Chapter

15

Software System Analysis Using Fault Trees

Joanne Bechta Dugan
University of Virginia

15.1 Introduction

This chapter will introduce you to the use of fault trees for the analysis of software systems. Fault trees provide a graphical and logical framework for analyzing the failure modes of systems. Their use helps the analyst to assess the impact of software failures on an overall system, or to prove that certain failure modes cannot occur (or occur with negligible probability). Fault tree models provide a conceptually simple modeling framework that can be used to compare different design alternatives or architectures for fault tolerance.

After introducing the fault tree modeling technique, we briefly discuss both qualitative and quantitative analysis techniques. We then describe two uses of fault trees as a design aid for software systems. Fault tree models can help the designer to efficiently combine off-line and on-line tests to prevent or detect software failures. Software safety validation uses software fault trees to qualitatively examine safety-critical software on a fine-grain (statement-by-statement) basis. On a coarser scale, fault trees can be used to qualitatively and quantitatively analyze fault-tolerant software systems. We consider both a qualitative and quantitative analysis of software-fault-tolerant systems, with respect to both reliability and safety. The need for parameter values for the software part of the models provides a case study for parameter estimation from experimental data.

15.2 Fault Tree Modeling

A fault tree model is a graphical representation of logical relationships between events (usually failure events). Fault trees were first devel-

oped in the 1960s 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 that 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 noncoherent fault tree [Henl82], 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 Fig. 15.1, 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 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 of a leak in the tub. 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.

15.2.1 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 is a cutset with no redundant elements; 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 15.2 shows an example fault tree whose cutset generation is shown in Fig. 15.3. The undesired top event occurs when either subevents G2 or G3 occur, which are themselves *AND* combinations of

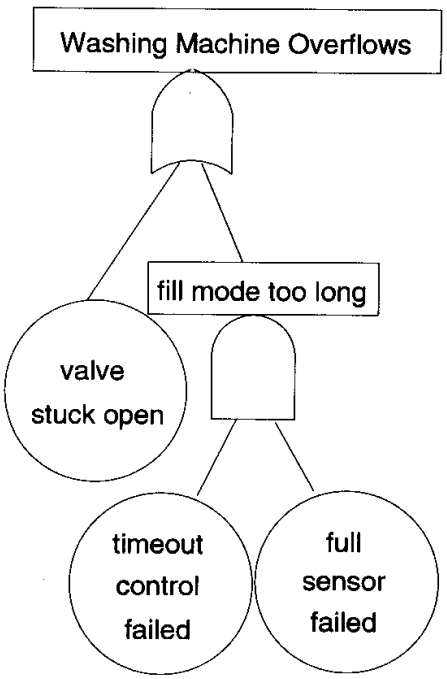


Figure 15.1 A simple fault tree model.

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 failure along two paths.

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

If a gate being expanded is a *k-out-of-n* gate, then its expansion is a combination of the *OR* and *AND* expansions. The *k-out-of-n* gate is expanded into the $\binom{n}{k}$ combinations of input events that can cause the gate to occur. For example, consider a cutset with a gate Gx that is a *3-out-of-4* gate, with inputs $I1$, $I2$, $I3$, and $I4$. Gate Gx gets split into four cutsets, replacing Gx with the four possibilities for selecting three of the inputs, namely, $\{I1, I2, I3\}$, $\{I1, I3, I4\}$, $\{I1, I2, I4\}$, and $\{I2, I3, I4\}$.

When using such an algorithm for generating cutsets, some reduction might be necessary. If a cutset contains the same basic event more

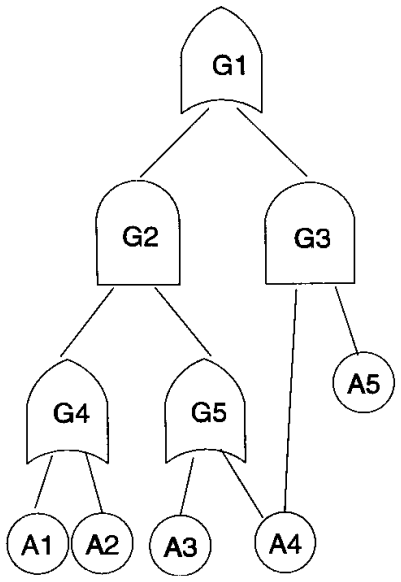


Figure 15.2 An example fault tree.

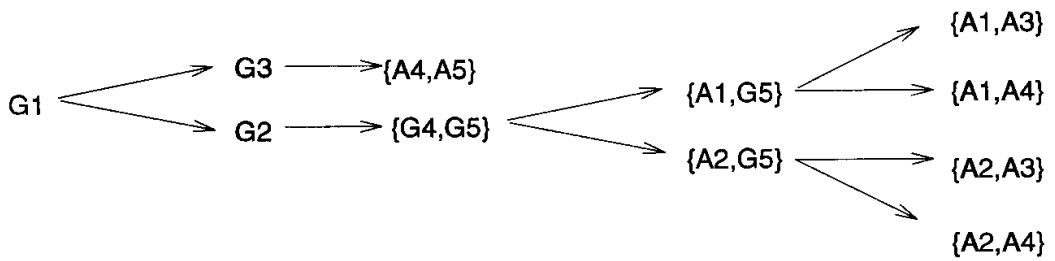


Figure 15.3 Cutset generation for a fault tree.

than once, then the redundant entries can be eliminated. If one cutset is a subset of another, the latter can be removed from further consideration, 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} }.

15.2.2 Fault tree analysis

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 Fig. 15.2, all cutsets had 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.

$$P\{\text{system failure}\} = P\{\bigcup_i C_i\} \tag{15.1}$$

where 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 probability 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.

Several methods exist for the evaluation of Eq. (15.1) [Henl82, Shoo90], 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:

$$P\{A \cup B\} = P\{A\} + P\{B\} - P\{\mathrm{A} \cap \mathrm{B}\} \tag{15.2}$$

and is given by

$$P\{\bigcup_{i=1}^{n} C_i\} = \sum_{i=1}^{n} P\{C_i\}$$

$$- \sum_{i<j} P\{C_i \bigcap C_j\}$$

$$+ \sum_{i<j<k} P\{C_i \bigcap C_j \bigcap C_k\} \tag{15.3}$$

$$\mp \cdots$$

$$\pm P\{\bigcap_{i=1}^{n} C_i\}$$

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 (15.3) 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 Eq. (15.3).

Consider the example fault tree shown in Fig. 15.2, whose cutsets are

$$C_1 = \{A_4, A_5\}$$

$$C_2 = \{A_1, A_3\}$$

$$C_3 = \{A_1, A_4\}$$

$$C_4 = \{A_2, A_3\}$$

$$C_5 = \{A_2, A_4\}$$

Assuming that the probability of occurrence for each of the basic events is

$$P\{A_1\} = P_{A1} = 0.05$$

$$P\{A_2\} = P_{A2} = 0.10$$

$$P\{A_3\} = P_{A3} = 0.15$$

$$P\{A_4\} = P_{A4} = 0.20$$

$$P\{A_5\} = P_{A5} = 0.25$$

then the probability of occurrence for each of the cutsets is

$$P\{C_1\} = P_{A4} \times P_{A5} = 0.05$$

$$P\{C_2\} = 0.0075$$

$$P\{C_3\} = 0.01$$

$$P\{C_4\} = 0.015$$

$$P\{C_5\} = 0.02$$

The sum of the probabilities for the singular cutsets, 0.1025, is an upper bound on the unreliability of the system.

$$0 \leq \text{unreliability} \leq 0.1025$$

The second term of Eq. (15.3) is the sum of the probability of occurrence for all the possible combinations of two cutsets; for the current example this is 0.015175. Subtracting this from the first term yields a lower bound on the unreliability of the system, 0.087325:

$$0.087325 \leq \text{unreliability} \leq 0.1025$$

Adding the third term, 0.0020875, the sum of the probabilities for all possible combinations of three cutsets yields a better upper bound:

$$0.087325 \leq \text{unreliability} \leq 0.0894125$$

Subtracting the fourth term, 0.0003, the sum of the probabilities for all the possible combinations of four cutsets yields a tighter lower bound:

$$0.0891125 \leq \text{unreliability} \leq 0.0894125$$

If we are interested in three decimal places of accuracy, the expansion can stop here, with a known unreliability of 0.089. Adding the final term, the probability that all five cutsets will occur (0.0000375) results in the exact unreliability:

$$\text{unreliability} = 0.08915$$

15.3 Fault Trees as a Design Aid for Software Systems

Although fault trees have traditionally been used to analyze hardware systems, they can provide a useful qualitative design aid for software systems as well [Hech86]. When designing robust software, fault trees can help the designer determine a good set of on-line reasonableness checks and off-line validation tests to cover a class of potential faults. In some sense, a fault tree analysis of a software system is complementary to a formal design review. A formal design review helps to ensure that a software system *does* what it *should* do. A fault tree analysis helps to ensure that a software system *does not do* what it *should not* do.

For each major function that a software system is expected to perform, there is an associated list of potential failures. For each potential failure with appreciable consequences, a fault tree model depicting the possible causes of the failure can be developed. In analyzing a software system, possible causes can include combinations of software faults, inputs, and operating modes. As in the analysis of a mechanical or hardware system, the undesired event is decomposed into its constituent causes, which become the basic events of the fault tree. In the analysis of hardware systems, the basic events are usually associated with physical component failures. In a software system, the basic events represent software modules which might produce an incorrect result or accept an invalid input. Or a basic event can represent the incorrect setting of an initialization parameter by a user or a buffer overflow.

A fault tree for a mechanical system is used to gain an understanding of how unavoidable events (component failures) can impact the system. If a system is determined to be particularly vulnerable to a certain failure mode, then the design is altered so as to reduce this vulnerability. Design alternatives might include choosing a more reliable substitute component, shielding the component from fault-inducing situations or environments, and enhancing redundancy.

How then are fault tree models applicable to software systems? A software component does not experience physical wearout or age degradation like a hardware component does. However, the sheer size and complexity of software systems currently being designed virtually guarantees that sometime in the lifetime of the system a software failure will occur. A fault tree analysis can help uncover the vulnerabilities of the system to particular classes of software failures, and can guide the designer in choosing preventive or protective measures. The basic events of a software fault tree analysis can be software modules which are decomposed to the point where they are either exhaustively testable or reliable failure detection and recovery mechanisms can be written. If a module is small enough to be exhaustively testable, then after testing, the analyst can be reasonably assured that a software

failure in that module will not occur. If, on the other hand, exhaustive testing is infeasible, it may be reasonable to define a detection and recovery routine to handle the potential failure.

The logical structure of the fault tree helps to determine where such tests or routines are best placed [Hech86]. If an undesired event in a fault tree is the output of an *OR* gate, then each input to the *OR* gate must be covered by exhaustive testing or detection/recovery routines in order to prevent the undesired outcome. If, however, the undesired event is the output from an *AND* gate, then the protection of *one* of the inputs will prevent the undesired event. At lower levels of the tree, testing and/or exception handling can be used to prevent intermediate events which ultimately can lead to the undesired top event. Testing or exception handling close to the top event in a fault tree has the greatest impact, and can sometimes provide a low-cost solution to satisfy the reliability requirements of less critical applications.

Consider again the fault tree shown in Fig. 15.2. Suppose that the basic events in the fault tree represent software modules whose failures combine to cause system failure as depicted by the fault tree. Module A4 inputs to the *AND* gate labeled G3; if module A4 does not fail, then the output of gate G3 will not occur. So, we can protect the system from one of its two major failure modes (labeled G2 and G3) by exhaustively testing module A4. But, even though we have protected G3, the output event for gate G2 could still occur. Since gate G2 is an *AND* gate also, its occurrence can be prevented by exhaustively testing A3 (in addition to A4). Alternatively, one could provide a detection and recovery routine for the gates G2 or G4. The system can be protected from software failure by exhaustively testing only two of the five modules, or by testing one and providing a single point of detection and recovery. Of course, the particulars of the software system dictate whether exhaustive testing of a module is feasible, or whether it is reasonable to insert an exception handler at the point represented by a gate in the fault tree.

The fault tree model depicting the failure modes in a software system explicitly enumerates the failure modes being considered, as well as those being ignored or overlooked. A later analysis of the system might well ask whether some particular failure mode was considered; the fault tree model can provide the answer. If the mode has been considered, it will appear in the fault tree; if not, it will be absent.

15.4 Safety Validation Using Fault Trees

In the previous section, we saw that the manual construction of a fault tree model can help a designer or analyst understand the potential failure modes of a software system, and it provides a logical structure for the placement of exception tests. In this section we will see how the for-

mal structure of the fault tree can provide the basis for automated tools to aid in safety validation.

Computer systems are often used to monitor or control a critical system where failure can have life-threatening or other severe consequences. Safety analysis of a critical system begins by identifying and classifying a set of hazards within the context of the operational environment. A *hazard* is a set of conditions (a state) that can lead to an accident, given certain environmental conditions, and can be classified as *critical* or *catastrophic,* depending on the potential outcome [Leve86, Leve91a]. A critical hazard is one that can lead to severe injury, severe occupational illness, or major system damage. A catastrophic hazard is one that can lead to death or system loss.

Once a set of critical or catastrophic system hazards is identified, fault tree analysis is used to identify the combinations of events which can lead to each hazard. The root of the fault tree is the hazard, and the causes of the hazard are resolved to the point where the critical software interfaces are identified. The critical behavior of the software (usually involving output values or missing or untimely outputs) is then analyzed using software fault trees [Leve86, Leve91a]. The goal of software fault tree analysis is to either find paths through the code from particular inputs to the hazardous outputs, or prove that such paths do not exist.

Software fault tree analysis uses failure templates for program structures to generate a fault tree model of the software. In each template, it is assumed that the execution of the statement causes the critical event, and the statement result is broken down into its constituent causes. This is similar to the approach used in formal axiomatic verification, where the weakest preconditions necessary to satisfy the given postconditions are derived. In fact, software fault tree analysis is a graphical application of axiomatic verification where the postconditions describe the hazardous conditions rather than the correctness conditions [Leve91a]. If the fault tree analysis leads to a contradiction (a false weakest precondition), then the analysis on this branch can be halted. If the fault tree analysis leads back to an input statement without reaching a contradiction, then the code must be redesigned to avoid the hazard.

