Safety Analysis Using Petri Nets

NANCY G. LEVESON AND JANICE L. STOLZY

Abstract—**The application of Time Petri net modeling and analysis techniques to safety-critical real-time systems is explored and procedures described which allow analysis of safety, recoverability, and fault-tolerance.**

Index Terms—**Fault-tolerance, Petri nets, requirements, software reliability, software safety.**

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.

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 [6]). Unfortunately, much more is known about how to engineer safe mechanical systems than safe computer-controlled 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 Safety Program Requirements or MIL-STD-1794; Safety Requirements for Space and Missile Systems). Modeling and analysis tools are desperately needed to aid in these tasks as the standard software tools and methods which currently exist do not satisfy these requirements.

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 fault may cause an accident only if there is a simultaneous human and/or hardware failure. Alternatively, an environmental event or failure may adversely affect the software. Accidents are often the result of multiple failure sequences which involve hardware, software, and human failures.

Manuscript received August 15, 1984; revised January 31, 1986. This work was supported in part by a MICRO Grant co-funded by the state of California and Hughes Aircraft Co., and by the National Science Foundation under Grant DCR-8406532.

The authors are with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

IEEE Log Number 8612565.

Petri nets have been used to model and analyze systems for such properties as deadlock and reachability. In this paper we show how they can be used in designing and analyzing such properties as safety and fault-tolerance. A systems approach is possible with Petri nets since hardware (e.g., [2], [5]), software (e.g., [13], [14], [17]), and human behavior can be modeled using the same language. 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. Techniques such as Failure Modes and Effects Analysis (FMEA) and Preliminary Hazard Analysis (PHA) 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 Time Petri net analysis techniques such as those described in this paper and thus to derive the software safety requirements.

Using Time Petri nets allows the incorporation of timing information into the analysis—a necessity for real-time embedded system analysis. In these systems, for example, basically correct software actions which are too early or too late can lead to unsafe conditions. Coolahan and Roussopoulos [4] have shown how Petri nets can be used to derive timing requirements for modules in real-time systems where the service involves repetitive performance of similar activities at a fixed, constant, and predetermined interval. A more general procedure which does not have these limitations can be found in Merlin [9] and Merlin and Farber [10]. In this paper we extend the definitions used by Merlin and Farber. Since our goal is not performance analysis using known times but the derivation of timing constraints and determination of worst cases (including the potential effects of timing failures), much of our analysis will involve deriving the untimed reachability graph (or parts of it) 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.

Petri net models can also be used to determine the most critical software functions which can then be augmented with fault tolerance facilities and to determine the conditions which must be incorporated into the run-time tests associated with these facilities such as watchdog timers and acceptance tests in recovery blocks.

The next section presents general definitions for Time Petri nets. Following that, procedures are described for eliminating hazards from a design without generating the entire Petri net reachability graph. The final section adds failures to the analysis procedures.

Definition of Time Petri Nets

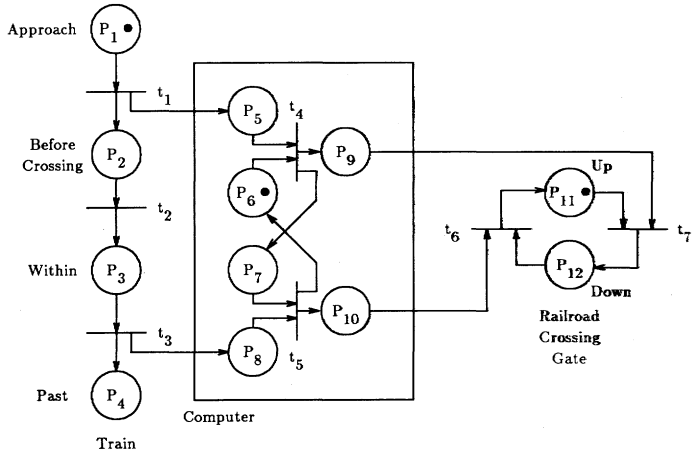
A formal definition of Petri nets can be found in the Appendix. The notation and terminology used in this paper closely follows that of Peterson [14]. For those unfamiliar with Petri nets, an informal definition follows.

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 that 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.

A graph structure is often used for illustration of Petri nets where a *circle* "○" represents a place and a *bar* "|" represents a transition. Fig. 1 shows a Petri net and the corresponding Petri net graph. 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. The execution of a Petri net is controlled by the number and distribution of tokens in the Petri net. A transition 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. When a transition is enabled, it may *fire*. When a transition fires, all enabling tokens are removed from its input places, and a token is deposited in each of its output places. Given the Petri net marking in Fig. 1, the next state after firing transition t_1 is shown in Fig. 2. 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. Fig. 1 can be interpreted as a model of a simple railroad crossing. Three parts of the system—the train (on the left), the computer or controlling device (in the large box), and the crossing gate (on the right)—are modeled. 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.



