ANALYZING SAFETY AND FAULT TOLERANCE USING TIME PETRI NETS[†]

N. G. Leveson and J. L. Stolzy
Information and Computer Science
University of California, Irvine
Irvine, California 92717

Abstract: The application of Time Petri net modelling and analysis techniques to safety-critical real-time systems is explored and procedures described which allow analysis of safety, recoverability, and fault-tolerance. These procedures can be used to help determine software requirements, to guide the use of fault detection and recovery procedures, to determine conditions which require immediate mitigating action to prevent accidents, etc. Thus it is possible to establish important properties during the synthesis of the system and software design instead of using guesswork and costly *a posteriori* analysis.

Introduction

Computers are increasingly being used as passive (monitoring) and active (controlling) components of real-time systems, e.g. air traffic control, aerospace, aircraft, industrial plants, and hospital patient monitoring systems. The problems of safety become important when these applications include systems where the consequences of failure are serious and may involve grave danger to human life and property.

Although in a batch system it is reasonable to abort execution and attempt to fix the problem when a failure occurs, control usually cannot be abandoned abruptly in an embedded system. Therefore, responses to hardware failures, software faults, human error, and undesired and perhaps unexpected environmental conditions must be built into the system. These responses can take three basic forms:

1) a *fault-tolerant* system continues to provide full performance and functional capabilities in the presence of operational faults.
2) a *fail-soft* system continues operation but provides only degraded performance or reduced functional capabilities until the fault is removed.
3) a *fail-safe* system attempts to limit the amount of damage caused by a failure. No attempt is made to satisfy the functional specifications except where necessary to ensure safety.

These responses are, for most situations, in the order of decreasing desirability although when the functional and safety requirements of the system are not identical (and especially when they are conflicting), they are not necessarily of decreasing importance.

The area of system safety is well-established, and procedures exist to identify and analyze electromechanical hazards along with techniques to eliminate or limit hazards in the final product (for a summary, see Malasky (1982)). Unfortunately, much more is known about how to engineer safe mechanical systems than safe software systems. With the increased use of software in safety-critical components of complex systems, government certification agencies and contractors are increasingly including requirements for software hazard analysis and verification of software safety (e.g. see MIL-STD-882b: System

[†]This work was partially supported by a MICRO grant co-funded by the State of California and Hughes Aircraft Co.

Safety Program Requirements). Modelling and analysis tools are desperately needed to aid in these tasks. This paper explores the application of Petri net modelling and analysis techniques to the design of safety-critical real-time systems. Because timing is crucial with respect to the control of real-time systems, Time Petri nets are used.

The next section describes the general approach to be taken. Following that, procedures are outlined for eliminating hazards from the system design. Then potential failures are added to the analysis procedures.

Safety Analysis

Whereas *system reliability* deals with the problems of ensuring that a system, including all hardware and software subsystems, performs a required task or mission for a specified time in a specified environment, *system safety* is concerned only with ensuring that a mishap does not occur in the process. Usually there are many possible system failures which have relatively little "cost" associated with them. Others have such drastic consequences that an attempt must be made to avoid them at all costs, perhaps even at the cost of attaining some or all of the goals of the system.[†] For example, an amusement park ride may have to be temporarily stopped because conditions are such (e.g. a foreign object is on the tracks) that a derailment is possible. Thus the response to a safety critical failure may focus on reduction of risk rather than attainment of mission [Leveson (1984)].

While software itself cannot be unsafe, it can issue commands to a system it controls which place the system in an unsafe state. Furthermore, the controlling software should be able to detect when factors beyond the control of the computer (e.g. environmental conditions) place the system in a hazardous state and to take steps to eliminate the hazard or, if that is not possible, initiate procedures to minimize the hazard. This then is the problem of *software safety.*

If software safety is to be studied and used as a measure of software quality, then some definitions are necessary. A *mishap* is an event or series of events which results in death, injury, illness, or damage to or loss of property or equipment. A *hazard* or *unsafe state* is a condition or state of the system with the potential for (i.e. some non-zero probability of) leading to a mishap. Hazards can be categorized by the aggregate probability of the occurrence of the individual conditions which make up the hazard and by the seriousness of the resulting mishap. Together these constitute *risk.*

The first step in a safety analysis is to identify the system hazards and assess their severity and probability (i.e. risk). The next step is to design the system so as to eliminate hazards or (if that is not possible) to minimize the risk by altering the design so that there is very little probability of the hazard occurring. This can be accomplished by first ensuring that the system as specified is safe, i.e. given that the specifications are correctly implemented and no failures occur, operation of the system will not result in a mishap. The next step in the design process is to identify and eliminate (by using fault tolerance techniques) single point failure modes which can lead to a hazard. Finally, techniques are used to ensure that the probability of multiple sequences of failures leading to a hazard is sufficiently low. If it is impossible to completely eliminate the possibility of a hazard, a design goal may be to minimize the effects of the hazard should it occur. In this case the system should detect the hazard and attempt to eliminate it, if possible; otherwise an attempt should be made to minimize any possible effects. In either case, in order to reduce risk, the *exposure time* (length of time of occurrence) of the hazardous conditions must be minimized. The goal of the techniques presented in this paper is to develop formal procedures to aid in this safety analysis process.

[†]In a system whose sole purpose is the sustaining of life, e.g. a pacemaker, these conflicts between safety and other system requirements do not occur.

It is important to stress the "system" nature of the problem. Software does not harm anyone -- only the instruments which it controls can do damage. Therefore, software safety procedures cannot be developed in a vacuum, but must be considered as part of the overall system safety. For example, a particular software error may cause a mishap only if there is a simultaneous human and/or hardware failure. Alternatively, an environmental event or failure may be involved in the software error. Mishaps are often the result of multiple failure sequences which involve hardware, software, and human failures. One modelling technique which has the potential for analyzing software for real-time systems within a system viewpoint is Time Petri nets. By combining hardware, software, and human components within one model, it is possible to determine, for example, the effects of a failure or fault in one component on another component. It is also possible to use the model to determine software safety and fault tolerance requirements. Writing correct software requirements is a difficult problem for which there are few analytical tools available. Techniques such as Failure Modes and Effects Analysis (FMEA) and Preliminary Hazard Analysis (PHA) [Malasky (1982)] have been developed to determine the system safety requirements. However, there is a need to be able to go from the system safety requirements to the software safety requirements. Using the hazardous states which have been identified in the PHA, it may be possible to work backward to the software interface using Petri net analysis techniques and thus to derive the software safety requirements.