Figures 15.4 and 15.5 [Leve91a] show example templates for the Assignment and If-Then-Else statements. The assignment statement fault tree shows that an assignment statement can fail because of an unsafe value or because of unsuccessful execution. The If-Then-Else statement can fail because of the condition evaluation, the "If" part or the "Then" part of the statement. The body of the Else and Then parts of the statements are replaced by the templates associated with the statement body.

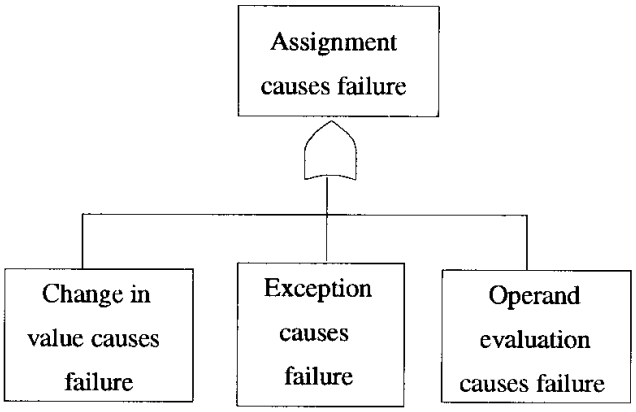


Figure 15.4 Template for assignment statement

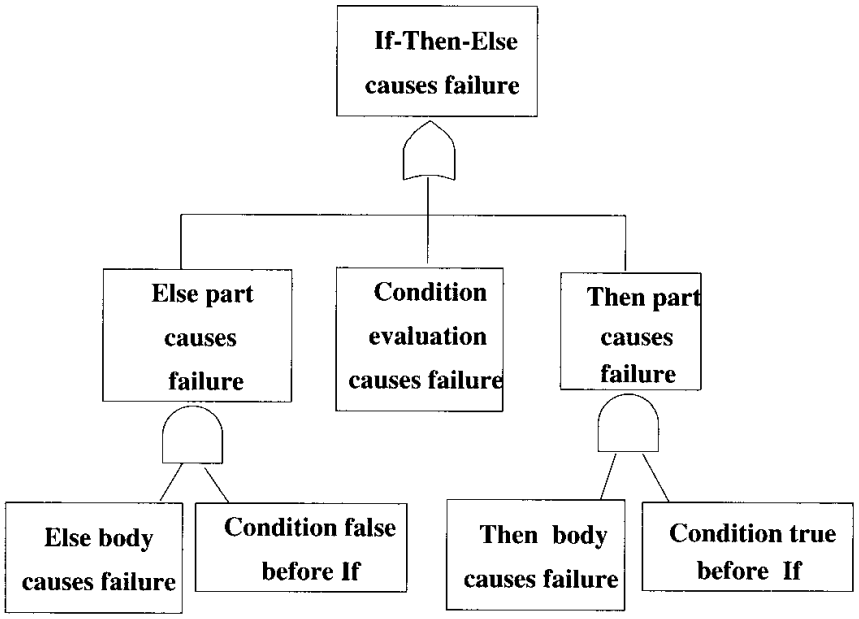


Figure 15.5 Template for If-Then-Else statement.

Consider the simple example of a program segment shown in Fig. 15.6, and suppose that the system fails if $x > 14$. The software fault tree model for this program segment is shown in Fig. 15.7. The software fault tree can be derived automatically from the program segment [Frie93], and can then be analyzed and reduced by the (human) analyst. Each of the "Then" and "Else" parts of the statement reduce to an AND condition. The software fault tree then reduces to that shown in Fig. 15.8. Each of the conditions at the lowest level of the fault tree in Fig. 15.8 can be further decomposed until either a contradiction is reached (which then proves that the hazard cannot occur) or until it is determined that a software repair is needed.

While software fault tree analysis has been applied to relatively small, industrial software at a reasonable cost, the practicality of its use has not been demonstrated on large-scale software [Leve91a]. However,

```
If y<6
   then x := z + y
   else
      begin
         q := 3
         x := (y+1)*q;
      end
```

Figure 15.6 A simple example program statement used for software fault tree analysis.

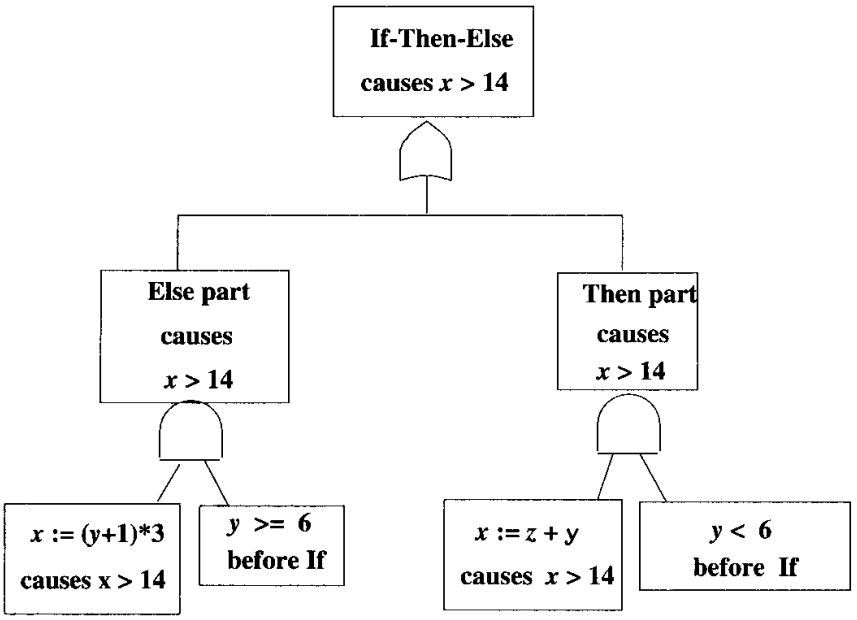


Figure 15.7 Software fault tree for program segment.

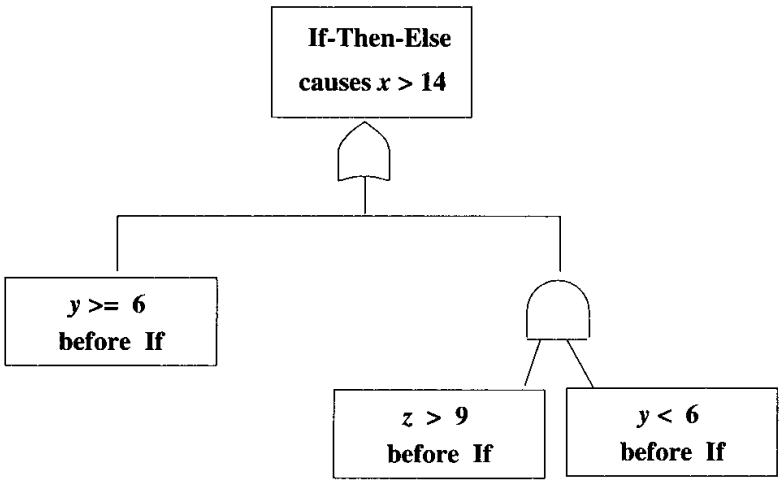


Figure 15.8 Reduced software fault tree for program segment.

since the software fault tree analysis is limited to identified hazards, if the error containment and isolation are satisfactory, only a small percentage of the code may need to be analyzed in detail. Also, the fault tree analysis can be stopped before reaching a contradiction if a run-time assertions or exception conditions are included in the code. The software fault tree provides the necessary information regarding content and placement of effective run-time assertions. Because run-time assertions are expensive, the guidance provided by software fault trees is useful from a practical consideration.

15.5 Analysis of Fault-Tolerant Software Systems

In addition to the detailed analysis of single-version critical software, fault tree models are useful for analyzing the failure modes associated with fault-tolerant software applications. Software fault trees for safety verification operate at a very fine granularity, considering each internal statement of the code. System analysis of fault-tolerant software, on the other hand, considers fault activation scenarios as basic events, and analyzes system-level failures. We will present fault tree models of three popular architectures for tolerating software faults: distributed recovery blocks, N-version programming, and N self-checking programming.

Figure 15.9 shows the hardware and error confinement areas [Lapr90] associated with the three architectures being considered. The systems are defined by the number of software variants, the number of hardware replications, and the decision algorithm. The hardware error confinement area (HECA) is the lightly shaded region; the software error confinement area (SECA) is the darkly shaded region. The HECA or SECA covers the region of the system affected by faults in that component. For example, the HECA covers the software component since the software component will fail if that hardware experiences a fault.

We make the following assumptions for this analysis.

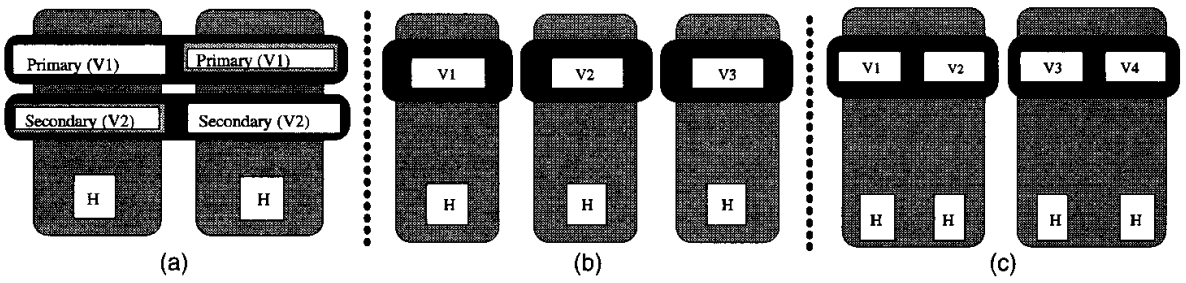


Figure 15.9 Structure of (*a*) distributed recovery block (DRB); (*b*) N-version programming (NVP); (*c*) N self-checking programming (NSCP).

Task computation. The computation being performed is a task (or set of tasks) which is repeated periodically. A set of sensor inputs is gathered and analyzed and a set of actuations are produced. Each repetition of a task is independent. The goal of the analysis is the probability that a task will succeed in producing an acceptable output, despite the possibility of hardware or software faults. More interesting task computation processes could be considered using techniques described in [Lapr92] and [Wei91]. We do not address timing or performance issues in this model. See [Tai93] for a performability analysis of fault-tolerant software techniques.

Software failure probability. Software faults exist in the code despite rigorous testing. A fault may be activated by some random input, thus producing an erroneous result. Each instantiation of a task receives a different set of inputs which are independent. Thus, a software task has a fixed probability of failure when executed, and each iteration is assumed to be statistically independent. Since we do not assign a failure rate to the software, we do not consider reliability growth models.

Coincident software failures in different versions. If two different software versions fail on the same input, they will produce either similar or different results. In this work, we use the Arlat/Kanoun/Laprie [Arla90] model for software failures and assume that similar erroneous results are caused by *related* software faults, and that different erroneous results that are simultaneously activated are caused by *unrelated* (called *independent* in their terminology) software faults. There is one difference between our model and that of Arlat/Kanoun/Laprie: our model assumes that related and unrelated software faults are statistically independent, while their's assumes that related and unrelated faults are mutually exclusive. Further, this treatment of *unrelated* and *related* faults differs considerably from models for correlated failures [Eckh85, Litt89, Nico90], in which unrelated and related software failures are not differentiated. Rather, software faults are considered to be statistically correlated and models for correlation are considered and proposed. A more detailed comparison of the two approaches is given in [Duga94a].

Transient hardware faults. A transient hardware fault is assumed to upset the software running on the processor and to produce an erroneous result that is indistinguishable from an input-activated software error. We assume that the lifetime of transient hardware faults is short compared to the length of a task computation, and thus we assign a fixed probability to the occurrence of a transient hardware fault during a single computation. Permanent hardware faults are considered separately, in Sec. 15.7.

Fault tree models will be used to describe the combinations of failure events (hardware and software) which can combine to produce an unacceptable output. This analysis will ignore failures of the common platform services (communication network, operating system, device drivers, etc.) and will concentrate on the fault tolerance of the application software instead.

Each of the fault tree models will use the following notation for basic events.

V# (where # is an integer between 1 and 4) For (up to) four versions of software, the input for a single computation activates an unrelated fault.

D An independent fault in the decider (acceptance test, majority voter, comparator, adjudicator).

RV## (where each # is an integer between 1 and 4) The input for a single computation activates a related fault between two versions. A related fault is one that occurs in two different versions, causing both to produce the same erroneous result.

RALL A related fault affects all versions as well as the decider, caused by imperfect specifications.

H# (where # is an integer between 1 and 4) A hardware fault affects the task computation.

15.5.1 Fault tree model for recovery block system

There are at least two different ways to combine hardware redundancy with the recovery block approach to software fault tolerance. In [Lapr90a], the RB/1/1 architecture duplicates the recovery block on two hardware components. Both hardware components execute the same variant, and hardware faults are detected by a comparison of the acceptance test and computation results. The distributed recovery block (DRB) advocated by Kim and Welch [Kim89] executes different alternates on the different hardware components in order to improve performance in case an error is detected. In the DRB system, one processor executes the primary alternate, while the other executes the secondary. If an error is detected in the primary results, the results from the secondary are immediately available. The fault tree model of both systems is identical.

The fault tree model of DRB is shown in Fig. 15.10. A single task computation will produce unacceptable results if one of three events occur. First, if both the primary and secondary fail on the same input because of two unrelated faults or a single related fault. Second, if both hardware components experience faults, then the computations being hosted will be upset and be unable to produce correct results. Third, if

the decider fails to either detect unacceptable results or to accept correct results, then the computation fails.