(a)

$P = \{ P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{11}, P_{12} \}$

$T = \{ t_1, t_2, t_3, t_4, t_5, t_6, t_7 \}$

$\mu_0 = (1,0,0,0,0,1,0,0,0,0,1,0)$

$I(t_1) = \{P_1\}$	$O(t_1) = \{P_2, P_5\}$
$I(t_2) = \{P_2\}$	$O(t_2) = \{P_3\}$
$I(t_3) = \{P_3\}$	$O(t_3) = \{P_4, P_8\}$
$I(t_4) = \{P_5, P_6\}$	$O(t_4) = \{P_7, P_9\}$
$I(t_5) = \{P_7, P_8\}$	$O(t_5) = \{P_6, P_{10}\}$
$I(t_6) = \{P_{10}, P_{12}\}$	$O(t_6) = \{P_{11}\}$
$I(t_7) = \{P_9, P_{11}\}$	$O(t_7) = \{P_{12}\}$

(b)

Fig. 1. (a) A Petri net graph. (b) Description of the above Petri net.

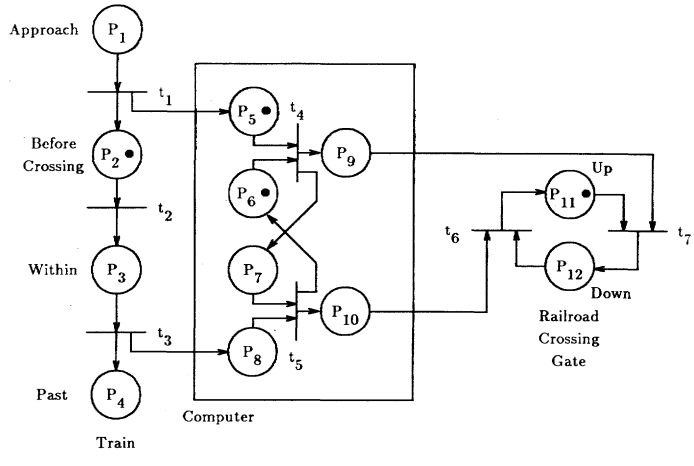


Fig. 2. A Petri net graph with the next state shown.

The state of the Petri net (and hence the state of the modeled 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. Given an initial state, the reachability set for the Petri net is the set of states that results from executing the Petri net.

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 Fig. 3(a)].

To model time requires enhancements to the basic Petri

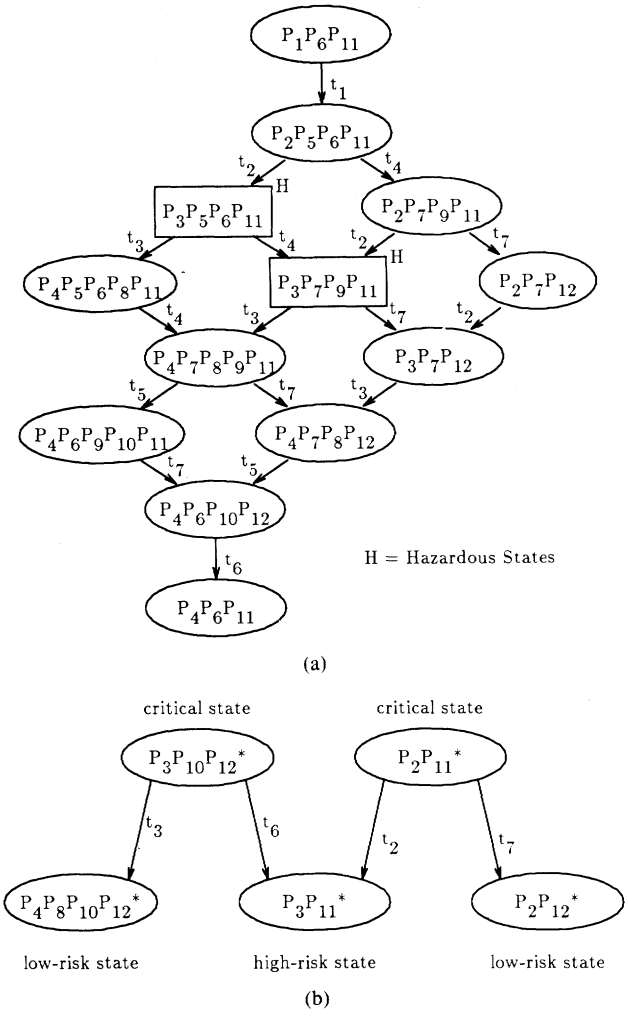


Fig. 3. (a) Reachability graph for Fig. 1. (b) Example of critical state algorithm.

