Contents

6

Analyses Using Stochastic Reward Nets

LORRIE A. TOMEK
IBM Corporation, Research Triangle Park, North Carolina

KISHOR S. TRIVEDI
Duke University

ABSTRACT

In this chapter, we examine the power of stochastic reward nets (SRNs), a variant of stochastic Petri nets, to model fault tolerant software systems. The three types of software fault tolerance we examine are: N-version programming, recovery blocks, and N self-checking programming. SRNs allow each fault tolerance technique to be specified in a concise manner. Complex dependencies, such as common-mode versus separate failures and detected versus undetected failures, associated with these systems are incorporated into each SRN model without undue complication. Underlying an SRN is a Markov reward model. Each SRN is automatically converted into a Markov reward model from which steady-state, transient, cumulative transient, and sensitivity measures are easily obtained. We study several measures, including reliability, safety, and performance measures, which are of interest in the evaluation of fault tolerant software techniques. We parameterize our model to account for common-mode failures between variants using distributions from experimental data, and then provide numerical results using the stochastic Petri net package (SPNP) [Cia89].

Software Fault Tolerance, Edited by Lyu
© 1995 John Wiley & Sons Ltd

6.1 INTRODUCTION

In this chapter, we explore the capability to model various software fault tolerance techniques using a generalization of stochastic Petri nets (SPNs) called *stochastic reward nets (SRNs)*. SRNs, like stochastic Petri nets, allow a particular system to be expressed concisely during the design phase. Tools have been developed to automatically convert the net into its underlying (equivalent) Markov model [Cia89, Chi85, Cou91]. Although the underlying Markov model can be very large, efficient solution techniques have been developed to obtain steady-state, transient, cumulative, and sensitivity measures. Stochastic Petri nets (SPNs) have been used successfully to represent systems under design in order to study their performance [Ibe90], safety [Tom94], reliability [Tom91], and availability [Ibe89]. We demonstrate that SRNs are well-suited to the task of modeling fault tolerant software systems.

The software fault tolerance strategies we will consider are N-version programming, recovery blocks, and N self-checking programming. These techniques require the development of two or more software modules which are termed *variants*. The variants are executed sequentially in the case of recovery blocks, and in parallel in the case of N-version programming and N self-checking programming. Each strategy employs a different *decision mechanism* to diagnose variant output as correct or incorrect. In *N-version programming (NVP)*, the results of all variants are collected and a voter determines the system output [Avi85]. In *recovery blocks (RB)*, the result of each sequentially executed variant is processed by an acceptance test; this acceptance test either accepts the variant's result or the next variant is executed (until all variants have been executed) [Ran75, And81]. In *N self-checking programming (NSCP)*, there are two possible decision mechanisms [Lap90]. The first possibility is that each variant has its own acceptance test. The second possibility is that each pair of variants is compared using a comparison algorithm associated with the pair of variants. The system's output is given by the active variant (or variant pair) that diagnoses its own output as correct. The active status cycles between the variants (or variant pairs), so that if at least one variant (or variant pair) diagnoses its output to be correct, the system does produce an output. In each of the above possibilities, the acceptance test associated with each variant or the comparison algorithm associated with a pair of variants can be identical or can be independently derived.

Associated with these techniques are several dependencies that are important to the accurate study of fault tolerant software systems. Dependencies in fault tolerant systems are generally classified according to the source of the failure. Laprie [Lap90] differentiates failures by the following two important classifications:

First, with respect to the type of failures:

1. *Separate failures* result from independent faults.
2. *Common mode failures* result from related faults or from independent faults with similar errors.

Second, with respect to error detection:

1. *Detected failures* occur when no unacceptable output is delivered.
2. *Undetected failures* occur when incorrect output is delivered.

In the SRN models, we consider both separate and common-mode failures by careful definition of system parameters. The distinction between detected and undetected failures is accounted for by the structure of the SRN model. There are many sources of common mode failures. One source is design faults that occur during the specification and/or implementation phase where the initial specification, design, or code is use in the development of more than one variant. Another source of common mode failure is that a single input is shared by all variants within the execution of the fault tolerant software block. If all inputs are equally difficult, then the variant failure probabilities are independent. If some inputs are more difficult than others (as has been shown in measurements of experimental systems), then the variant failure probabilities are dependent random variables.

SRNs are well suited to express each software fault tolerance technique and account for these complexities. In addition, since SRNs are equal in modeling power to Markov reward models (MRMs), the wide range of measures available for MRMs can be easily obtained for SRNs. Using SRNs, we are able to numerically study safety, reliability, and performance of each software fault tolerance technique.

The chapter is organized as follows. In Section 6.2, we describe SRNs including the measures that can be readily obtained for SRNs. In Section 6.3, we describe three fault tolerant software techniques which we model using SRNs. The techniques considered are N-version programming, recovery blocks, and N self-checking programming. Section 6.4 discusses issues defined by software fault tolerant systems including detected versus undetected failures and common-mode versus separate failures. The SRN models of each software fault tolerant technique are then revisited to explore how these issues are incorporated into the model. In Section 6.5, we study numerical results for performance, reliability, and safety measures. The numerical study incorporates parameters measured in the multi-version software experiment of Lyu and He [Lyu93]. Section 6.6 summarizes the chapter.

6.2 INTRODUCTION TO STOCHASTIC REWARD NETS

Stochastic reward nets (SRNs) are a generalization of generalized stochastic Petri nets (GSPNs), which in turn are a generalization of stochastic Petri nets (SPNs). We first describe SPNs, then GSPNs, and then SRNs.

6.2.1 Stochastic Petri Nets (SPNs)

A Petri net (PN) is a bipartite directed graph with two disjoint sets called *places* and *transitions* [Pet81]. Directed arcs in the graph connect places to transitions (called *input arcs*) and transitions to places (called *output arcs*). Places may contain an integer number of entities called *tokens*. The state or condition of the system is associated with the presence or absence of tokens in various places in the net. The condition of the net may enable some transitions to fire. This *firing of a transition* is the removal of tokens from one or more places in the net and/or the arrival of tokens in one or more places in the net. The tokens are removed from places connected to the transition by an input arc; the tokens arrive in places connected to the transition by an output arc. A *marked Petri net* is obtained by associating tokens with places. The *marking* of a PN is the distribution of tokens in the places of the PN. A marking is represented by a vector $M = (\#(P_1), \#(P_2), \cdots, \#(P_n))$ where $\#(P_i)$ is the number of tokens in place i and n is the number of places in the net.

In a graphical representation of a PN, places are represented by circles, transitions are represented by bars and the tokens are represented by dots or integers in the circles (places). The *input places* of a transition are the set of places which connect to that transition through input arcs. Similarly, *output places* of a transition are those places to which output arcs are drawn from that transition.

A transition is considered *enabled* to fire in the current marking if each of its input places contains the number of tokens assigned to the input arc (called the *arc multiplicity*). The firing of a transition is an atomic action in which the designated number of tokens are removed from each input place of that transition and one or more tokens (as specified by the output arc multiplicity) are added to each output place of that transition, possibly resulting in a new marking of the PN.

If exponentially distributed *firing times* correspond with the transitions, the result is a *stochastic Petri net* [Flo91, Mol82, Nat80]. Allowing transitions to have either zero firing times (immediate transitions) or exponentially distributed firing times (timed transitions) gives rise to the *generalized stochastic Petri net (GSPN)* [Ajm84]. The transitions with exponentially distributed firing time are drawn as unfilled rectangles; immediate transitions are drawn as lines.

6.2.2 Generalized Stochastic Petri Nets (GSPNs)

Other extensions to SPNs in the development of GSPNs include the inhibitor arc. An *inhibitor arc* is an arc from a place to a transition that inhibits the firing of the transition when a token is present in the input place.

Each distinct marking of the PN constitutes a separate state of the PN. A marking is reachable from another marking if there exists a sequence of transition firings occur starting from the original marking that results in the new marking. The *reachability set (graph)* of a PN is the set (graph) of markings that are reachable from the other markings.

In any marking of the PN, a number of transitions may be simultaneously enabled. The tie between simultaneously enabled transitions can be broken by specifying priorities, by specifying probabilities, or by a race. *Priorities* are non-negative integers that permit a partial ordering of transitions. Whenever a transition with a priority δ is enabled, all transitions with priorities less than δ are inhibited from firing. If *probabilities* are assigned to the transitions, they are interpreted as weights of each transition. With some probability, any of the enabled transitions may be the first to fire; any enabled transition with positive probability may fire next. Immediate transitions which can be simultaneously enabled must have either priorities or probabilities assigned to avoid ambiguity in the net. For timed transitions, the decision as to which transition fires next can be decided by a *race*; the transition with the minimal delay prior to firing will fire next.

The markings of a GSPN are classified into two types: tangible and vanishing. A marking is *tangible* if the only transitions enabled (if any are enabled) are timed transitions. A marking is *vanishing* if one or more immediate transitions are enabled in the marking.

Ajmone Marsan et al [Ajm84] showed that a GSPN is equivalent in power to a continuous time Markov chain (CTMC). All CTMCs can be converted to an equivalent GSPN model; any GSPN model can be converted to an equivalent CTMC. The same techniques used for steady state, transient, cumulative, and sensitivity measures of a CTMC can be used to solve for the same measures of a GSPN (once the GSPN is converted to a CTMC.)

6.2.3 Stochastic Reward Nets (SRNs)

Stochastic reward nets (SRNs) are a superset of GSPNs [Cia89, Cia92]. SRNs substantially increase the modeling power of the GSPN by adding guard functions, marking dependent arc multiplicities, general transition priorities, and reward rates at the net level.

A *guard function* is a Boolean function associated with a transition [Cia89, Cia92]. Whenever the transition satisfies all the input and inhibitor conditions in a marking M, the guard is evaluated. The transition is considered enabled only if the guard function evaluates to true.