Formal Definitions

A Petri net is composed of a set of *places* P, a set of *transitions* T, an *input* function I, an *output* function O, and an *initial marking* μ_0. The input function I is a mapping from the transition t_i to a bag of places $I(t_i)$ where a bag is a generalization of a set which allows multiple occurrences of an element. Similarly, the output function O maps a transition t_i to a bag of places $O(t_i)$. The initial placement of tokens on the places of the net is specified by μ_0 [Peterson (1981)]. Formally, this is written:

Definition: A *Petri net structure*, Φ, is a five-tuple, $\Phi=(P,T,I,O,\mu_0)$.

$P=\{p_1,p_2,\ldots,p_n\}$ is a finite set of *places*, $n\geq 0$.

$T=\{t_1,t_2,\ldots,t_m\}$ is a finite set of *transitions*, $m\geq 0$. The set of places and the set of transitions are disjoint, $P\cap T=\emptyset$.

$I:T \rightarrow P^{\infty}$ is the *input* function, a mapping from transitions to bags of places.

$O: T\rightarrow P^{\infty}$ is the *output* function, a mapping from transitions to bags of places.

Finally, $\mu_0:P \rightarrow N$ is the *initial marking* for the net where N is the set of non-negative integers.

Definition: The *multiplicity* of an input place p_i for a transition t_j is the number of occurrences of the place in the input bag of the transition, denoted $\#(p_i,I(t_j))$. The multiplicity of an output place is defined similarly and denoted $\#(p_i,O(t_j))$.

A graph structure is often used for illustration of Petri nets where a *circle* " ○ " represents a place and a *bar* " | " represents a transition. Figure 1 shows a petri net. An arrow from a place to a transition defines the place to be an input to the transition. Similarly, an output place is indicated by an arrow from the transition to the place.

The dynamic aspects of Petri net models are denoted by markings which are assignments of *tokens* to the places of a Petri net. Markings may change during *execution* of a Petri net.

Definition: A *marking* μ of a Petri net Φ is a function from the set of places P to the nonnegative integers N, μ: $P\rightarrow N$.

The execution of a Petri net is controlled by the number and distribution of tokens in the Petri net.

Definition: A transition t_j is *enabled* if and only if each of its input places contains at least as many tokens as there exists arcs from that place to the transition, i.e. $\mu(p_i) \geq \#(p_i,I(t_j))$ for all $p_i \in P$.

When a transition fires, all enabling tokens are removed from its input places, and a token is deposited in each of its output places. Transition firings continue as long as there exists at least one enabled transition.

When using Petri nets to model systems, places represent conditions and transitions are used to represent events. Figure 1 can be interpreted as a model of a simple railroad crossing. P_1, P_2, P_3, and P_4 represent the different conditions that can hold for the train (i.e. approaching, just before, within, and past the crossing, respectively). Similarly, transitions 1, 2, and 3 denote the events of signalling the train's approach, entering the crossing, and signalling the train's departure. The large box represents the controlling device or computer -- either hardware or software based. The states of the gate are represented by two places P_{11} (the gate is up) and P_{12} (the gate is down). Transitions 6 and 7 represent the events of raising and lowering the gate respectively.

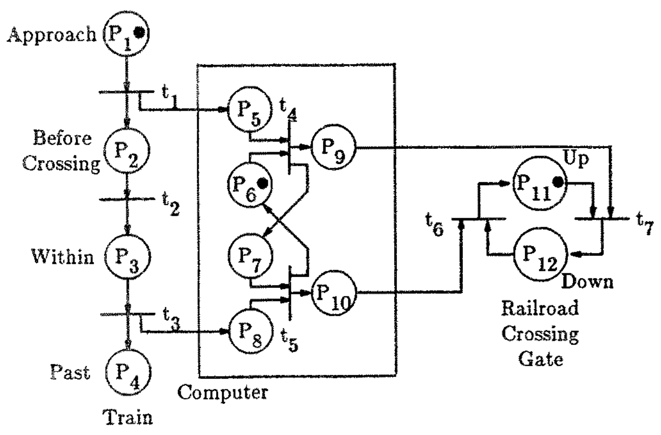


Figure 1. A Petri Net Graph

The state of the Petri net (and hence the state of the modelled system) is defined by the marking (the existing conditions). The change in state caused by firing a transition is defined by the *next-state function* δ.

Definition: The *next-state function* δ: $N^n \times T \rightarrow N^n$ for a Petri net $\Phi=(P,T,I,O,\mu_0)$ with marking μ and transition $t_j \in T$ is defined if and only if t_j is enabled. If $\delta(\mu,t_j)$ is defined, then $\delta(\mu,t_j) = \mu'$ where

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j)) \text{ for all } p_i \in P$$

Definition: For a Petri net $\Phi=(P,T,I,O,\mu_0)$ with marking μ, a marking μ' is *immediately reachable* from μ if there exists a transition $t_j \in T$ such that $\delta(\mu,t_j)=\mu'$.

The "reachability" relationship is the reflexive transitive closure of the "immediately reachable" relationship.

Definition: The *reachability set* $R(\Phi,\mu)$ for a Petri net $\Phi=(P,T,I,O,\mu_0)$ with marking μ is the smallest set of markings defined by:

1. $\mu \in R(\Phi,\mu)$
2. If $\mu' \in R(\Phi,\mu)$ and $\mu'' = \delta(\mu',t_j)$, for some $t_j \in T$, then $\mu'' \in R(\Phi,\mu)$.

Both trees and graphs have been used to represent the reachability state. In this paper, a reachability graph is used where the nodes of the graph are labeled with the present marking (i.e. the state) and the arcs represent transitions between states (see figure 2a).

Definition: A *path* in the reachability graph is a sequence of transitions $t_i,...,t_j$ starting at marking μ_{i-1} to μ_j such that $\delta(\mu_{n-1},t_n) = \mu_n$ for n = i...j

Definition: The *extended next-state function* δ^* is defined for a marking μ, and a sequence of transitions $s \in T^*$ by

$$\delta^*(\mu,t_j s) = \delta^*(\delta(\mu,t_j),s)$$
$$\delta^*(\mu,\lambda)= \mu$$

To model time requires enhancements to the basic Petri net model. There have been several proposals for extending standard Petri nets to include time. We use the Time Petri net approach of Merlin (1974) as it provides a very flexible modelling tool while retaining the instantaneous firing feature of the untimed Petri net.