net model. There have been several proposals for extending standard Petri nets to include time. Ramchandani [15] proposed associating delays with transitions. Merlin [9] proposed using two values, Min and Max times, to define a range of delays for each transition. This approach has also been used by Berthomieu and Menasche [3]. Sifakis [18] proposed instead associating the delays with places. Coolahan and Roussopoulos [4] employed an approach similar to Sifakis. Associating delays with places does not increase the power of the model, but does retain the instantaneous firing feature of the basic Petri net model. In fact, transition delays and place delays are equivalent since one can be translated into the other. Razouk [16] has proposed using firing times along with enabling times. In his model, the tokens are absorbed by the transition after the enabling time has elapsed and do not reappear on the output places until after the transition finishes firing (i.e., after the firing time has elapsed). This model is less flexible than the Merlin and Farber model, but does make performance analysis easier.

Since our goal is not performance analysis using known times but the derivation of timing constraints, we have chosen to use the Merlin and Farber model. Tokens are allowed to remain on the input places during the transition delay so the model retains the instantaneous firing feature of untimed Petri nets while also providing a very flexible modeling tool.

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, 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,

$\Phi = (P, T, I, O, \text{Min}, \text{Max}, \mu_0)$.

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

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

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.

Min and Max are the *min time function* and *max time function*, respectively, where

$\text{Min}: T \rightarrow R$ and $\text{Max}: T \rightarrow R$, R is the set of nonnegative real numbers

and

$\text{Min}_i \leq \text{Max}_i$ for all i such that $t_i \in T$.

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

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

Definition: The state of the net σ consists of the tuple (μ, E) where μ is the marking and E is the remaining enabling time vector. E is a function of a set of tuples of real numbers R, $E: (R \times R) \rightarrow (R \times R)$.

An excellent description of the next-state function for Time Petri nets can be found in [3]. Added complexity over the untimed Petri net arises because of the continuous nature of time. Since transitions may fire at any time in their allowed interval, the states have in general an unbounded number of successors. Berthomieu and Menasche solve this problem by defining state classes that consider the set of all states reachable from the initial state by a given sequence of transitions.

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.

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.[1] 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.

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. In general, from a safety standpoint, the first priority of the response to a safety-critical situation is reduction of risk rather than attainment of mission [8].

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.

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 $\delta^*(\sigma, s) = \sigma_m$.

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). Often early in the design of a system, the probabilities are unknown and the analysis is done considering only severity (the procedure which will be followed in this paper). 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."[2]

The overall goal in designing a safety-critical system is to eliminate hazards from the design or (if that is not possible) to minimize risk by altering the design so that there is a very low probability of the hazard occurring. To show that a system is *safe*[3] or *low-risk*, it is necessary to first ensure that given that the specifications are correctly implemented and no failures occur, operation of the system will not result in a mishap. Second, the risk of faults or failures leading to a mishap must be eliminated or minimized by using fault-tolerance or fail-safe procedures. If it is not possible to eliminate completely the possibility of a hazard occurring, then in order to reduce risk the *exposure time* (length of time of occurrence) of the hazardous conditions must be 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 identifies all possible states that the system can reach from the initial state by any legal sequence of transition firings. In the train example, a

[1] 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.

[2] What is acceptable risk is often determined by appropriate government licensing agencies. For example, mishaps have been defined by the NRC for all nuclear systems. 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.

[3] 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.

hazard occurs when both a train is approaching the crossing (P_3) and the guard gate is up (P_{11}). The hazardous states are shown in Fig. 3(a). Generating the entire reachability graph may well be impractical due to the size of the 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 set.

One way to do a safety hazard analysis is to work backward from the high-risk state to determine if it is reachable. 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 [19]. 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 [19]. It is important to note that the concern here is not with correctness, but with system safety. That is, a system is "safe" if it is free from mishaps even if it does not accomplish its mission or functional objectives.

By using the inverse Petri net (where the input and output functions are reversed), it can be determined if a high-risk state is reachable by using the high-risk state as the initial state and determining whether the original initial state is reachable. Unfortunately, it is possible for the backward reachability graph to be as large as or even larger than the original graph. Our solution is an algorithm which does not require the entire backward reachability graph to be generated. The algorithm requires the definition of a particular type of state that we call a *critical* state.

