

# Analyzing Concurrent and Fault-Tolerant Software using Stochastic Reward Nets

Gianfranco Ciardo  
Software Productivity Consortium  
Herndon, VA 22070

Jogesh K. Muppala  
Software Productivity Consortium  
Herndon, VA 22070

Kishor S. Trivedi \*  
Dept. of Electrical Engineering  
Duke University  
Durham, NC 27706

## Abstract

We present two software applications and develop models for them. The first application considers a producer-consumer tasking system with an intermediate buffer task and studies how the performance is affected by different selection policies when multiple tasks are ready to synchronize. The second application studies the reliability of a fault-tolerant software system using the recovery block scheme. The model is incrementally augmented by considering clustered failures or the effective arrival rate of inputs to the system.

We use stochastic reward nets, a variant of stochastic Petri nets, to model the two software applications. In both models, each quantity to be computed is defined in terms of either the expected value of a reward rate in steady-state or at a given time  $\theta$ , or as the expected value of the accumulated reward until absorption or until a given time  $\theta$ . This allows extreme flexibility while maintaining a rigorous formalization of these quantities.

## 1 Introduction

Many applications demand high performance and reliability/availability from computer systems. Higher levels of integration and newer techniques in VLSI design have made hardware with high performance and reliability, relatively inexpensive. Software, on the other hand, is becoming a major component in the overall cost of these systems [27]. Often, though, the software poses performance and reliability bottlenecks which should be discovered and eliminated. Improvements in software assessment methods for the design phase of the software life cycle are required to minimize costly redesigns and changes due to unanticipated performance or reliability problems.

---

\*This work was supported in part by the National Science Foundation under Grant CCR-9108114 and by the Naval Surface Weapons Center under the ONR Grant N00014-91-J-4162.

Markov models have been used for software performance assessment [10], software reliability assessment [17], and for analyzing software fault-tolerance [9, 14, 21]. Markov models have been quite popular in hardware performance models and hardware reliability models as well. Reasons for the popularity of Markov models include the ability to capture various dependencies, the equal ease with which steady-state, transient and cumulative transient measures can be computed and the extension to Markov reward models useful in performability analysis. The main drawbacks of Markov models include the size of the state space and the assumption of exponentially distributed sojourn times. It is possible to remove the assumption of exponential sojourn time distributions by using phase-type expansions of non-exponential distributions [13, 29]. This method converts a non-Markovian problem into a Markovian one with an even larger state space.

Stochastic Petri nets (SPNs) can be used to specify the problem in a concise fashion and the underlying Markov chain can then be generated automatically. Algorithms for storing and efficiently solving relatively large Markov chains have emerged and have been implemented in several packages [6, 8, 23]. Our version of SPNs, called stochastic reward nets (SRNs), not only allows the compact specification of large Markov models but also permits the concise specification of reward structures at the net level. In this way, automatic generation of large Markov reward models is facilitated. Steady-state, transient, and cumulative transient measures of the resulting Markov reward models can be computed [8]. We illustrate our approach with two examples: software performance assessment of a producer-consumer system and reliability assessment of the recovery block, a software fault-tolerance scheme.

As we will show, detailed behavior of the system can be described concisely and the effects of various design decisions can be predicted easily. We note that SRNs are also suitable for hardware performance, reliability, and performability analysis, hence they can be used for combined hardware-software analysis. Some aspects of system hardware are indeed represented in the models described in this paper.

Several papers are relevant to our study. Performance modeling of concurrent software has been carried out using Markov chains [11, 12], series parallel graphs [16], queueing networks [28], stochastic rendezvous networks [31, 30], and SPNs [18, 5, 26]. A recent study by Leu *et al.* uses SPNs to model fault-tolerant aspects of software [15].

Section 2 gives a brief review of the SRN concepts; it also contains an explanation for the symbols used in the paper. In Section 3 we present the analysis of a producer-consumer tasking system and in Section 4 we present the analysis of the recovery block scheme. Conclusions are presented in Section 5.

## 2 Stochastic Reward Nets

There are several definitions for Petri nets [19, 20] and even more for stochastic Petri nets. Our SRN formalism allows only exponentially distributed or constant zero times, so its underlying stochastic process is independent semi-Markov with either exponentially distributed or constant zero holding times. We assume that the semi-Markov process is regular, that is, the number of transition firings in a finite interval of time is finite with probability one. Such a process can then be transformed into a continuous-time Markov chain as it is done for the generalized stochastic Petri net (GSPN) formalism [1].

