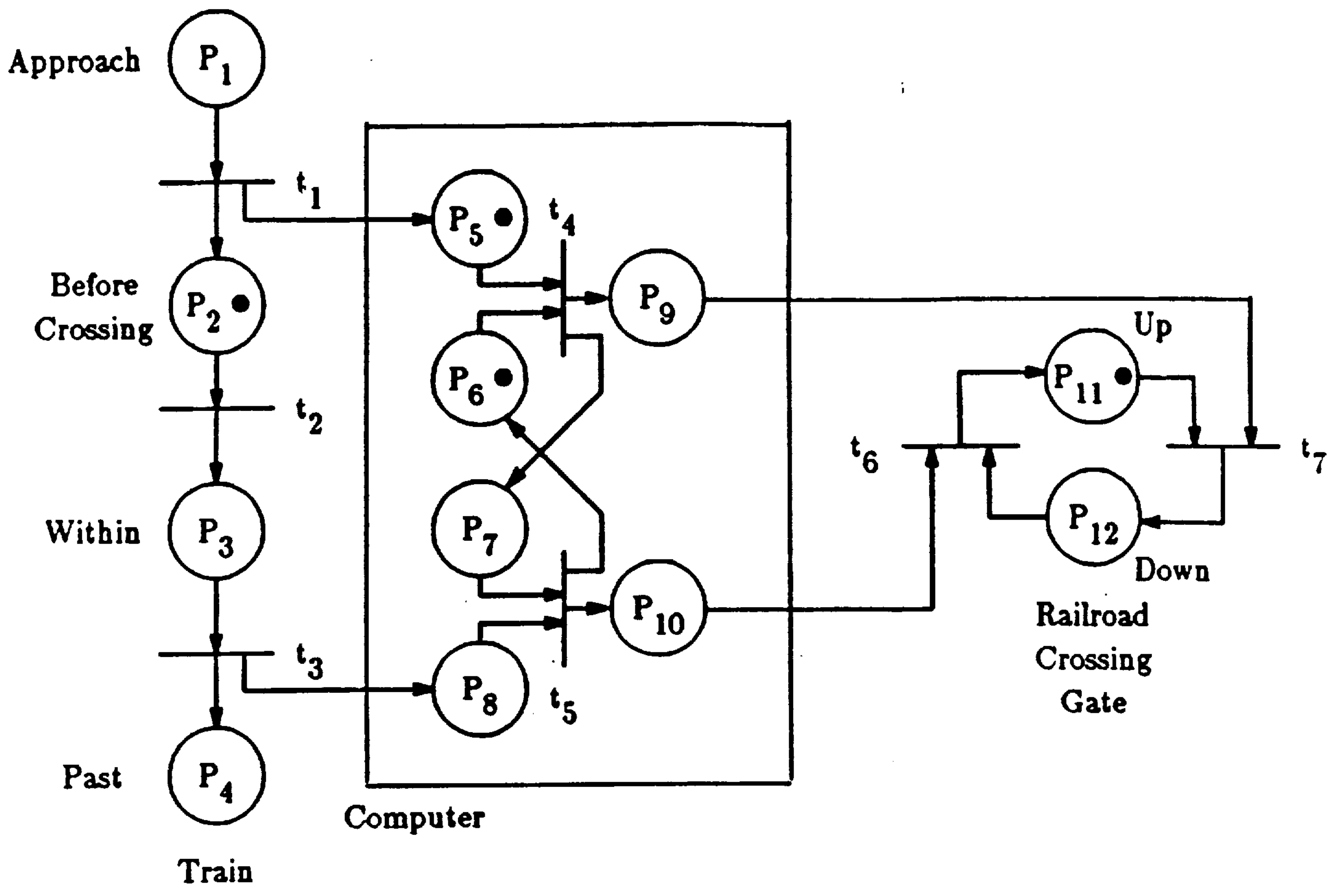


Figure 2. A Petri Net Graph with the Next State Shown

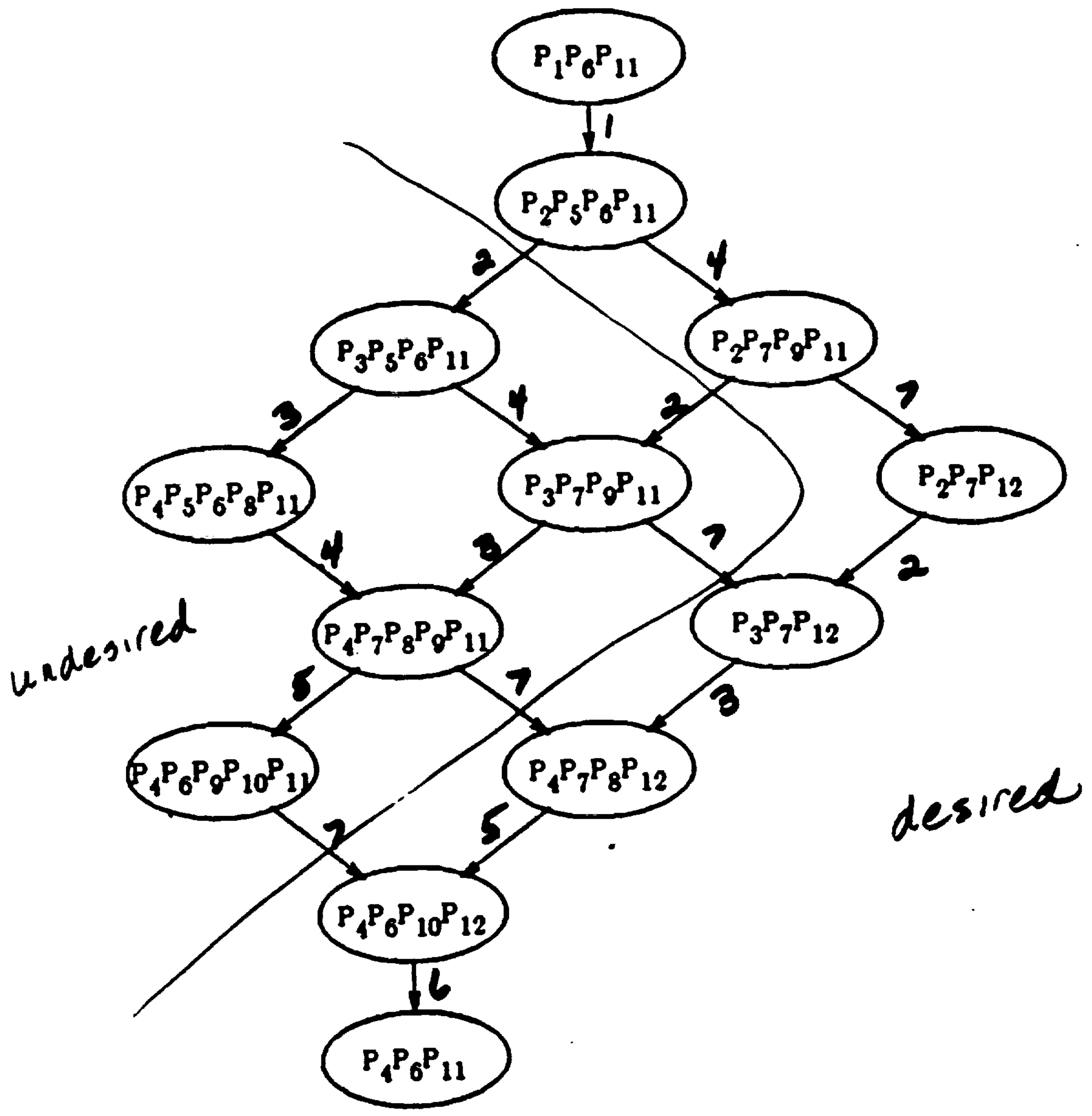


Figure 3. Reachability Graph for Figure 1

critical state: path to high-risk and possibly low-risk states as well as path to only low-risk states.

Algorithm:

Start with high-risk states. Generate those one step back and see if can go forward from them (Look forward one).

Work back to first potential critical state (state with two successors) and eliminate bad path.

- What if state not really reachable?
- What if not really a critical state?
- How do we know what states to start with. i.e. what about miscellaneous conditions?

Modify design to disallow traversal of undesired path

- change design - e.g. add interlocks, lockouts, etc.

- add timing constraints

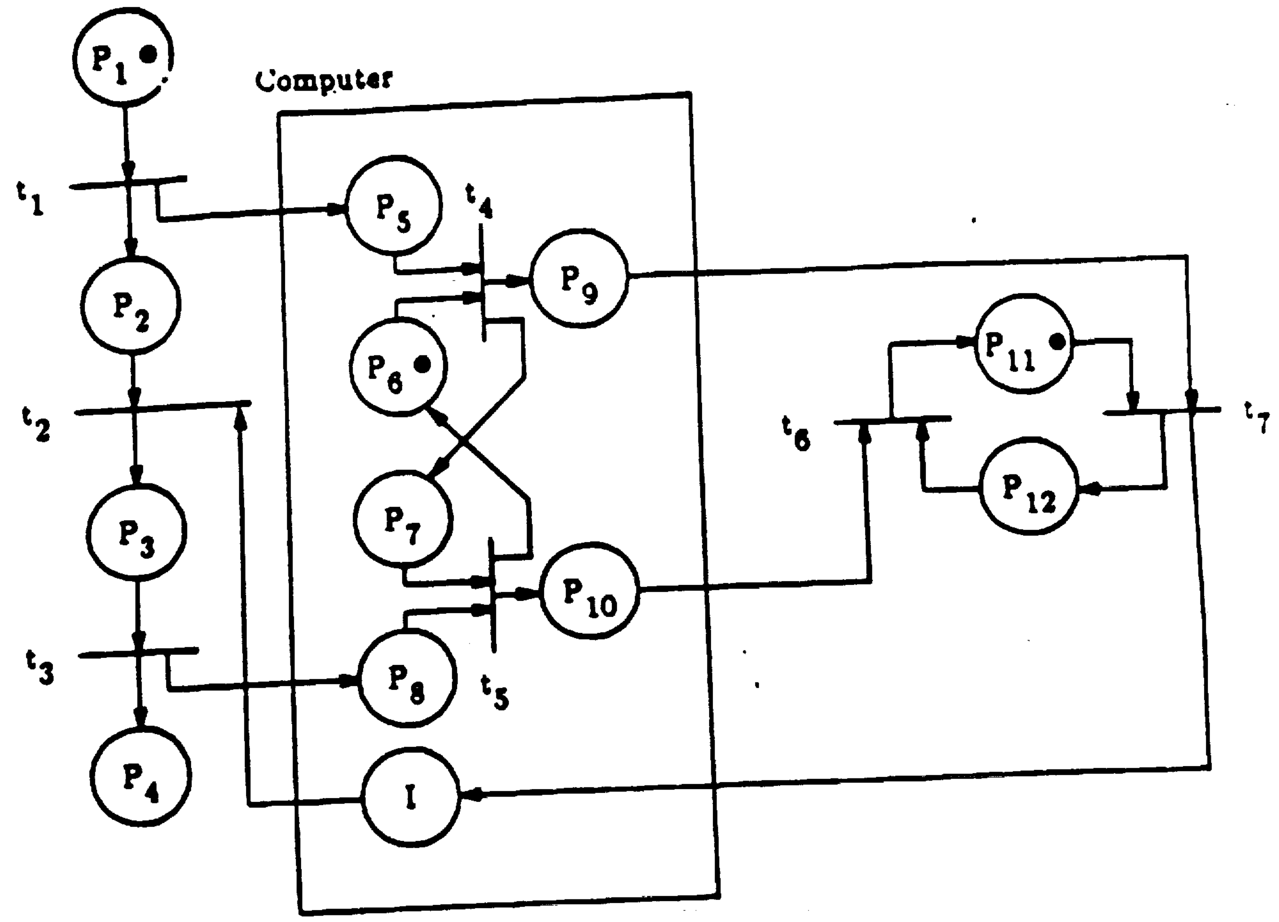


Figure 4c. A Petri Net Graph with an Interlock (I)

$$\begin{tikzcd}
P_1P_6P_{11} \arrow[d, "t_1"] & P_3P_7P_{12} \arrow[d, "t_3"] \\
P_2P_5P_6P_{11} \arrow[d, "t_4"] & P_4P_7P_8P_{12} \arrow[d, "t_5"] \\
P_2P_7P_9P_{11} \arrow[d, "t_7"] & P_4P_6P_{10}P_{12} \arrow[d, "t_6"] \\
I\,P_2P_7P_{12} \arrow[uuur, "t_2"] & P_4P_6P_{11}
\end{tikzcd}$$