The states of a reachability set 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 nonempty 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 (e.g., the crossing gate always remains up when the train is approaching).

To ensure that high-risk states can never be reached, it is possible simply to work backward to the first critical state (i.e., to a state in the reachability graph that has two successors) and to use design techniques such as those outlined below to ensure that the bad path is never taken. The technique is 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 the possibility of a mishap that would not have occurred. Also, as will be seen in the next section when failures are discussed, eliminating a nonexistent path may have the effect of eliminating or lessening the possibility of mishaps caused by run-time faults and failures. 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 the second path will also be eliminated.

The algorithm 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 start not with complete states but only with partial states. That is, some conditions in the state are unimportant as far as risk goes, and thus it is not known at the beginning of the algorithm the complete composition of the reachable high-risk states (the complete states from which to start the backward analysis). The "don't care" places in each state are "filled in" with those conditions that are possible 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 that 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 following describes the details of the algorithm to identify and eliminate critical states:

```
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 = φ
            for each transition t ∈ T {determine which transitions are enabled}
               do
                  let R = O(t) ∩ c;
                  if R ≠ φ then {t is enabled, generate the corresponding next backward states}
                     Next_back_states = Next_back_states ∪ δ⁻¹(R ∪ (O(t) − R) ∪ (c − R),t);
               od
```

```
        for each next_back_state b
            do
                forward_states = φ
                for each transition t ∈ T {determine which transitions are enabled}
                    do
                        let R = I(t) ∩ b;
                        if R ≠ φ then {t is enabled, generate the corresponding forward states}
                            forward_states = forward_states ∪ ∪(R̄ ∪ (I(t) − R) ∪ (b − R),t);
                    od
                Other_states = Forward_states − [Forward_states ∩ {S ∪ Next_back_states}]
                case b
                    b ∈ 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 f ∈ 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 transition paths from critical states;
    od
od
```

Using the train example again, Fig. 3(b) 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 derive the information that in order to avoid the high-risk state, the design must be modified to ensure that transition t_3 has precedence over transition t_6 and that transition t_7 has precedence over transition t_2.

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 precedence over the critical transition.[4]

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 precedence over transition t_j. There may be multiple desired transitions and an interlock must be applied to each. See Fig. 4(a) 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 Fig. 4(b)]. This corresponds to a critical section in software.

In the train example, an interlock can be added between t_7 and t_2 [see Fig. 4(c) and (d)] 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 fire 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 desired transition (t_i) to fire must be less than the minimum time for the other 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 that has the required conditions continually enabled. In the system modeled in Figs. 1 and 3, the desired goal is to have condition P_{12} occur before 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

[4]To require that a transition t_i always have precedence 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.