A Time Petri net (TPN) is a Petri net, i.e. it is composed of a set of places P, a set of transitions T, an input function I, an output function O, and an initial marking μ_0 along with the added *firing time functions* Min and Max. The firing time functions specify the conditions under which a transition may fire. Formally, this is written:

Definition: A *Time Petri net structure*, Φ, is a seven-tuple, $\Phi=(P,T,I,O,Min,Max,\mu_0)$. P, T, I, O, and μ_0 are defined as above.
Min and Max are the *min time function* and *max time function*, respectively, where
Min:T $\rightarrow$ R and Max:T $\rightarrow$ R, R is the set of non-negative real numbers and
$Min_i \leq Max_i$ for all i such that $t_i \in T$.

Definition: A transition is *firable* at time τ if and only if it has been continuously enabled during the interval $\tau - Min(t_j)$ to τ. The firable transition may fire at any time τ for $Min(t_j) \leq \tau \leq Max(t_j)$. A transition must fire at time τ if it has been continuously enabled during the interval $\tau - Max(t_j)$ to τ.

Note that the Time Petri net is equivalent to a standard Petri net if all Min times are 0 and all Max times are set to ∞. Also note that the markings of the states of the Time Petri net reachability graph will be equal to or a subset of the markings of the equivalent untimed Petri net. This is true since the enabling rules for the time Petri net are the same as for a Petri net. The difference lies in the additional restrictions placed on the firing rules. Thus adding timing may restrict the set of possible markings, but will never increase it. Since we are basically interested in determining worst cases (including the potential effects of timing failures), much of our analysis will involve deriving the untimed reachability graph and then determining 1) the timing constraints of the final system necessary to avoid high-risk states, and 2) the run-time checks, e.g. watchdog timers, needed to detect critical timing failures.

Eliminating High-Risk States from the Design

A mishap is an unplanned event or series of events that results in death, injury, illness, or damage to or loss of property or equipment. Mishaps can be classified as to severity from catastrophic to negligible.

Definition: A *hazard* is a set of conditions within a state from which there is a path to a mishap. A state σ is hazardous if and only if there exists a mishap state σ_m and a sequence of transitions $s \in T^*$ such that $\delta^*(\sigma,s)=\sigma_m$.

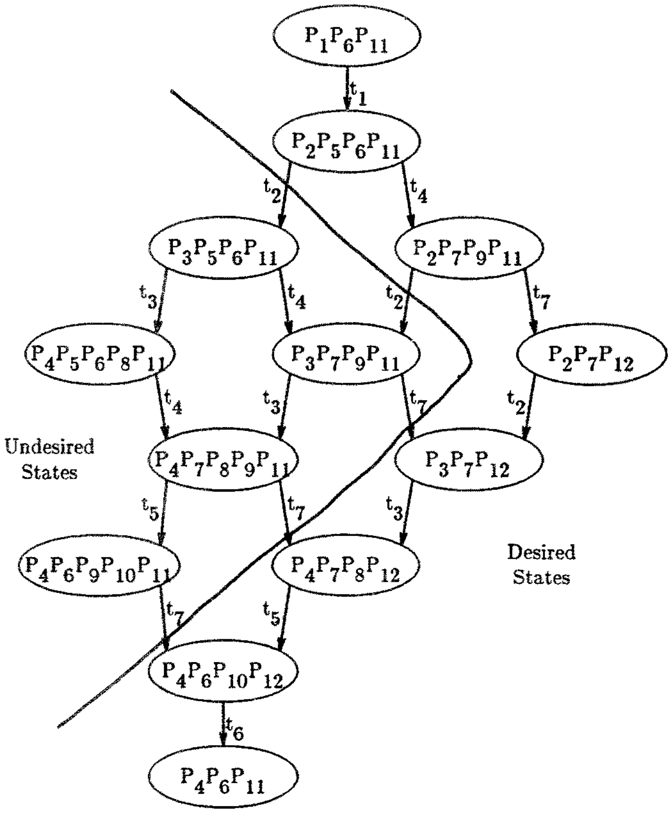


Figure 2a. Reachability Graph for Figure 1

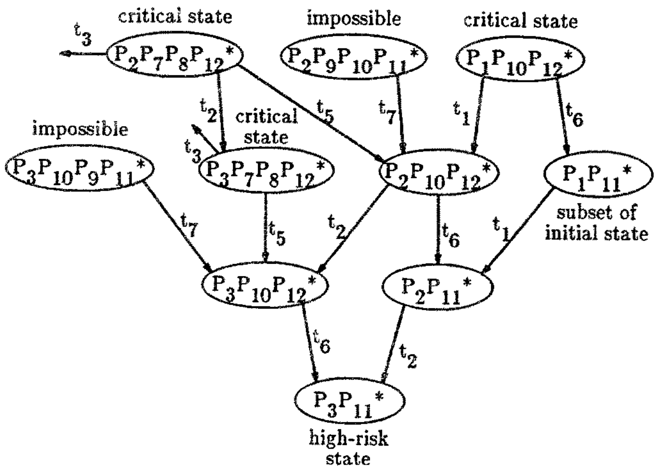


Figure 2b. Example of Critical State Algorithm

Hazards can be classified according to the severity of any possible resulting mishap. For simplicity we will divide hazards into two groups -- high-risk and low-risk -- where high-risk hazards can lead to catastrophic (unacceptable) losses. Of course more categories can and often are used. It is important to note that in many, if not most, realistic systems it is impossible to completely eliminate risk. The goal instead is to design a system with "acceptable risk." [†]

To show that a system is *safe* [‡] or *low-risk*, it is necessary to first ensure that given that the specifications are correctly implemented, no mishaps will result. Second, the risk of faults or failures leading to a mishap must be eliminated or minimized. In this section we discuss how to identify and eliminate high-risk hazards which have been designed into the system. The next section will treat the problem of failures.

Creating the reachability graph allows the designer of a system to determine if the system design can "reach" any high-risk states since it determines all possible states that the system can reach from the initial state by any legal sequence of transition firings. However, this may well be impractical due to the size of the reachability graph for a complex system. In the rest of this section, we describe techniques which may allow the design to be analyzed for safety without producing the entire reachability graph.