Figure 4d
Reachability Graph for Figure 4c

Time Petri Nets

Add two times to each transition:

minimum required enabling time

maximum time before transition must fire

Adding time makes analysis more difficult, but since interested in worst case behavior:

1) Derive non-time reachability graph

2) Then use this to determine timing requirements

Adding Failures to the Analysis

Types of control failures:

- a required event that does not occur
- an undesired event
- an incorrect sequence of required events
- two incompatible events occurring simultaneously
- timing failures in event sequences
  - exceeding maximum time constraints between events
  - failing to ensure minimum time constraints between events
  - durational failures

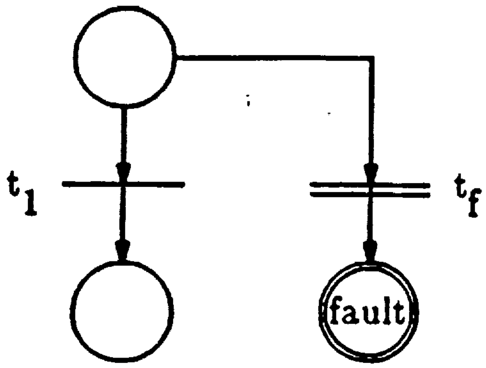


Figure 5a. Desired Event t_1 Does Not Occur

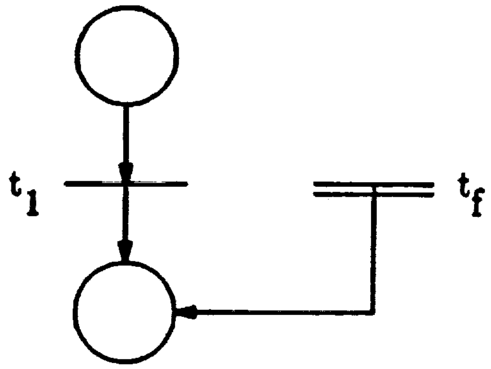


Figure 5b. Undesired Event t_1 Occurs

Important safety properties of a design:

Recoverable

Fault Tolerant

Fail Safe

faulty state: every path to it from the initial state contains a failure transition.

Recoverable: after the occurrence of a failure, the control of the process is not lost, and in an acceptable amount of time, it will return to normal execution.

1) the number of faulty states is finite

2) there are no terminal faulty states

3) there are no directed loops containing only faulty states

4) the sum of the max times on all paths from the failure transition to a correct state is less than a predefined acceptable amount of time.

correct behavior path: a path in the failure reachability graph from the initial state to a final state which contains no failure transitions.

Fault Tolerant Process:

1) a correct behavior path is a subsequence of every path from the initial state to any terminal state.

2) the sum of the maximum times on all paths is less than a predefined acceptable amount of time.

Fail-Safe: all paths from a failure F contain only low-risk states.

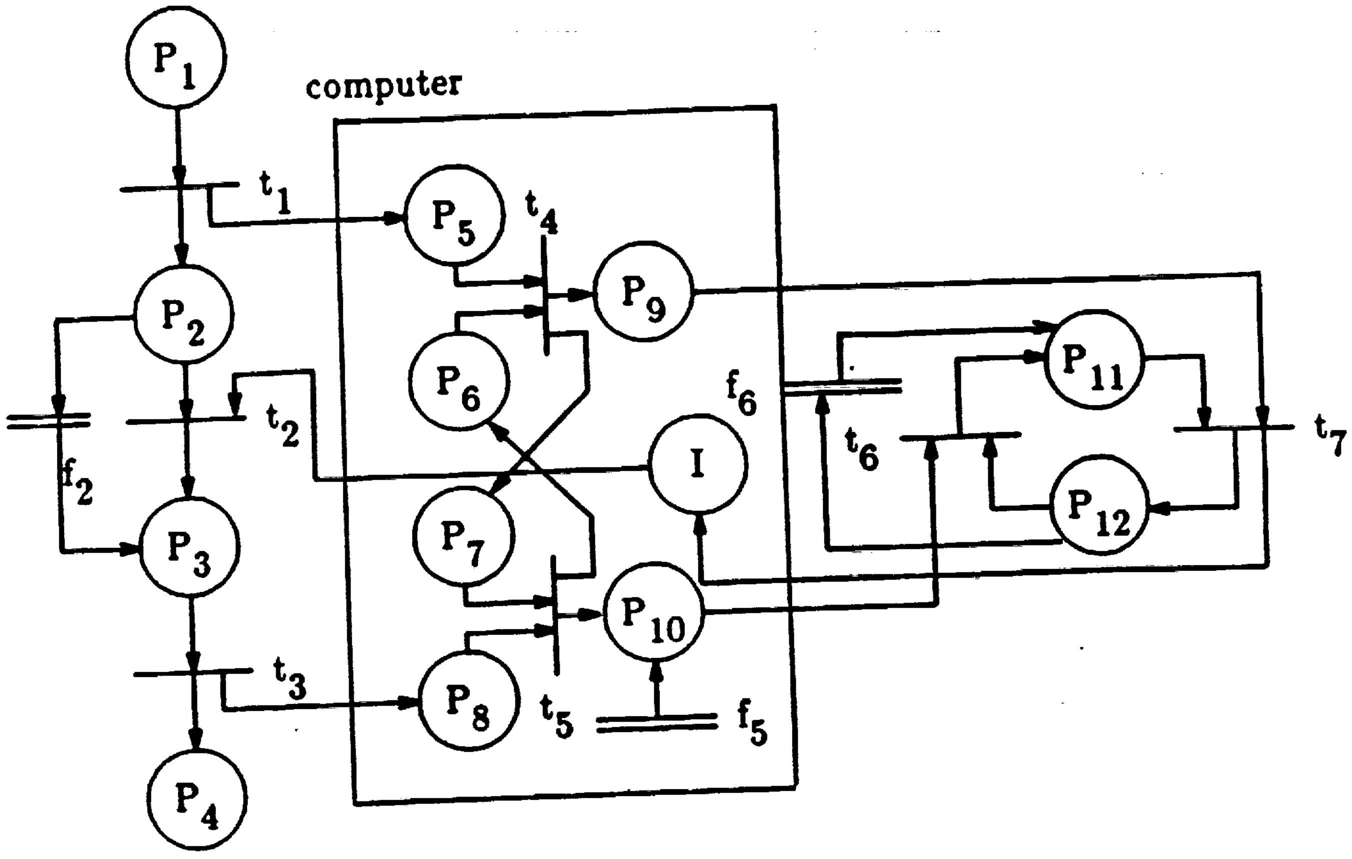


Figure 8a. A Petri Net Graph with Failures

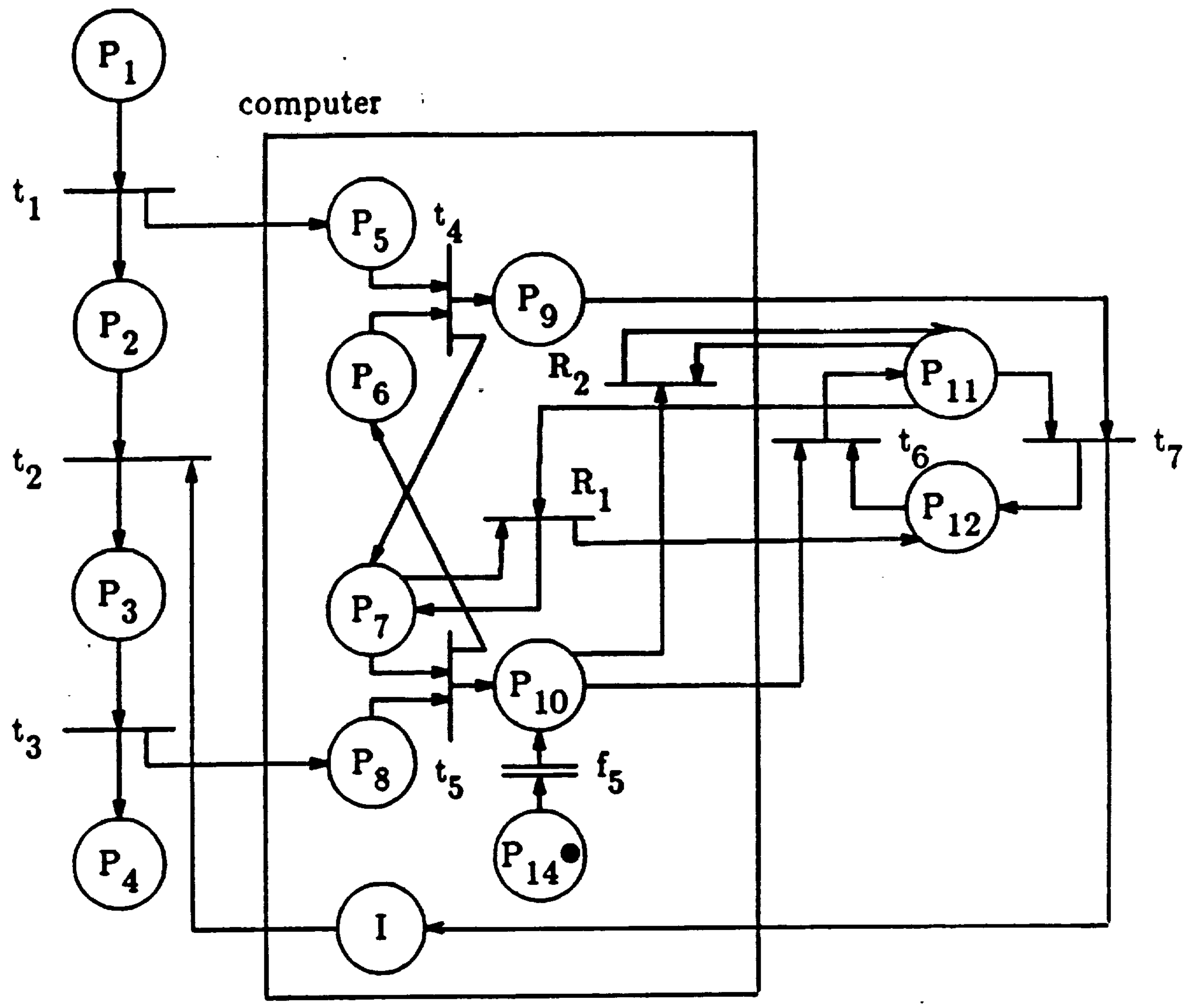


Figure 9. A Petri Net Graph with Failure Transition and Recovery

- Included in Mil-Std-882B (System Safety) and AF Handbook on Software Safety

- Used experimentally on some real projects

- Never meant to be practical — changing to a better model.

External Interaction Model

Goal is to interface between system engineering and software engineering.

EIM includes the software behavior and assumptions about the behavior of the environment within which the software will operate.

Uses:

- To verify system correctness (software requirements satisfy system requirements) including satisfying constraints.

- To minimize effects of system requirements and design changes on software through appropriate design.

- To determine and specify appropriate responses by software to violations of environmental assumptions (robustness).

Cannot just "scale-up" techniques for specifying interface between software components.

Can use model to determine whether the components (including the software) of the larger system working together exhibit certain properties.

> Partial responsibility for ensuring some properties may be assigned to software.
>
> Their existence may be affected by behavior of software.

Static analysis of properties vs. dynamic control during execution

> In both cases, must prepare system under development, systematically from the outset, to satisfy them.

Safety is an emergent or non-hierarchical property.

> Appears only when system components considered as a whole not in individual components.
>
> Accidents most often occur in the interfaces of systems — a consequence of undesired and unhandled interactions between components.

Statecharts:

- extension of finite-state machines to include:

  hierarchy
  modularity
  orthogonality
  generalized transitions

- Provides graphical language with rigorous semantics.

- But doesn't have:

  communication other than broadcast
  straightforward notation for assigning
  attributes to inputs and outputs
  probabilities
  analysis techniques defined on it.