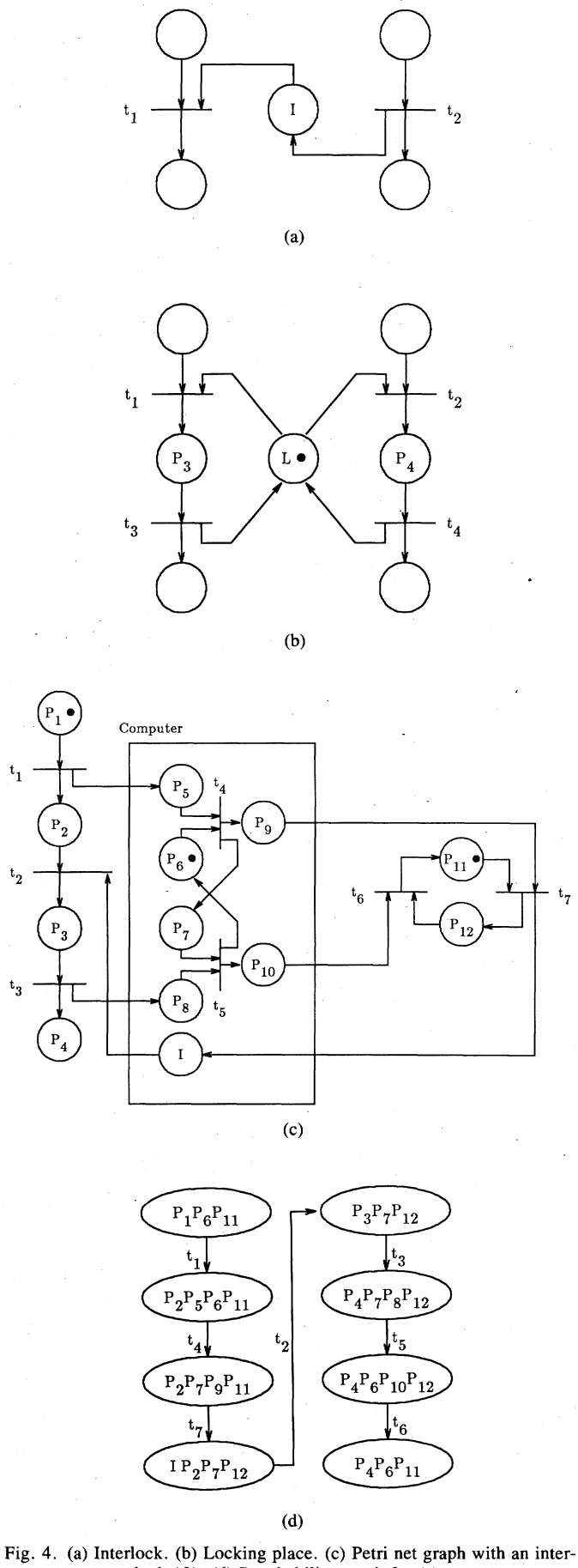


Fig. 4. (a) Interlock. (b) Locking place. (c) Petri net graph with an interlock (I). (d) Reachability graph for (c).

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 (either hardware or software) into the system design. An example is shown in the next section. It should be noted that this procedure only identifies possible ways to augment a design to make it safer. The actual interlocks and timers that are used must be considered from an engineering feasibility and cost standpoint. If the design is found to involve many hazards, a complete redesign may be preferable to patching the original design.

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 [7], 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 modeled in the Petri net. Merlin and Farber [10] modeled failures in Petri nets as a loss of a token or generation of a spurious token. Azema and Diaz [1] 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 1, it is important to know whether the failure results in a "stuck at 1" 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 that 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

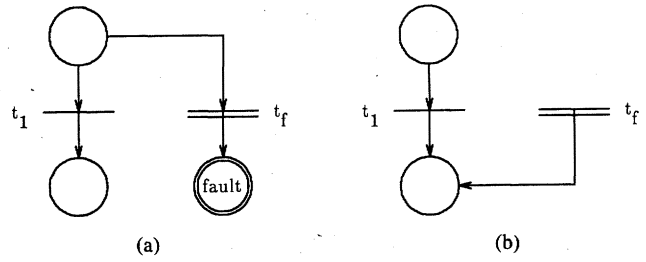


Fig. 5. (a) Desired event t_1 does not occur. (b) Undesired event t_1 occurs.

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.[5] For 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 $T_L \cap T_F = \phi$. 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 $P_L \cap P_F = \phi$. Examples of modeling some of the above types of control failures can be found in Fig. 5. The failure transitions shown are infinitely firable. 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.

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 that 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. 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,

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

2) there are no terminal faulty states,

for all $\sigma \in \Sigma_F$, there exists a $t \in T$ such that $\delta(\sigma, t_i) = \sigma'$

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

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

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

This definition is similar to that of Merlin and Farber [10], 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.

The problem with this definition is that it does not allow for any type of system degradation. Once a permanent failure has occurred, by definition the state cannot return to normal unless some repair action has taken place. For example, a hardware system using standby sparing is not recoverable because once a spare has been "swapped-in" the system cannot return to a legal state (one which existed before the failure) since there is obviously one less spare than originally. Recoverability has been used in modeling communication protocols since the loss of a message can be thought of as a transient fault. To define general fault tolerance, different conditions are needed.

Definition: A correct behavior path is a path in the FRG from the initial state (σ_0) to final state (σ_n) which contains no failure transitions, i.e., a sequence of transitions $t_1 \cdots t_n \in T^*$ such that for all i, $t_i \in T_L$ and $\delta(\sigma_{i-1}, t_i) = \sigma_i$, for $i = 1 \ldots n$.

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) a correct behavior path is a subsequence of *every* path from the initial state to any terminal state. b) the sum of the maximum times on all paths is less than a predefined acceptable amount of time.

[5]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.