The states of a reachability graph can be separated into two disjoint sets: states from which it is possible to reach high-risk and possibly also low-risk states and those from which it is possible to reach only low-risk states.

Definition: A state (marking) μ_c is a *critical state* if and only if

a) $\mu_c \in$ low-risk states and

b) there exist two distinct sequences of transitions s_1 and s_2 and two markings μ_i and μ_j such that $\delta^*(\mu_c, s_1) = \mu_i$ and $\delta^*(\mu_c, s_2) = \mu_j$ where $\mu_i \in$ high-risk states and $\mu_j \in$ low-risk states.

If a high-risk state is reachable, then there must be a critical state on the path from the initial state to the high-risk state (this includes the possibility that the critical state is the initial state). Otherwise, the design needs to be completely redone since all executions result in high-risk states.

To eliminate hazards, it is not necessary to produce the entire reachability graph but only to determine the critical states and to disallow the unwanted transition in each case. Some of our techniques are conservative, i.e. in order to reduce the large amount of computing to produce the entire graph, a larger number of critical states may be identified than actually exist. But note that it does no harm to eliminate a hazard which never existed. Also, as will be seen in the next section when failures are discussed, eliminating a non-existent path may have the effect of eliminating or lessening the possibility of mishaps caused by run-time failures and faults.

One way to locate critical states without necessarily producing the entire reachability graph is to start with the set of high-risk states and to work backward to determine if they are reachable from the initial state. This approach is useful when the goal of the analysis is to prove only that the system cannot reach certain hazardous states. This is often a requirement for safety-critical systems, e.g. see MIL-STD-882b. Fault tree analysis is a similar technique used for the same purpose [Vesely *et. al.* (1981), Leveson and Harvey (1983)]. The backward approach is itself practical only if one considers a relatively small number of high-risk states. This has been found to be adequate in practice [Vesely *et. al.* (1981)]. It is important to note that the concern here is not with correctness, but with system safety. That is, a

[†]What is acceptable risk is often determined by appropriate government licensing agencies. If not predetermined by law, the definition and categorization of mishaps as to severity must be done in the early stages of the system design.

[‡]Because the term "safe" has a specific meaning in Petri net theory (a place is safe if it never contains more than one token), we will use the term "low-risk" where necessary to avoid confusion.

system is "safe" if it is free from mishaps even if it also does not accomplish its "mission" or functional objectives.

To determine if a state can be reached using backward reachability graphs, it is necessary to temporarily ignore timing constraints. The procedure is to first construct the *inverse* untimed Petri net.

Definition: The *inverse* Petri net, Φ^{-1} for a Petri net Φ=(P,T,I,O) is defined by interchanging the input and output functions, Φ^{-1}= (P,T,O,I).

A reachability graph is then constructed using the inverse Petri net and the high-risk state as the initial marking. If the original initial state is reachable, then the mishap may be possible.

Theorem: A high-risk state σ_m is in the reachability set R(Φ,σ_0) if and only if given an initial state σ_0, $\sigma_0 \in$ R(Φ^{-1},σ_m).

The proof can be shown by induction on the sequence of transition firings. By definition if $\mu = \delta(\mu',t)$ then $\mu' = \delta^{-1}(\mu,t)$. This allows the sequence of transitions from σ_0 to σ_m to be traversed in reverse order.

Even though a high-risk state is reachable in the untimed Petri net, it may not be reachable when time constraints are considered. Two approaches are possible. The first is to use the time constraints and work forward from the initial state to determine if the timing constraints have eliminated this path from the timed reachability graph. The other is to assume the worst and just modify the design to ensure that the path is eliminated.

This backward approach is helpful only if the resulting reachability graph is smaller than the original. If the state is reachable, then the backward reachability graph can never be larger than the original reachability graph. Unfortunately, if the high-risk state is not reachable, it is possible for the backward reachability graph to be larger than the original graph and even to be infinite. Therefore, again it may be impractical to generate the entire backward reachability graph.

But if the goal is to ensure that high-risk states can never be reached, it is possible to simply work backward to the first "critical" state (in this case to a state in the reachability graph which has two successors) and to use design techniques such as those outlined below to ensure that the bad path is never taken. It is unimportant as to whether this path actually is reachable since eliminating the possibility of a mishap which would not have occurred does no harm. It is also unimportant if this is truly a critical state as defined above (one path leads to low-risk states) since if the uneliminated path also leads to a mishap, this will be determined in a later step, and this second path will also be eliminated.

The analysis procedure starts with the set of high-risk conditions. For each member of this set, the immediately prior state or states are generated. Each of these "one-step-backward" states is then examined to see if it is a potentially critical state and can be used to eliminate one path to the high-risk state. Note that we are not dealing with complete states but only with partial states. That is, some conditions in the state are unimportant as far as risk goes. Furthermore, we do not know what the complete final states are. Therefore there may be some "don't care" places in each state which are "filled in" in the process of executing the algorithm. Finally, we need only to look forward one step from each potentially critical state in order to label it as critical (i.e. there exists a next-state which is low-risk). This is because if this path also leads to a high-risk state, then it will be eliminated by the algorithm in a later step. The details of the algorithm follow:

```
Put initial set of high-risk conditions into S = states_to_process
while  S is not empty
  do
    let c be one of S;
    if c is a subset of the initial state then
       high-risk state reachable and need to redesign
    else
       do   {work backwards to critical states}
          next_back_states = $\emptyset$
         {determine which transitions are enabled}
          for each transition $t \in T$
             do
               let $R = O(t) \cap c$;
               $TE = O(t) - R$;
               $SE = c - R$;
               if $R \neq \emptyset$ then {t is enabled -  generate the corresponding next backward states}
                  Next_back_states = Next_back_states $\cup\ \delta^{-1}(R \cup TE \cup SE, t)$;
             od
          for each next_back_state b
             do
               Forward_states = set of immediately reachable states $\delta(b,t)$
               Other_states = Forward_states $-$ [Forward_states $\cap$ {S $\cup$ Next_back_states}]
               case b
                  $b \in$ states_considered : exit;
                  b is illegal according to system invariants : exit;
                  b is high risk : add b to S;
                  b is low-risk and there exists a $f \in$ other_states such
                       that f is low-risk {therefore b is potentially critical}: add b to set of critical states;
                  else {b is low-risk but not critical - necessary to go backwards again}
                        add b to S;
               esac
             od
         move c from S to states_considered;
         augment design by eliminating bad transitions from critical states;
       od
end while
```

Using the train example again, figure 2b shows the partial graph generated by the algorithm for the high risk state where the train is approaching (P_3), the gate is up (P_{11}), and any other "don't care conditions" (denoted by the "*") may also hold. Propagating this state backwards, we reach the initial state, impossible states, and critical states. From this we derive the information that in order to avoid the high-risk state, the design must be modified to ensure that transition t_3 has priority over transition t_5 and that transition t_6 has priority over transition t_1.