The fault tree model provides a compact format for describing the effects of both software and hardware faults. Even without generating the cutsets for the DRB system, one can easily visualize the effects of a decider failure or a related fault between the versions. There are five minimal cutsets for the DRB system. Three of these cutsets have a single element, those being *RV, RALL,* and *D,* as a related fault and a decider fault are single points of failure. The other two cutsets have two members, one for both versions failing, one for both hardware hosts failing. The probability that an unacceptable result is produced during a single task iteration is given by

$$P_{RV} + Q_{RV}P_D + Q_{RV}P_{RALL}Q_D + Q_{RV}Q_{RALL}Q_D P_H{}^2 + P_V{}^2 Q_{RV}Q_{RALL}Q_D(1 - P_H{}^2) \tag{15.4}$$

where P_X is the probability that event X occurs, and $Q_X = 1 - P_X$.

15.5.2 Fault tree model for N-version programming system

The example NVP system consists of three identical hardware components, each running a distinct software version. It is a direct mapping of the NVP method onto hardware. If any of the hardware components experiences a fault, it causes the software version running on that

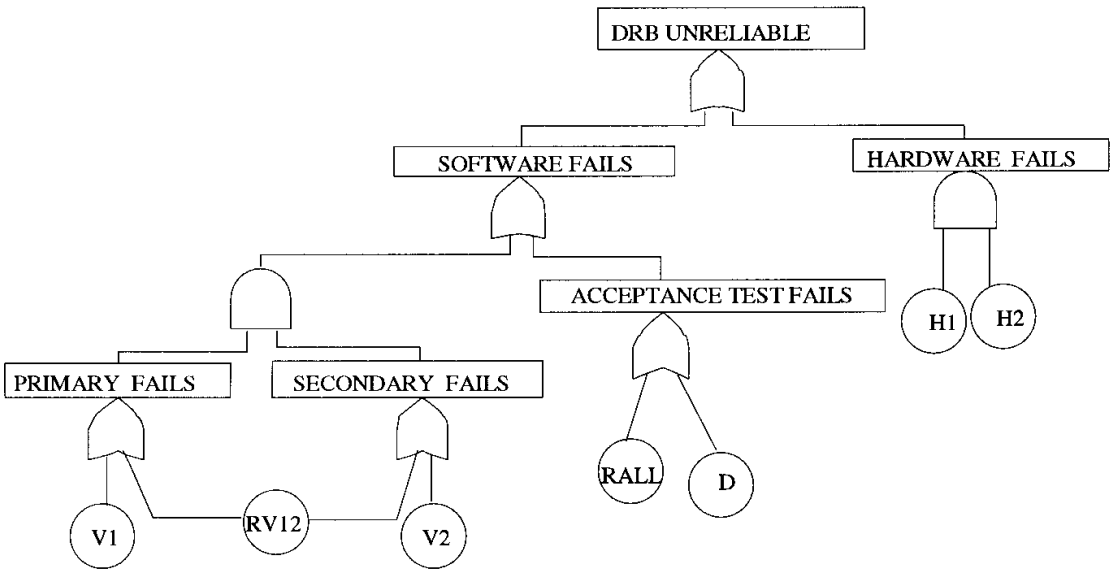


Figure 15.10 Reliability model of DRB.

hardware to produce inaccurate results. If all of the other hardware and software modules are functioning properly, the system will still produce a correct result, since two out of the three versions, a majority, are correct. If two software or hardware components have faults, the system will fail, since there will be only one correct result. The system will also fail if at least one hardware component and one software component fail, but only if the faulty hardware does not host the faulty software version.

Figure 15.11 shows the fault tree model of NVP. With three software versions running on three separate processors, several different failure scenarios must be considered, including coincident unrelated faults as well as related software faults, and combinations of hardware and software faults. A single task iteration can fail from several causes: first, if two of the three versions activate unrelated faults, or if any related fault between two versions is activated; second, if the input activates a fault which affects all three versions or a fault in the decider; third, if two of the three processors experience faults during the same task; finally, if a hardware host fails and one of the software components on another host also fails (via an unrelated or related fault).

The NVP fault tree has 20 minimal cutsets, of which 5 have a single member (*RV*12, *RV*13, *RV*23, *RALL,* and *D*). The remaining cutsets enumerate the 15 ways in which software and/or hardware components combine to fail two of the three versions. The solution of the NVP fault tree model is given by

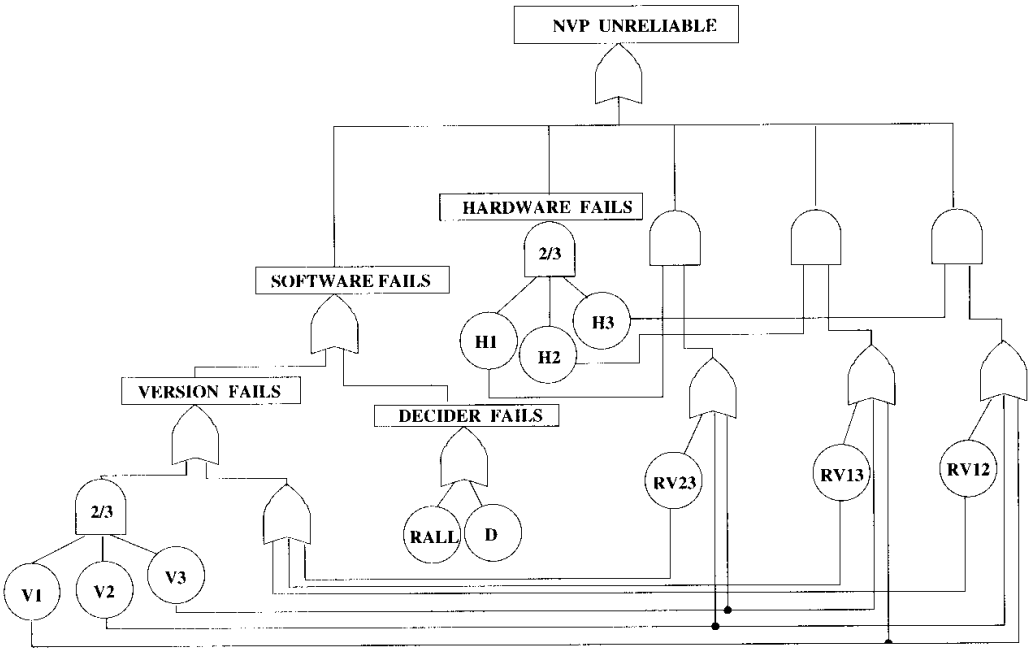


Figure 15.11 Reliability model of NVP.

$$
\begin{aligned}
&P_{RV} + \\
&Q_{RV}P_{RV} + \\
&Q_{RV}{}^{2}P_{RV} + \\
&Q_{RV}{}^{3}P_{D} + \\
&Q_{RV}{}^{3}P_{RALL}Q_{D} + \\
&P_{V}{}^{2}Q_{RV}{}^{3}Q_{RALL}Q_{D} + \\
&P_{V}{}^{2}Q_{V}Q_{RV}{}^{3}Q_{RALL}Q_{D} + \\
&P_{V}{}^{2}Q_{V}Q_{RV}{}^{3}Q_{RALL}Q_{D} + \\
&Q_{V}P_{V}Q_{RV}{}^{3}Q_{RALL}Q_{D}P_{H}{}^{2}Q_{H}(1-P_{V}) + \\
&Q_{RV}{}^{3}Q_{RALL}Q_{D}P_{H}{}^{2}Q_{H}(1-P_{V})(1-P_{V}{}^{2}) + \\
&Q_{V}P_{V}Q_{RV}{}^{3}Q_{RALL}Q_{D}P_{H}{}^{2}Q_{H}(1-P_{V}) + \qquad (15.5)\\
&Q_{RV}{}^{3}Q_{RALL}Q_{D}P_{H}{}^{2}Q_{H}(1-P_{V})(1-P_{V}{}^{2}) + \\
&Q_{V}P_{V}Q_{RV}{}^{3}Q_{RALL}Q_{D}P_{H}{}^{2}Q_{H}(1-P_{V}) + \\
&Q_{RV}{}^{3}Q_{RALL}Q_{D}P_{H}{}^{2}Q_{H}(1-P_{V})(1-P_{V}{}^{2}) + \\
&Q_{V}P_{V}Q_{V}Q_{RV}{}^{3}Q_{RALL}Q_{D}P_{H}Q_{H}(1-P_{H}) + \\
&Q_{V}{}^{2}P_{V}Q_{RV}{}^{3}Q_{RALL}Q_{D}P_{H}Q_{H}(1-P_{H}) + \\
&P_{V}Q_{V}Q_{RV}{}^{3}Q_{RALL}Q_{D}Q_{H}{}^{2}P_{H}(1-P_{V}) + \\
&Q_{V}{}^{2}P_{V}Q_{RV}{}^{3}Q_{RALL}Q_{D}Q_{H}{}^{2}P_{H} + \\
&P_{V}Q_{V}Q_{RV}{}^{3}Q_{RALL}Q_{D}Q_{H}{}^{2}P_{H}(1-P_{V}) + \\
&Q_{V}P_{V}Q_{V}Q_{RV}{}^{3}Q_{RALL}Q_{D}Q_{H}{}^{2}P_{H}
\end{aligned}
$$

15.5.3 Fault tree model for N self-checking programming system

The example NSCP architecture is comprised of four software versions and four hardware components, each grouped in two pairs, essentially dividing the system into two halves. The hardware pairs operate in hot

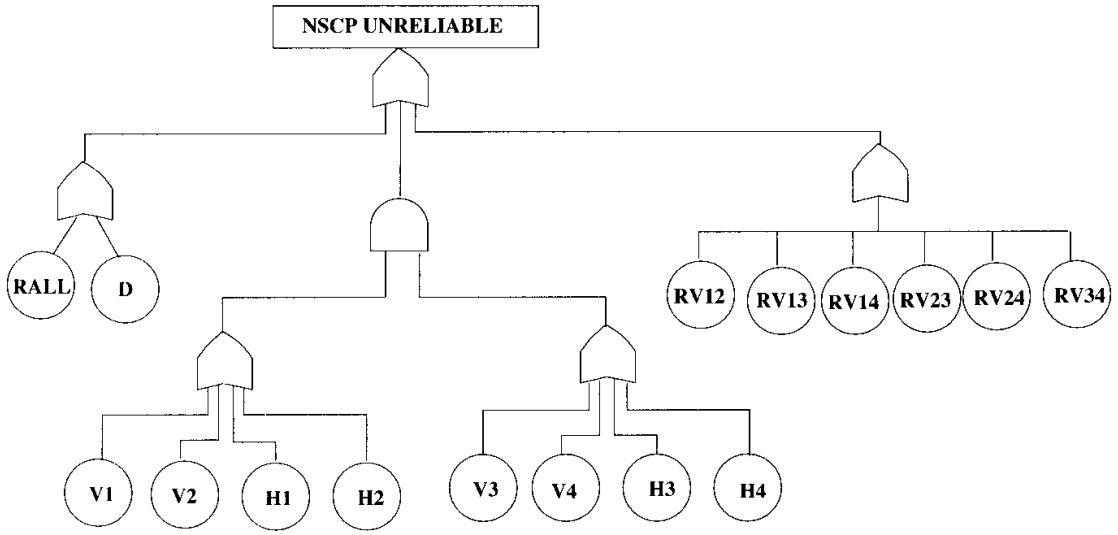


Figure 15.12 Reliability model of NSCP.

standby redundancy with each hardware component supporting one software version. The version pairs form self-checking software components. A self-checking software component consists of either two versions and a comparison algorithm or a version and an acceptance test. In this case, error detection is done by comparison. The four software versions are executed and the results of V1 and V2 are compared against each other, as are the results of V3 and V4. If either pair of results do not match, they are discarded and only the remaining two are used. If the results do match, the results of the two pairs are then compared. A hardware fault causes the software version running on it to produce incorrect results, as would a fault in the software version itself. This results in a discrepancy in the output of the two versions, causing that pair to be ignored.

The fault tree model of the NSCP system (Fig. 15.12) shows that the system is vulnerable to related faults, whether they involve versions in the same error confinement area or not. (To allow later comparison with the NVP and DRB systems, we have ignored the possibility of a related fault affecting three versions.) There are 24 cutsets for the NSCP system. Six singleton cutsets enumerate the related faults affecting two versions, and two singleton cutsets reflect the single points of failure in the decider and in a related fault affecting all versions (*RALL*). There are also 16 cutsets with two elements, enumerating all combinations of version and/or hardware failures affecting one version in each error confinement area. The solution of the model is given by

$$
\begin{aligned}
&P_{RV} + \\
&Q_{RV}P_{RV} + \\
&Q_{RV}{}^2P_{RV} + \\
&Q_{RV}{}^3P_{RV} + \\
&Q_{RV}{}^4P_{RV} + \\
&Q_{RV}{}^5P_{RV} + \\
&Q_{RV}{}^6P_{RALL} + \\
&Q_{RV}{}^6P_DQ_{RALL} + \\
&P_V{}^2Q_{RV}{}^6Q_DQ_{RALL} + \\
&P_V{}^2Q_VQ_{RV}{}^6Q_DQ_{RALL} + \\
&P_VQ_V{}^2Q_{RV}{}^6Q_DQ_{RALL}P_H + \\
&P_VQ_V{}^2Q_{RV}{}^6Q_DQ_{RALL}Q_HP_H + \\
&Q_VP_V{}^2Q_{RV}{}^6Q_DQ_{RALL} + \\
&P_V{}^2Q_VQ_{RV}{}^6Q_DQ_{RALL}(1-P_V) + \\
&P_VQ_V{}^2Q_{RV}{}^6Q_DQ_{RALL}P_H(1-P_V) + \\
&P_VQ_V{}^2Q_{RV}{}^6Q_DQ_{RALL}Q_HP_H(1-P_V) + \\
&Q_V{}^2P_VQ_{RV}{}^6Q_DQ_{RALL}P_H + \\
&Q_VP_VQ_{RV}{}^6Q_DQ_{RALL}P_H(1-P_V)^2 + \\
&Q_V{}^2Q_{RV}{}^6Q_DQ_{RALL}P_H{}^2(1-P_V)^2 + \\
&Q_V{}^2Q_{RV}{}^6Q_DQ_{RALL}P_H{}^2Q_H(1-P_V)^2 + \\
&Q_V{}^2P_VQ_{RV}{}^6Q_DQ_{RALL}Q_HP_H + \\
&Q_VP_VQ_{RV}{}^6Q_DQ_{RALL}P_H(1-P_V)^2(1-P_H) + \\
&Q_V{}^2Q_{RV}{}^6Q_DQ_{RALL}P_H{}^2(1-P_V)^2(1-P_H) + \\
&Q_V{}^2Q_{RV}{}^6Q_DQ_{RALL}P_H{}^2Q_H(1-P_V)^2(1-P_H)
\end{aligned}
\tag{15.6}
$$

15.6 Quantitative Analysis of Fault-Tolerant Software

The qualitative analysis of the fault-tolerant software systems described in the previous section was useful for understanding system structure and behavior. However, it is difficult to compare the systems unless the probabilities associated with the basic events are known. The fact that one system has more cutsets (or even more singleton cutsets) does not necessarily mean that it is less reliable. A quantitative assessment of the probability of failure using a common set of assumptions and parameter values provides a clearer comparison.