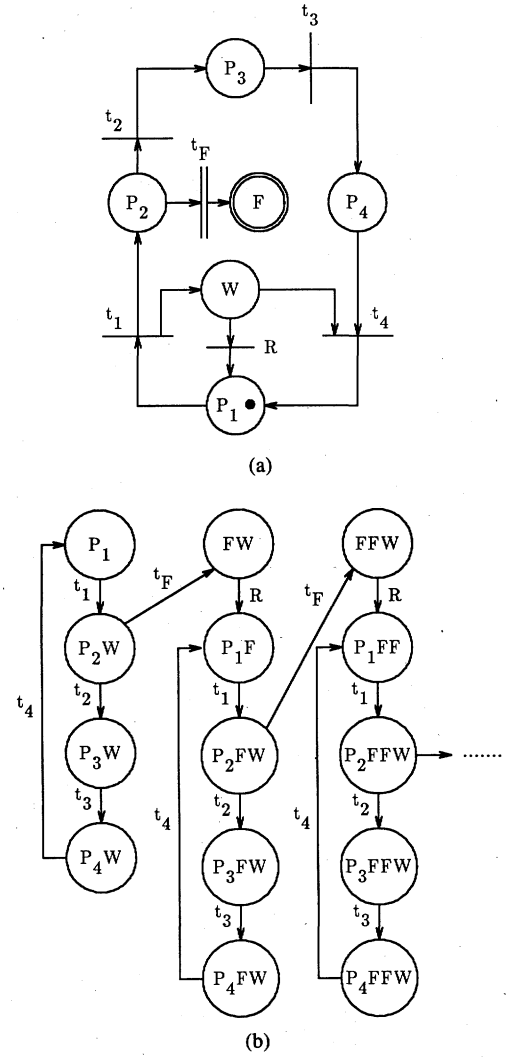


Fig. 6. (a) A fault-tolerant process using a watchdog timer. (b) Reachability graph for (a).

$$\text{For every path } (t_1, \cdots, t_n) \text{ from } \sigma_0 \text{ to } \sigma_n,$$
$$\Sigma \text{ Max } (t_j) < T_{\text{acceptable}} \text{ for } j = 1 \ldots n$$

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

Fig. 6(a) shows an example of a fault tolerant design that uses a watchdog timer to detect and recover from the failure. A different type of detection and recovery scheme is shown in a later example. In Fig. 6(a), the initial state has a token in P_1. When transition t_1 fires, it both puts a token into P_2 and starts the watchdog timer by putting a token in W. If everything works successfully, transition t_4 pulls the token out of W thus stopping the timer. If a failure t_F occurs (for simplicity shown only for P_2 although it could occur anywhere), then transition R should fire and start a token on P_1 again. However, in the design as it stands, R could fire any time after t_1. What is desired is that the timer should fire after a period that indicates the process could not be working correctly. So it is necessary to make Min $(R) \geq$ Max (t_2) + Max (t_3) + Max (t_4). Fig. 6(b) shows the resulting reachability graph given this timing constraint.

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 $\delta^*(\sigma_0, s_1F) = \sigma_f$ there does not exist a sequence s_2 and state $\sigma_h \in$ high-risk states such that $\delta^*(\sigma_f, Fs_2) = \sigma_h$. 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. Often a system can be designed only to be fault-tolerant for a fixed number of faults. For example, there may be only n spares available. Therefore, a design goal may be to ensure that the system is n-fault tolerant and n + 1-fail-safe. Note also that by the definitions it is possible for a system to be fault-tolerant but not fail-safe. That is, the failure may put the system into a high-risk state (the gate is up and the train is coming). If the failure cannot be avoided, then it is necessary to minimize risk. Since the mishap occurs only when another event occurs, i.e., a car approaches the train crossing, risk is reduced by minimizing the time that the fault is present in the system (exposure time). This in turn is the min time for the recovery transition (or transitions).

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 that 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 that attempt to ensure that if a hazardous state is reached, the risk will be eliminated or minimized by fail-safe techniques that change the state to a no-risk or lesser-risk state 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 after putting in the interlock described above. If interested in failures that 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 [Fig. 7(b)]. The high-risk state is not reachable from the regular Petri net, but examination of the reachability graph in Fig. 4(d) shows that three single failures (each by themselves) would allow the high-risk