When a critical state is identified, it is necessary to modify the Petri net in some way to ensure that the good path is always taken, i.e. that another transition always is performed before or has priority over the critical transition.[†]

There are many possible ways to modify the system design in order to eliminate the high-risk states. One common approach is to use an interlock. Interlocks are used to ensure correct sequences of events. An example of a hardware interlock is an access panel or door to equipment where a high voltage exists. Software interlocks include monitors and batons. To model an interlock in a Petri net, assume that t_i is the desired transition, while t_j is the undesired transition. It is possible to force the system always to take the desired path (i.e. to eliminate the undesired path from the reachability graph) by making the following changes to the two transitions in the Petri net. Add a new place (the interlock I) to the output bag of t_i and to the input bag of t_j. This ensures that transition t_i always has priority over transition t_j. There may be multiple desired transitions and an interlock must be applied to each. See figure 3a for an example.

The above type of interlock is used to ensure that one event always precedes another event (e.g. a baton in software). Another type involves ensuring that an event does not occur while a condition is true. This is implemented in the Petri net by using a locking place (see figure 3b). This corresponds to a critical section in software.

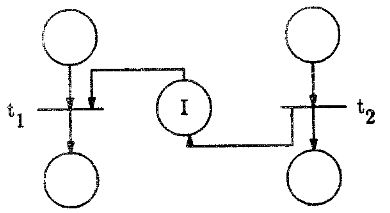


Figure 3a. Interlock

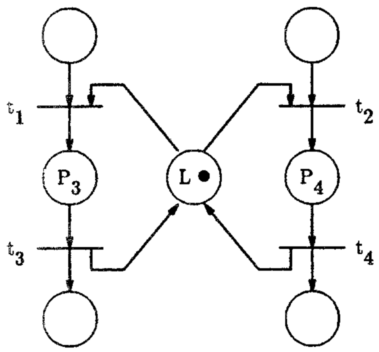


Figure 3b. Locking Place

In the train example, an interlock can be added between t_7 and t_2 (see figure 5a) in order to eliminate the high-risk states. The interlock is included within the computer-controller, but alternatively it might have been part of the hardware. One physical implementation of such an interlock might be a computer-controlled warning signal for the train.

Another way to ensure that one transition will always fire when both are enabled is to enforce timing constraints or timing conditions in the designed system. In order to ensure that a transition t_j (which leads to the high-risk state) does not fires whenever t_i and t_j are both enabled (i.e. the high-risk state is eliminated from the reachability graph), the following timing constraint must be enforced: the maximum time that it may take for the higher priority transition (t_i) to fire must be less than the minimum time for the lower priority transition (t_j) to become enabled and to fire. Each of these time quantities must be the total time that the enabling conditions have been met, not just the individual transition time limit.

One method of determining these quantities is to use the reachability graph to find the maximum (or minimum) valued path leading to the transition which has the required conditions continually enabled. In the system modelled in figure 1, the desired goal is to have condition P_{12} occur before

[†]To require that a transition t_i always have priority over a transition t_j in all situations may be more strict than absolutely necessary but this is true of most safety devices and is one reason why safety occasionally conflicts with other system qualities such as performance.

condition P_3. In terms of the reachability graph this means that when in state $P_2P_5P_6P_{11}$ or $P_2P_7P_9P_{11}$, transition t_2 must not be firable. In the first case, the constraint necessary for t_4 to fire before t_2 is simply that $Min(t_2) > Max(t_4)$. For the second case it is a bit more complicated since firing t_1 results in t_2 being enabled. The constraint in this case is $Min(t_2) > Max(t_7)+Max(t_4)$.

Timing constraints are enforced in systems by either verifying that the design makes it impossible for the constraint to be violated or by using watchdog timers and other devices to determine when the constraint is about to fail and to insert recovery techniques into the system design (either software or hardware). An example is shown in the next section.

Adding Failures to the Analysis

Once the design is determined to have an acceptable level of risk, run-time faults and failures must be considered. Designing for fault tolerance and safety requires being able to model failures and faults and to analyze the resulting model. Using definitions from Kopetz (1982), a failure is defined as an event while a fault is a state. A failure always results in a fault and is called a fault-starting event. The fault remains in the system until the occurrence of a terminating event for this fault. In this paper, we are concerned with control failures. Control failures include:

- 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 (i.e. a condition or set of conditions fail to hold for a particular amount of time)

Each of these types of failures must be able to be modelled in the Petri net. Merlin and Farber (1976) modelled failures in Petri nets as a loss of token or generation of a spurious token. Azema and Diaz (1977) took a similar approach. This was appropriate since Merlin's goal was to analyze failures in communication systems where the primary type of fault is the loss of a message due to failure of the underlying communication medium. However, when dealing with analysis of failures in more general situations, it is often useful to be able to determine the state that a system is in after the failure has occurred (i.e. the fault). For example, if a token is lost when the system is in a state where a particular bit is one, it is important to know whether the failure results in a "stuck at one" state or a "zero" state for the bit. This is because a fault remains in the system until a terminating event for the fault (the faulty condition is no longer true or loses its token). Because of the faulty state or condition, it is possible for further failures to occur which cause further faults. Thus the type of fault which results from the failure must be included in the model in order to analyze the consequences of failures on the system (and thus to differentiate between high and low cost failures). For analysis and readability purposes, it is also useful to model failure events in a different way than normal, expected events.