Marking dependent arc multiplicities allow either the number of tokens required for the transition to be enabled, or the number of tokens removed from the input place, or the number of tokens placed in an output place to be a function of the current marking of the PN. Such arcs are called *variable cardinality arcs*.

6.2.4 Measures

Stochastic Reward Nets (SRNs) provide the same modeling capability as Markov reward models (MRMs). A *Markov reward model* is a Markov chain with reward rates (real numbers) assigned to each state [How71]. A state of an SRN is actually a marking (labeled $(\#(P_1), \#(P_2), ..., \#(P_n))$ if there are n places in the net). We label the set of all possible markings that can be reached in the net as Ω. These markings are subdivided into tangible markings $\Omega_{\mathcal{T}}$ and vanishing markings $\Omega_{\mathcal{V}}$. For each tangible marking i in $\Omega_{\mathcal{T}}$, a reward rate r_i is assigned. This reward is determined by examining the overall measures to be obtained. In Section 6.5, we examine the reward definitions needed to generate reliability, safety, and performance measures.

Several measures are obtained using Markov reward models. These include the expected reward rate both in steady state and at a given time, the expected accumulated reward until either absorption or a given time, and the distribution of accumulated reward either until absorption or a given time.

The *expected reward rate in steady state* can be computed using the steady state probability of being in each marking i for all $i \in \Omega_{\mathcal{T}}$. For steady state distribution π_i, the expected reward rate is given by

$$E[\mathcal{R}] = \sum_{i \in \Omega_{\mathcal{T}}} r_i \pi_i$$

The *expected reward rate at time t* can be computed by using the transient probability of being in each marking $i \in \Omega_{\mathcal{T}}$, labeled $p_i(t)$. The expected reward rate at time t is then given by

$$E[\mathcal{R}(t)] = \sum_{i \in \Omega_{\mathcal{T}}} r_i p_i(t)$$

The *distribution of reward rate at time t* $P\{\mathcal{R}(t) \leq x\}$ is given by

$$P\{\mathcal{R}(t) \leq x\} = \sum_{r_i \leq x, i \in \Omega_{\mathcal{T}}} p_i(t)$$

The *accumulated reward* in $(0, t]$, $Y(t)$, is defined as $Y(t) = \int_0^t \mathcal{R}(u)du$. The *expected accumulated reward in (0,t]* can be computed as

$$E[Y(t)] = E[\int_0^t \mathcal{R}(u)du] = \int_0^t E[\mathcal{R}(u)]du = \sum_{i \in \Omega_{\mathcal{T}}} r_i \int_0^t p_i(u)du$$

The *expected accumulated reward until absorption*, labeled $E[Y(\infty)]$, can be computed as

$$E[Y(\infty)] = \sum_{i \in \Omega_{\mathcal{T}}} r_i \int_0^\infty p_i(u)du$$

The *distribution of accumulated reward* is a measure of considerable interest. The distribution of accumulated reward until absorption is defined as $\mathcal{Y}(x) = P\{Y(\infty) \leq x\}$. This distribution was first studied by Beaudry [Bea78] for an underlying CTMC model with strictly positive reward rates, and was extended by Ciardo et al. [Cia90] to allow an underlying semi-Markov model with non-negative reward rates.

Sensitivities (derivatives) of the above measures with respect to one or more model parameters can also be obtained [Cia92].

6.3 FAULT TOLERANT SOFTWARE MODELS

Next, we develop SRN models for recovery blocks, N-version programming blocks, and N self-checking programming blocks. In this section, we focus on the basic model. We revisit each model in Section 6.4 to discuss issues such as detected versus undetected failures and common-mode versus separate failures.

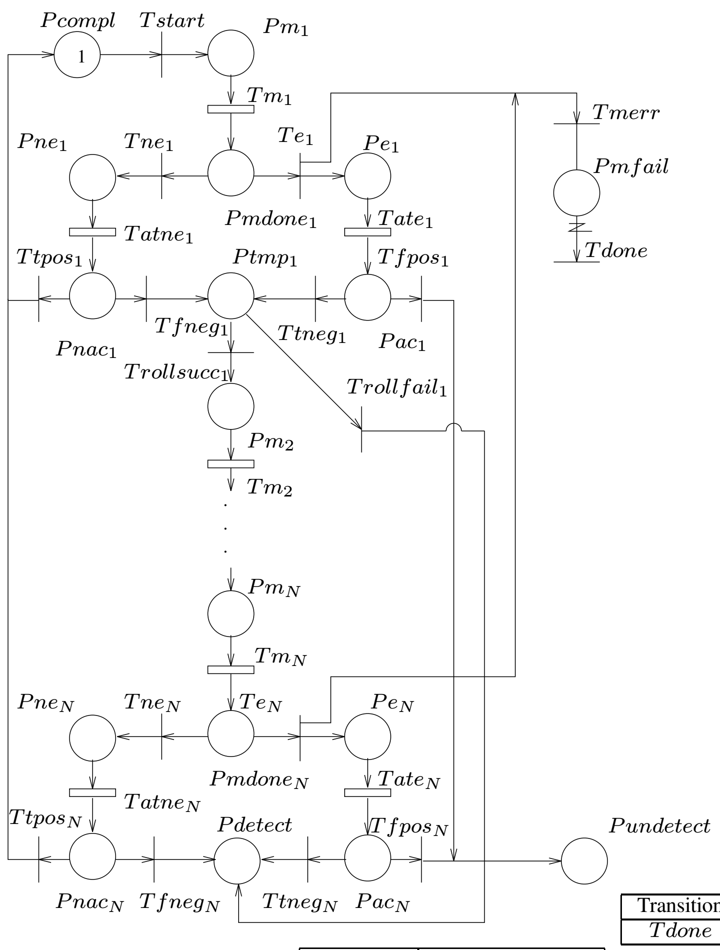
6.3.1 Recovery Blocks

A recovery block (RB) consists of two or more variants and a single acceptance test (AT). The variants are ordered with the first variant called the *primary* and the others called *alternates*. The primary and the alternate variants are independently developed, based on different algorithms and implemented by different programmers. For each input to the recovery block, the primary is executed first and its output is evaluated using the AT. If the AT fails to accept the output, a rollback recovery is attempted; this process is repeated for each alternate variant in succession until either (1) a variant produces an output that is accepted by the AT, (2) the rollback recovery fails, or (3) all variants execute without satisfying the AT. In the last case, the RB is said to have failed on this input dataset. The pseudocode for a RB with N variants (a primary and $N - 1$ alternates) is shown below.

```
ensure acceptance test
    by primary variant (#1)
    else by alternate variant (#2)
    else by alternate variant (#3)
    ...
    else by alternate variant $(\#N)$
else error
```

The parameters required for a recovery block model are difficult to obtain. Following the categories of Pucci [Puc90, Puc92], events in the recovery blocks are classified into the following four types of events.

1. Variant i produces correct output which the AT accepts.
2. Variant i produces correct output which the AT rejects.



Transition	Trans. Priority
$Tdone$	HIGH

Transition	Guard Function
$Tdone$	$\#(Pcompl) > 0$

Arc	Arc Multiplicity
$Pmfail \rightarrow Tdone$	$max(\#(Pmfail), 1)$

Figure 6.1 SRN model of a software recovery block (RB)

3. Variant i produces incorrect output which the AT rejects.
4. Variant i produces incorrect output which the AT accepts.

In addition, we will consider both successful and unsuccessful rollback recovery attempts following a negative AT diagnosis.

The SRN model of a recovery block is shown in Figure 6.1. The net is nearly self-explanatory. Place $Pcompl$ is the starting point of the RB. The firing of transition $Tstart$, which places a token in place Pm_1, indicates that the recovery block has begun executing the next (or first in this case) dataset. A token in place Pm_1 indicates that the primary variant in the recovery block has begun execution on the current dataset. The firing of transition Tm_1 corresponds to the completion of the execution of the primary variant. Transitions Tne_1

and Te_1 correspond to the events that the output produced by the variant are correct and incorrect respectively. Transition Tne_1 moves the token from place $Pmdone_1$ to place Pne_1 indicating that the variant produced a correct output. Transition Te_1 moves the token from place $Pmdone_1$ to both places Pe_1 and $Pmfail$. A token in place Pe_1 indicates that the first variant produced an incorrect output. Place $Pmfail$ counts the number of variants producing an incorrect result on the current dataset; this is needed to represent common-mode failures. Transition $Tatne_1$ represents the execution of the AT after the variant produces a correct output. The immediate transitions $Ttpos_1$ and $Tfneg_1$, which correspond to a correct positive diagnosis by the AT and a false negative diagnosis by the AT respectively, are then enabled. Transition $Tate_1$ represents the execution of the AT after the variant produces an incorrect output. The immediate transitions $Ttneg_1$ and $Tfpos_1$, which correspond to correct negative diagnosis by the AT and a false positive diagnosis by the AT respectively, are then enabled. A false positive AT diagnosis causes the token to be moved to place $Pundetect$ indicating an undetected block failure. A true positive AT diagnosis causes the token to be moved to place $Pcompl$ indicating the block has completed execution on the current dataset. The block then begins operation on the next dataset.

If an error is discovered, represented by the firing of either $Tfneg_1$ and $Ttneg_1$, the system initiates a rollback recovery action. Transition $Trollsucc_1$ represents a successful rollback. Transition $Trollfail_1$ represents an unsuccessful rollback resulting in an RB failure. The output arc from $Trollsucc_1$ leads to Pm_2, the starting place of the first alternate variant, while the output arc from $Trollfail_1$ leads to $Pdetect$ which represents a detected RB failure. The alternate variants are similarly modeled by the other places and transitions indexed from 2 to N. The structure of the last variant is slightly different, since the failure of the last variant automatically results in a detected system failure. Thus, the output arcs from transitions $Tfneg_N$ and $Ttneg_N$ lead to place $Pdetect$.