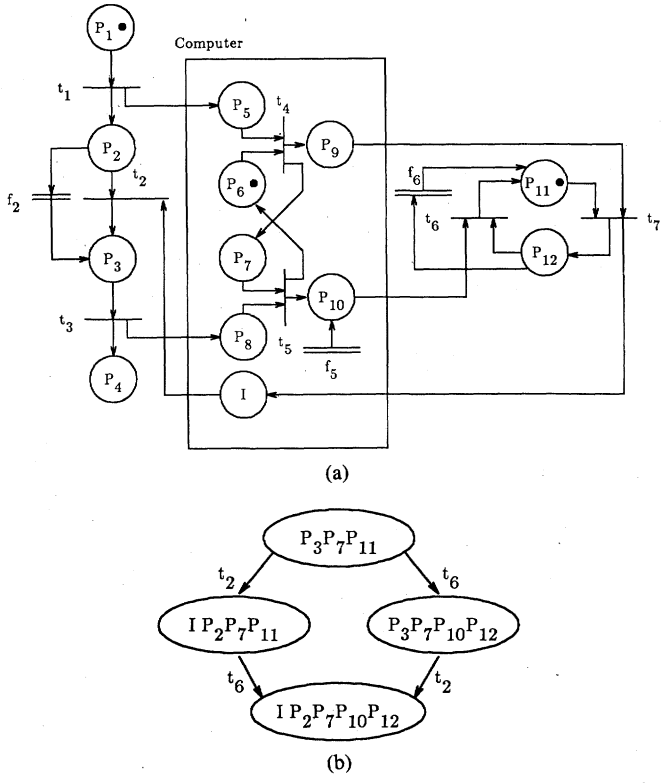


Fig. 7. (a) Petri net graph with failures. (b) Backwards reachability graph.

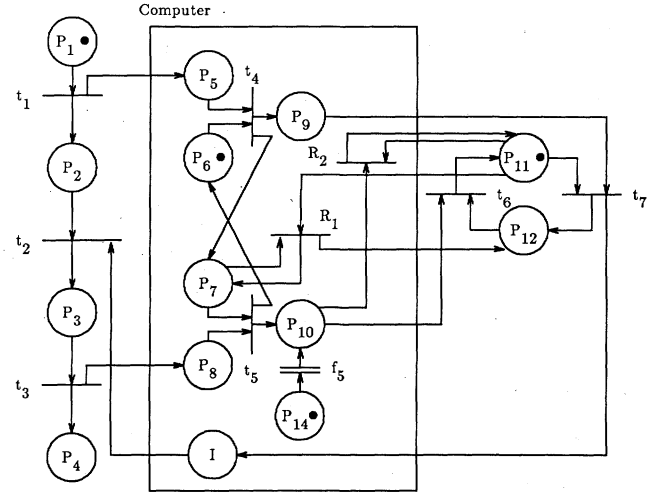


Fig. 8. A Petri net graph with failure and recovery.

state to be reached, i.e., a failure transition f_2 that takes a token from P_2 and puts one in P_3, a failure transition f_6 that does the same for P_{12} and P_{11}, and a failure transition f_5 that 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 that 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.

Failure transition f_5 in Fig. 7(a) was chosen as the basis for the fault tolerance mechanism shown in Fig. 8. This failure models a spurious output signal from the computer. The number of tokens in P_{14} represent the maximum number of failures that can occur during analysis. The analysis performed here is for at most one failure. Transitions R_1 and R_2 are used for fault detection and subsequent recovery. After a failure, the system can be in 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 is approaching, the gate is down, and the erroneous signal is given to raise the gate. In order to detect the situation, 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. Fig. 9 shows the reachability graph for this net. From the untimed reachability graph we see that for the state labeled 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. States labeled 9 and 19 were not developed further because the timing constraint made the states unreachable.

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 nonsoftware components may be considered or more detailed failures of only those particular

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

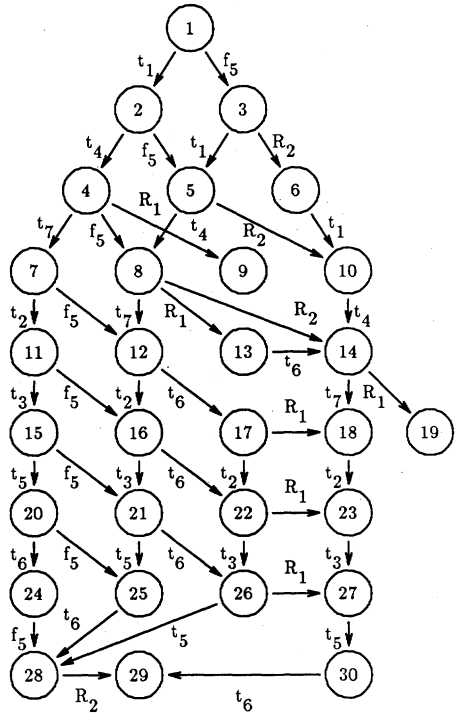


Fig. 9. Reachability graph for Fig. 8.

modules which are determined to be critical may be modeled.

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 modeling 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 that 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, although 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 modeled. Some techniques that 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 that 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 is 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).

Appendix

The formal definition of Petri nets follows. In general, the notation used in [14] is followed.

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

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

$T = \{t_1, t_2, \cdots, t_m\}$ is a finite set of *transitions*, m

≥ 0. The set of places and the set of transitions are disjoint, $P \cap T = \phi$.

$I: T \rightarrow P^{\infty}$ 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 nonnegative 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 $\#(p_i, O(t_j))$.