Use Statecharts to describe state information of all components and add:

1) Input and output exchange declarations

input exchange chart:

value(Y) $\in$ ValuRange(Y)
time(Y) $\in$ TimeRange(Y)
timetype(Y) $\in$ {continual, periodic, S-R}
source(Y) $\in$ C
capacity
exceptions

output exchange chart:

value(X) $\in$ ValuRange(X)
time(X) $\in$ TimeRange(X)
timetype(X) $\in$ {continual, periodic, S-R}
destination(X) $\in$ C
load

2) Rules and mappings for these exchanges

Make explicit the assumptions and allowed interactions according to given, implied, and derived constraints.

Static checking for inconsistencies between matched exchanges

Source for dynamic checking of inconsistencies between real state of environment and computer model of it.

As external state changes, inputs provide current status and feedback information to update inter-
nal
model.

Two types of safety analysis defined on EIM model:

- Safety analysis assuming no failures — provide confidence that if system performs as specified, will not reach a hazardous state.

  backwards reachability analysis based on critical states

- Analysis with failure — fault tree analysis generated directly from model's state representation.

  Determine erroneous software states that can lead to system hazards.

TESTBED:

TCAS II: Traffic Alert and Collision Avoidance System

- Family of airborne devices functioning independently of the ground-based ATC system.
- Provides traffic advisories to assist pilot in avoiding intruder aircraft.
- Provides resolution advisories (recommended escape maneuvers) in a vertical direction to avoid conflicting traffic.
- Communicates with intruder aircraft TCAS systems, transponders on intruder aircraft, pilot, and ground-based radar beacon system.
- Used by airline aircraft and larger commuter and business aircraft.
- We will provide a system requirements specification and a safety analysis of the specification.

ESTABLISHING SOFTWARE SAFETY REQUIREMENTS

Goal: Rewriting software hazards identified by the SHA as software requirements

Need to consider:

- what system *shall not* do
- means for eliminating and controlling damage in case of an accident
- ways in which software can fail safely and to what extent failure is tolerable.

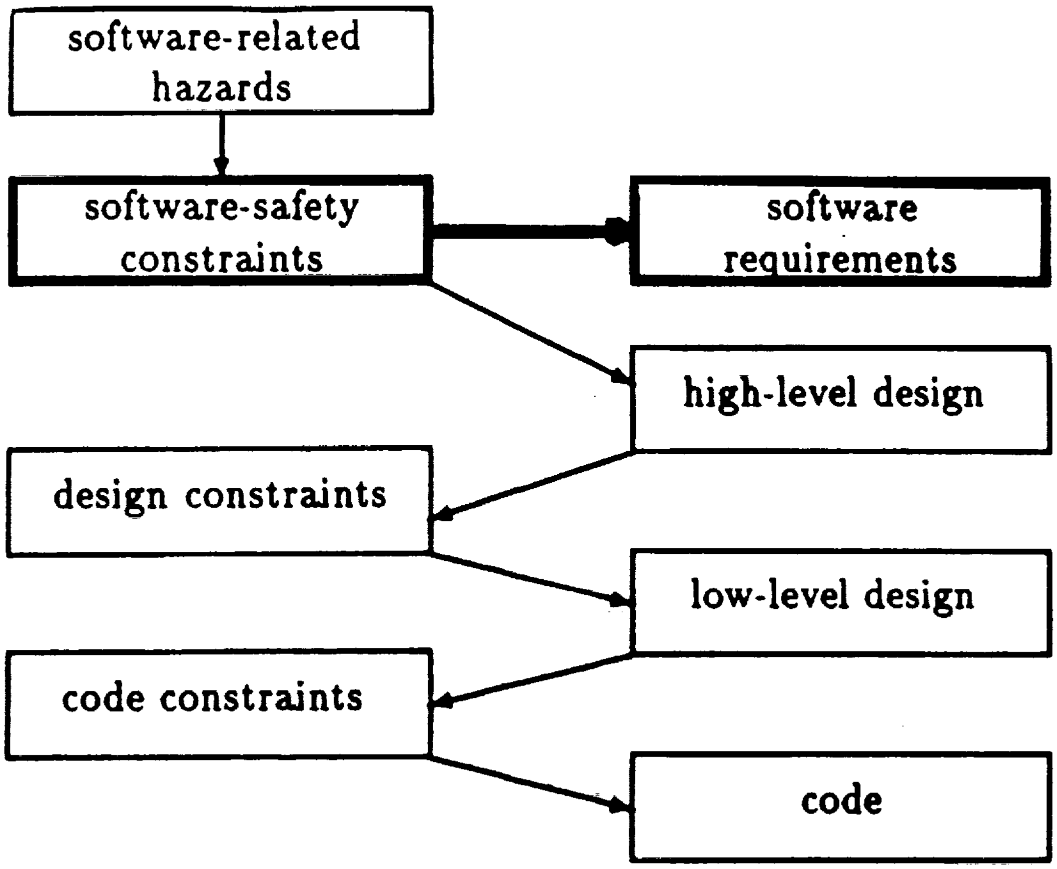
SOFTWARE SAFETY REQUIREMENTS REVIEW

Goal: identify critical requirements, missing requirements, requirements that may conflict with safety.

Participants: software engineers
system engineers
application engineers
safety engineers

Results used to: improve software requirements specification
update Software Hazards List
update detailed safety design criteria
test and evaluation criteria

Techniques: Ad hoc techniques
Real Time Logic



Requirements Analysis:

- Analyze software requirements for robustness, lack of ambiguity, consistency with system requirements

- Verify consistency with safety constraints.

- Identify conflicts and tradeoffs.

What is Correctness?

A system is a set of component working together to achieve some common objective or function.

Three components of system requirements:

1) Basic function or objective

2) Constraints on operating conditions

Define range of conditions within which system may operate while achieving its objective.

Limit the set of acceptable designs.

Arise from several sources:

quality considerations
physical limitations on equipment
process characteristics
safety considerations

3) Prioritized quality goals to:

Judge which alternative design is best

Resolve tradeoffs between conflicting requirements.

Two aspects of specification correctness:

- Implementation correctness

  Constructed component satisfies its requirements specification.

  Specification must distinguish behavior of desired software from any other, undesired program that might be designed.

  Requires specification to be sufficiently unambiguous.

- System correctness

  Component behavior, if implementation satisfies its requirements, together with specified behavior of the other components will satisfy the system requirements.

Approaches to finding errors in requirements specifications:

- Prototyping
- Executable specifications
- Scenarios
- Informal reviews
- Formal modeling and analysis

  Build model of software behavior and its interface with other components and analyze to ensure behavior and properties of model match desired behavior and properties.

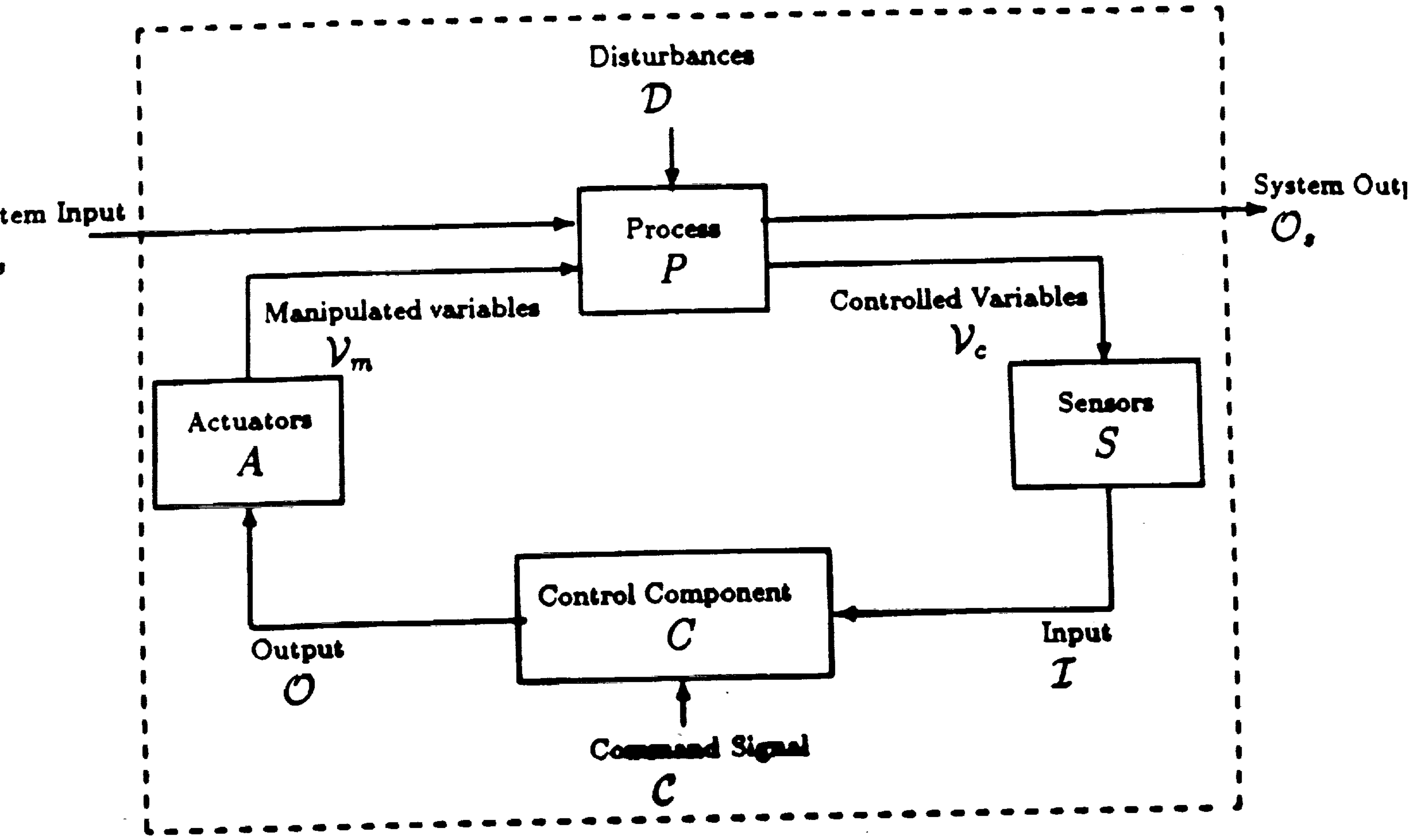


Figure 1: The control loop

The RSM is denoted as a seven-tuple $(\Sigma, Q, q_0, P_T, P_O, \delta, \gamma)$ wherе:

- Σ is the set of input/output variables, $\mathcal{I}$ and $\mathcal{O}$,
- Q is the set of states of the control component C.
- $q_0 \in Q$ is the initial state of C; the software is in this state before startup.
- P_T is the set predicates on the values and timing of the inputs $(\mathcal{I})$. They t state change in the RSM.
- P_O is the set of predicates on the outputs $(\mathcal{O})$
- δ is the state transition function $Q \times P_T$ to Q.

(is the trigger-to-output relationship $Q \times P_T$ to P_O.