When the recovery block completes (the token is returned to place Pm_1) then transition $Tdone$ fires and all tokens are removed from place $Pmfail$ for the next execution of the recovery block.

6.3.2 N-Version Programming

In N-version programming (NVP), all variants operate on the same input in parallel. The results of all variants are collected and a voter determines the system output [Avi85]. The reliability of this mechanism is dependent upon individual variant results. If more than half of the variants produce results that are within the required error tolerance, the prevailing result is declared correct. If half or less than half of the variants produce the same result, the result is declared incorrect. In this case, no output would be released by the block.

In Figure 6.2, an SRN model of an N-version programming system is shown. Initially, a single token is in place $Pcompl$. The software block begins operating on the next input immediately with the firing of transition $Tstart$. This transition places one token in each of the places Pm_1, Pm_2, ..., Pm_N, representing the fact that each variant begins operation on the provided input. Transitions Tm_i for $i \in 1, 2, ..., N$ represents the completion of execution of variant i. When all variants have completed, place $Pvote$ will contain N tokens, enabling transition $Tvote$. Transition $Tvote$ represents the execution of the voting mechanism. When voting is complete, a single token is moved to place $Pdiag$ where the voting result is diagnosed. If less than half of the variants produced correct output, then the voting result can be either a true negative, represented by the firing of transition $Ttneg$, or a false positive,

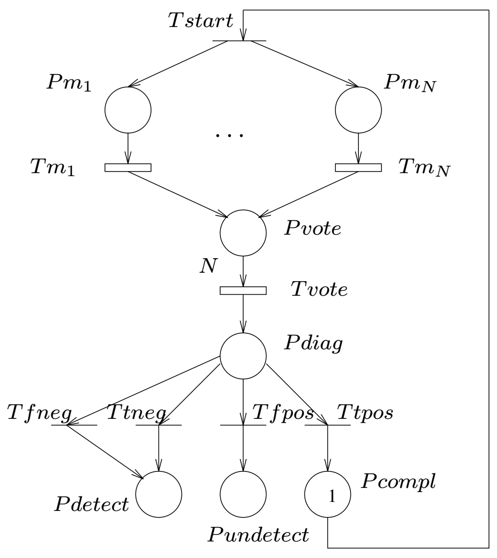


Figure 6.2 SRN model of N-version programming

represented by the firing of transition $Tfpos$. If at least half of the variants produced correct output, then the voting result can be either a true positive, represented by the firing of transition $Ttpos$, or a false negative, represented by the firing of transition $Tfneg$. If the voting result is negative, either transition $Ttneg$ or transition $Tfneg$ fire, moving the token to place $Pdetect$. This represents a detected error. If the voting result is a false positive, transition $Tfpos$ fires, moving the token to place $Pundetect$, indicating an undetected error. If the voting result is a true positive, transition $Ttpos$ fires, moving the token to place $Pcompl$, indicating the software block as successfully completed execution on the current dataset. Once the token is returned to place $Pcompl$, the N-version programming block begins operating on the next input.

6.3.3 N Self-Checking Programming

In N *self-checking programming (NSCP)*, all variants operate in parallel on the same input. There are two possible decision mechanisms [Lap90] using this technique. The first possibility is that each variant has its own acceptance test. The second possibility is that each pair of variants is compared using a comparison algorithm associated with the pair. In each of the above possibilities, the acceptance test associated with each variant or comparison algorithm associated with a pair of variants can be identical or independently derived. One variant is always considered to be the active variant. If the active variant produces an output that is diagnosed as correct, then that output is the block output. If the active variant diagnoses its output to be incorrect, then the status of active variant is passed to one of the alternate variants.

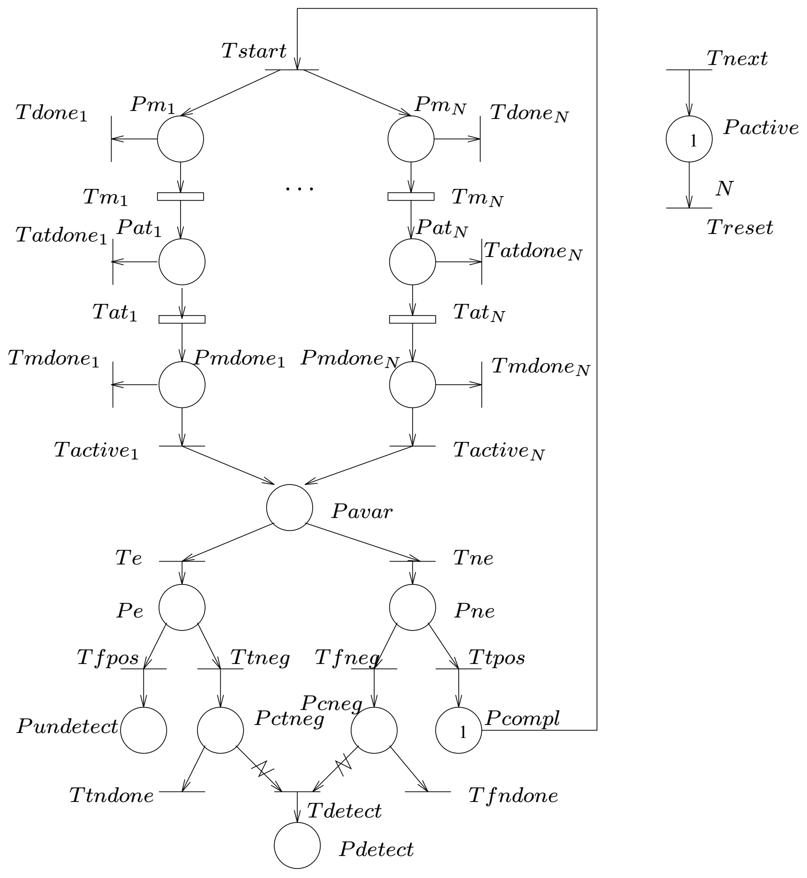


Figure 6.3 SRN model of N self-checking programming using an acceptance test for error diagnosis

6.3.3.1 NSCP WITH ACCEPTANCE TEST

First, we study the case where each variant diagnoses the correctness or incorrectness of its output using an acceptance test as shown in Figure 6.3. The transition priorities, guard functions, and arc multiplicities associated with this model are given in Figure 6.4. Initially, there is a single token in both places $Pcompl$ and $Pactive$. The software block begins operating on the next input immediately with the firing of transition $Tstart$. This transition places one token in each of places Pm_1, Pm_2, ..., Pm_N, representing the fact that each variant begins operation on the provided input. Transitions Tm_i for $i \in 1, 2, ..., N$ represents the execution of each variant i. As each variant completes execution, a token is placed in Pat_i (for variant i). The self-checking procedure (acceptance test execution) is modeled by transition Tat_i. When the acceptance test completes, the token is moved from place Pat_i to place $Pmdone_i$ (for variant i). Place $Pactive$ contains the number of tokens representing the number of the active variant (a number between 1 and N). When the active variant completes its acceptance test, then a token is in place $Pmdone_i$ enabling transition $Tactive_i$, where i is the number of tokens in place $Pactive$. The firing of transition $Tactive_i$ moves the token to place $Pavar$.

Transition	Trans. Priority
$Tnext$	LOW
$Treset$	HIGH
$Tstart$	LOW
$Tactive_i$, for $i \in [1, N]$	HIGH
$Tdetect$	HIGH
$Tdone_i$, for $i \in [1, N]$	HIGH
$Tatdone_i$, for $i \in [1, N]$	HIGH
$Tmdone_i$, for $i \in [1, N]$	HIGH
$Ttndone$	HIGH
$Tfndone$	HIGH

Transition	Guard Function
$Tnext$	$\#(Pm_i) + \#(Pat_i) + \#(Pmdone_i) + \#(Pavar) + \#(Pe)+$ $\#(Pne) + \#(Pdetect) + \#(Pundetect) + \#(Pcompl) == 0$ where $i = \#(Pactive)$
$Treset$	$\#(Pactive) > N$
$Tactive_i$, for $i \in [1, N]$	$\#(Pactive) == i$
$Tdetect$	$\#(Pctneg) + \#(Pcfneg) == N$
$Tdone_i$, for $i \in [1, N]$	$\#(Pcompl) + \#(Pdetect) + \#(Pundetect) > 0$
$Tatdone_i$, for $i \in [1, N]$	$\#(Pcompl) + \#(Pdetect) + \#(Pundetect) > 0$
$Tmdone_i$, for $i \in [1, N]$	$\#(Pcompl) + \#(Pdetect) + \#(Pundetect) > 0$
$Tfndone$	$\#(Pcompl) + \#(Pdetect) + \#(Pundetect) > 0$
$Ttndone$	$\#(Pcompl) + \#(Pdetect) + \#(Pundetect) > 0$

Arc	Arc Multiplicity
$Pctneg \to Tdetect$	$\#(Pctneg)$
$Pcfneg \to Tdetect$	$\#(Pcfneg)$
$Pctneg \to Ttndone$	$max(\#(Pctneg), 1)$
$Pcfneg \to Tfndone$	$max(\#(Pcfneg), 1)$

Figure 6.4 Function definitions for SRN model of N self-checking programming using an acceptance test for error diagnosis

This enables transitions Te, representing the fact that the active variant has produced incorrect output, and transition Tne, representing the fact that the variant has produced correct output. If the variant produced incorrect output, the firing of transition Te moves the token to place Pe. The diagnosis of incorrect output can be either a false positive diagnoses or a true negative diagnosis represented by transitions $Tfpos$ and $Ttneg$ respectively. If a false positive diagnosis occurs, the token is moved to place $Pundetect$, representing an undetected error. If a true negative diagnosis is detected, the token is moved to place $Pctneg$. Place $Pctneg$ counts the number of incorrect variant outputs which are diagnosed as true negative. The tokens remain in place $Pctneg$ until either all variants are diagnosed as incorrect or the block completes execution without detecting a failure.