15.6.1 Methodology for parameter estimation from experimental data

Given the probability of occurrence for the basic events, the determination of the probability of system failure is relatively straightforward, using well-known methods [Henl82, Shoo90]. The estimation of parameters provides a more difficult problem. The estimation of failure probabilities for hardware components has been considered for a number of years, and reasonable estimates exist for generic components (such as a processor). However, the estimation of software version failure probability is less accessible, and the estimation of the probability of related faults is more difficult still. In this section we will describe the methodology for estimating model parameter values from experimental data, followed by a case study using a set of experimental data.

Several experiments in multiversion programming have been performed, as described in Chap. 14. Among other measures, most experiments provide some estimate of the number of times different versions fail coincidentally. For example, the NASA Langley Research Center study involving 20 programs from four universities [Eckh91] provides a table listing how many instances of coincident failures were detected. The Knight-Leveson study of 28 versions [Knig86] provides an estimated probability of coincident failures. The Lyu-He study [Lyu93a] considered three- and five-version configurations formed from 12 different versions. These sets of experimental data can be used to estimate the probabilities for the basic events in a fault tree model of a fault tolerant software system.

Coincident failures in different versions can arise from two distinct causes. First, two (or more) versions may both experience unrelated faults that are activated by the same input. If two programs fail independently, there is always a finite probability that they will fail coincidentally, else the programs would not be independent. A coincident failure does not necessarily imply that a related fault has been acti-

vated. Second, the input may activate a fault that is related between the two versions. In order to estimate the probabilities of unrelated and related faults, we will determine the (theoretical) probability of failure by unrelated faults. To the extent that the observed frequency of coincident faults exceeds this value, we will attribute the excess to related faults.

The experimental data are necessarily coarse. As it is infeasible to exhaustively test a single version; it is more difficult to exhaustively observe every possible instance of coincident failures in multiple versions. The sampling techniques used to gather the experimental data provide an estimate of the probabilities of coincident failures rather than the exact value. Considering the coarseness of the experimental data, we will limit ourselves to the estimation of three parameter values: P_V, the probability of an unrelated fault in a single version; P_{RV}, the probability of a related fault between two versions; and P_{RALL}, the probability of a related fault in all versions. To attempt to estimate more (for example, the probability of a related fault that affects exactly three versions or exactly four versions) seems unreasonable. Notice that we will assume that the versions are all statistically identical, and we do not attempt to estimate different probabilities of failure for each individual version or for each individual case of two simultaneous versions.

The following notation will be used throughout the remainder of this section.

N	The number of different versions involved in the experiment
P_V	The estimated probability (for each version) that an unrelated fault is activated in a single version
P_{RV}	The estimated probability (for each pair of versions) that a related fault affects two versions
P_{RALL}	The estimated probability that a related fault affects all versions
F_i	The observed frequency that i of the N versions fail coincidentally

The first parameter that we estimate is P_V, the probability that a single version fails unrelated. The estimate for P_V comes from considering F_0 (the observed frequency of no failures in the N versions) and F_1 (the observed frequency of exactly one failure in the N versions). When considering N different versions processing the same input, the probability that there are no failures is set equal to the observed frequency of no failures.

$$F_0 = (1 - P_V)^N (1 - P_{RV})^{\binom{N}{2}} (1 - P_{RALL}) \tag{15.7}$$

Then, considering the case where only a single failure occurs, we observe that a single failure can occur in any of the N programs, which implies that a related fault does not occur (else more than one version

would be affected). This is then set equal to the observed frequency of a single failure of the N versions.

$$F_1 = N(1 - P_V)^{(N-1)}P_V(1 - P_{RV})^{\binom{N}{2}}(1 - P_{RALL}) \tag{15.8}$$

Dividing Eq. (15.7) by Eq. (15.8) yields an estimate for P_V.

$$P_V = \frac{F_1}{NF_0 + F_1} \tag{15.9}$$

Estimating the probability of a related fault between two versions, P_{RV}, is more involved, but follows the same basic procedure. First, consider the case where exactly two versions are observed to fail coincidentally. This event can be caused by one of three events:

- The simultaneous activation of two unrelated faults
- The activation of a related fault between two versions
- Both (the activation of two unrelated and a related fault between the two versions)

The probabilities of each of these events will be determined separately. The probability that unrelated faults are simultaneously activated in two versions (and no related faults are activated) is

$$\binom{N}{2}P_V^2(1 - P_V)^{(N-2)}(1 - P_{RV})^{\binom{N}{2}}(1 - P_{RALL}) \tag{15.10}$$

The probability that a single related fault (and no unrelated fault) is activated is given by

$$\binom{N}{2}(1 - P_V)^N P_{RV}(1 - P_{RV})^{\left(\binom{N}{2}-1\right)}(1 - P_{RALL}) \tag{15.11}$$

Finally, the probability that both an unrelated fault and two related faults are simultaneously activated is given by

$$\binom{N}{2}P_V^2 P_{RV}(1 - P_V)^{(N-2)}(1 - P_{RV})^{\left(\binom{N}{2}-1\right)}(1 - P_{RALL}) \tag{15.12}$$

Because the three events are disjoint, we can sum their probabilities and set the sum equal to F_2, the observed frequency of two coincident errors.

$$F_2 = \binom{N}{2}(P_V^2 + P_{RV} - P_V^2 P_{RV})(1 - P_V)^{(N-2)}(1 - P_{RV})^{\left(\binom{N}{2}-1\right)}(1 - P_{RALL}) \tag{15.13}$$

Dividing Eq. (15.13) by Eq. (15.8) and performing some algebraic manipulations yields an estimate for P_{RV} which depends on the experimental data and the previously derived estimate for P_V.

$$P_{RV} = \frac{2F_2P_V(1 - P_V) - (N - 1)F_1P_V^2}{2F_2P_V(1 - P_V) + (N - 1)F_1(1 - P_V^2)} \tag{15.14}$$

The estimate for P_{RALL} is more involved, as there are many ways in which all versions can fail. There may be a related fault between all versions that is activated by the input, or all versions might simultaneously fail from a combination of unrelated and related faults. Consider the case where there are three versions. In addition to the possibility of a single fault affecting all three versions, all three versions could experience a simultaneous activation of unrelated faults, or one of three combinations of an unrelated and related fault affecting different versions may be activated. The fault tree in Fig. 15.13 illustrates the combinations of events that can cause all three versions to fail coincidentally. A simple methodology for estimating P_{RALL} could use the previously determined estimates for P_V and P_{RV} and repeated guessing for P_{RALL} in the solution of the fault tree in Fig. 15.13, until the fault tree solution for the probability of simultaneous errors approximates the observed frequency of all versions failing simultaneously.

The fault tree model for three versions can easily be generalized to the case where there are N versions. The top event of the fault tree is an *OR* gate with two inputs, an *AND* gate showing all versions failing, and a basic event, representing a related fault that affects all versions

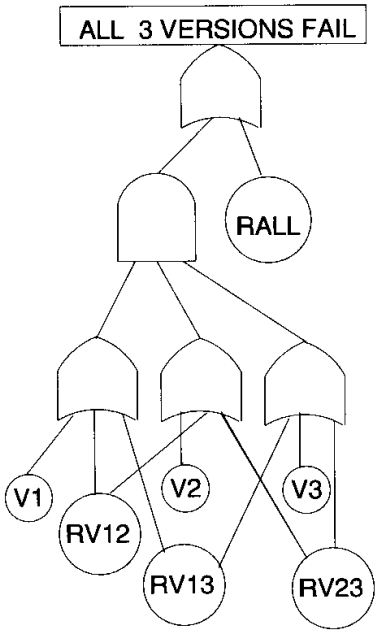


Figure 15.13 Fault tree model used to estimate P_{RALL} for a 3-version system.

simultaneously. The *AND* gate has N inputs, one for each version. Each of the N inputs to the *AND* gate is itself an *OR* gate with N inputs, all basic events. Each *OR* gate has one input representing an unrelated fault in the version, and $N - 1$ inputs representing related faults with each other possible version.

As an example parameter calculation, consider an experimental data set from an early implementation of a three-version system [Chen78, Hech86]. From a sample set of seven versions, twelve different three-version configurations were constructed. Each three-version configuration was subjected to 32 test cases, with the results as tabulated in Table 15.1. For this data set, we can estimate $P_V = 0.0109$.

$$P_V = \frac{F_1}{3F_0 + F_1} = 0.075$$

The estimate for P_{RV} is then

$$P_{RV} = \frac{2F_2P_V(1 - P_V) - (2)F_1P_V^2}{2F_2P_V(1 - P_V) + (2)F_1(1 - P_V^2)} = 1.185 \times 10^{-2}$$

The fault tree shown in Fig. 15.13 is used to estimate $P_{RALL} = 9.7 \times 10^{-3}$.

15.6.2 A case study in parameter estimation

In this section we will use the data from the Lyu-He study [Lyu93a] to determine parameter values for models of three example fault-tolerant software systems (Fig. 15.9). The systems being considered are DRB (distributed recovery block, Fig. 15.10), NVP (N-version programming, three versions, Fig. 15.11), and NSCP (N self-checking programming, Fig. 15.12).

For the Lyu-He data, two levels of granularity were used to define software execution errors and coincident errors: by case or by frame. The first level was defined based on test cases (1000 in total). If a version failed at any time in a test case, it was considered failed for the

TABLE 15.1 Experimental Results from Three-Version Programming

Number of failures	Number of test cases	Observed frequency
0	290	$F_0 = 0.755$
1	71	$F_1 = 0.185$
2	18	$F_2 = 0.047$
3	5	$F_3 = 0.013$

whole case. If two or more versions failed in the same test case (no matter at the same time or not), they were said to have coincident errors for that test case. The second level of granularity was defined based on execution time frames (5,280,920 in total). Errors were counted only at the time frame upon which they manifested themselves, and coincident errors were defined to be the multiple program versions failing at the same time frame in the same test case (with or without the same variables and values).

The 12 programs accepted in the Lyu-He experiment were configured in pairs, whose outputs were compared for each test case. Table 15.2 shows the number of times that 0, 1, and 2 errors were observed in the two-version configurations. The data from Table 15.2 yield an estimate of $P_V = 0.095$ for the probability of activation of an unrelated fault in a two-version configuration, and an estimate of $P_{RV} = 0.0167$ for the probability of a related fault for the by-case data. The by-frame data in Table 15.2 produces $P_V = 0.000026$ and $P_{RV} = 1.3 \times 10^{-7}$ as estimates.

Next, the 12 versions were configured in sets of three programs. Table 15.3 shows the number of times that 0, 1, 2, and 3 errors were observed in the three-version configurations. The data from Table 15.3 yield an estimate of $P_V = 0.0958$ for the probability of activation of an unrelated fault in a three-version configuration. Table 15.4 compares the probability of activation of 1, 2, and 3 faults as predicted by a model, assuming independence between versions, with the observed values. The observed frequency of two simultaneous errors is lower than predicted by the independent model; thus there is no support in the data for related faults affecting two versions, and P_{RV} is estimated to be zero. The observed frequency of three simultaneous errors is higher than predicted by the independent model, so we derive an estimate for P_{RALL}, based on the $P_{RV} = 0$ assumption.

Using the assumption that $P_{RV} = 0$, the probability that three simultaneous errors are activated is given by

$$F_3 = P_V^3 + P_{RALL} - P_V^3 P_{RALL} \tag{15.15}$$

yielding an estimate of $P_{RALL} = 0.0003$ for the by-case data.

TABLE 15.2 Error Characteristics for Two-Version Configurations

	By case		By frame	
Category	Number of cases	Frequency	Number of cases	Frequency
F_0 (no errors)	53,150	0.8053	348,522,546	0.99994786
F_1 (single error)	11,160	0.1691	18,128	0.00005201
F_2 (two coincident)	1,690	0.0256	46	0.00000013
Total	66,000	1.0000	348,540,720	1.00000000

TABLE 15.3 Error Characteristics for Three-Version Configurations

	By case		By frame	
Category	Number of cases	Frequency	Number of cases	Frequency
F_0 (no errors)	163,370	0.7426	1,161,707,015	0.99991790
F_1 (single error)	51,930	0.2360	94,835	0.00008163
F_2 (two coincident)	4,440	0.0202	550	0.00000047
F_4 (three coincident)	260	0.0012	0	0.00000000
Total	220,000	1.0000	1,161,802,400	1.00000000

TABLE 15.4 Comparison of Independent Model with Observed Data for Three Versions, By Case

No. errors activated	Independent model	Observed frequency
0	0.7393	0.7426
1	0.2350	0.2360
2	0.0249	0.0202
3	0.0009	0.0012

The by-frame data in Table 15.3 produces $P_V = 0.000027$ as an estimate. For this by-frame data, when the failure probabilities that are predicted by the independent model are compared to the actual data (Table 15.5), the observed frequency of two errors is two orders of magnitude higher than the predicted probability. There were no cases for which all three programs produced erroneous results. Thus, we will estimate $P_{RALL} = 0$ and derive an estimate for $P_{RV} = 1.57 \times 10^{-7}$.