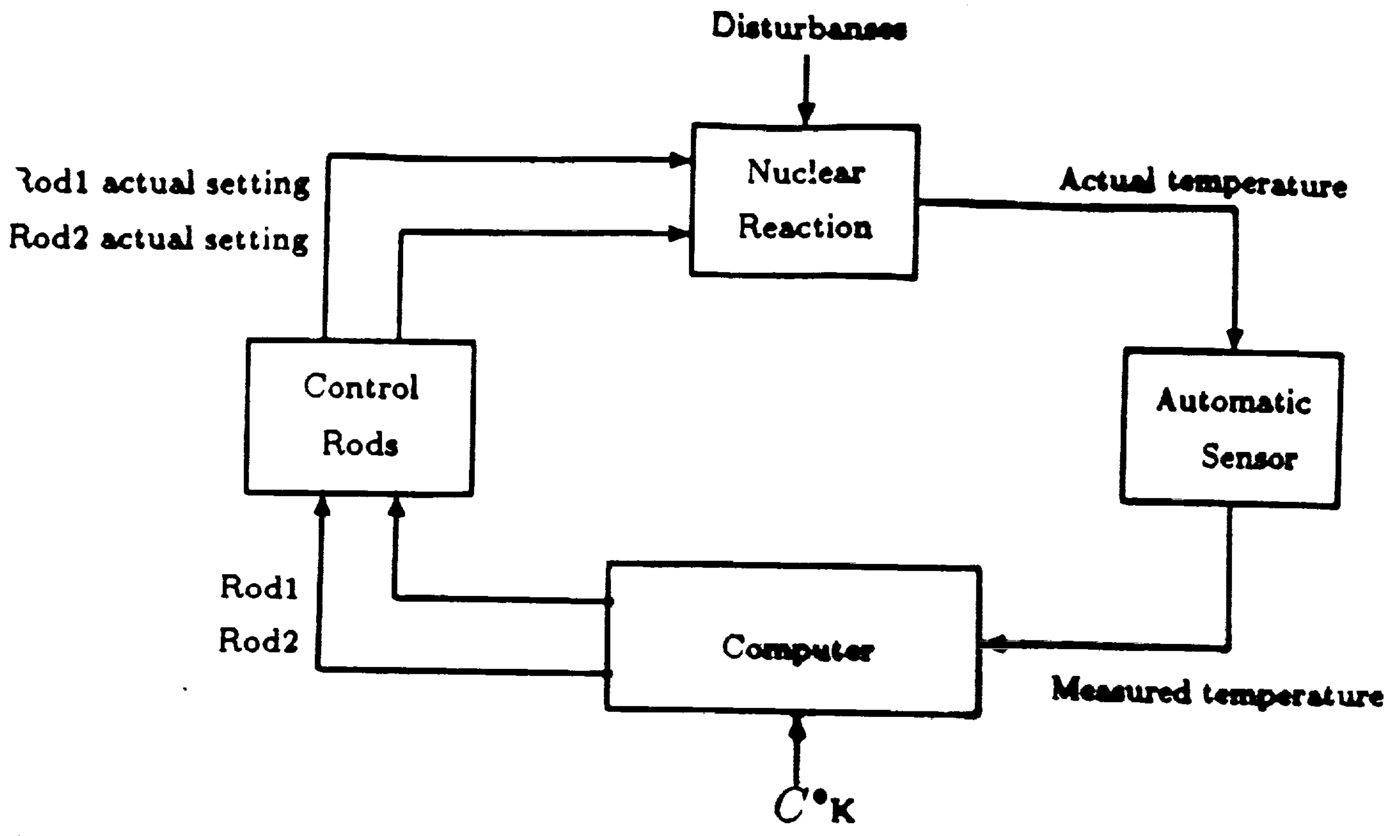


Figure 2: Block diagram of the temperature control system.

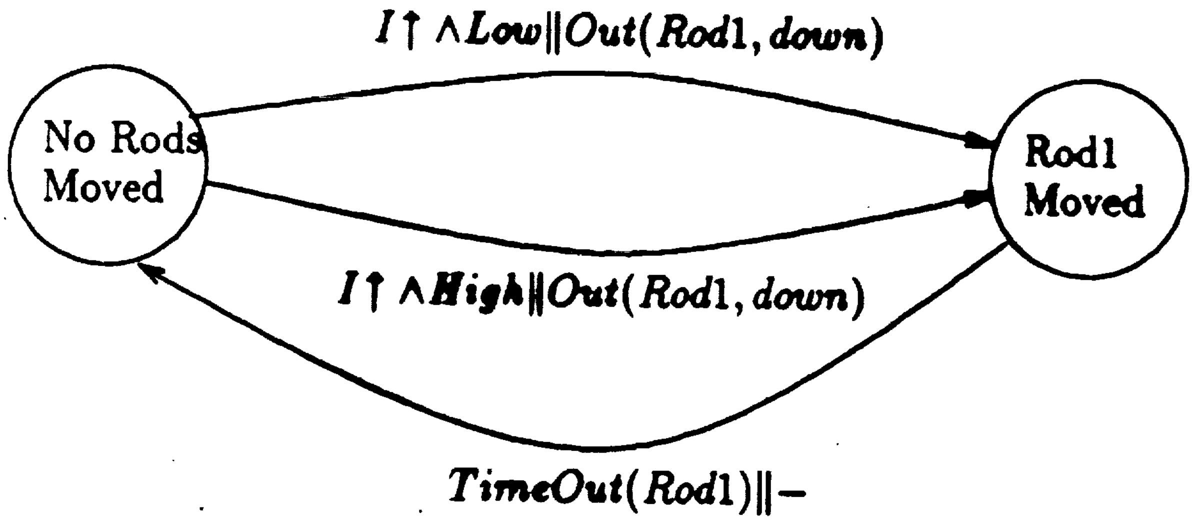
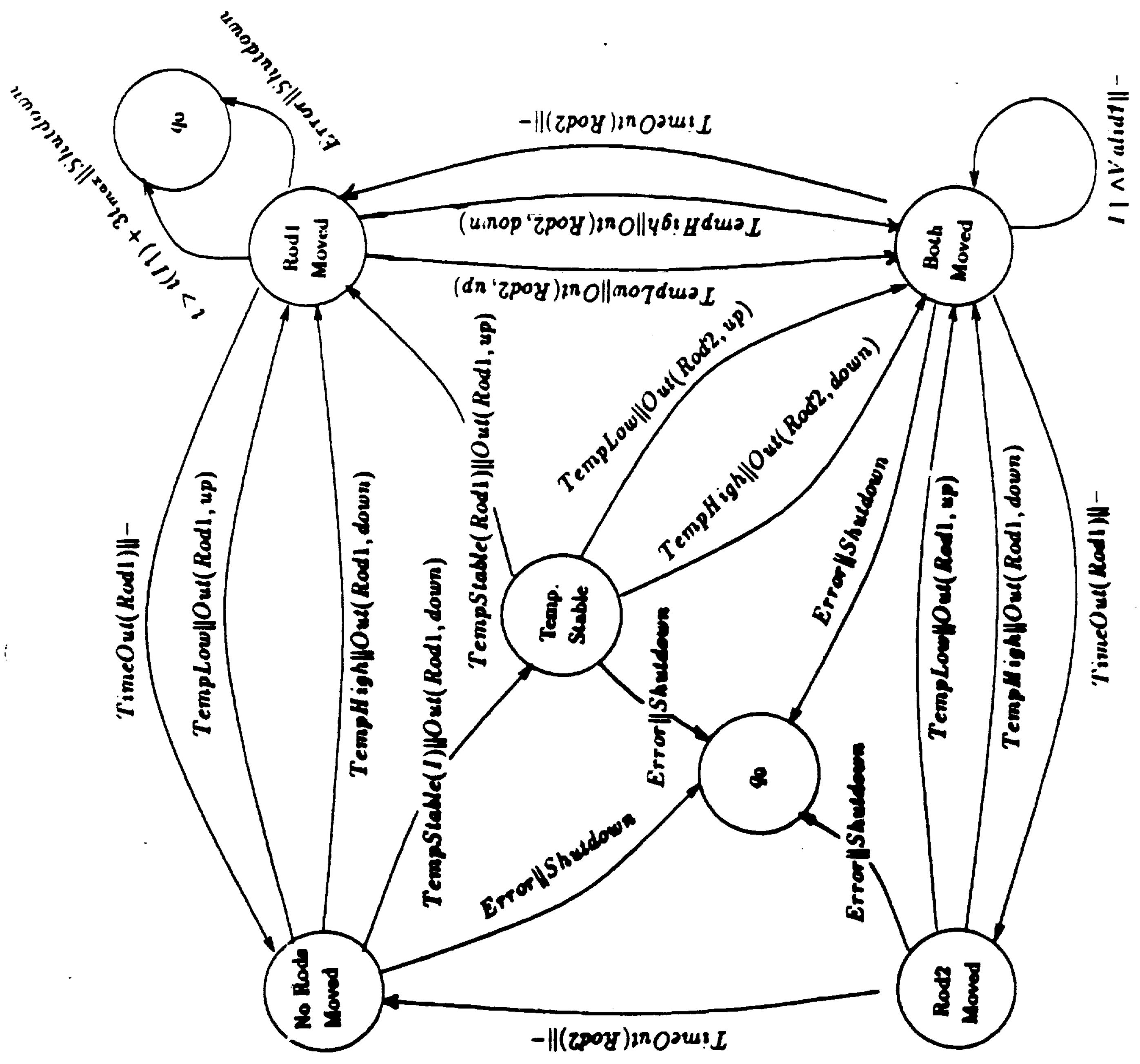

$I\uparrow \wedge Low \| Out(Rod1, down)$
No Rods Moved
Rod1 Moved
$I\uparrow \wedge High \| Out(Rod1, down)$
$TimeOut(Rod1) \| -$

Figure 3: A fragment of an RSM

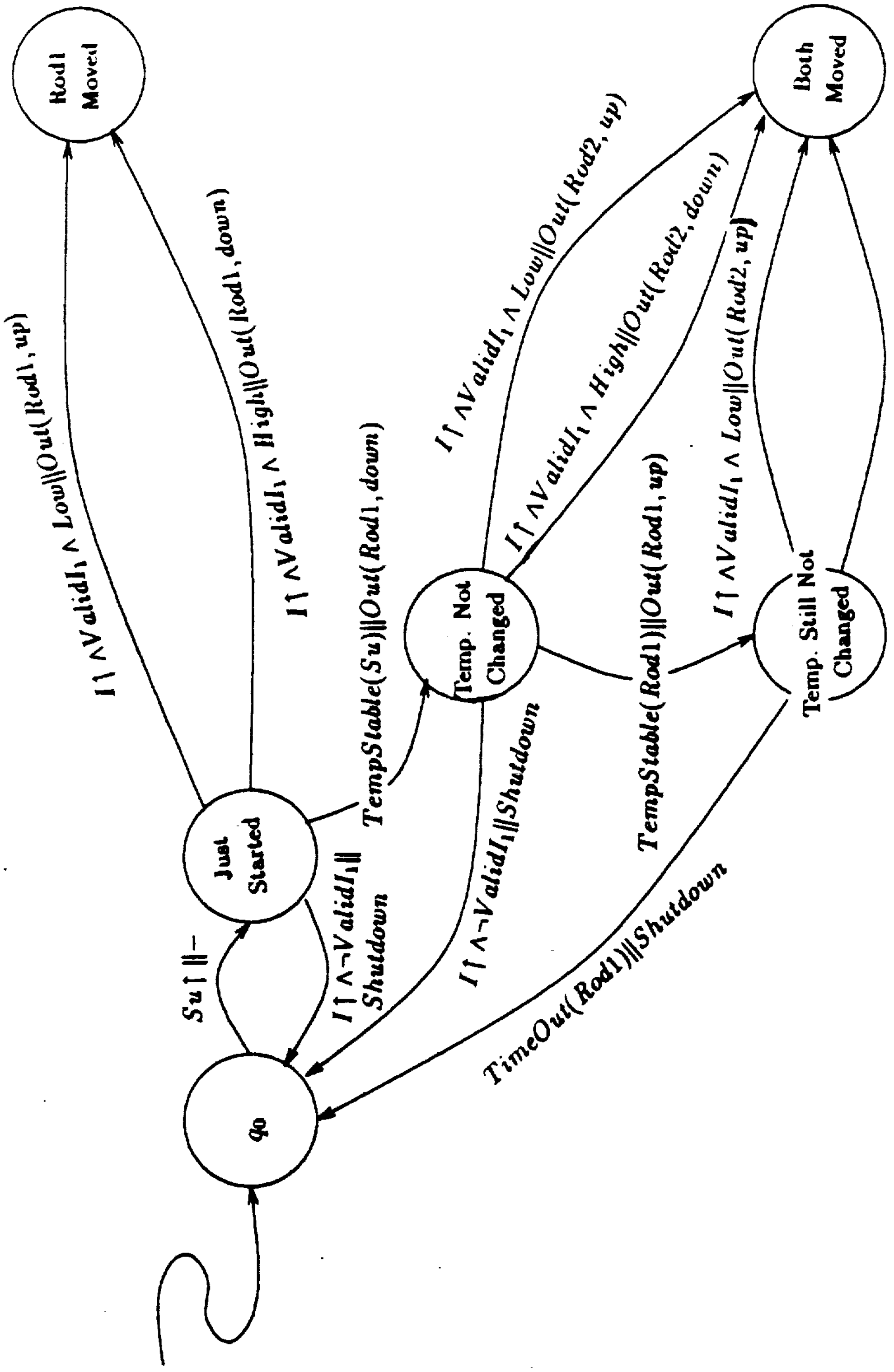
$$\begin{aligned} Low &= v(I) < C^{\circ}K \\ High &= v(I) \geq C^{\circ}K \\ TimeOut(x) &= t > t(x\uparrow) + 30 \\ Out(O, x) &= O\uparrow \wedge (v(O) = x) \end{aligned}$$

Normal Operation.