If the variant produced incorrect output, transition Tne fires moving the token from place $Pavar$ to place Pne. The diagnosis of correct output can be either a false negative or a true positive represented by transitions $Tfneg$ and $Ttpos$. Transition $Tfneg$ moves the token from place Pne to place $Pcfpos$. Place $Pcfpos$ counts the number of correct variant outputs which are diagnosed as false negative. The tokens remain in place $Pcfpos$ until either all variants are diagnosed as incorrect or the active variant pair diagnoses its output to be correct. If the sum of the number of tokens in place $Pctneg$ and $Pcfpos$ is equal to N, all variants have been diagnosed as incorrect. This enables transition $Tdetect$, which removes all tokens from places $Pctneg$ and $Pcfpos$ and places a single token in place $Pdetect$; this represents a

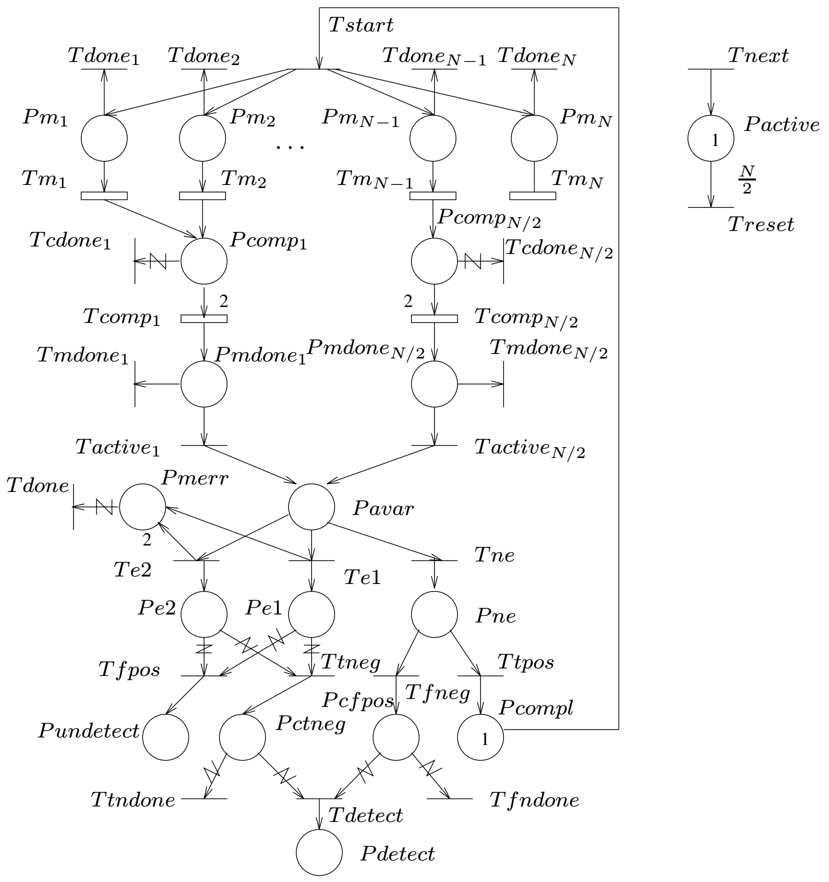


Figure 6.5 SRN model of N self-checking programming using a comparison test for error diagnosis

detected failure. If the diagnosis of the correct output is a true positive, transition $Ttpos$ fires, moving the token from place Pne to place $Pcompl$.

A token in place $Pcompl$ satisfies the guard function for transitions $Tdone_i$, $Tatdone_i$, and $Tmdone_i$ for $i \in [1, N]$ and transitions $Ttndone$ and $Tfndone$. All tokens in these places are removed from the net, effectively resetting the net to its initial state. After this reset, the N self-checking programming block begins execution of the next input.

6.3.3.2 NSCP WITH COMPARISON ALGORITHM

Now, we study the case where the outputs of pairs of variants are diagnosed by a comparison algorithm as shown in Figure 6.5. The transition priority, guard, and arc multiplicity functions are given in Figure 6.6. This model is very similar to the previously discussed NSCP with acceptance test model. Initially, there is a single token in both places $Pcompl$ and $Pactive$. The software block begins operating on the next input immediately with the firing of transition $Tstart$. This transition places one token in each of places Pm_1, Pm_2, ...,

Transition	Trans. Priority
$Tnext$	LOW
$Treset$	HIGH
$Tstart$	LOW
$Tactive_i$, for $i \in [1, N/2]$	HIGH
$Tdetect$	HIGH
$Tdone_i$, for $i \in [1, N]$	HIGH
$Tcdone_i$, for $i \in [1, N/2]$	HIGH
$Tmdone_i$, for $i \in [1, N/2]$	HIGH
$Ttndone$	HIGH
$Tfndone$	HIGH
$Tedone$	HIGH

Transition	Guard Function
$Tnext$	$\#(Pm_{i\times 2}) + \#(Pcomp_i) + \#(Pmdone_i) + \#(Pavar) + \#(Pe)+$ $\#(Pne) + \#(Pdetect) + \#(Pundetect) + \#(Pcompl) == 0$ where $i = \#(Pactive)$
$Treset$	$\#(Pactive) > N/2$
$Tactive_i$, for $i \in [1, N/2]$	$\#(Pactive) == i$
$Tfpos$	$\#(Pe1) + \#(Pe2) > 0$
$Ttneg$	$\#(Pe1) + \#(Pe2) > 0$
$Tdetect$	$\#(Pctneg) + \#(Pcfneg) == N/2$
$Tdone_i$, for $i \in [1, N]$	$\#(Pcompl) + \#(Pdetect) + \#(Pundetect) > 0$
$Tcdone_i$, for $i \in [1, N/2]$	$\#(Pcompl) + \#(Pdetect) + \#(Pundetect) > 0$
$Tmdone_i$, for $i \in [1, N/2]$	$\#(Pcompl) + \#(Pdetect) + \#(Pundetect) > 0$
$Ttndone$	$\#(Pcompl) + \#(Pdetect) + \#(Pundetect) > 0$
$Tfndone$	$\#(Pcompl) + \#(Pdetect) + \#(Pundetect) > 0$
$Tedone$	$\#(Pcompl) + \#(Pdetect) + \#(Pundetect) > 0$

Arc	Arc Multiplicity
$Pctneg \to Tdetect$	$\#(Pctneg)$
$Pcfneg \to Tdetect$	$\#(Pcfneg)$
$Pctneg \to Ttndone$	$max(\#(Pctneg), 1)$
$Pcfneg \to Tfndone$	$max(\#(Pcfneg), 1)$
$Pcomp_i \to Tcdone_i$, for $i \in [1, N/2]$	$max(\#(Pcomp_i), 1)$
$Pei \to Tfpos$, for $i = 1, 2$	$\#(Pe_i)$
$Pei \to Ttneg$, for $i = 1, 2$	$\#(Pe_i)$

Figure 6.6 Function definitions for SRN model of N self-checking programming using a comparison test for error diagnosis

Pm_N, representing the fact that each variant begins operation on the provided input. Transitions Tm_i for $i \in 1, 2, ..., N$ represents the execution of each variant i. As each variant completes execution, a token is placed in $Tcomp_{\lceil i/2 \rceil}$ (for variant i). The self-checking procedure (comparison algorithm execution) is modeled by transition $Tcomp_i$. This transition is enabled only when both variants in a pair have completed execution (when two tokens are in place $Pcomp_i$). When the comparison test completes, the token is moved from place $Pcomp_i$ to place $Pmdone_i$ (for variant pair i). Place $Pactive$ contains the number of tokens representing the number of the active variant pair (a number between 1 and $N/2$). When the active variant pair completes its comparison algorithm, a token is in place $Pmdone_i$, enabling transition $Tactive_i$, where i is the number of tokens in place $Pactive$. The firing of transition $Tactive_i$ moves the token to place $Pavar$. This enables transitions $Te1$, $Te2$, and Tne. The firing of transition $Te1$ means one of the two variants produced an incorrect output and therefore the comparison test result should be negative. The firing of transition $Te2$ means both of the two variants produced an incorrect output and therefore the comparison test result should be negative. The firing of transition Tne means neither of the two variants produced an incorrect output and therefore the comparison test result should be positive. The firing of transition $Te1$ moves a token to place $Pe1$ and to place $Pmerr$. The firing of transition $Te2$ moves a token to place $Pe2$ and two tokens to place $Pmerr$. The number of tokens in place $Pmerr$ represents the number of variants that produced an incorrect output on the current dataset. A token in place $Pe1$, indicating one of the variant pair produced an incorrect output. A token in place $Pe2$ indicates that both of the variants in the pair produced an incorrect output. A token in either $Pe1$ or $Pe2$ enables transitions $Tfpos$ and $Ttneg$, which indicate a false positive diagnoses and a true negative diagnoses respectively. A false positive diagnoses causes the token to move to place $Pundetect$, indicating an undetected error. If a true negative diagnosis is detected, the token is moved to place $Pctneg$. Place $Pctneg$ counts the number of incorrect variant pair outputs which are diagnosed as true negative. The tokens remain in place $Pctneg$ until either all variants pairs are diagnosed as incorrect or the active variant pair diagnoses its output to be correct.