The same 12 programs which passed the acceptance testing phase of the software development process were analyzed in combinations of four programs; the results are shown in Table 15.6. The by-case data from Table 15.6 yields an estimate of $P_V = 0.106$ for the probability of activation of an unrelated fault in a four-version configuration. Table 15.7 compares the probability of activation of 1, 2, 3, and 4 faults as predicted by a model, assuming independence between versions, with the observed values. The observed frequency of two simultaneous

TABLE 15.5 Comparison of Independent Model with Observed Data for Three Versions, By Frame

No. errors activated	Independent model	Observed frequency
0	0.999919	0.999918
1	0.000081	0.0000816
2	2×10^{-9}	5×10^{-7}
3	2×10^{-14}	0.0

errors is lower than predicted by the independent model, while the observed frequency of three simultaneous errors is higher than predicted. For this set of data we will assume that $P_{RV} = 0$. The observed frequency of four simultaneous failures is also lower than predicted by the independent model, so we will also assume that $P_{RALL} = 0$. The by-frame data in Table 15.6 produces $P_V = 0.000026$ and $P_{RALL} = 1.3 \times 10^{-7}$ as estimates.

15.6.3 Comparative analysis of three software-fault-tolerant systems

Table 15.8 summarizes the parameters estimated from the Lyu-He data. The parameter values for the three systems were applied to the fault tree models shown in Fig. 15.14, using both the by-case and by-frame data. The fault tree models in Figure 15.14 represent systems that use simple comparison of results for error detection, and thus do not directly relate to the DRB, NVP, and NSCP models. The predicted failure probability using the derived parameters in the fault tree models agrees quite well with the observed data, as listed in Table 15.8. The observed failure frequency for the four-version configuration is difficult to estimate because of the possibility of a 2–2 split vote. The data for

TABLE 15.6 Error Characteristics for Four-Version Configurations

	By case		By frame	
Category	Number of cases	Frequency	Number of cases	Frequency
F_0 (no errors)	322010	0.65052	2,613,781,410	0.99989519
F_1 (single error)	152900	0.30889	271,920	0.00010402
F_2 (two coincident)	16350	0.03303	2,070	0.00000079
F_3 (three coincident)	3700	0.00747	0	0.00000000
F_4 (four coincident)	40	0.00008	0	0.00000000
Total	495000	1.00000	2,614,055,400	1.00000000

TABLE 15.7 Comparison of Independent Model with Observed Data for Four Versions, By Case

No. errors activated	Independent model	Observed frequency
0	0.63878	0.65052
1	0.30296	0.30889
2	0.05388	0.03303
3	0.00426	0.00747
4	0.00013	0.00008

the occurrences of such a split are not available. Thus the observed failure frequency in Table 15.8 is a lower bound (it is the sum of the observed cases of three or four coincident failures). If the data on a 2–2 split were available, then the probability of a 2–2 split would be added to the observed frequency values listed in Table 8. For the by-frame data, for example, if 5 percent of the two-coincident failures produced similar wrong results, then the model and the observed data would agree quite well.

TABLE 15.8 Summary of Parameter Values Derived from Lyu-He Data

2-version model	3-version model	4-version model
	By-Case Data	
$P_V = 0.095$ $P_{RV} = 0.0167$	$P_V = 0.0958$ $P_{RV} = 0$ $P_{RALL} = 0.0003$	$P_V = 0.106$ $P_{RV} = 0$ $P_{RALL} = 0$
Predicted failure probability (from the model)		
0.0265	0.0262	0.0044
Observed failure probability (from the data)		
0.0256	0.0214	0.0076
	By-Frame Data	
$P_V = 0.000026$ $P_{RV} = 1.3 \times 10^{-7}$	$P_V = 0.000027$ $P_{RV} = 1.57 \times 10^{-7}$ $P_{RALL} = 0$	$P_V = 0.000026$ $P_{RV} = 1.3 \times 10^{-7}$ $P_{RALL} = 0$
Predicted failure probability (from the model)		
1.31×10^{-7}	4.73×10^{-7}	7.8×10^{-7}
Observed failure probability (from the data)		
1.32×10^{-7}	4.73×10^{-7}	0

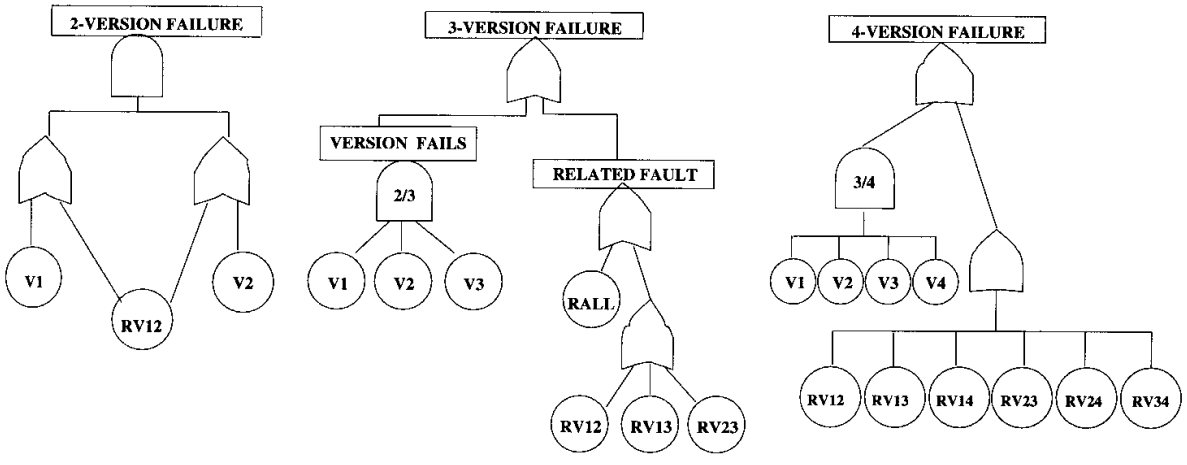


Figure 15.14 Fault tree models for two-, three-, and four-version systems.

These parameters are derived from a single experimental implementation and so may not be generally applicable. Similar analysis of other experimental data will help to establish a set of reasonable parameters that can be used in models that are developed during the design phase of a fault-tolerant system.

The first observation that we can make from Table 15.8 is that there is good agreement between the model and the experimental data. Second, we note that for the by-case data the individual versions were not highly reliable (failure probabilities on the order of 0.1) and yet the fault-tolerant software systems achieved respectable levels of reliability.

The difference in reliability between the by-case and by-frame data is striking. The increased reliability associated with the by-frame data suggests a strong advantage associated with frequent error-checking, since forward-error recovery may be possible. A more complete analysis would also reveal any disadvantages, and would allow a useful trade-off analysis. For example, more frequent comparisons could introduce more overhead for synchronization and message passing, which might jeopardize a hard deadline.

A more complete analysis of the fault-tolerant systems for a single task computation includes the effects of both hardware faults and decider failures and is shown in Table 15.9. For the hardware faults, we assume a fairly typical failure rate of 10^{-4} per hour. In the by-case scenario, a typical test case contained 5280 time frames, each time frame being 50 ms, so a typical computation executed for 264 seconds. Assuming that hardware faults occur at a rate ($10^{-4}/3600$) *per second,* we see that the probability that a hardware fault occurs during a typical test case is

$$1 - e^{-10^{-4}/3600 \times 264 \text{ seconds}} = 7.333 \times 10^{-6} \tag{15.16}$$

We conservatively assume that a hardware fault that occurs anywhere during the execution of a task disrupts the entire computation running on the host. For the by-frame data, the probability that a hardware fault occurs during a time frame is

$$1 - e^{-10^{-4}/3600 \times 0.05 \text{ seconds}} = 1.4 \times 10^{-9} \tag{15.17}$$

If we further assume that the lifetime of a hardware fault is 1 second, then it can affect as many as 20 time frames. We thus take the probability of a hardware fault to be 20 times the value calculated in Eq. (15.17), or 2.8×10^{-8}.

Since no decider failures were observed during the experimental implementation, it is difficult to estimate this probability. The decider used for the recovery block system is an acceptance test, and for this

application is likely to be significantly more complex than the comparator used for the NVP and NSCP systems. For the sake of comparison, for the by-case data we will assume that the comparator used in the NVP and NSCP systems has a failure probability of only 0.0001 and that the acceptance test used for the DRB system has a failure probability of 0.001. For the by-frame data, the decider is considered to be extremely reliable, with a failure probability of 10^{-7} for all three systems. If the decider were any less reliable, then its failure probability would dominate the system analysis, and the results would be far less interesting.

15.7 System-Level Analysis of Hardware and Software System

Computer systems that are used for critical applications are designed to tolerate both hardware and software faults by executing multiple software versions on redundant hardware and by actively reconfiguring the system in response to a permanent failure of a hardware component. In the previous section, the impact of hardware failures was limited to the analysis of a single task; a more complete analysis considers the dynamic reconfiguration of the system configuration in response to hard permanent faults.

Sophisticated techniques exist for the separate analysis of fault-tolerant hardware [Geis90, John88] and software [Lapr84, Scot87,

TABLE 15.9 Comparison of Base Case with More General Case

RB model	NVP model	NSCP model
	By-case data	
Probability of decider failure used for system analysis		
0.001	0.0001	0.0001
Predicted failure probability (perfect decider, no HW faults)		
0.0256	0.0261	0.0403
Predicted failure probability (imperfect decider, HW faults)		
0.0266	0.0262	0.0404
	By-frame data	
Probability of decider failure used for system analysis		
1×10^{-7}	1×10^{-7}	1×10^{-7}
Predicted failure probability (perfect decider, no HW faults)		
1×10^{-6}	2.07×10^{-6}	1.23×10^{-5}
Predicted failure probability (imperfect decider, HW faults)		
1.1×10^{-6}	2.17×10^{-6}	1.24×10^{-5}

Shin84], and a few authors have considered their combined analysis [Lapr84, Star87, Lapr92b]. We will combine the fault tree analysis of a single repetitive task with a Markov model representing the evolution of the hardware configuration as permanent faults occur. The fault tree model captures the effects of software bugs and transient hardware faults which can affect a single task computation, while the Markov model describes how the system on which the software is running can change with time.

A reliability model of an integrated fault-tolerant system must include at least three different factors: computation errors, system structure, and coverage modeling. The fault tree models for the fault-tolerant software systems which we have already considered will describe the computation error process. In these fault tree models, we have deliberately remained vague as to the hardware faults being considered. Here, let us be more precise. In the computation error model, we consider only transient hardware faults that affect the computation but cause no permanent hardware damage. A transient hardware fault is assumed to upset the software running on the processor and produce an erroneous result that is indistinguishable from an input-activated software error. Permanent hardware faults, which require automatic system reconfiguration, are included in the Markov model of system structure.

The longer-term system behavior is affected by permanent faults and component repair, which require system reconfiguration to a different mode of operation. The system structure is modeled by a Markov chain, where the Markov states and transitions model the long-term behavior of the system as hardware and software components are reconfigured in and out of the system. Each state in the Markov chain represents a particular configuration of hardware and software components and thus a different level of redundancy.

The short-term behavior of the computation process and the long-term behavior of the system structure are combined as follows. For each state in the Markov chain, there is a different combination of hardware transients and software faults that can cause a computation error. The fault tree model solution produces, for each state i in the Markov model, the probability q_i that an output error occurs during a single task computation while the state is in state i. The Markov model solution produces $P_i(t)$, the probability that the system is in state i at time t. These two measures are combined to produce $Q(t)$, the probability that an unacceptable result is produced at time t:

$$Q(t) = \sum_{i=1}^{n} q_i P_i(t)$$

The models of the three systems being analyzed (DRB, NVP, and NSCP; see Fig. 15.9) will consist of two fault trees and one Markov

model. Since each of the systems can tolerate one permanent hardware fault, there are two operational states in the Markov chain. The initial state in each of the Markov chains represents the full operational structure, and an intermediate state represents the system structure after successful automatic reconfiguration to handle a single permanent hardware fault. (For the sake of simplifying the comparisons, we assume that the systems are not repairable. Repair can easily be considered in the Markov model of the system structure.) There is a single failure state which is reached when the second permanent hardware fault is activated or when a coverage failure occurs.

A coverage failure occurs when the system is unable to detect and recover from the activation of a permanent hardware fault. The probability that the system can correctly detect, isolate, and reconfigure in response to a permanent hardware fault is the parameter c in the Markov models. If the fault is not covered, then a coverage failure is said to occur, which leads to immediate system failure. The coverage parameter (c) can be determined from a coverage model that considers such effects as physical fault behavior, error and fault detection, and recovery and reconfiguration mechanisms. Coverage modeling is described in more detail in [Duga89a, Duga93a].

The safety models for the three systems are similar to the reliability models in that they consist of a Markov model and two associated fault trees. The major difference between a reliability and safety analysis is in the definition of failure. In the reliability models, any unacceptable result (whether or not it is detected) is considered a failure. In a safety model, a detected error is assumed to be handled by the system in a fail-safe manner, so an unsafe result occurs only if an undetected error is produced.

In the Markov part of the safety models, two failure states are defined. The fail-safe state is reached when the second covered permanent hardware fault is activated. The fail-unsafe state is reached when any uncovered hardware fault occurs. The system is considered safe when in the absorbing fail-safe state. This illustrates a key difference between a reliability analysis and a safety analysis. A system which is shut down safely (and thus is not operational) is inherently safe, although it is certainly not reliable.

15.7.1 System reliability and safety model for DRB

The Markov model for the long-term behavior of the DRB system is shown in Fig. 15.15. The Markov model details the initial state configuration, where the recovery block structure is executed on redundant hardware, and the reconfigured state, after the activation of a permanent hardware fault. There are two processors available in the initial

state. On one processor, the primary is executed first and the secondary remains idle until needed, while the other processor executes the secondary software module first. The idle software component is shaded.