For these reasons, we introduce a new type of transition, a *failure transition* which acts like other transitions but is denoted by a double bar and a *fault condition* which is denoted by a double circle.† For

†Merlin actually includes failure transitions in his reachability graph (which he calls the error token machine), but does not put them in the Petri net itself.

a Petri net, Φ, the set of transitions becomes $T = T_L \cup T_F$ where T_L are legal transitions and T_F are failure transitions and $T_L \cap T_F = \emptyset$. Similarly, the set of places is now $P = P_L \cup P_F$ where P_L are legal places and P_F are faults and $P_L \cap P_F = \emptyset$. Examples of modelling some of the above types of control failures can be found in figure 4. The failure transitions shown are infinitely fire-able. To make analysis practical, a place which acts as a counter can be added to the failure transition. The number of tokens initially contained in this place controls the maximum number of times the transition (failure) can fire. Realistically, most systems are designed to handle and recover from a maximum number of faults, and the tokens in the counter are the Petri net equivalent of this ceiling value.

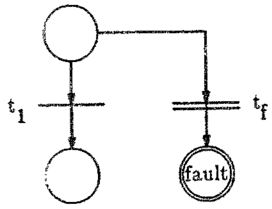


Figure 4a. Desired Event t_1 Does Not Occur

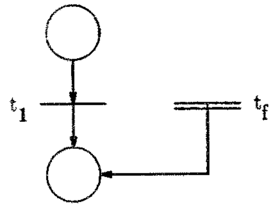


Figure 4b. Undesired Event t_1 Occurs

We now have two types of states: faulty states and legal states.

Definition: A state σ is a *legal state* if and only if there exists a path in the failure reachability graph from the initial state σ_0 to σ which contains only legal transitions, i.e. if σ_0 is the initial state, there exists a sequence of legal transitions $s \in T_L^*$ such that $\delta^*(\sigma_0,s) = \sigma$.

Definition: A state σ is a *faulty state* if and only if every path to σ from the initial state σ_0 contains a failure transition i.e. for every sequence $s \in T^*$ where $\delta^*(\sigma_0,s) = \sigma$ there exists a t_f such that $t_f \in T_F$ and $t_f \in s$

Once failures are included in the model, it is necessary to decide what qualities of the design are important to analyze with respect to control failures. Three such qualities are control fault tolerance, recoverability, and fail-safety. Control fault tolerance implies that a system continues to function correctly (i.e. to provide the service required by its specification) in the presence of component failure. Recoverability implies that a system continues to provide service although the service may be (temporarily) degraded (i.e. may not satisfy *all* the requirements of the specification). A system is fail-safe if component faults do not lead to a catastrophic system failure (mishap) although the system may not provide any service except that required to prevent the catastrophic failure. Each of these qualities can be defined in terms of Petri nets as follows:

Definition: A process is *recoverable* if 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. Formally, a process is recoverable from a failure $t_f \in T_F$ if and only if in the failure reachability graph (FRG):

Let Σ_F be the set of faulty states and let Σ_L be the set of legal states

1) the number of faulty states is finite,

$\quad$ *cardinality*$(\Sigma_F) < \infty$

2) there are no terminal faulty states,

$\quad$ for all $\sigma \in \Sigma_F$, σ is firable

3) there are no directed loops including *only* faulty states,

there does not exist a sequence $t_1 \ldots t_n$ in the FRG such that for $\sigma_i \in \Sigma_F$,
$\delta(\sigma_i, t_i) = \sigma_{i+1}$ for i=1...n-1 and $\sigma_1 = \sigma_{n+1}$

4) the sum of the maximum times on all paths from the failure transition to a correct state is less than a pre-defined acceptable amount of time.

For every path $(t_1, \ldots, t_n)$ from $\sigma_1 \in \Sigma_F$ to $\sigma_2 \in \Sigma_L$,
$\Sigma \text{ Max}(t_j) < T_{\text{acceptable}}$ for j = 1...n

This definition is similar to that of Merlin and Farber (1976), but they allow any finite amount of time to return to normal execution. For many real-time systems, timing constraints are more strict than this. Thus doing nothing for a certain amount of time can be as dangerous under certain conditions as performing an incorrect action even though control is ultimately restored.

Definition: A string A is a *subsequence* of string B if and only if A can be obtained from B by deleting zero or more elements of B.

Definition: A process is *fault-tolerant* for a control failure $t_f \in T_F$ if and only if a) it is recoverable and b) a correct behavior path is a subsequence of every path from the initial state to any terminal state. A correct behavior path is a path in the FRG from the initial state to final state which contains no failure transitions, i.e. a sequence of transitions $t_1 \ldots t_n \in T^*$ such that for all i, $t_i \in T_L$ and $\delta(\sigma_{i-1}, t_i) = \sigma_i$, for i=1...n, σ_n is not firable

Note that for nonterminating or cyclic processes, σ_n may not be a terminal state but may instead be the initial state.

Definition: A system is *fail-safe* if and only if all paths from a failure F in the FRG contain only low-risk states, i.e. for all states σ_f and sequences s_1 such that $\delta^*(\sigma_0, s_1 F) = \sigma_f$ there does not exist a sequence s_2 and state $\sigma_h \in$ high-risk states such that $\delta^*(\sigma_f, F s_2) = \sigma_h$. Note that the system may never get back to a legal state.

The above definitions can be extended to include the possibility of n failures, thus a system, for example, may be n-fault tolerant, n+1-recoverable, and n+2-fail-safe.

Two analysis approaches are possible. The first is to determine, perhaps through past experience, which failures are most likely, and then to create the resulting Failure Reachability Graph (FRG) and analyze it for the above properties. This may be very costly (and possibly impractical) for complex systems with many possible failure modes. Also, in software it is difficult to determine directly which failures are the most likely.