If both variants in the pair produced incorrect output, transition Tne fires, moving the token from place $Pavar$ to place Pne. The diagnosis of correct output can be either a false negative or a true positive represented by transitions $Tfneg$ and $Ttpos$. Transition $Tfneg$ moves the token from place Pne to place $Pcfneg$. Place $Pcfneg$ counts the number of correct variant pair outputs diagnosed as false negative. The tokens remain in place $Pcfneg$ until either all variants are diagnosed as incorrect or the active variant diagnoses its output as correct. If the sum of the number of tokens in place $Pctneg$ and $Pcfpos$ is equal to $N/2$, all variant pairs have been diagnosed as incorrect. This enables transition $Tdetect$, which removes all tokens from places $Pctneg$ and $Pcfpos$ and places a single token in place $Pdetect$; this represents a detected failure. If the diagnosis of the correct output is a true positive, transition $Ttpos$ fires, moving the token from place Pne to place $Pcompl$.

A token in place $Pcompl$ satisfies the guard function for transitions $Tdone_i$ for $i \in [1, N]$, transitions $Tcdone_i$ and $Tmdone_i$ for $i \in [1, N/2]$, and transitions $Ttndone$, $Tfndone$, and $Tedone$. All tokens in these places are removed from the net, effectively resetting the net to its initial state. After this reset, the N self-checking programming block begins execution of the next input.

6.4 DEPENDENCIES IN THE SRN MODELS

Dependencies in fault tolerant systems are generally classified according to the source of the failure. Laprie [Lap90] classified failures in fault tolerant software systems using two criteria. First, failures are classified as either *separate* or *common-mode*. Sources of common-mode failures include *design faults* from shared specification or implementation, *similar errors* from independent faults, and the inherent difficulty of *shared input*. Next, failures can either be *detected* or *undetected*. It is most important in the development of a model to account for these dependencies. In our SRN models, these dependencies are accounted for by the structure of the model, and by judicious definition of the immediate transition probabilities.

6.4.1 Detected versus Undetected Failures

First, consider the distinction between detected and undetected failures. In the previous section, we developed the SRN model of each software fault tolerance technique which included places $Pdetect$, for detected failures, and $Pundetect$, for undetected failures. Defining separate places for detected and undetected failures, rather a single place (to indicate any type of failure), allows numerical study of several additional measures of interest.

Safety measures include both steady state and transient measures. A steady state measure of interest is the probability the system will eventually fail due to an unsafe failure. An unsafe failure is indicated by the existence of a token in place $Pundetect$. A transient measure of interest is $S(t)$, the safety distribution, defined to be the probability the system does not enter an unsafe state by time t.

Reliability measures can be obtained by considering block failure as the existence of a token in either place $Pdetect$ or $Pundetect$. Mean time to failure is a cumulative measure of the expected time until a token arrives in either place $Pdetect$ or $Pundetect$. The transient reliability function, $R(t)$, is the probability that there are no tokens in either place $Pdetect$ or $Pundetect$ at time t.

6.4.2 Common-Mode versus Separate Failures

Next, consider the distinction between separate and common-mode failures. Separate failures result from independent faults with distinct errors. Common-mode failures result from related faults or independent faults subject to similar errors. Measurements have shown that software variants do not exhibit separate failures. Measurements provide a probability mass function $p_N(\cdot)$ where $p_N(i)$ is the probability that i of the N variants produce incorrect output. If all variant failures are separate, then $p_N(\cdot)$ is a binomial probability mass function.

Common-mode variant failures can be easily accounted for in the SRN model by carefully structuring the model to retain tokens in places which provide needed information; state dependent transition probabilities can then be defined to use this information. The state information needed to model common-mode variant failures includes n_{vdone}, the number of variants in the program block which have completed execution, and n_{fail}, the number of the variants which have completed and produced incorrect results.

In addition, to simplify the probability functions needed in the SRN models, we include in the state information $n_{totfail}$, the number of variants out of N producing incorrect output. This variable is computed using probability mass function $p_N(\cdot)$ each time a new dataset begins processing.

Using the assumption that variants are stochastically identical, we can compute several probabilities of interest in fault tolerant software systems. The probability a variant produces incorrect output is given by

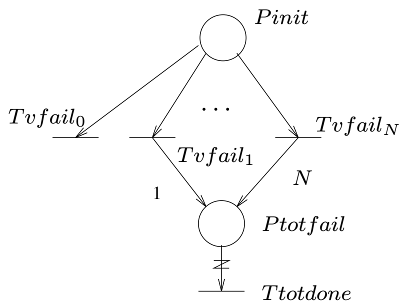
$$prob(\text{variant failure}) = \frac{n_{totfail} - n_{fail}}{N - n_{vdone}}$$

The probability that less than $N/2$ variants produce incorrect output is given by

$$prob(\text{no. variants fail } < N/2) = 1_{n_{totfail} < N/2}$$

where 1_x is an indicator function which evaluates to 1 if x is true and 0 otherwise. If we consider a pair of variants, we can compute the probabilities that one, both, or neither of the pair of variants fail. The probability both variants produce correct output is

$$prob(\text{neither variant fails}) = \left(1.0 - \frac{n_{totfail} - n_{fail}}{N - n_{vdone}}\right) \times \left(1.0 - \frac{n_{totfail} - n_{fail}}{N - (n_{vdone} + 1)}\right)$$



Transition	Trans. Priority
$Ttotdone$	HIGH

Transition	Guard Function
$Ttotdone$	$\#(Pcompl) > 0$

Arc	Arc Multiplicity
$Ptotfail \rightarrow Ttotdone$	$max(\#(Ptotfail), 1)$

Figure 6.7 Subnet added to SRN models for common-mode failures

The probability both variants produce incorrect output is

$$prob(\text{both variants fail}) = \frac{n_{totfail} - n_{fail}}{N - n_{vdone}} \times \frac{n_{totfail} - (n_{fail} + 1)}{N - (n_{vdone} + 1)}$$

The probability that one of the two variants produce incorrect output and the other produces correct output is

$$prob(\text{one of two variants fail}) = 1.0 - (prob(\text{neither variant fails})$$

$$+prob(\text{both variants fail}))$$

6.4.3 SRN Models with Common-Mode and Separate Failures

Figure 6.7 shows a subnet that is added to each previously described SRN model to simplify the incorporation of common-mode failures. A token arrives in transition $Pinit$ as a result of the firing of transition $Tstart$. Transition $Tstart$ (which is part of each previously developed SRN model) is modified to include another output arc associated with place $Pinit$. When a token arrives in place $Pinit$, the software block is ready to operate on a new dataset. Transitions $Tvfail_0$, $Tvfail_1$, ... $Tvfail_N$ become enabled. The probability associated with each transition $Tvfail_i$ is $p_N(i)$, the probability that i out of the N variants produce an incorrect result. The firing of transition $Tvfail_i$ causes i tokens to be placed in $Ptotfail$. Place $Ptotfail$ represents the number of variants that will produce incorrect output on the current dataset. The number of tokens in place $Ptotfail$ is used to determine the variant failure probabilities in each SRN model. This is discussed in detail for each SRN model in the next section. When the software block completes execution on the current dataset (a token is moved to place $Pcompl$), all tokens in place $Ptotfail$ are removed by the firing of transition $Ttotdone$. This resets this subnet prior to the arrival of the next dataset.

6.4.3.1 RECOVERY BLOCK SRN

In the SRN model of the RB scheme, transitions Tne_i and Te_i are immediate transitions representing the correctness or incorrectness of variant i. These transitions are enabled only after $i-1$ variants have completed execution. Of these $i-1$ variants, $\#(Pmfail)$ produced incorrect results. The immediate transition probabilities for Te_i and Tne_i incorporating common-mode failures are given by

$$\begin{aligned} prob(Te_i) &= \frac{n_{totfail} - n_{fail}}{N - n_{vdone}} \\ &= \frac{\#(Ptotfail) - \#(Pmfail)}{N - (i-1)} \end{aligned}$$

and

$$prob(Tne_i) = 1 - prob(Te_i)$$

6.4.3.2 N-VERSION PROGRAMMING SRN

In the SRN model of the NVP scheme, out of the N available variants $\#(Ptotfail)$ variants produce incorrect results. The voter diagnosis is dependent on the number of the N variants producing incorrect output. If less than half of the variants produce correct output (that is if $\#(Ptotfail) < N/2$), then the vote should be positive. However, similar errors may cause the voter to diagnose a false positive when less than half of the variants produce a correct output. In addition, the implementation of the voter may be incorrect (e.g. the voter's variant error tolerance may be too small) and the voter may diagnose a false negative even though more than half of the variants produced a correct output.

$$\begin{aligned} prob(Tfpos) &= prob(\text{at least half of variants were incorrect}) \times \\ &\quad\; prob(\text{diagnosis on incorrect input is positive}) \\ prob(Ttneg) &= prob(\text{at least half of variants were incorrect}) \times \\ &\quad\; (1.0 - prob(\text{diagnosis on incorrect input is positive})) \\ prob(Tfneg) &= prob(\text{less than half of variants were incorrect}) \times \\ &\quad\; prob(\text{diagnosis on correct input is negative}) \\ prob(Ttpos) &= prob(\text{less than half of variants were incorrect}) \times \\ &\quad\; (1.0 - prob(\text{diagnosis on correct input is negative})) \end{aligned}$$

The variant probabilities used in the above equations are given by

$$prob(\text{at least half of variants are incorrect}) = 1_{\#(Ptotfail)>=N/2}$$

$$prob(\text{less than half of variants are incorrect}) = 1_{\#(Ptotfail)<N/2}$$

6.4.3.3 N SELF-CHECKING PROGRAMMING WITH ACCEPTANCE TEST SRN MODEL

In the SRN model of the NSCP with acceptance test, a token in place $Pavar$ enables transitions Te, which indicates an incorrect variant result, and transition Tne, which indicates a correct variant result. The number of previously completed and diagnosed variants is

$\#(Pctneg) + \#(Pcfneg)$. The number of these variants which produced incorrect results is $\#(Pctneg)$. The probability that the variant represented by a token in place $Pavar$ produces an incorrect result is

$$\begin{aligned} prob(Te) &= \frac{n_{totfail} - n_{fail}}{N - n_{vdone}} \\ &= \frac{\#(Ptotfail) - \#(Pctneg)}{N - (\#(Pctneg) + \#(Pcfneg))} \end{aligned}$$

and

$$prob(Tne) = 1 - prob(Te)$$

6.4.3.4 N SELF-CHECKING PROGRAMMING WITH COMPARISON ALGORITHM SRN MODEL

Similarly, in the SRN model of the NSCP with comparison tests, a token in place $Pavar$ enables transitions $Te1$, $Te2$, and Tne. The firing of transition $Te1$ indicates that one variant in the pair produced an incorrect output while the other variant produced a correct output. The firing of transition $Te2$ indicates that both variants in the pair produced incorrect output. The firing of transition Tne indicates that both of the variants in the pair produced correct output.