In the initial configuration of the DRB system, there are two active processors, which can fail independently at rate λ. If the system can properly respond to the failure of one of the processors (with probability c), then the system is reconfigured to a single processor; thus the rate $2\lambda c$ from the initial state to the intermediate state and the rate $2\lambda(1-c)$ for an uncovered failure. When a single processor remains, the system survives until that processor fails.

The fault tree model for the computation process associated with the initial state was analyzed previously (see Fig. 15.10). The fault tree model for the computation process in the reconfiguration state is derived from that for the initial state and is shown in Fig. 15.16. In the reconfiguration state, a single recovery block structure executes on the single remaining processor.

The safety model of DRB, shown in Fig. 15.17, shows that an acceptance test failure is the only software cause of an unsafe result. As long as the acceptance test does not accept an incorrect result, then a safe output is assumed to be produced. The hardware redundancy does not increase the safety of the system, as the system is vulnerable to the acceptance test in both states. The hardware redundancy can actually decrease the safety of the system, since the system is perfectly safe when in the fail-safe state, and the hardware redundancy delays absorption into this state.

15.7.2 System reliability and safety model for NVP

The Markov model for the long-term behavior of the NVP system is shown in Fig. 15.18. In the initial state there are three active processors,

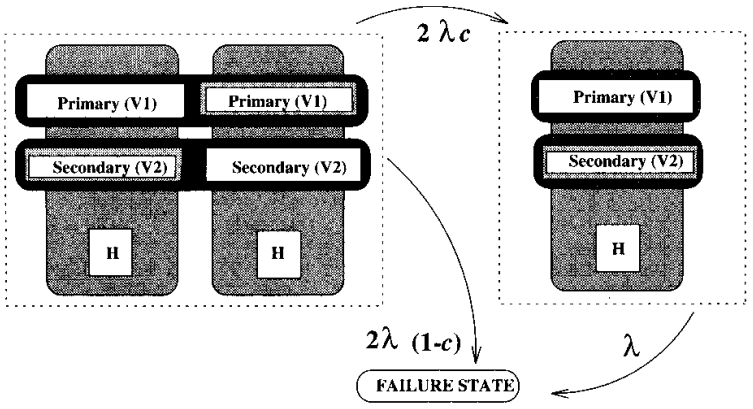


Figure 15.15 Markov reliability model of DRB.

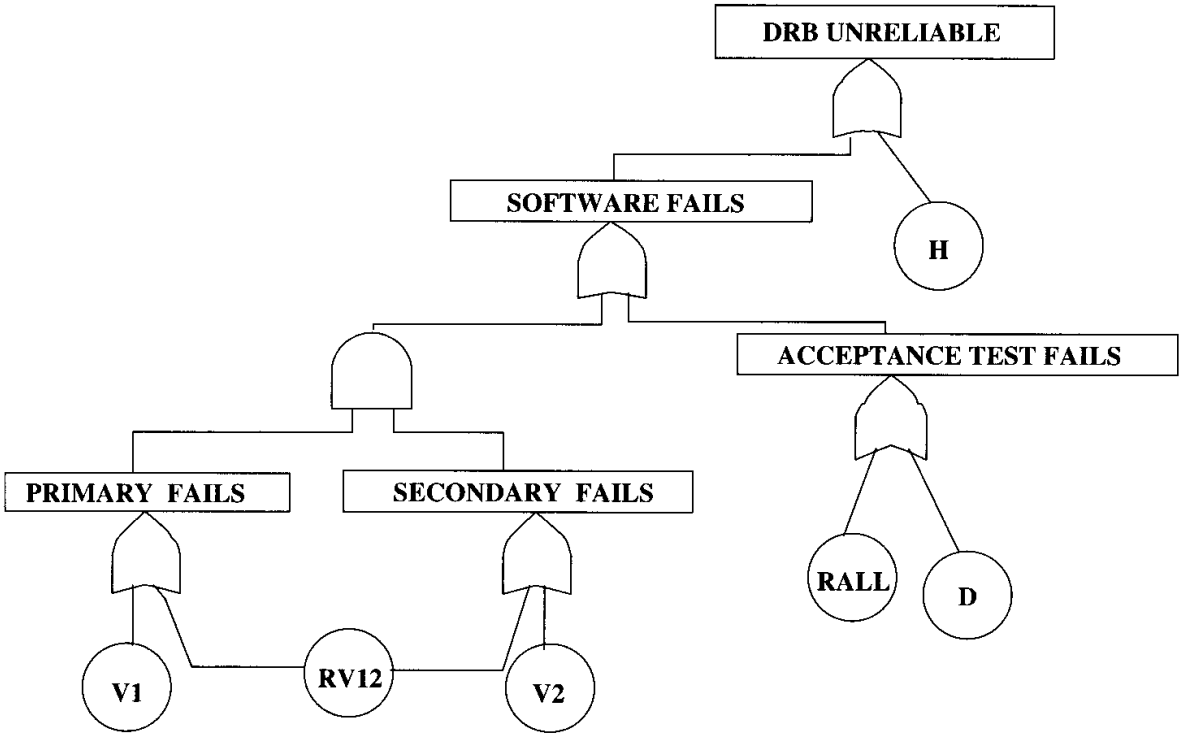


Figure 15.16 Fault tree model of computation process in DRB reconfiguration state.

so the transition rate to the reconfiguration state is $3\lambda c$ and the transition rate to the failure state caused by an uncovered failure is $3\lambda(1-c)$. We assume that the system is reconfigured to simplex mode after the first permanent hardware fault. (See [Doyl95] for a discussion of the TMR-simplex reconfiguration scheme). In the reconfigured state, an unreliable result is caused by either a hardware-transient or a software-

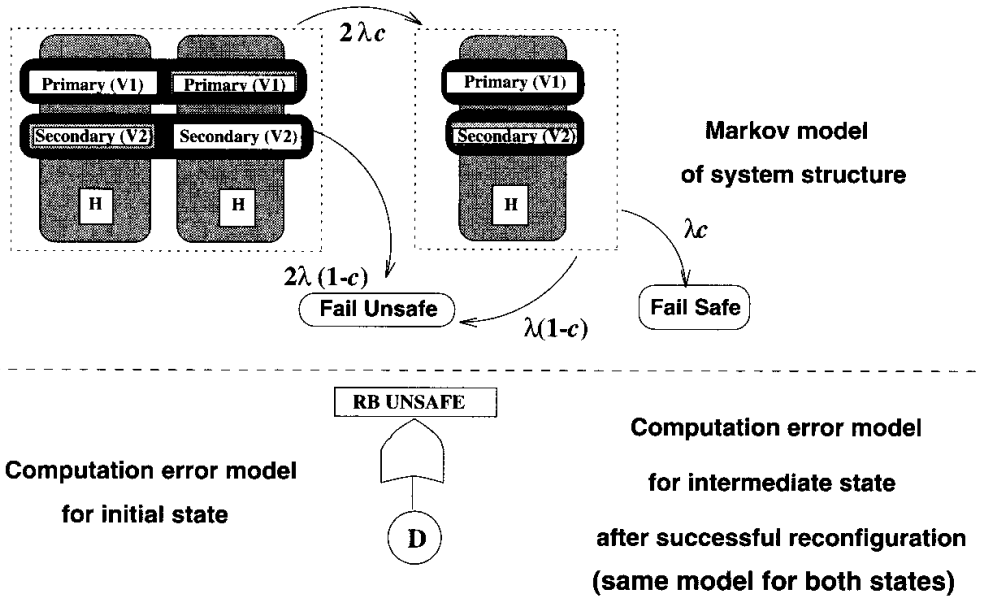


Figure 15.17 Safety model of DRB.

fault activation, as shown in the fault tree of Fig. 15.19. The system fails when the single remaining processor fails; thus the transition rate λ from the reconfiguration state to the failure state.

The NVP safety model (Fig. 15.20) shows that the safety of the NVP system is vulnerable to related faults as well as decider faults. In the Markov model, we assume that the reconfigured state uses two versions (rather than one, as was assumed for the reliability model) so as to increase the opportunity for comparisons between alternatives and thus increase error detectability.

15.7.3 System reliability and safety model for NSCP

The Markov model for the long-term behavior of the NSCP system is shown in Fig. 15.21, while the fault tree model for the reconfiguration state is shown in Fig. 15.22.

The NSCP safety model (Fig. 15.23) shows the same vulnerability of the NSCP system to related faults. When the system is fully operational, all two-way related faults will be detected by the self-checking arrangements, leaving the system vulnerable only to a decider fault, and a fault affecting all versions similarly. After reconfiguration, a related fault affecting both remaining versions could also produce an undetected error.

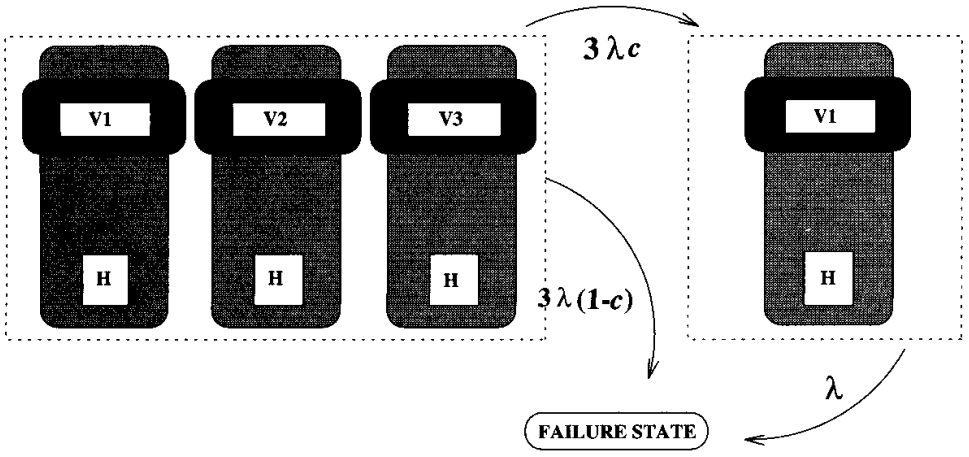


Figure 15.18 Markov reliability model of NVP.

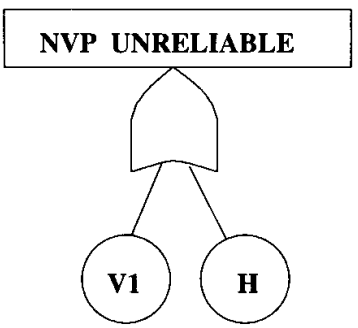


Figure 15.19 Fault tree model of computation process in NVP reconfiguration state.

15.7.4 A case study in system-level analysis

This section contains a quantitative analysis of the system-level reliability and safety models for the DRB, NVP, and NSCP systems. The software parameter values used in this study are those derived earlier from the Lyu-He data. Typical permanent failure rates for processors range in the 10^{-5} *per hour* range, with transients perhaps an order of magnitude larger. Thus we will use $\lambda_p = 10^{-5}$ per hour for the Markov model. The fault and error recovery process is captured in the coverage parameters used in the Markov chain [Duga89a]. We assume a commonly used value for the coverage parameter in the Markov model, $c = 0.999$.

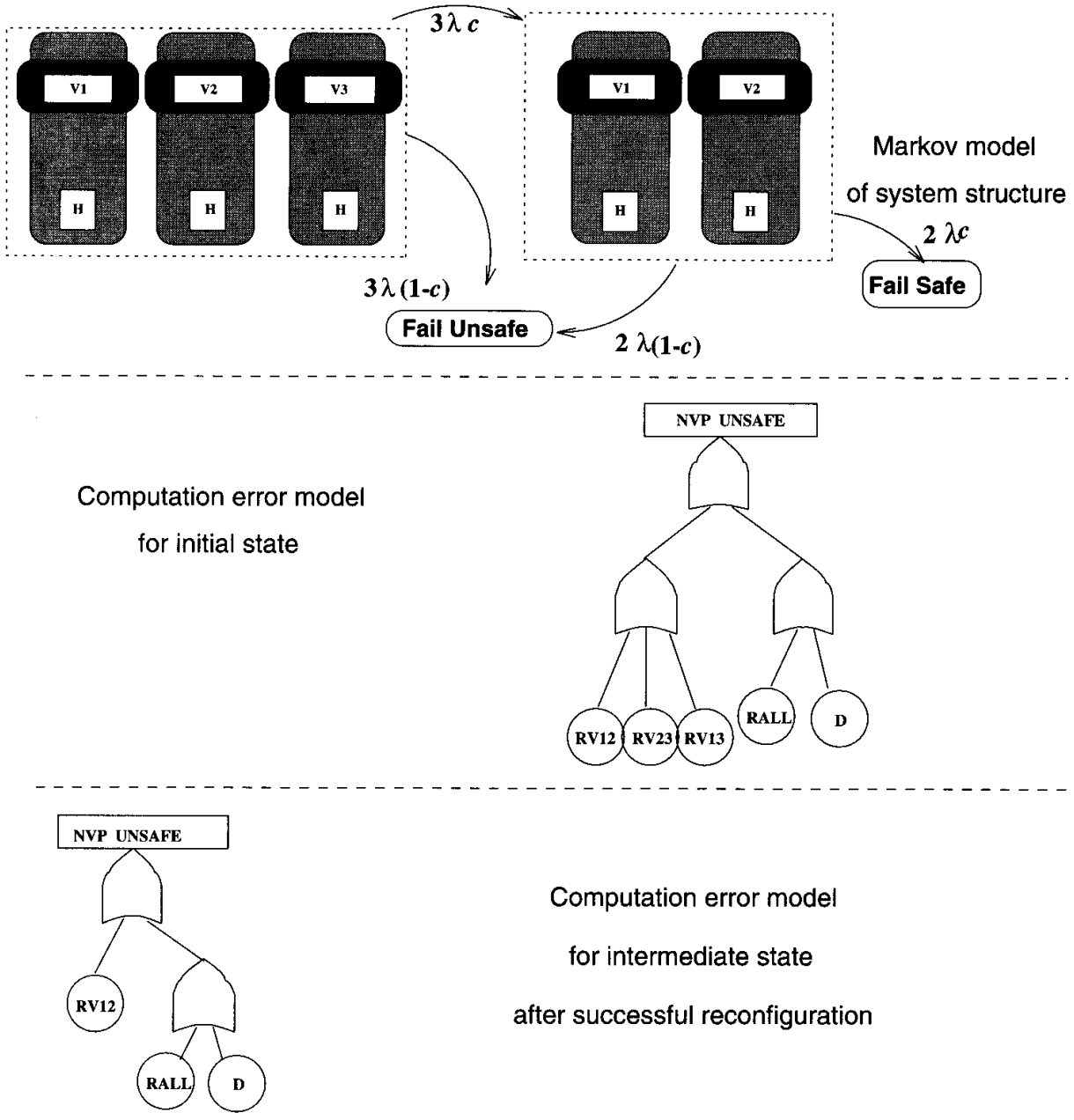


Figure 15.20 Safety model of NVP.