Definition: The *next-state function* $\delta: N^n \times T \rightarrow N^n$ 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

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

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 $\delta(\mu, t_j) = \mu'$.

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 $\mu'' \in R(\Phi, \mu)$.

Definition: A path in the reachability graph is a sequence of transitions $t_i, \cdots, t_j$ starting at marking μ_{i-1} to μ_j such that

$$\delta(\mu_{n-1}, t_n) = \mu_n \quad \text{for} \quad n = i \ldots j.$$

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

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

$$\delta^*(\mu, \lambda) = \mu.$$

REFERENCES

[1] P. Azema and M. Diaz, "Checking experiments for concurrent systems," in *Proc. FTCS-7*, June 1977, p. 206.

[2] P. Azema, R. Valette, and M. Diaz, "Petri nets as a common tool for design verification and hardware simulation," in *Proc. 13th IEEE Design Automation Conf.*, June 1976, pp. 109–116.

[3] B. Berthomieu and M. Menasche, "An enumerative approach for analyzing time Petri nets," in *Proc. 1983 IFIP Congress*, Paris, Sept. 1983.

[4] J. E. Coolahan and N. Roussopoulos, "Timing requirements for time-driven systems using augmented Petri nets," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 603–616, Sept. 1983.

[5] M. Hack, "Analysis of production schemata by Petri nets," Massachusetts Inst. Technol., Tech. Rep. 94, Project MAC, Feb. 1972.

[6] W. Hammer, *Handbook of System and Product Safety*. Englewood Cliffs, NJ: Prentice-Hall, 1972.

[7] H. Kopetz, "The failure fault (FF) model," in *Proc. FTCS-12*, Santa Monica, CA, June 1982, pp. 14–17.

[8] N. G. Leveson, "Software safety in process-control systems," *Computer*, Feb. 1984.

[9] P. M. Merlin, "A study of the recoverability of computing systems," Ph.D. dissertation, Dep. Inform. Comput. Sci., Univ. of California, Irvine, 1974.

[10] P. M. Merlin and D. J. Farber, "Recoverability of communication protocols—Implications of a theoretical study," *IEEE Trans. Commun.*, vol. COM-24, pp. 1036–1043, Sept. 1976.

[11] *System Safety Program Requirements*, U.S. Dep. Defense, Standard MIL-STD-882b, Apr. 1984.

[12] *Safety Requirements for Space and Missile Systems*, U.S. Dep. Defense, Standard MIL-STD-1794, Apr. 1984.

[13] R. A. Nelson, L. M. Haibt, and P. B. Sheridan, "Casting Petri nets into programs," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 590–602, Sept. 1983.

[14] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[15] C. Ramchandani, "Analysis of asynchronous concurrent systems by timed Petri nets," Ph.D. dissertation, Massachusetts Inst. Technol., Project MAC Rep. MAC-TR-120, 1974.

[16] R. R. Razouk, "The derivation of performance expressions for communication protocols from timed Petri net models," Dep. Inform. Comput. Sci., Univ. California, Irvine, Tech. Rep. 211, Nov. 1983.

[17] M. Rose, "Modeling and analysis of concurrent systems using contour/transition-nets," Ph.D. dissertation, Dep. Inform. Comput. Sci., Univ. California, Irvine, 1984.

[18] J. Sifakis, "Petri nets for performance evaluation," in *Measuring, Modeling, and Evaluating Computer Systems (Proc. 3rd Symp., IFIP Working Group 7.3)*, H. Beilner and E. Gelenbe, Eds. Amsterdam, The Netherlands: North-Holland, 1977, pp. 75–93.

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

Nancy G. Leveson received the B.A. degree in mathematics, the M.S. degree in management, and the Ph.D. degree in computer science from the University of California, Los Angeles.

She has worked for IBM and is currently an Associate Professor of Computer Science at the University of California, Irvine. Her current interests are in software reliability, software safety, and software fault tolerance. She heads the Software Safety Project at UCI which is exploring a range of software engineering topics involved in specifying, designing, verifying, and assessing reliable and safe real-time software.

Dr. Leveson is a member of the Association for Computing Machinery, the IEEE Computer Society, and the System Safety Society.

Janice L. Stolzy received the B.S. degree in mathematics and applied science from the University of California, Riverside, in 1976, and the M.S. degree in computer science from the University of California, Los Angeles, in 1978.

She is currently working toward the Ph.D. degree in information and computer science at the University of California, Irvine. Her interests lie in the areas of software safety and reliability.

Ms. Stolzy is a member of the Association for Computing Machinery and the IEEE Computer Society.