ORIGINAL PAGE IS OF POOR QUALITY

The Startup Sequence.


Rod1 Moved
Both Moved
Just Started
q0
Temp. Not Changed
Temp. Still Not Changed
I↑ ∧ValidI₁ ∧ Low||Out(Rod1, up)
I↑ ∧ValidI₁ ∧ High||Out(Rod1, down)
TempStable(Su)||Out(Rod1, down)
I↑ ∧ValidI₁ ∧ Low||Out(Rod2, up)
I↑ ∧ValidI₁ ∧ High||Out(Rod2, down)
TempStable(Rod1)||Out(Rod1, up)
I↑ ∧ValidI₁ ∧ Low||Out(Rod2, up)
I↑ ∧ValidI₁ ∧ High||Out(Rod2, down)
Su↑ ||−
I↑ ∧¬ValidI₁|| Shutdown
I↑ ∧¬ValidI₁||Shutdown
TimeOut(Rod1)||Shutdown

(Criteria + Heuristics)

- Input/output Variables Σ
- States Q, q_0

  Startup and Shutdown
  Modes

- Trigger Predicates P_T

  Tautology Requirements
  Essential Value Assumptions
  Essential Timing Assumptions
  - Properly bounded ranges
  - Capacity and load
  - Mimimum arrival rates, etc.

Criterion 6.1 Every state must have a behavior (transition) defined for every possible input. Formally,

$$\forall I, q \exists q_1, p : (\delta(q, p) = q_1) \wedge (p \in P_{T_I})$$

where $I \in \Sigma$, q, $q_1 \in Q$ and P_{T_I} is defined as in section 4.

Criterion 6.2 The logical OR ($\vee$) of the input predicates on the transitions out of any state must form a tautology:

$$\models \bigvee_i p_i$$

where the p_is are the input predicates leading out of the state of interest.

Criterion 6.3 Every state must have a behavior (transition) defined in case there is *no* input for a given period of time, i.e., a timeout.

Criterion 6.4 The RSM must to be deterministic. Let p_i represent the input predicate on the ith transition out of a state. Then deterministic behavior is guaranteed by:

$$\forall i \forall j (i \neq j) \Rightarrow \neg(p_i \wedge p_j)$$

- Output Predicates P_o

  Environmental capacity considerations
  Data Age
  Latency

- Trigger-to-Output Relationship γ

  Graceful Degradation

  Hysteresis

  Responsiveness and Spontaneity (Feedback)

- Transitions δ

  Basic Reachability

  Recurrent Behavior

  Reversibility

  Reachability of Safe States

  Path Robustness

  Constraint Analysis

Criterion 9.3 Reversibility of an operation x (performed in a state q_x) by an operation y (performed in a state $q \in Q_y$) requires a path between q_x and a state belonging to Q_y. Formally,

$$\exists q \exists s : (\hat{\delta}(q_x, s) = q) \wedge (\phi(s_i)).$$

where $q \in Q_y$.

PATH ROBUSTNESS

Criterion 9.5 Soft and hard-failure modes should be eliminated for all hazard-reducing outputs. Formally, let Q_x and Q_y be the sets of states where actions x and y are performed. The loss of the ability to receive I is a soft-failure mode for the paths from action x to action y iff

$$\exists q \forall q_1, s[(\hat{\delta}(q, s) = q_1) \Rightarrow (\neg\phi(s_i) \vee I\uparrow)]$$

where $q \in Q_x$ and $q_1 \in Q_y$.
The loss of the ability to receive I is a hard-failure mode iff

$$\forall q \forall q_1, s[(\hat{\delta}(q, s) = q_1) \Rightarrow (\neg\phi(s_i) \vee I\uparrow)]$$

where $q \in Q_x$ and $q_1 \in Q_y$.

Future Goals:

- Extend criteria on RSM
- Define analysis procedures for the criteria
- Demonstrate by applying to TCAS

Transactions on Software Engineering, March 1991.

SAFE SOFTWARE DESIGN PRINCIPLES

- Preventing Hazards
- Detecting and Recovering from Hazards
- Example

SOFTWARE SAFETY DESIGN CONCEPTS

Analysis and verification alone not enough because:

- techniques are error-prone
- cost may be prohibitive
- elimination of all hazards may require too severe a performance penalty

Two general principles:

(1) design should provide leverage for certification effort

(2) avoid adding complexity

Two categories:

- Prevention of hazards through software design

  (tends to involve reduction of functionality or design freedom)

- Detection and treatment at run-time

  (difficult and unreliable)

PREVENTING HAZARDS THROUGH SOFTWARE DESIGN

General goals: make software intrinsically safe so that software faults and failures cannot lead to system hazards.

General approach: reduce amount of software that affects safety and change as many potentially critical faults into non-critical faults as possible.

- Design to limit actions of software

  modularization
  data access limitations
  separate critical from non-critical functions
  firewalls
  hierarchical design

- Authority limitation to protect critical functions and data
- Minimize hazardous states or time in them
- Use software interlocks to ensure sequencing or prevent hazardous outputs
- Protect against hardware failures

DETECTION AND TREATMENT AT RUN-TIME

Detection:

assertions
acceptance tests
external monitors
watchdog timers
voting

- Mechanisms not as much of a problem as formulating the checks

Recovery:

- Safety recovery routines needed when:

  unsafe state detected externally

  determined that software cannot provide required output within a prescribed time limit

  continuation of regular routine would lead to a hazard if no intervention

- Backward adequate if can be guaranteed that software faults will be detected and successful recovery completed before fault affect external state.

- Forward recovery usually also needed

  robust data structures
  dynamic alteration of flow of control
  reconfiguration
  ignoring single cycle errors
  reduced function or fail-safe modes

- Design for a safe-side

Fail-Safe Design

Design system to have a safe-side: state that is always reachable from any other state and that is always safe.

- Often has performance penalties.
- Besides shutting down, may need to take some action to avoid harm.
- Safety system itself may cause harm.
- May be intermediate safe states with limited functionality, especially in systems where shutdown itself would be hazardous.
- Reconfiguration or dynamic alteration of control is a form of partial shutdown.

2 level structure:

TOP LEVEL

less important governing functions
supervisory, coordination, management functions
separate processor
loss cannot endanger turbine nor cause it to shutdown

BASE LEVEL

secure software core that can detect significant failures of hardware

self checks of:
- sensibility of incoming signals
- whether processor functioning correctly

failure of self-check leads to reversion to safe state through fail-safe hardware.

No interrupts except for fatal store fault (nonmaskable)

— timing and sequencing defined
— more rigorous and exhaustive testing

Uses polling

all messages unidirectional
→ no recovery or contention protocols required
→ higher level of predictability

State table defines:

scheduling of tasks

self-check criteria appropriate under particular conditions

SOFTWARE DESIGN AND RECOVERY ANALYSIS

Two goals:

(1) Identify safety-critical items.

(2) Identify self-test, fault-tolerance, and fail-safe facilities needed for safety-critical items.

Safety-Critical Items:

- software processes, data items, or states whose inadvertent occurrence, failure to occur when required, occurrence out of sequence, occurrence in combination with other functions, or erroneous value can be involved in development of a hazard.

- Includes erroneous program states and data items that could cause a hazard even if function or algorithm is correct.

- Identify through backward flow analysis on top-level design to locate critical paths and data.

  Manual procedures
  Software Fault Tree Analysis
  Uses Hierarchy

- Used for:

  feedback to software and system design

  e.g. minimizing critical items
  isolating critical items
  designing fault tolerance facilities

  planning load shedding and reconfiguration

Recovery Analysis

- Evaluate software and hardware failures for potential effect on safety-critical items.
- identify self-test, fault-tolerance, and fail-safe facilities needed for critical items.

Results:

- Identification of assumptions about failures and undesired events
- Fault-tolerance and fail-safe guidelines for rest of software development
- Evaluation of safety design requirements
- Description of planned safety aspects of the design including prevention, detection, and treatment of hazards.
- Evaluation of planned safety aspects of design including fault detection and recovery facilities planned for each critical item.

SOFTWARE SAFETY DESIGN REVIEW

As part of regular design review:

(1) verify that safety requirements implemented in detailed design

(2) verify that software safety design criteria and fault tolerance guidelines implemented in design

(3) produce a final safety test recommendations report.

VERIFICATION OF SAFETY

- What is it?
- Software Fault Tree Analysis

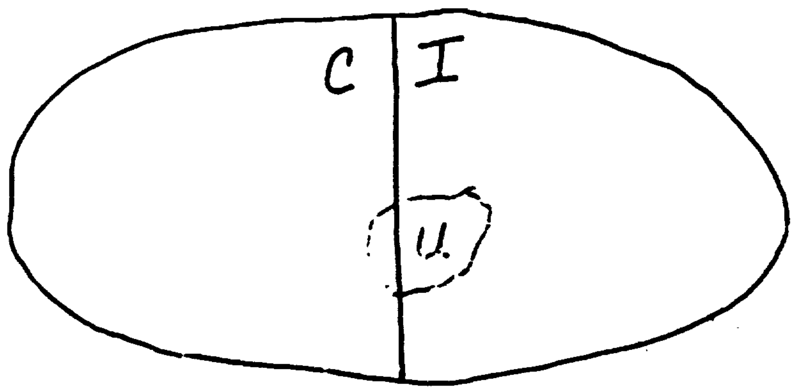
CODE VERIFICATION AND VALIDATION

- If rest of program followed, need for most costly procedures will be minimized.
- Walkthroughs and formal verification (e.g. Software Fault Tree Analysis) needed only on modules determined to be so critical that testing or other assurance procedures alone will not suffice to ensure acceptable risk.
- Need to verify that detailed feature related to safety-critical items and fault-tolerance facilities have been correctly implemented.
- Need to verify that assumptions and models upon which analyses have been based are correct.

verification of correctness

vs.

verification of safety



Software Fault Tree Analysis

- A symbolic logic diagram which shows cause and effect relationship (AND or OR) between an undesired event and one or more contributory causes.