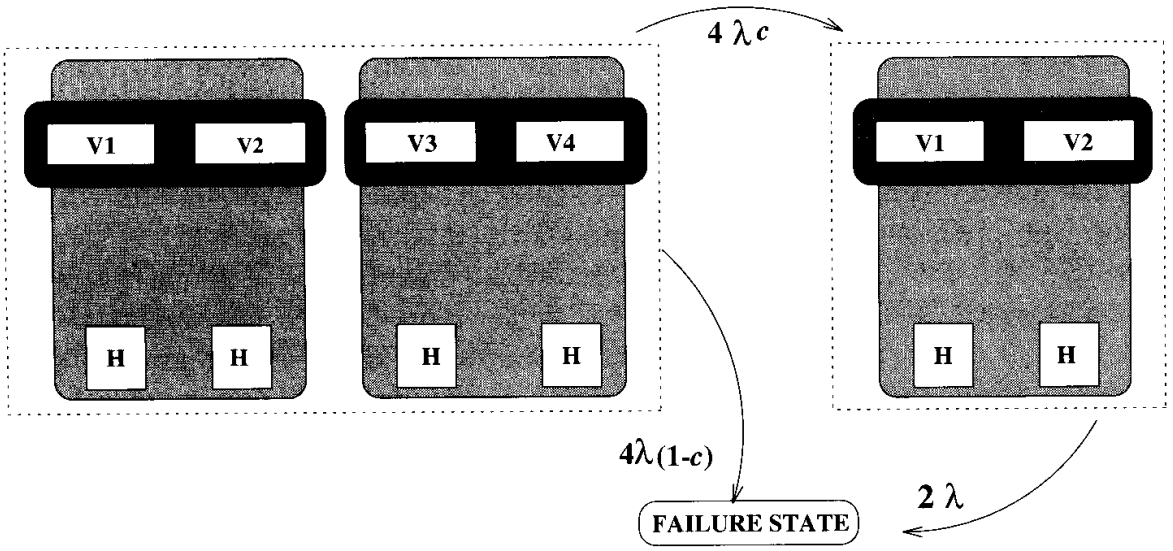


Figure 15.21 Markov reliability model of NSCP.

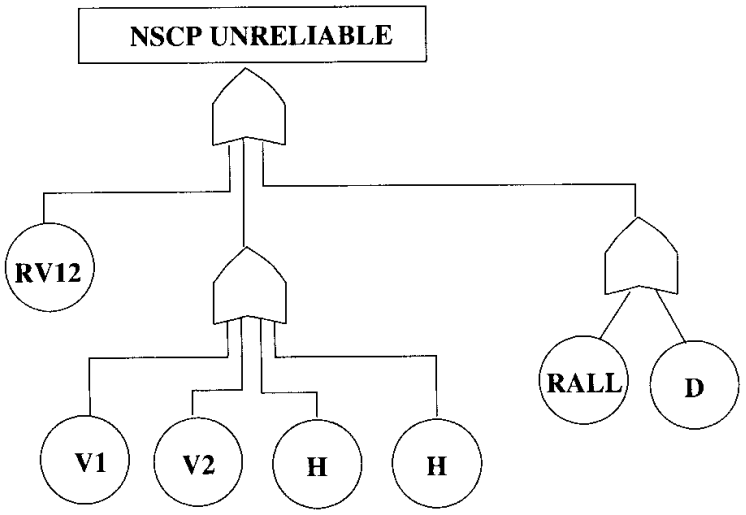


Figure 15.22 Fault tree model of computation process in NSCP reconfiguration state.

Figure 15.24 compares the predicted reliability of the three systems. Under both the by-case and by-frame scenarios, the recovery block system is most able to produce a correct result, followed by NVP. NSCP is the least reliable of the three. Of course, these comparisons are dependent on the experimental data used and assumptions made. More experimental data and analysis are needed to enable a more conclusive comparison.

Figure 15.25 gives a closer look at the comparisons between the NVP and DRB systems during the first 200 hours. The by-case data show a crossover point where NVP is initially more reliable but is later less reliable than DRB. Using the by-frame data, there is no crossover point, but the estimates are so small that the differences may not be statistically significant.

Figure 15.26 compares the predicted safety of the three systems. Under the by-case scenario, NSCP is the most likely to produce a safe result, and DRB is an order of magnitude less safe than NVP or NSCP. This difference is caused by the difference in assumed failure probability associated with the decider. Interestingly, the opposite ordering results from the by-frame data. Using the by-frame data to parameterize the models, DRB is predicted to be the safest, while NSCP is the least safe. The reversal of ordering between the by-case and by-frame parameterizations is caused by the relationship between the probabilities of related failure and decider failure. The by-case data parameter values resulted in related fault probabilities that were generally lower than the decider failure probabilities, while the by-frame data resulted in related fault probabilities that were relatively high. In the safety models, since there were fewer events that lead to an unsafe result, this relationship between related faults and decider faults becomes significant.

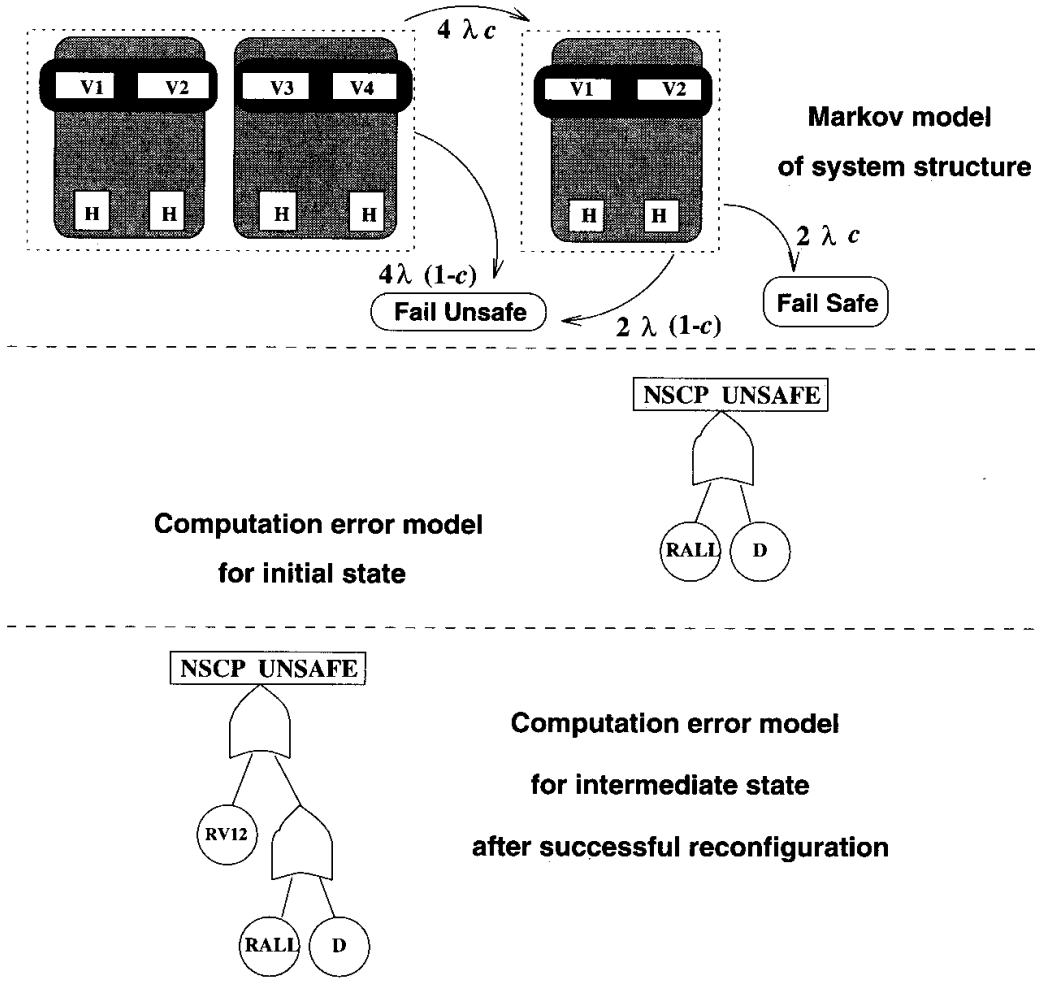


Figure 15.23 Safety model of NSCP.

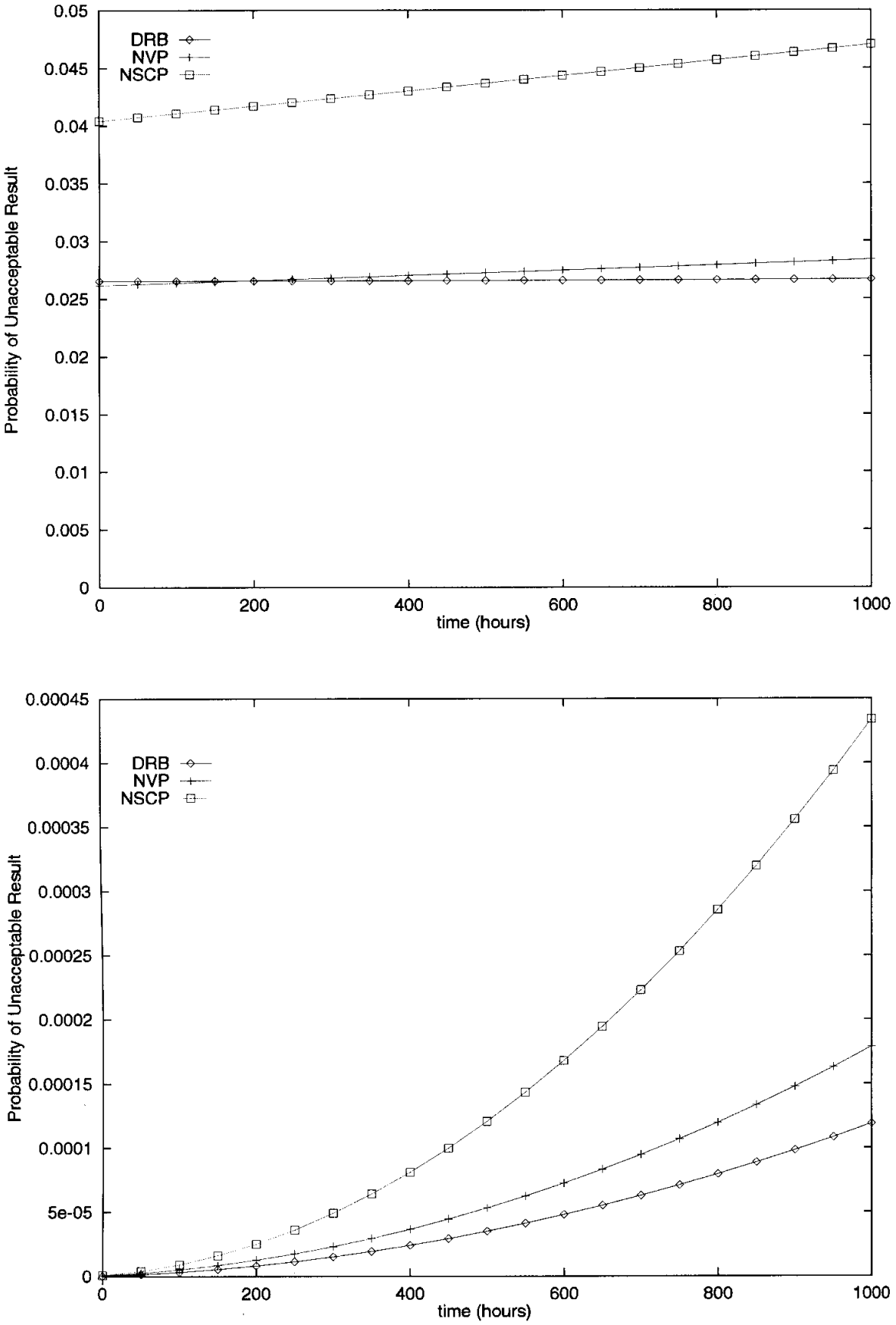
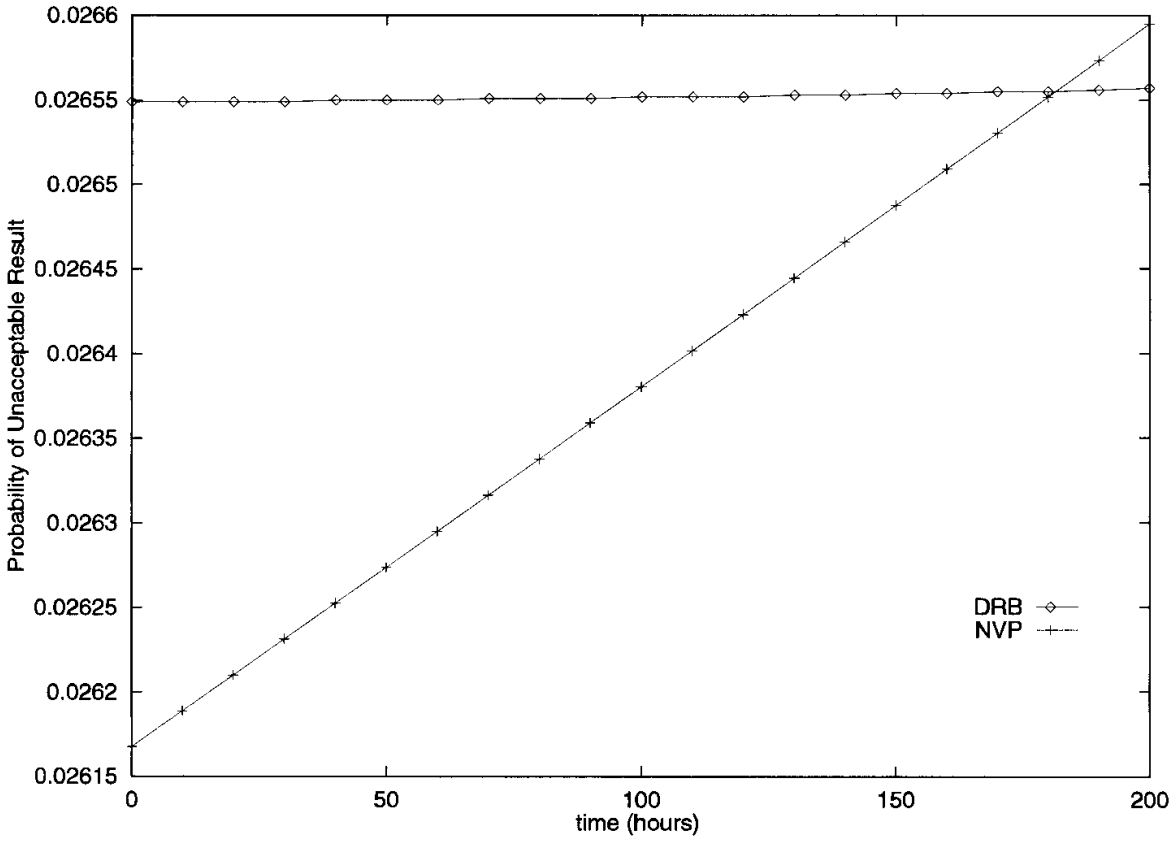