The number of variant pairs that have completed execution and diagnosis is given by $\#(Pctneg) + \#(Pcfneg)$; the number of variants that have completed execution is therefore $2 \times (\#(Pctneg) + \#(Pcfpos))$. Of the variant pairs that have completed, the number of pairs where at least one variant did not produce correct output is given by $\#(Pmerr)$.

The probability neither variant produces incorrect output is

$$\begin{aligned} prob(Tne) &= \left(1.0 - \frac{n_{totfail} - n_{fail}}{N - n_{vdone}}\right) \times \left(1.0 - \frac{n_{totfail} - n_{fail}}{N - (n_{vdone} + 1)}\right) \\ &= \left(1.0 - \frac{\#(Ptotfail) - \#(Pmerr)}{N - 2 \times (\#(Pctneg) + \#(Pcfneg))}\right) \times \\ &\quad \left(1.0 - \frac{\#(Ptotfail) - \#(Pmerr)}{N - 2 \times (\#(Pctneg) + \#(Pcfneg)) - 1}\right) \end{aligned}$$

The probability both variants produce incorrect output is

$$\begin{aligned} prob(Te2) &= \frac{n_{totfail} - n_{fail}}{N - n_{vdone}} \times \frac{n_{totfail} - (n_{fail} + 1)}{N - (n_{vdone} + 1)} \\ &= \frac{\#(Ptotfail) - \#(Pmerr)}{N - 2 \times (\#(Pctneg) + \#(Pcfneg))} \times \\ &\quad \frac{\#(Ptotfail) - \#(Pmerr) - 1}{N - 2 \times (\#(Pctneg) + \#(Pcfneg)) - 1} \end{aligned}$$

The probability that one of the two variants produce incorrect output and the other produces correct output is

$$prob(Te1) = 1.0 - prob(Tne) - prob(Te2)$$

Table 6.1 Error characteristics for $N = 1$ variant case

Category	Obs. Frequency	Computed Probability
No Errors	0.8549	0.8549
One Error	0.1451	0.1451

Table 6.2 Error characteristics for $N = 2$ variant case

Category	Obs. Frequency	Computed Probability
No Errors	0.8053	0.791833
One Error	0.1691	0.196034
Two Errors	0.0256	0.112133

6.5 NUMERICAL RESULTS

In this section, we analyze the SRN for each software fault tolerant scheme. We compute steady state, transient, cumulative, and sensitivity measures for safety, reliability, and performance using the same SRN definitions. We vary parameters such as the number of variants N, variant execution rates, decider execution rates, as well as the distribution of failed variants for separate and common-mode failures. The flexibility of the SRN model makes this analysis quite easy.

6.5.1 Parameterizing the Model

In order to obtain numerical results, we need to select parameter values for the variant and decider execution rates, the decider failure probabilities, and the distribution of common-mode failures. We choose realistic values for the execution rates, as well as the decider failure probabilities. The common-mode failure distribution is obtained from experimental data obtained by Lyu and He [Lyu93]. This experimental data was also used in reliability analysis of fault tolerant software systems by Dugan and Lyu [Dug93b, Dug93a].

The execution rate of each variant is chosen to be $\mu = 1.0$. The execution rate of the AT is $\mu_{at} = 100 \times \mu$. The execution rate of the voter is $\mu_{voter} = 10 \times \mu$. The execution rate of the comparison algorithm is $\mu_{comp} = 50 \times \mu$.

The results of the experiment by Lyu and He are shown in Tables 6.1, 6.2, 6.3, and 6.4 for $N = 1, 2, 3$, and 4 respectively. These results are used to account for common-mode variant failure. In each table, the number of variant errors (category), the observed frequency (or

Table 6.3 Error characterists for the $N = 3$ variant case

Category	Obs. Frequency	Computed Probability
No Errors	0.7426	0.74532
One Error	0.2360	0.230172
Two Errors	0.0202	0.0236942
Three Errors	0.0012	0.000813037

Table 6.4 Error characteristics for the $N = 4$ variant case

Category	Obs. Frequency	Computed Probability
No Errors	0.65052	0.657793
One Error	0.30889	0.29047
Two Errors	0.03303	0.0480998
Three Errors	0.00747	0.00354
Four Errors	0.00008	$9.77001E^{-}5$

probability) of exhibiting that number of variant errors, and the computed frequency (assuming separate failures) is given. The observed frequency data provides the probability mass function $p_N(\cdot)$ used to account for common-mode failures. In our numerical study, we can compare the model results obtained using the common-mode failures indicated by this experimental data to separate failures. Separate variant failures are modeled by assuming that each variant fails independently with a probability computed using the common-mode $p_N(\cdot)$ pmf. The $p_N(\cdot)$ probability mass function for the separate failure case is binomial with parameter $p = prob(\text{variant failure})$.

Experimental data is available only for $N = 1$ through 4. In our numerical study, we compute the separate failure probability mass function for $N > 4$ to be a binomial distribution using the same value for parameter p as in the $N = 4$ case.

Next, we consider the decider failure probabilities for the acceptance test, voter, and comparison algorithm. We assume that the AT diagnoses a false positive with probability 0.002 and a false negative with probability 0.001. The probability that the voter diagnoses a false positive is 0.0005 and a false negative is 0.0001. The probability that the comparison algorithm diagnoses a false positive is 0.0001 and a false negative is 0.0002. These probabilities could be generalized. For example, the probability that the voter in an N-version programming scheme produces a false positive or false negative may depend on the number of variants which produce incorrect output. The same observation holds for the comparison algorithm in the N-self checking programming scheme. This dependence could be incorporated in the SRN. In the NVP SRN model, we maintain the number of variants that produced incorrect results in place $Ptotfail$ (therefore in the state space description) when the voter diagnosis probabilities are computed. In the NSCP with comparison algorithm SRN, we maintain the number of variants in the pair that produced incorrect by either placing the token in place $Pe1$ (indicating one of the two variants failed), place $Pe2$ (indicating both of the two variants failed), or place Pne (indicating neither of the two variants failed). Since the information is available, state dependent failure probabilities could be used to make the comparison algorithm diagnosis probabilities dependent on this status.

6.5.2 Reliability Measures

Reliability measures include the system unreliability, $UR(t) = 1 - R(t)$, which is the probability that the system fails prior to time t, and the mean time to failure, $MTTF$.

First, we examine how the mean time until system failure, $MTTF$, is affected by the number of variants available in the software block. The $MTTF$ calculation is accomplished using the expected accumulated reward until absorption. Each non-absorbing state i is assigned the reward rate $r_i = 1$. Each absorbing state i is assigned the reward rate $r_i = 0$. In our SRN

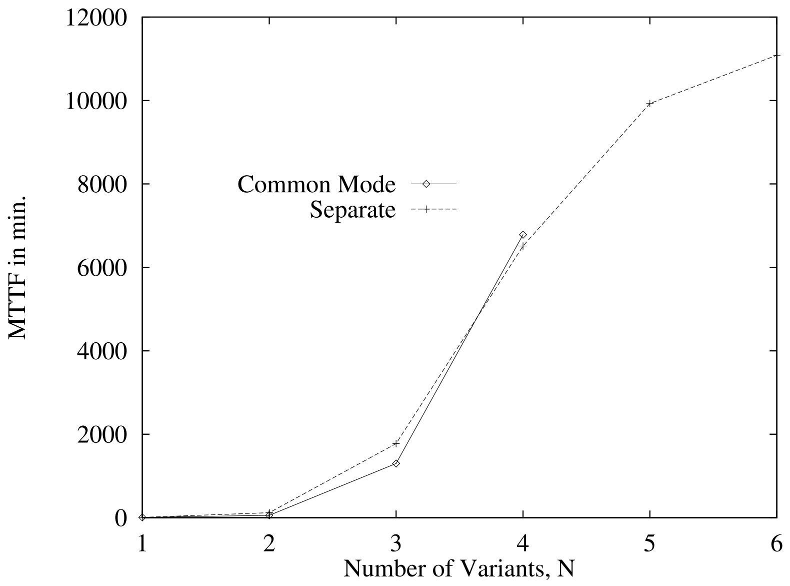


Figure 6.8 $MTTF$ of a NSCP with acceptance test as a function of the number of variants

models, absorbing states are those with a token in either place $Pdetect$ or $Pundetect$. The $MTTF$ is then computed by

$$MTTF = \sum_{i \in \Omega_{\mathcal{T}}} r_i \int_0^{\infty} P_i(\tau) d\tau$$

In Figure 6.8, the mean time to failure of the NSCP block with acceptance test error diagnosis is shown. We examine how the $MTTF$ calculation differs for separate failures and common-mode failures. At $N = 1$, the $MTTF$ is the same in each case. At $N = 2$ and $N = 3$, we see that the separate failure computation is optimistic. This is intuitive, since we expect common-mode failures to decrease block reliability. At $N = 4$, the $MTTF$ of the separate failure computation is pessimistic. This non-intuitive result occurs because in this case, the probability that one or two variants out of the four variants fail is less in the case of separate failure mode than in the common-mode case (as shown in Table 6.4). In general, we would not expect this to be true if the sample size of the experiment were increased. For both the separate failure case and the common-mode failure case, the $MTTF$ increases significantly with the number of variants.