- A Systems Tool

- Related to formal axiomatic program verification BUT:

  specification derived from system requirements

  include consideration of hardware failures, environmental conditions, human errors

  proof by contradiction

(1) $A := F(Y)$; (2) $B := X - 5.0$; (3) **if** $A > B$ **then** Sub1; **end if**;

Figure 8: Sample Assignment Statements

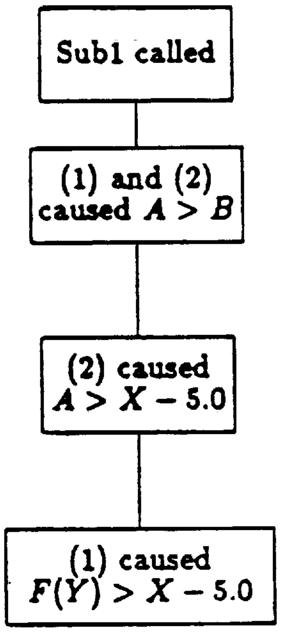


Figure 9: Fault Tree for Assignment Statements

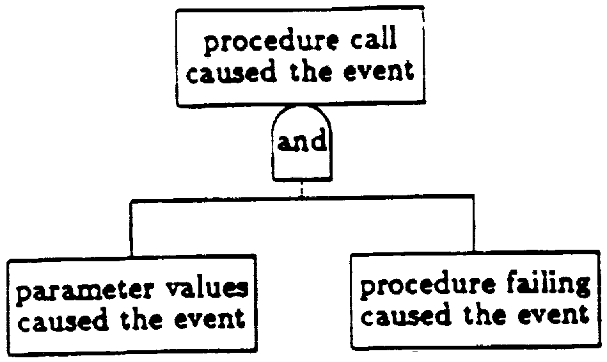

procedure call caused the event
and
parameter values caused the event
procedure failing caused the event

Figure 10: Fault Tree for a Procedure Call

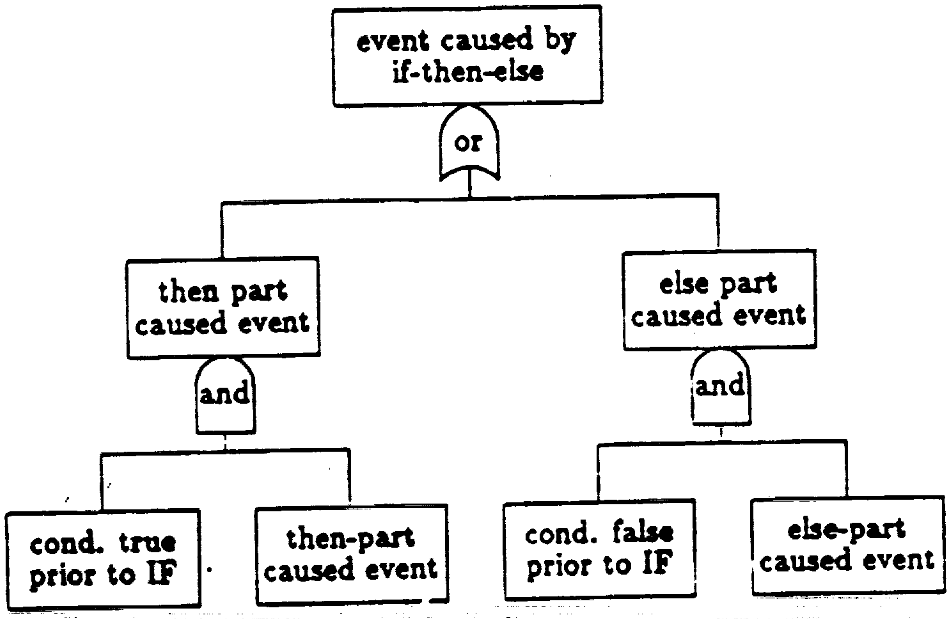


Figure 11: Fault Tree for an If-Then-Else Statement

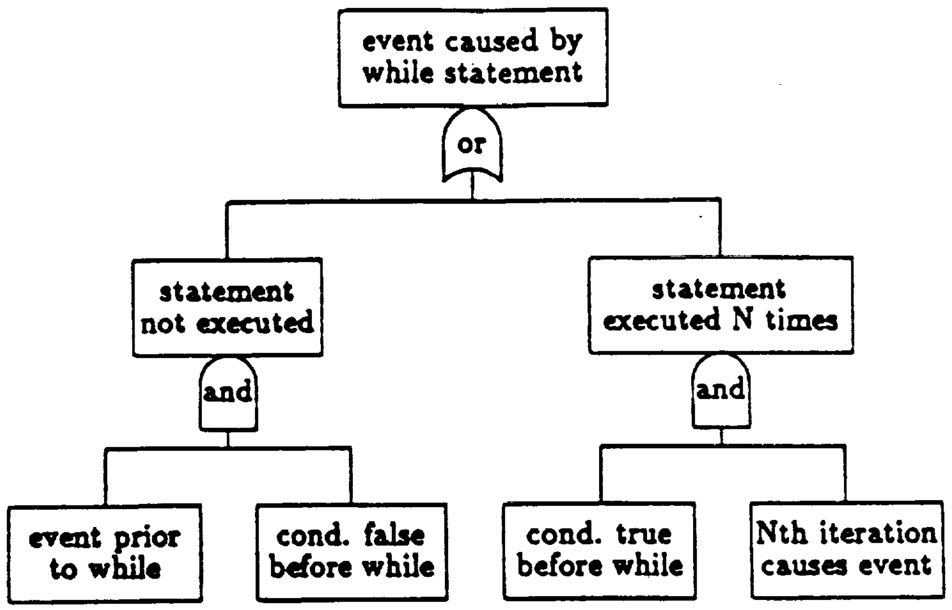


Figure 12: Fault Tree for a While Statement

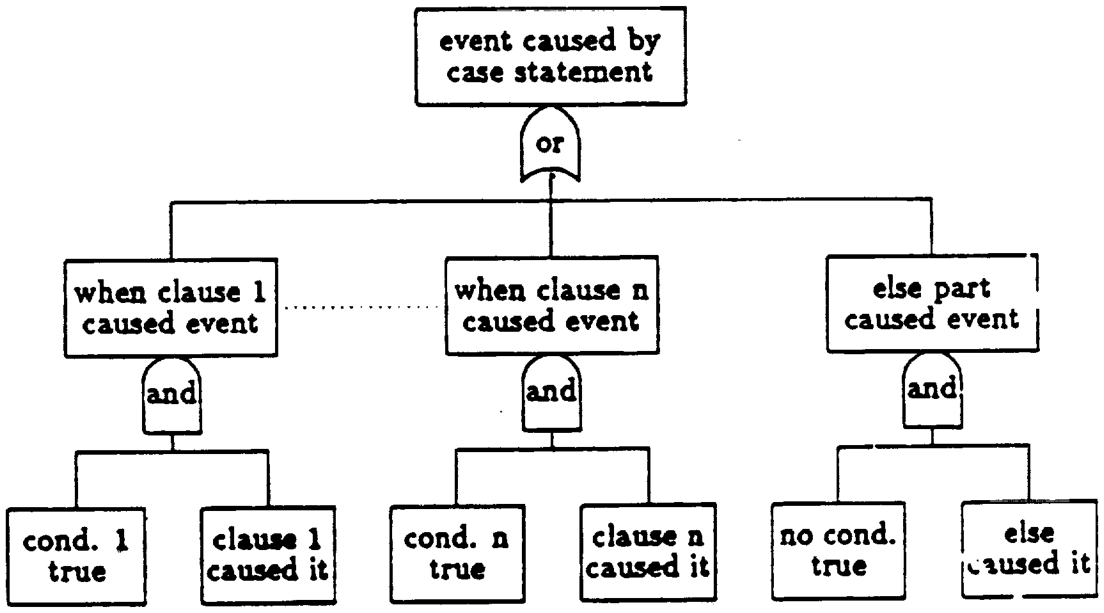


Figure 13: Fault Tree for a Case Statement

Software fault tree has two possible patterns:

(1) A contradiction is found.

(2) Fault tree runs through code and out to controlled system or its environment.

FIREWHEEL Example

Used SFTA to analyze the flight and telemetry program for a U.C. Berkeley spacecraft.

- **Mission:** to sample electric fields in the earth's magnetotail.

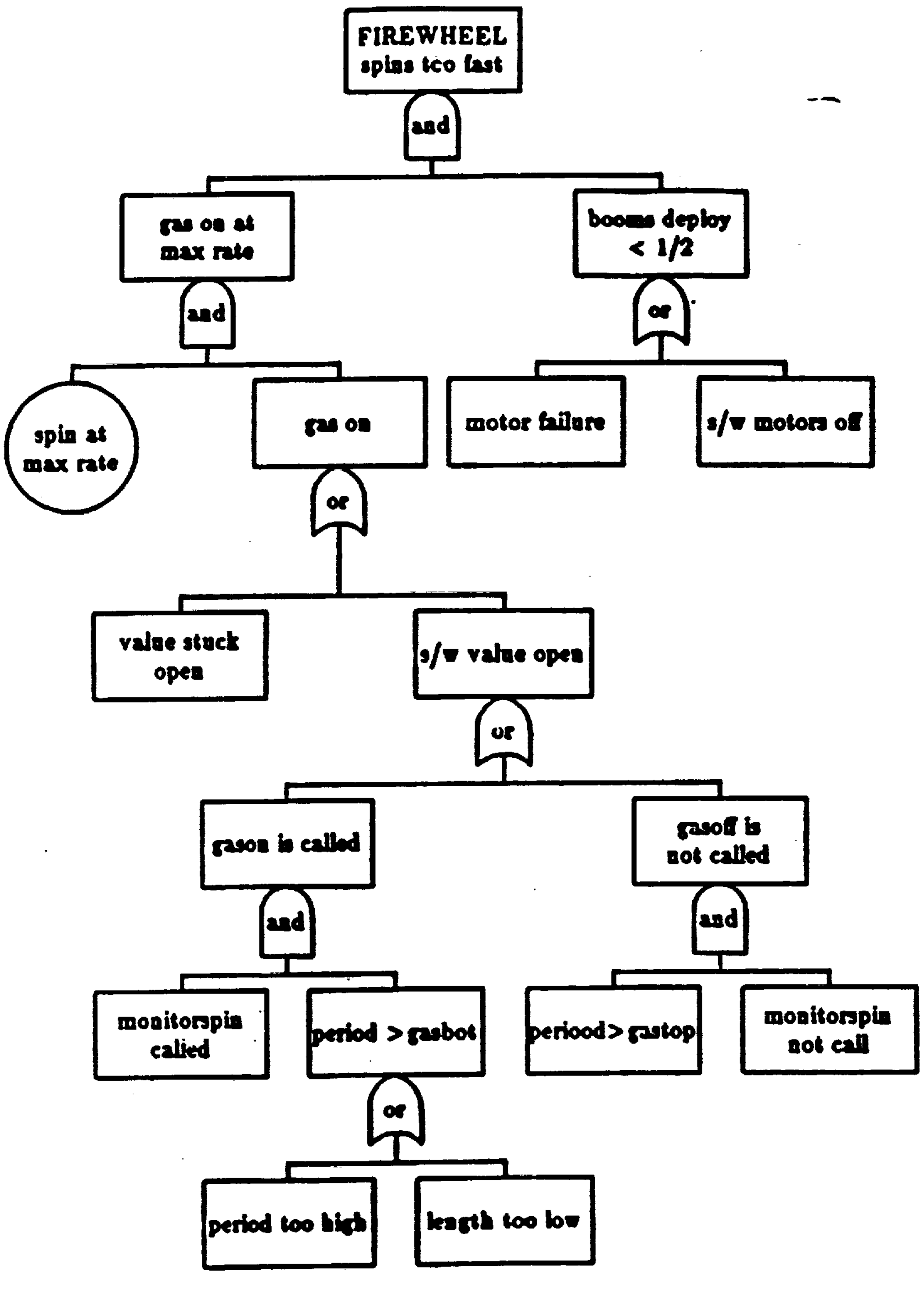
- **Critical Failure Event:** ripping wire booms off spacecraft.

- **Cost:** needed to examine 12% of code (out of approximately 300 lines of Pascal code), took two man days