An alternative approach is to take the safety viewpoint and consider only those failures with the most serious consequences. Since this is the requirement of most safety certification programs, there is a practical application for this type of analysis. In this approach, single-point failures and failure sequences which can lead to high-risk states are determined through the analysis after which the design can be augmented with fault-detection and recovery devices to minimize the risk of a mishap. If risk cannot be lowered sufficiently through these devices (e.g. there is an unacceptable probability they will fail or there are uncontrollable variables such as human error involved), it is also possible to add additional safety devices to the design. For example, the designer may add hazard-detection and risk-minimization mechanisms which attempt to ensure that if a hazardous state is reached, the risk will be eliminated or minimized by fail-safe techniques which change the state to a no-risk or lesser-risk state

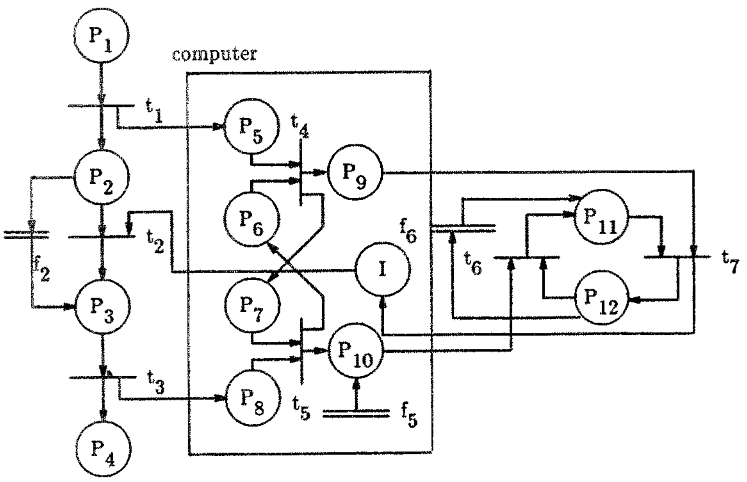


Figure 5a. A Petri Net Graph with Failures

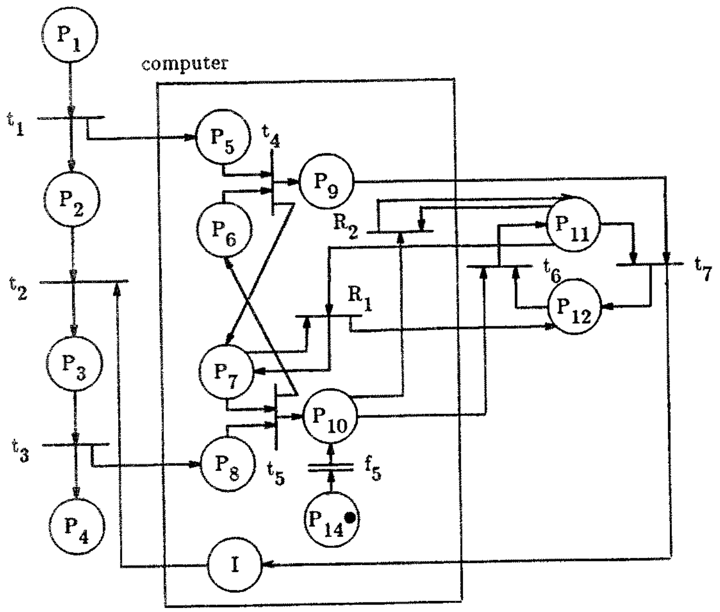


Figure 6. A Petri Net Graph with Failure Transition and Recovery

while at the same time minimizing the exposure time of the hazard.

As an example of the process, consider the Petri-net model in the previous examples. If interested in failures which could result in high-risk states (e.g. the train is approaching, P_3, and the gate is up, P_{11}), a backward reachability graph can be constructed (figure 5b). The high-risk state is not reachable from the regular Petri net, but examination of the reachability graph in figure 5c shows that three single failures (each by themselves) would allow the high-risk state to be reached, i.e. a failure transition f_2 which takes a token from P_2 and puts one in P_3, a failure transition f_6 which does the same for P_{12} and P_{11}, and a failure transition f_5 which involves an erroneous generation of a token in P_{10}. Failure transition f_2 is a human failure where the train ignores the warning signal. Transition f_6 is a gate failure which results in a premature gate raising. The last failure, f_5, could be caused by a spurious signal from the controlling computer. Normally, the designer would now include standard failure detection mechanisms in the design along with recovery procedures.

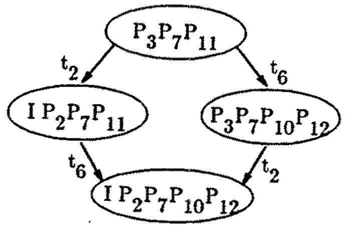


Figure 5b
Backwards Reachability Graph

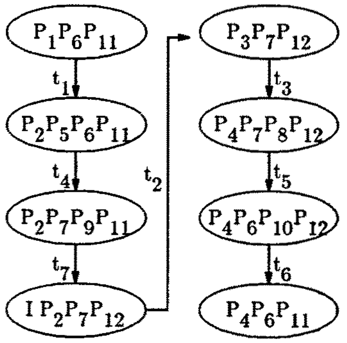
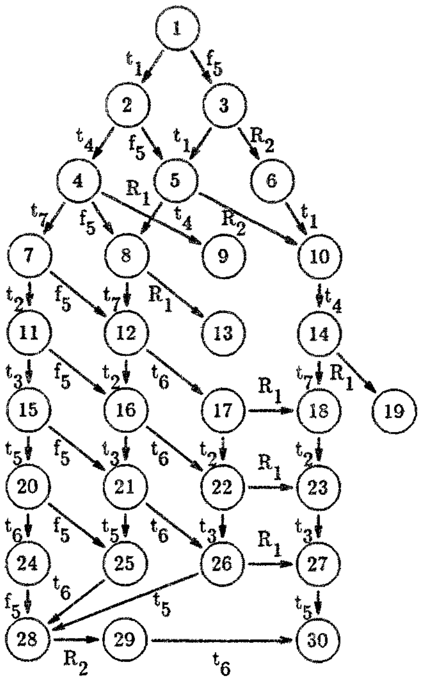


Figure 5c
Reachability Graph for Figure 5a

Failure transition f_5 in figure 5a was chosen as the basis for the fault tolerance mechanism shown in figure 6. This failure models a spurious output signal from the computer. Transitions R_1 and R_2 are used for fault detection and subsequent recovery. After a failure, there are two possible situations depending on the current state of the gate. If the gate is up then one response to a spurious up signal is to ignore it (shown in transition R_2). The enabling conditions are P_{11} (gate up) and P_{10} (signal from the computer).

The second possibility is the safety critical situation. In this case a train in approaching, the gate is down, and the erroneous signal is given to raise the gate. In order to detect the problem, redundant information must be contained in the system. The model has an internal "view of the world" contained in P_6 and P_7 which correspond directly to the actual conditions P_{11} and P_{12}. Fault detection is accomplished by checking to see if P_7 and P_{11} occur at the same time. If so, there is a discrepancy between the real world and the internal state.