Next, the unreliability of the N self-checking programming block with comparison algorithm for error diagnosis is plotted in Figure 6.9. The unreliability, $UR(t) = 1 - R(t)$, is the probability that a token arrives at either place $Pdetect$ or place $Pundetect$ by time t. The reliability, $R(t)$, of each model is computed using transient analysis. If $P_i(t)$ is the probability that the software block is in state i at time t then the reliability is computed simply by summing over the probability that the software block is in any state where there are no tokens in either place $Pdetect$ or $Pundetect$.

$$R(t) = \sum_{i \in \Omega_{\mathcal{T}} : \#(Pdetect)=0 \wedge \#(Pundetect)=0} P_i(t)$$

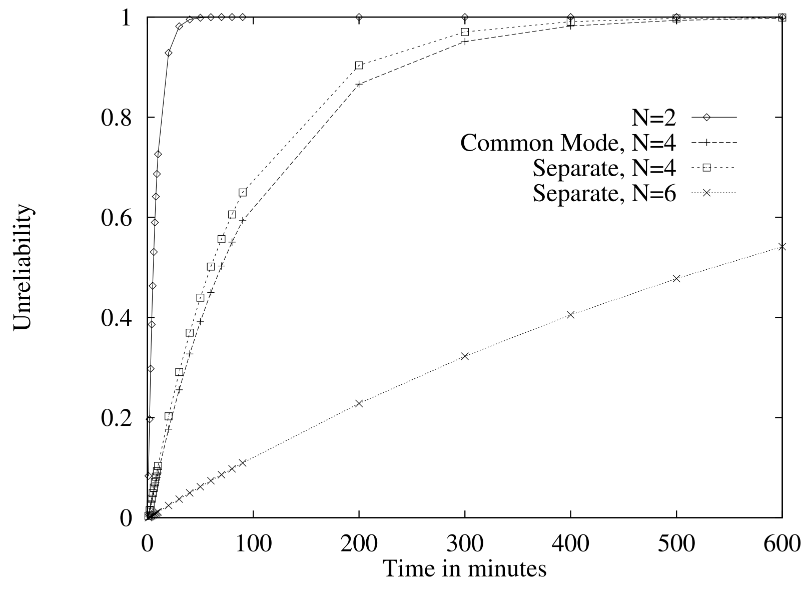


Figure 6.9 Failure probability of NSCP with comparison algorithm as a function of time

$R(t)$ is determined by first performing transient analysis of the SRN to determine $P_i(t)$. Then the reward rates for each state are defined: $r_i = 1$ if i is an operational state (a state with no tokens in either place $Pundetect$ or $Pdetect$) and $r_i = 0$ otherwise. The reliability is simply the expected reward at time t.

$$R(t) = \sum_{i \in \Omega_{\mathcal{T}}} r_i P_i(t)$$

The unreliability is plotted for $N = 2$, where (due to the fact that the comparison algorithm operates on two variant outputs at a time) the separate failure case is stochastically identical to the common-mode failure case. Next, the unreliability is plotted for $N = 4$ and 6. Only $N = 4$ data is available for common-mode failures. We see that reliability improves significantly by increasing N from 2 to 6.

6.5.3 Safety Measures

In safety analysis, the distinction between undetected and detected failures is most significant. Only undetected failures are unsafe. Detected failures are viewed as an alternate (and safe) way to complete an execution of the software block. The measures of interest in safety analysis include, $S(t)$, the probability an undetected error occurs by time t and $MTTUF$, the mean time to an undetected failure.

A slight modification of the SRN models is needed to compute the safety measures. Detected failures and successful software block completion are both safe executions of the software block. This is accomplished the SRN model by modifying each transition into place $Pdetect$ so that it goes instead to place $Pcompl$. This is the *only* modification to the SRN that is needed to generate safety measures. After making this modification, the computation of $S(t)$ and $MTTUF$ follow the same procedures as the computation of $R(t)$ and $MTTF$ respectively.

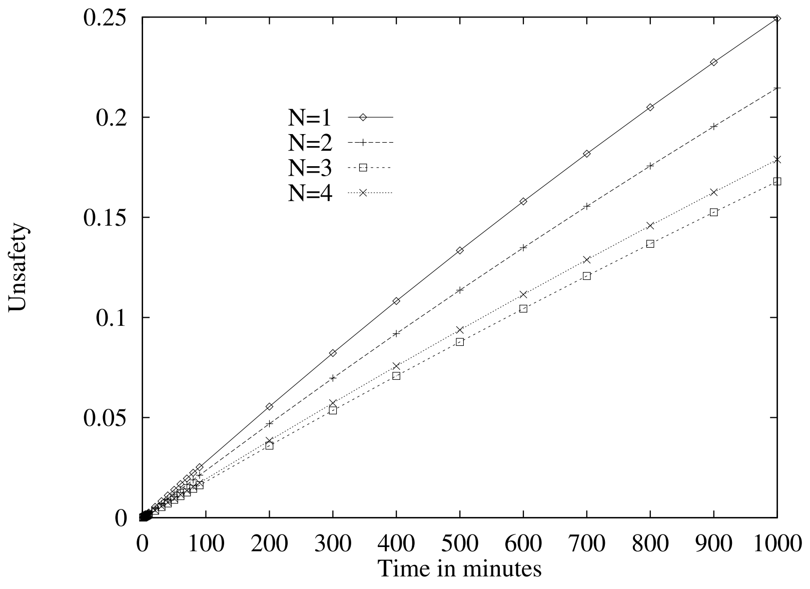


Figure 6.10 Probability of an unsafe failure in a RB as a function of time assuming common-mode failures

Figure 6.10 shows the probability of unsafe failure in a recovery block as a function of time, $US(t) = 1.0 - S(t)$. The software block is considered to be in a safe state whenever there is no token in place $Pundetect$. The safety distribution, $S(t)$ is computed using the transient solution vector. The block safety is computed simply by summing over the probability that the software block is in any state where there are no tokens in $Pundetect$.

$$S(t) = \sum_{i \in \Omega_{\mathcal{T}}:\#(Pundetect)=0} P_i(t)$$

This is easily computed once the transient analysis of the (modified) SRN produces $P_i(t)$. The reward rates for each state are defined: $r_i = 1$ if i is a safe state (a state with no tokens in place $Pundetect$) and $r_i = 0$ otherwise. The block safety is simply the expected reward at time t.

$$S(t) = \sum_{i \in \Omega_{\mathcal{T}}} r_i P_i(t)$$

The complement of block safety, $US(t) = 1.0 - S(t)$, is computed for the recovery block strategy assuming common-mode failures. We see that as N increases from 1, to 2, and to 3, the safety of the block increases. As N increases from 3 to 4, the safety decreases slightly. This is again due to experimental sampling. We expect that the unsafety of the block would decrease as N increases.

Figure 6.11 shows the mean time until an unsafe failure in an NVP block as a function of the number of variants. The $MTTUF$ is computed using the expected accumulated reward until absorption which was used to compute the $MTTF$. Each non-absorbing state i is assigned the reward rate $r_i = 1$. Each absorbing state i is assigned the reward rate $r_i = 0$. In our SRN models, absorbing states are those with a token in place $Pundetect$. (Recall that the

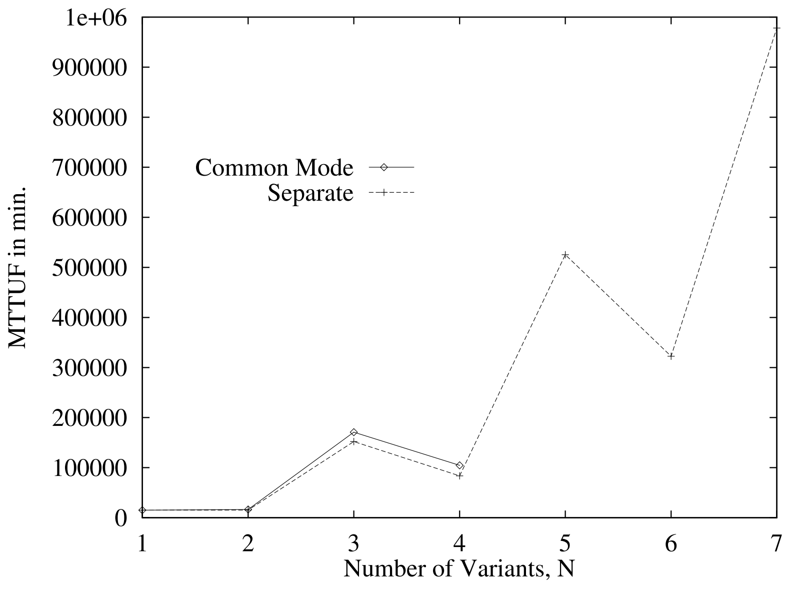


Figure 6.11 Mean time until an undetected failure in an NVP block as a function of the number of variants

modification made to the nets to facilitate safety analysis prevents a token from arriving in place $Pdetect$.) The $MTTUF$ is then computed by

$$MTTUF = \sum_{i \in \Omega_{\mathcal{T}}} r_i \int_0^\infty P_i(\tau) d\tau$$

In Figure 6.11, we plot the mean time until an unsafe failure in an NVP block for both common-mode and separate failures. The saw toothed nature of the graph is due to the NVP strategy requirement that an output is diagnosed to be correct only if more than half of the variants produce the same output.