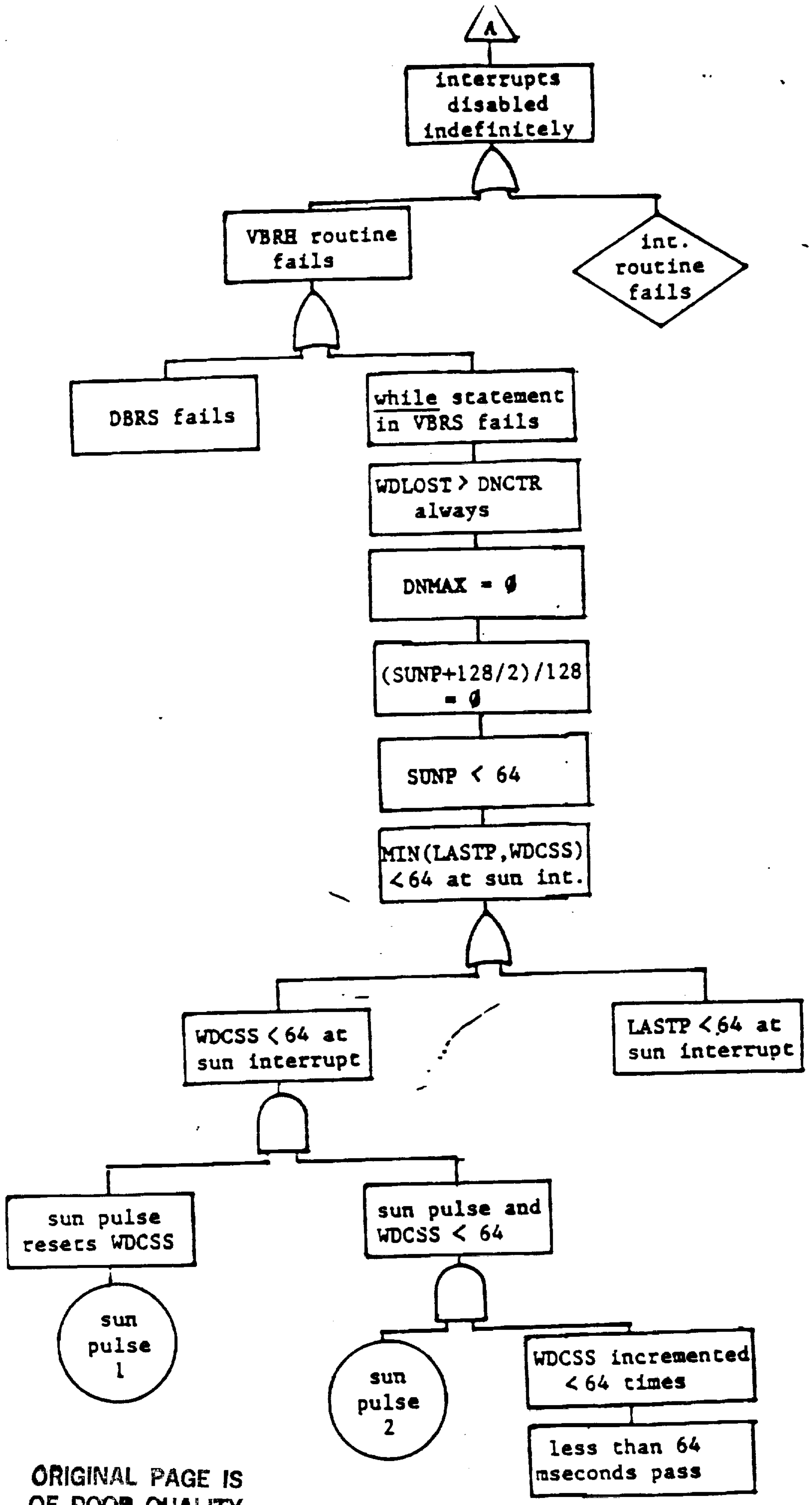
- **Results:** A critical scenario detected that was undiscovered during a thorough test and evaluation by an independent group.

PETER HARVEY

figure 7. Firewheel Spin Control – Software Opens gas Value
file location: 7
initial fault: FireWheel spins too fast

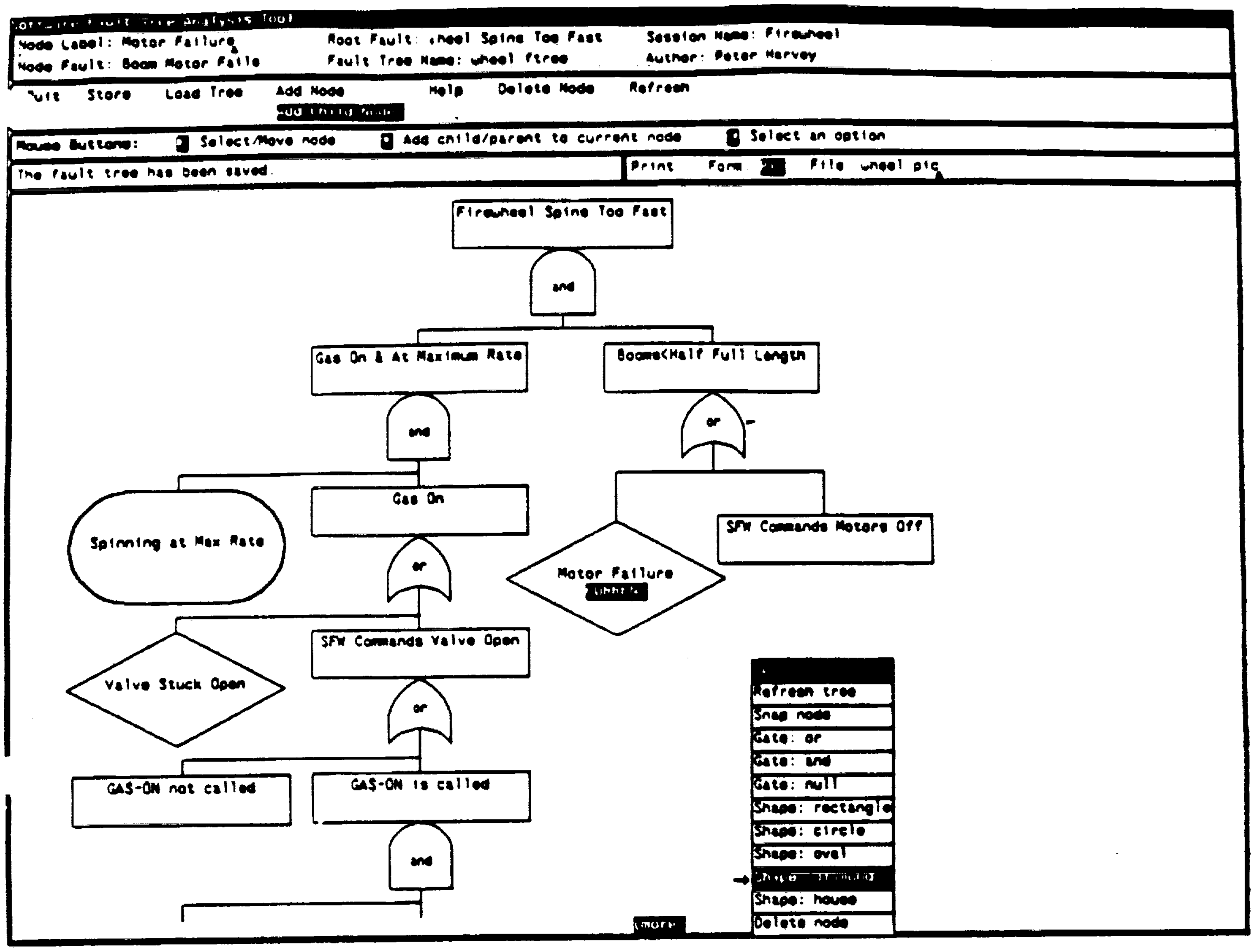


ORIGINAL PAGE IS OF POOR QUALITY


A
interrupts disabled indefinitely
VBRH routine fails
inc. routine fails
DBRS fails
while statement in VBRS fails
WDLOST > DNCTR always
DNMAX = ∅
(SUNP+128/2)/128 = ∅
SUNP < 64
MIN(LASTP,WDCSS) < 64 at sun int.
WDCSS < 64 at sun interrupt
LASTP < 64 at sun interrupt
sun pulse resets WDCSS
sun pulse and WDCSS < 64
sun pulse 1
sun pulse 2
WDCSS incremented < 64 times
less than 64 mseconds pass

ORIGINAL PAGE IS OF POOR QUALITY

Figure 9b. Boom Length Too Low (continued)



Software Fault Tree Analysis Tool

Node Label: Motor Failure

Node Fault: Boom Motor Fails

Root Fault: wheel Spins Too Fast

Fault Tree Name: wheel ftree

Session Name: Firewheel

Author: Peter Harvey

Mouse Buttons: Select/Move node · Add child/parent to current node · Select an option

The fault tree has been saved.

Print Form File wheel pic

ORIGINAL PAGE IS
OF POOR QUALITY

Software Fault Tree Analysis useful to:

- identify software faults potentially leading to accidents OR "verify" (increase confidence) they do not exist

- provide information for testing

- provide guidance for content and placement of run-time assertions (fault detection)

- provide information on fail-safe requirements

ASSESSMENT OF RISK

Certification of system partly based on system safety report including

- Description of procedures used to ensure software safety
- Results of software analyses
- Quantification of risk.

Physical devices vs. software

- They "fail" differently: failure vs. design errors

- No historical reliability and safety assessments on standard designs for software.

- Repair involves redesign rather than replacement by equivalent part.

Software Reliability Models

- Estimation of reliability model parameters made from measurements of time between failures during testing.
- Most of controversy rests on assumptions models make about software. Some typical ones:
  - Software faults, and thus the failures they may cause, are independent of each other.
  - Inputs for software are selected randomly from an input space.
  - Test-input space is representative of the operational input space.
  - Software under test constitutes a functional unit to which no new software modules are added during testing.
  - Each software failure is observed.
  - Faults are corrected without introducing new ones.
  - All errors are of equal severity.
  - Each fault contributes equally to the failure rate.
  - No major revisions or changes in staffing or aspects of development or maintenance environment.

- Even if believe models, cannot exercise enough during testing to provide very low failure probabilities with high confidence.

Doug Miller:

> To assure failure rate less than 10^{-9} failures/hour, must test for more than 10^9 hours and experience no failures (110,000 years of testing).
>
> To be 99% confident that failure probability less than 10^{-9} requires 4.6×10^9 test cases without failure (525,000 years of testing if unit of time hours and assume reasonable amount of time to execute a test case).

CONCLUSIONS

- Standard reliability and fault tolerance techniques will not solve the safety problem for the present.

- A new attitude required:

  Looking at what you do *not* want software to do along with what you want it to do.

  Assuming things will go wrong.

- New procedures and changes to entire software development process will be necessary.

  Special software safety analysis techniques are needed.

  Design techniques, especially eliminating complexity, will help.

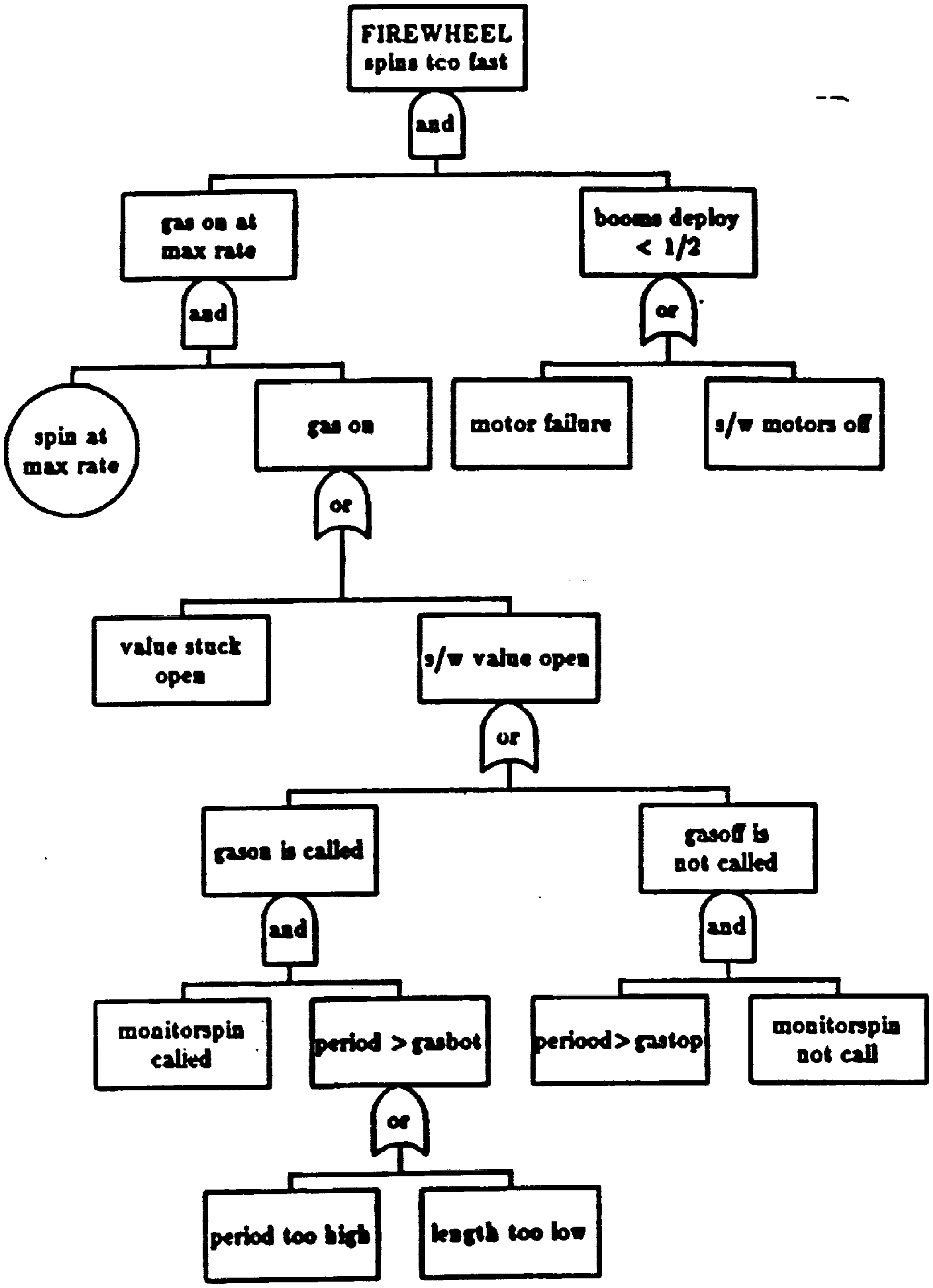
FIREWHEEL Example

Used SFTA to analyze the flight and telemetry program for a U.C. Berkeley spacecraft.