Figure 15.24 Predicted reliability, by-case data (top) and by-frame data (bottom).



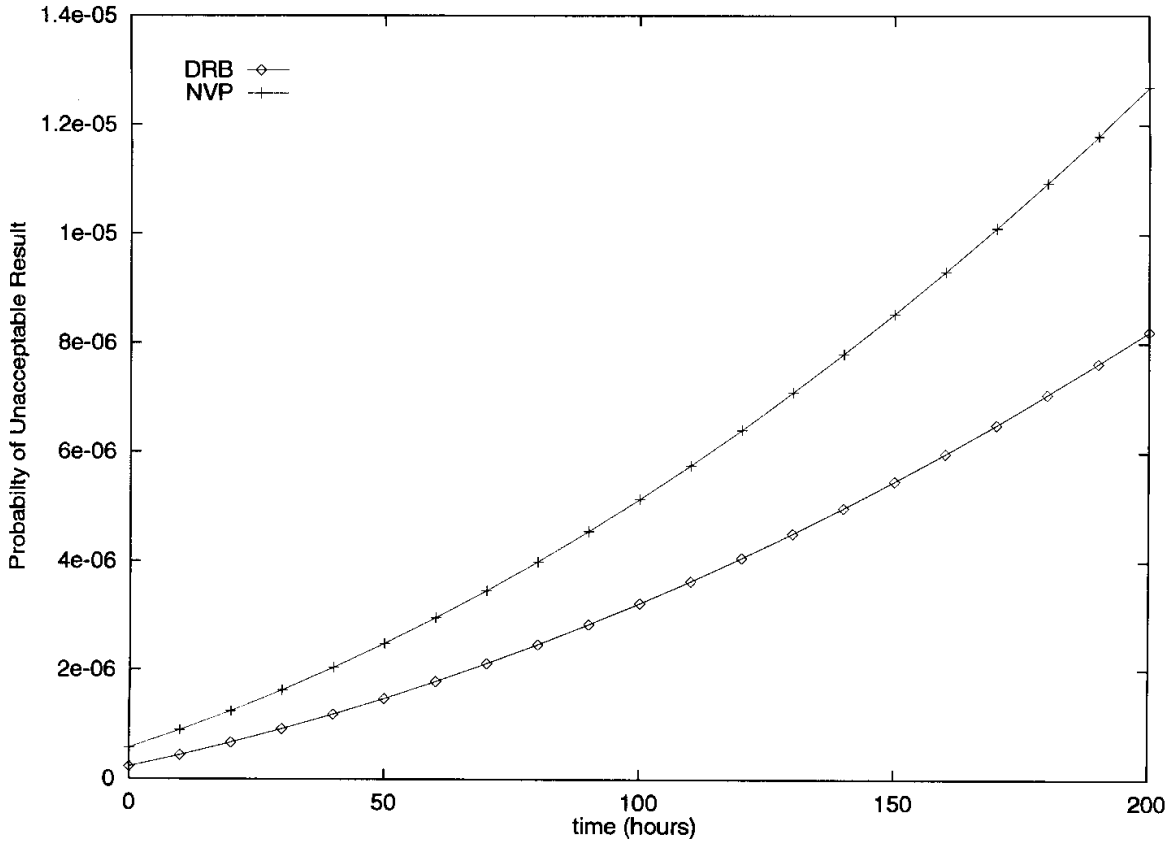


Figure 15.25 Predicted reliability, by-case data (top) and by-frame data (bottom).

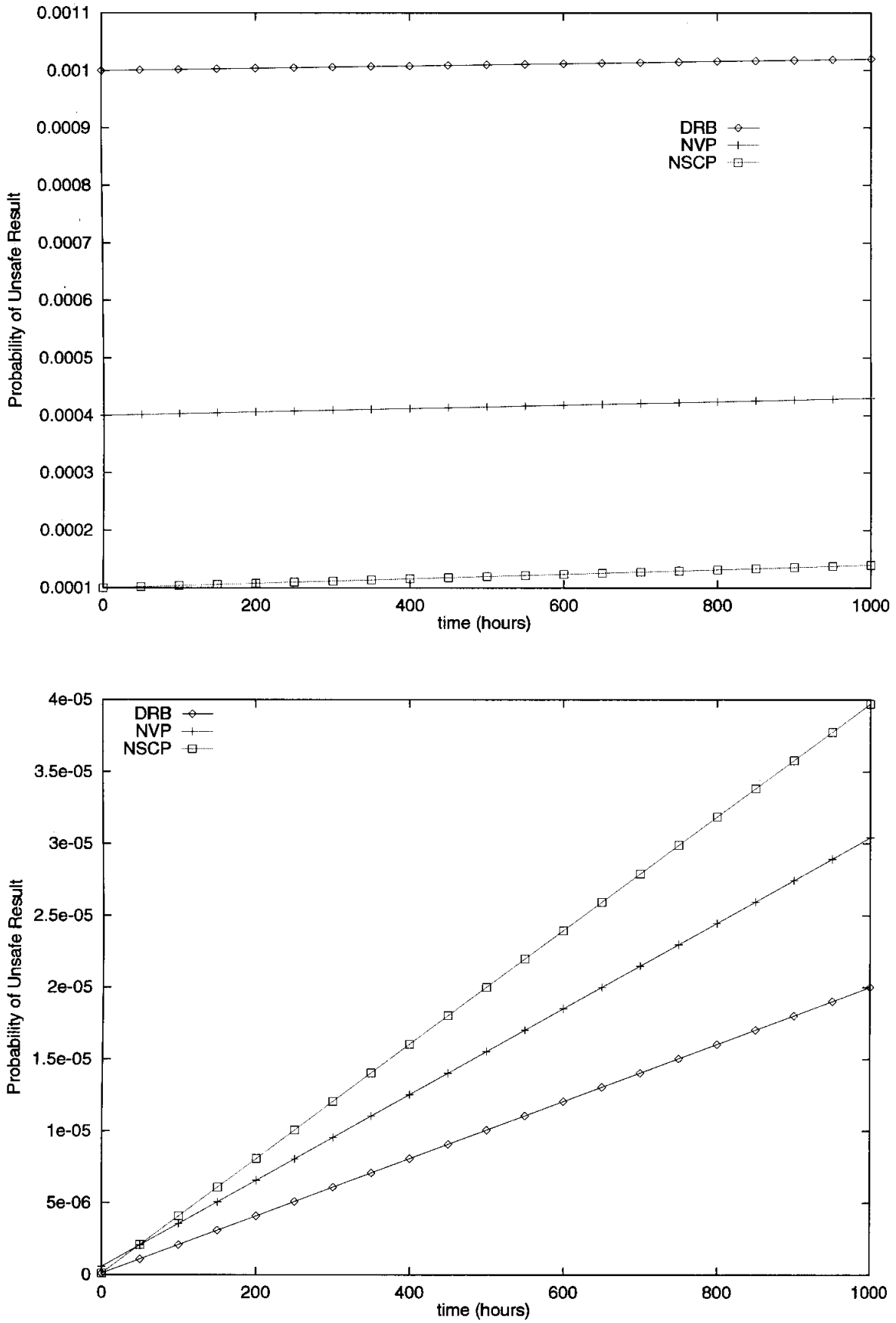


Figure 15.26 Predicted safety, by-case data (top) and by-frame data (bottom).

A more complete comparative analysis of these three systems, including a sensitivity analysis and an assessment of the impact of the decider failure probability, appear in [Duga95]. A major disadvantage associated with fault trees is the exponential solution time, but fault trees share this disadvantage with every other comparable modeling technique. Approaches to this problem include the development of good approximate solution techniques [Duga89b] and the recent use of binary decision diagrams (BDD) for quantitative analysis [Coud93, Rauz93]. A second disadvantage associated with fault tree modeling is the inability to model sequence-dependent failures. Fault trees are a *combinatorial* model (as are reliability block diagrams) that represent combinations of events which lead to system failure. As such, combinatorial models cannot capture information concerning the *order* in which failures have occurred. As an approach to this problem, a dynamic fault tree has been defined, which used a Markov chain for solution [Duga92].

15.8 Summary

Fault tree models, which have traditionally been used for the analysis of hardware systems, are well suited to the analysis of software. Fault trees can serve as a design aid to help determine the effective use of on-line and off-line testing. Software safety validation is aided by the use of software fault trees, where the code is analyzed on a statement-by-statement basis. At the systems level, where a software program or a processor are each considered as basic components, fault trees combine well with Markov models to predict overall system reliability and safety. Such system models may be parameterized using experimental data if field experience is insufficient.

The advantages associated with the use of fault trees are the graphical and mathematical foundations, which give rise to good qualitative and quantitative solution methods. Since fault trees are applicable to many different systems, they can provide a common framework for comparative analysis.

Problems

15.1 What are the minimal cutsets for the fault tree in Fig. 15.1? What is the probability of occurrence of the top event in the tree, given the probabilities of occurrence for the basic events: P_{valve}, P_{timeout}, and P_{full}?

15.2 For the fault tree shown in Fig. 15.2, suppose that we know that event $A4$ has already occurred. Given this information, what is now the set of minimal cutsets, and what is the probability of occurrence for the top event in the tree? (Define P_{Ai} to be the probability of occurrence for the basic event A_i).

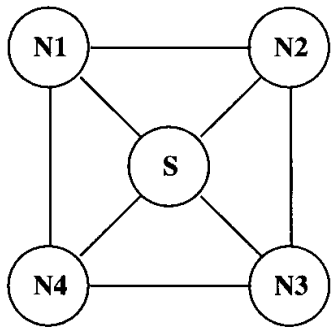


Figure 15.27

15.3 For Probs. 15.3 to 15.5, consider the system shown in Fig. 15.27. The circles in the figure represent processing nodes and the lines represent bidirectional links between the nodes. The nodes labeled $N1$, $N2$, $N3$, and $N4$ are active nodes, while node S is a spare node. Active and spare nodes have the same probability of failure p_N, while links fail with probability p_l.

Assume that links do not fail, and that the system is operational as long as four operational nodes can communicate. A failed node disables the attached nodes. Draw the fault tree model for the system and list the minimal cutsets.

15.4 For the fault tree derived in Prob. 15.3, suppose that the probability of node failure is $p_N = 0.05$. What is the probability that four nodes are connected?

15.5 Assume that both the nodes and links can fail. When a node fails, it can still relay messages on unfailed links. So the system fails when two of the five nodes either fail or are disconnected from the rest of the network. Draw a fault tree model for the system.

15.6 Figure 15.8 showed part of a fault tree used for safety validation. Normally, we would continue expanding each of the cases. Suppose instead that we try to detect a potential hazard on the fly, and take some corrective or preventive action. Suppose during run time that we could detect whether $y > 0$ and whether $z > 0$ at any point in the program. How would that help?

15.7 Consider the consensus recovery block (CRB) [Scot87] described in Chap. 14. Determine a fault tree model to analyze the reliability of CRB (similar to the fault tree models derived in Sec. 15.5).

15.8 Develop a fault tree model for reliability and safety analysis of a two-version NVP system.

15.9 Develop a fault tree model for reliability and safety analysis of a three-version DRB system.

15.10 Develop a fault tree model for reliability and safety analysis of a four-version NVP system.

15.11 Consider an experimental implementation of a multiversion programming system that resulted in 27 versions [Knig86]. Testing with more than a million test cases revealed the failure behavior shown in Table 15.10. If a 2-version system that used comparison matching were formed by randomly selecting 2 of the 27 versions, what would be the expected reliability of the software

TABLE 15.10

Number of failed versions	Observed frequency
0	0.983539
1	0.15206e-1
2	0.551e-3
3	0.343e-3
4	0.242e-3
5	0.73e-4
6	0.32e-4
7	0.12e-4
8	0.2e-5
More than 8	0

system? (Assume that the system fails if it is unable to produce a correct result. Thus a mismatch results in a failure.)

15.12 For the same system described in Prob. 15.7, what would be the expected safety of a similarly constructed two-version system? (The system fails when an incorrect result is delivered.)