Upon failure detection, there are several possible recoveries -- depending on which model is accepted as the true state of the system (i.e. is the computer state wrong or is the gate really up when it should be down). The safest solution is to assume the gate is up and lower it. This is the purpose of transition R_1. Figure 7 shows the reachability graph for this net. The untimed reachability graph shows that for the state labelled 4 (conditions P_2, P_7, P_9, P_{11}, and P_{14}), recovery is initiated when a failure has not occurred. Further investigation reveals that there is a point in time when the computer state is legitimately inconsistent with the actual world (after t_4 has fired but before t_7 fires). One solution is to put a time constraint on R_1 such that the minimum time of R_1 is greater than the maximum time of t_7. This forces failure detection to wait until a consistent state has been permitted.



State #	Places	State #	Places
1	P_1 P_6 P_{11} P_{14}	16	P_3 P_7 P_{10} P_{12}
2	P_2 P_5 P_6 P_{11} P_{14}	17	P_2 P_7 P_{11} I
3	P_1 P_6 P_{10} P_{11}	18	P_2 P_7 P_{12} I
4	P_2 P_7 P_9 P_{11} P_{14}	19	P_2 P_7 P_9 P_{12}
5	P_2 P_5 P_6 P_{10} P_{11}	20	P_4 P_6 P_{10} P_{12} P_{14}
6	P_1 P_6 P_{11}	21	P_4 P_7 P_8 P_{10} P_{12}
7	P_2 P_7 P_{12} P_{14} I	22	P_3 P_7 P_{11}
8	P_2 P_7 P_9 P_{10} P_{11}	23	P_3 P_7 P_{12}
9	P_2 P_7 P_9 P_{12} P_{14}	24	P_4 P_6 P_{11} P_{14}
10	P_2 P_5 P_6 P_{11}	25	P_4 P_6 P_{10} P_{10} P_{12}
11	P_3 P_7 P_{12} P_{14}	26	P_4 P_7 P_8 P_{11}
12	P_2 P_7 P_{10} P_{12} I	27	P_4 P_7 P_8 P_{12}
13	P_2 P_7 P_9 P_{10} P_{12}	28	P_4 P_6 P_{10} P_{11}
14	P_2 P_7 P_9 P_{11}	29	P_4 P_6 P_{11}
15	P_4 P_7 P_8 P_{12} P_{14}	30	P_4 P_6 P_{10} P_{12}

Figure 7. Reachability Graph for Figure 6

In summary, analysis of the failure reachability graph with respect to the definitions of fault tolerant, recoverable, and fail-safe design will aid the designer in adding appropriate failure detection and recovery techniques to the system. When interested solely in a safety analysis, backward procedures can be used to determine which failures and faults are potentially the most costly and thus need to be augmented with fault tolerance mechanisms and also to determine where and how safety mechanisms should be used. This may be particularly useful for the software components of the system since it is difficult to determine which faults are most likely to occur and the potential number of failures to model may be very large. Furthermore, it is possible to treat the software at various levels of abstraction, e.g. only failures of the interfaces of the software and non-software components may be considered or more detailed failures of only those particular modules which are determined to be critical may be modelled.

Conclusions

The use of Time Petri nets in design and analysis of safety-critical, real-time systems has been described and the basic model extended to allow modelling failures and faults. This allows the system to be analyzed for properties such as fault-tolerance and safety, to determine which functions are most critical and thus may need to be made fault-tolerant (assuming that it may be too costly to ensure complete fault-tolerance), to determine conditions which require immediate mitigating action to prevent accidents, to determine possible sequences of failures which can lead to accidents, etc. Thus it is possible to establish important properties during the synthesis of the design instead of using guesswork and costly *a posteriori* analysis (including formal analysis and testing).

Unfortunately, Petri nets can be difficult to analyze. For general Petri nets, the reachability problem, though decidable, has been shown to be exponential time- and space-hard. Although this is not a necessary property of Petri net models (many important and real systems can be analyzed efficiently), it is a possible result when complex systems are modelled. Some techniques which are useful even if the entire reachability graph is not completed have been presented in this paper. It is also possible to use the failure-enhanced Time Petri net model as the basis for a simulation in order to answer some of the same questions which could have been answered by the failure reachability graph. Finally, many real-time systems require the computer software to be written and tested before the hardware components have been completed. Since the Time Petri net model is executable, the hardware parts can be used as a test bed for the software development process.

In this paper, only severity of hazards was considered and not the probability of the hazard occurring or of leading to a mishap. This is a pessimistic approach (i.e. all hazards are considered to have equally high probabilities). We are currently devising techniques to include probabilities in the analysis. This will enable the designer to use a more sophisticated definition of risk and to derive measurements for risk (and thus safety) from the model. This in turn can provide the information required by the designer to make difficult tradeoff decisions, e.g. what if there are two possible recovery methods, one of which is more likely to work but also has worse penalties in the event of failure (perhaps in terms of taking so long to execute that no other alternatives or fail-safe procedures are still feasible).

References

[1] Azema, P., and Diaz, M. "Checking Experiments for Concurrent Systems," *FTCS-7*, June 1977, p. 206.

[2] Malasky, S.W. *System Safety: Technology and Application*, Garland STPM Press, New York, 1982.

[3] Kopetz, H. "The Failure Fault (FF) Model," *FTCS-12*, Santa Monica, Calif., June 1982, pp. 14-17.

[4] Leveson, N.G. and Harvey, P.R. "Analyzing Software Safety," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 5, Sept. 1983.

[5] Leveson, N.G. "Software Safety in Process-Control Systems," *IEEE Computer*, February 1984.

[6] Merlin, P.M. "A Study of the Recoverability of Computing Systems," Ph.D. Thesis, Information and Computer Science Department, University of California, Irvine, 1974.

[7] Merlin, P.M. and Farber, D.J. "Recoverability of Communication Protocols -- Implications of a Theoretical Study," *IEEE Transactions on Communications*, vol. COM-24, no. 9, September 1976, pp. 1036-1043.

[8] MIL-STD-882b, System Safety Program Requirements, U.S. Department of Defense, April 1984.

[9] Peterson, J.L. *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.

[10] Vesely, W.E., Goldberg, F.F., Roberts, N.H., and Haasl, D.F. *Fault Tree Handbook*, NUREG-0492, U.S. Nuclear Regulatory Commission, January 1981.