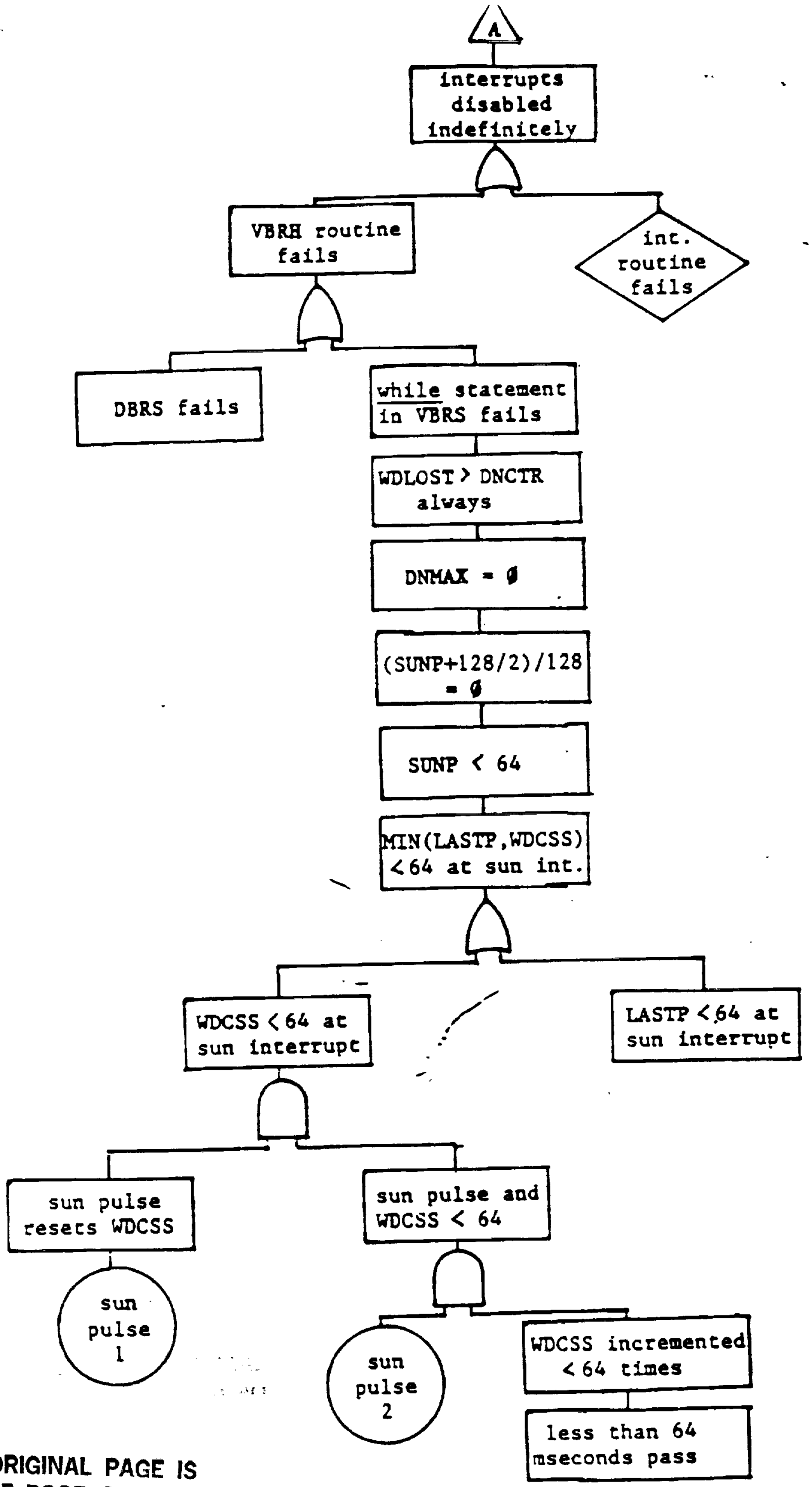
- **Mission**: to sample electric fields in the earth's magnetotail.
- **Critical Failure Event**: ripping wire booms off spacecraft.
- **Cost**: needed to examine 12% of code (out of approximately 300 lines of Pascal code), took two man days
- **Results**: A critical scenario detected that was undiscovered during a thorough test and evaluation by an independent group.

PETER HARVEY

figure 7. Firewheel Spin Control – Software Opens gas Value
file location: 7
initial fault: FireWheel spins too fast

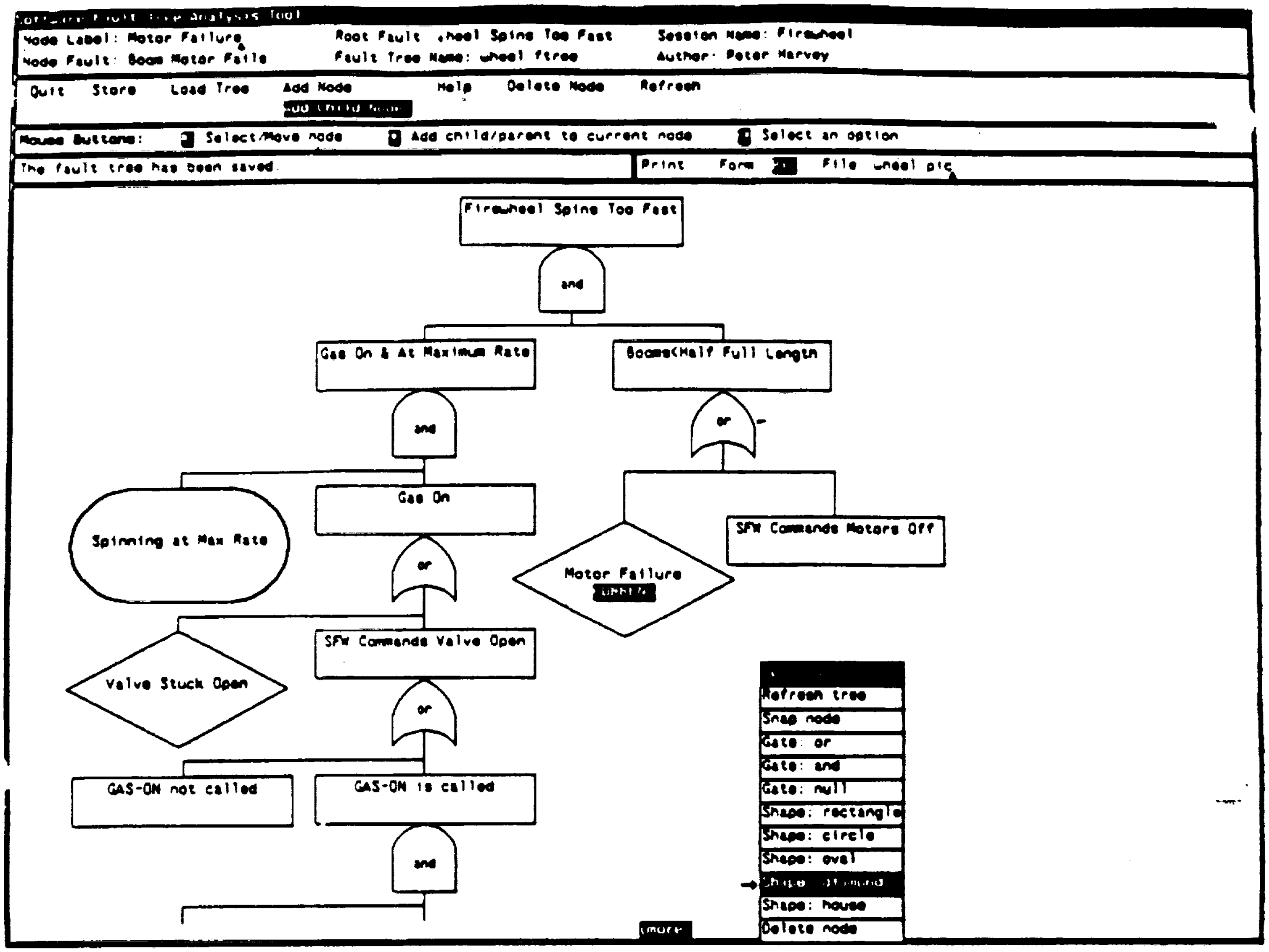


ORIGINAL PAGE IS OF POOR QUALITY



A
interrupts disabled indefinitely
VBRH routine fails
inc. routine fails
DBRS fails
while statement in VBRS fails
WDLOST > DNCTR always
DNMAX = Ø
(SUNP+128/2)/128 = Ø
SUNP < 64
MIN(LASTP,WDCSS) <64 at sun int.
WDCSS < 64 at sun interrupt
LASTP < 64 at sun interrupt
sun pulse resets WDCSS
sun pulse and WDCSS < 64
sun pulse 1
sun pulse 2
WDCSS incremented < 64 times
less than 64 mseconds pass

Figure 9b. Boom Length Too Low (continued)



Software Fault Tree Analysis Tool

Node Label: Motor Failure   Root Fault: wheel Spins Too Fast   Session Name: Firewheel

Node Fault: Boom Motor Fails   Fault Tree Name: wheel ftree   Author: Peter Harvey

Quit   Store   Load Tree   Add Node   Help   Delete Node   Refresh

Add Child Node

Mouse Buttons: Select/Move node   Add child/parent to current node   Select an option

The fault tree has been saved.

Print   Form   File wheel pic

Software Fault Tree Analysis useful to:

- identify software faults potentially leading to accidents OR "verify" (increase confidence) they do not exist
- provide information for testing
- provide guidance for content and placement of run-time assertions (fault detection)
- provide information on fail-safe requirements

ASSESSMENT OF RISK

Certification of system partly based on system safety report including

- Description of procedures used to ensur~~ing~~e software safety
- Results of software analyses
- Quantification of risk.

Physical devices vs. software

- They "fail" differently: failure vs. design errors

- No historical reliability and safety assessments on standard designs for software.

- Repair involves redesign rather than replacement by equivalent part.

Software Reliability Models

- Estimation of reliability model parameters made from measurements of time between failures during testing.

- Most of controversy rests on assumptions models make about software. Some typical ones:

  — Software faults, and thus the failures they may cause, are independent of each other.

  — Inputs for software are selected randomly from an input space.

  — Test-input space is representative of the operational input space.

  — Software under test constitutes a functional unit to which no new software modules are added during testing.

  — Each software failure is observed.

  — Faults are corrected without introducing new ones.

  — All errors are of equal severity.

  — Each fault contributes equally to the failure rate.

  — No major revisions or changes in staffing or aspects of development or maintenance environment.

- Even if believe models, cannot exercise enough during testing to provide very low failure probabilities with high confidence.

Doug Miller:

To assure failure rate less than 10^{-9} failures/hour, must test for more than 10^{9} hours and experience no failures (110,000 years of testing).

To be 99% confident that failure probability less than 10^{-9} requires 4.6×10^{9} test cases without failure (525,000 years of testing if unit of time hours and assume reasonable amount of time to execute a test case).

CONCLUSIONS

- Standard reliability and fault tolerance techniques will not solve the safety problem for the present.

- A new attitude required:

  Looking at what you do *not* want software to do along with what you want it to do.

  Assuming things will go wrong.

- New procedures and changes to entire software development process will be necessary.

  Special software safety analysis techniques are needed.

  Design techniques, especially eliminating complexity, will help.