6.5.4 Performance Measures

The performance measures we compute include $E[D(t)]$, the number of datasets processed prior to time t and $E[D]$, the number of datasets processed prior to failure. We examine how the expected number of datasets is affected by the number of variants available in the system. In Figure 6.12, the expected number of dataset processed in a RB as a function of the number of variants is shown. The expected number of datasets is computed by assigning a positive reward to each state where there is a token in place Pm_1 and zero reward in all other states. Since the expected sojourn of a token in place Pm_1 is $1/\text{rate}(Tm_1)$, the reward assigned to states where there is a token in place Pm_1 is the rate of transition Tm_1.

$$E[D] = \sum_{i \in \Omega_{\mathcal{T}}} r_i \int_0^\infty P_i(\tau) d\tau$$

We observe in this graph, that as N increases beyond five variants, the increase in the expected number of datasets is small.

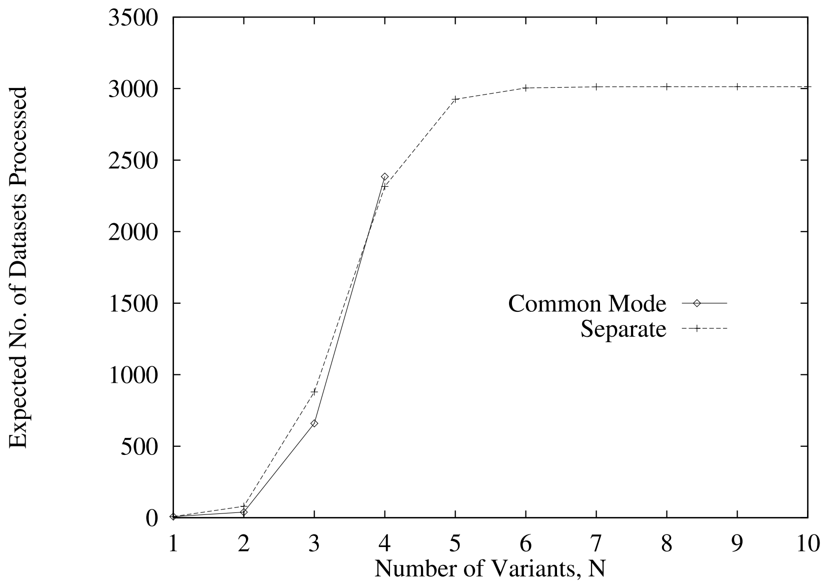


Figure 6.12 Expected number of datasets processed in a RB as a function of the number of variants

In Figure 6.13, the expected number of datasets in an NVP block is shown as a function of time. The expected number of datasets processed until time t, $E[D(t)]$, is computed using the same reward definitions as were used in the computation of $E[D]$.

$$E[D(t)] = \sum_{i \in \Omega_{\mathcal{T}}} r_i \int_0^t P_i(\tau) d\tau$$

The expected number of datasets is computed for $N = 1, 3, 5, 7$, and 9. In the case $N = 1$, the common-mode and separate failure cases are stochastically identical. We see that as N increases, the expected number of datasets processed over time continues to increase. This is due to the reduction in the probability of software block failure as N increases.

6.6 CONCLUSIONS

In this chapter, we explored the capability of stochastic reward nets (SRNs) to model fault tolerant software techniques. The software fault tolerance techniques we considered are N-version programming, recovery blocks, and N self-checking programming. SRNs are well suited to allow for concise expression of each software fault tolerance technique. Associated with these techniques are various dependencies. We have shown that model structure and state dependent transition probabilities allow common-mode and separate failures to be easily modeled. In addition, since SRNs are equal in their modeling power to Markov reward models (MRMs), the wide range of steady state, transient, cumulative, and sensitivity measures available for MRMs, were shown to easily obtained for SRNs. We numerically demonstrated that safety, reliability, and performance measures were easily obtained for the above three software fault tolerance techniques using SRNs.

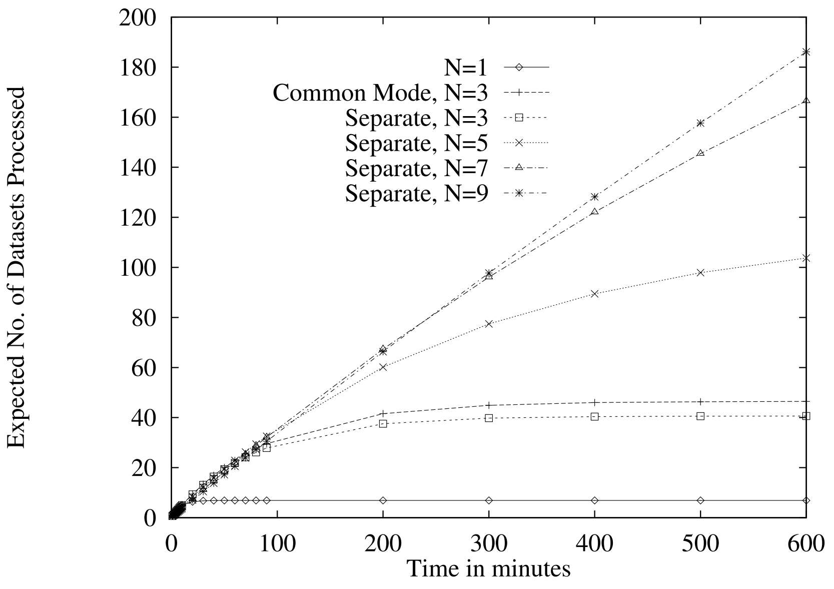


Figure 6.13 Expected number of datasets processed in an NVP block as a function of time

ACKNOWLEDGEMENTS

The work of K.S. Trivedi was supported in part by the National Science Foundation under Grant CCR-9108114.

REFERENCES

[Ajm84] M. Ajmone-Marsan, G. Conte, and G. Balbo. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.

[And81] T. Anderson and P.A. Lee. *Fault Tolerance-Principles and Practice*. Prentice Hall, 1981.

[Avi85] A. Avižienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[Bea78] M. D. Beaudry. Performance related reliability for computer systems. *IEEE Transactions on Computers*, C-27(6):540–547, June 1978.

[Chi85] G. Chiola. A software package for the analysis of generalized stochastic Petri net models. In *Proceedings of the International Workshop on Timed Petri Nets*, pages 136–143, Los Alamitos, CA, July 1985. IEEE Computer Society Press.

[Cia92] G. Ciardo, A. Blakemore, P. F. Chimento, J. K. Muppala, and K. S. Trivedi. Automated generation and analysis of Markov reward models using stochastic reward nets. In Carl Meyer and R. J. Plemmons, editors, *Linear Algebra, Markov Chains, and Queueing Models, IMA Volumes in Mathematics and Applications*, volume 48, Heidelberg, 1992. Springer-Verlag.

[Cia90] G. Ciardo, R. Marie, B. Sericola, and K. S. Trivedi. Performability analysis using semi-Markov reward processes. *IEEE Transactions on Computers*, C-39(10):1251–1264, 1990.

[Cia89] G. Ciardo, J. Muppala, and K. Trivedi. SPNP: stochastic Petri net package. In *Proceedings of the International Workshop on Petri Nets and Performance Models (PNPM '89)*, pages 142–150, Los Alamitos, CA, December 1989. IEEE Computer Society Press.

[Cou91] J.A. Couvillion, R. Freire, R. Johnson, W.D. Obal, M.A. Qureshi, M. Rai, W.H. Sanders, and J.E. Tvedt. Performability modeling with ultrasan. *IEEE Software*, 8(5):69–80, September 1991.

[Dug93a] Joanne Bechta Dugan and Michael R. Lyu. System-level reliability and sensitivity analyses for three fault-tolerant system architectures. In *Proceedings of the Fourth IFIP Working Conference on Dependable Computing for Critical Applications (DCCA 4)*, December 1993.

[Dug93b] Joanne Bechta Dugan and Michael R. Lyu. System reliability analysis of an N-version programming application. In *Proceedings of the International Symposium on Software Reliability Engineering*, November 1993.

[Flo91] Gérard Florin, C. Fraize, and Stéphane Natkin. Stochastic Petri nets: properties, applications and tools. *Microelectronics and Reliability*, 31(4):669–697, 1991.

[How71] R. A. Howard. *Dynamic Probabilistic Systems, Vol.II: Semi-Markov and Decision Processes*. John Wiley & Sons, New York, 1971.

[Ibe90] O.C. Ibe and K.S. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(10), December 1990.

[Ibe89] O. C. Ibe, K. S. Trivedi, A. Sathaye, and R. C. Howe. Stochastic Petri net modeling of VAXcluster system availability. In *Proc. International Conference on Petri Nets and Performance Models*, Kyoto, Japan, December 1989.

[Lap90] J. Laprie, J. Arlat, C. Béounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. 23(7), July 1990.

[Lyu93] Michael R. Lyu and Yu-Tao He. Improving the N-version programming process through the evolution of a design paradigm. *IEEE Transactions on Reliability*, June 1993.

[Mol82] M. K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, C-31(9):913–917, September 1982.

[Nat80] S. Natkin. *Reseaux de Petri stochastiques*. PhD thesis, CNAM-Paris, June 1980.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.

[Puc90] G. Pucci. On the modeling and testing of recovery block structures. In *Proceedings of the Twentieth International Symposium on Fault Tolerant Computing (FTCS 20)*, pages 356–363, Newcastle upon Tyne, UK, 1990.

[Puc92] Geppino Pucci. A new approach to the modeling of recovery block structures. *IEEE Transactions on Software Engineering*, SE-18(2):356–363, February 1992.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Tom94] Lorrie A. Tomek, Varsha Mainkar, Robert Geist, and Kishor S. Trivedi. Reliability modeling of life-critical, real-time systems. *IEEE Proceedings, Special Issue on Real-Time Life-Critical Systems*, 82(1), January 1994.

[Tom91] L. A. Tomek and K. S. Trivedi. Fixed point iteration in availability modeling. In M. Dal Cin, editor, *Proceedings of the Fifth International GI/ITG/GMA Conference on Fault-Tolerant Computing Systems*, pages 229–240. Springer-Verlag, Berlin, September 1991.