The SRNs differ from the GSPNs in several key aspects. From a structural point of view, both formalisms are equivalent to Turing machines. But the SRNs provide enabling functions, marking-dependent arc cardinalities, a more general approach to the specification of priorities, and the ability to decide in a marking-dependent fashion whether the firing time of a transition is exponentially distributed or null, often resulting in more compact nets. Perhaps more important, though, are the differences from a stochastic modeling point of view. The SRN formalism considers the measure specification as an integral part of the model. Underlying an SRN is an independent semi-Markov reward process with reward rates associated to the states and reward impulses associated to the transitions between states. Our definition of SRN explicitly includes parameters (inputs) and the specification of multiple measures (outputs). A SRN with  $m$  inputs and  $n$  outputs defines a function from  $\mathbb{R}^m$  to  $\mathbb{R}^n$ .

We define a non-parametric SRN as an 11-tuple  $\mathbf{A} = \{P, T, D^-, D^+, D^\circ, e, >, \mu_0, \lambda, w, M\}$ , where:

- $P = \{p_1, \dots, p_{|P|}\}$  is a finite set of places. Each place contains a non-negative number of tokens. The multiset describing the number of tokens in each place is called a marking. The notation  $\#(p, \mu)$  is used to indicate the number of tokens in place  $p$  in marking  $\mu$ . If the marking is clear from the context, the notation  $\#(p)$  is used.
- $T = \{t_1, \dots, t_{|T|}\}$  is a finite set of transitions ( $P \cap T = \emptyset$ ).
- $\forall p \in P, \forall t \in T, D_{p,t}^- : \mathbb{N}^{|P|} \rightarrow \mathbb{N}, D_{p,t}^+ : \mathbb{N}^{|P|} \rightarrow \mathbb{N},$  and  $D_{p,t}^\circ : \mathbb{N}^{|P|} \rightarrow \mathbb{N}$  are the marking-dependent multiplicities of the input arc from  $p$  to  $t$ , the output arc from  $t$  to  $p$ , and the inhibitor arc from  $p$  to  $t$ , respectively. If an arc multiplicity evaluates to zero in a marking, the arc is ignored (does not have any effect) in that marking.

We say that a transition  $t \in T$  is arc-enabled in marking  $\mu$  iff

$$\forall p \in P, D_{p,t}^-(\mu) \leq \#(p, \mu) \wedge (D_{p,t}^\circ(\mu) > \#(p, \mu) \vee D_{p,t}^\circ(\mu) = 0)$$

When transition  $t$  fires in marking  $\mu$  the new marking  $\mu'$  satisfies:

$$\forall p \in P, \#(p, \mu') = \#(p, \mu) - D_{p,t}^-(\mu) + D_{p,t}^+(\mu)$$

- $\forall t \in T, e_t : \mathbb{N}^{|P|} \rightarrow \{true, false\}$  is the enabling function of transition  $t$ . If  $e_t(\mu) = false$ ,  $t$  is disabled in  $\mu$ .
- $>$  is a transitive and irreflexive relation imposing a priority among transitions. In a marking  $\mu$ ,  $t_1$  is marking-enabled iff it is arc-enabled,  $e_{t_1}(\mu) = true$ , and no other transition  $t_2$  exists such that  $t_2 > t_1$ ,  $t_2$  is arc-enabled, and  $e_{t_2}(\mu) = true$ . This definition is more flexible than the one adopted by other SPN formalisms, where integers are associated with transitions (e.g., imagine the situation where  $t_1 > t_2$ ,  $t_3 > t_4$ , but  $t_1$  has no priority relation with respect to  $t_3$  and  $t_4$ ).
- $\mu_0$  is the initial marking.
- $\forall t \in T, \lambda_t : \mathbb{N}^{|P|} \rightarrow \mathbb{R}^+ \cup \{\infty\}$  is the rate of the exponential distribution for the firing time of transition  $t$ . If the rate is  $\infty$  in a marking, the transition firing time is zero.

This is a generalization of [1], where transitions are *a priori* classified as “timed” or “immediate”. In this paper, though, there are transitions for which the rate is always  $\infty$ . We still call them immediate and we represent them with a thin bar instead of a hollow rectangle. The distinction between vanishing and tangible markings introduced in [1] is still applicable: a marking  $\mu$  is said to be vanishing if there is a marking-enabled transition  $t$  in  $\mu$  such that  $\lambda_t = \infty$ ;  $\mu$  is said to be tangible otherwise. We additionally impose the interpretation that, in a vanishing marking  $\mu$ , all transitions  $t$  with  $\lambda_t(\mu) < \infty$  are implicitly inhibited. Hence, a transition  $t$  in a marking  $\mu$  is enabled in the usual sense and can actually fire iff it is marking-enabled and either  $\mu$  is tangible or  $\mu$  is vanishing and  $\lambda_t(\mu) = \infty$ .

- $\forall t \in T, w_t : \mathbb{N}^{|P|} \rightarrow \mathbb{R}^+$  describes the weight assigned to the firing of enabled transition  $t$ , whenever its rate  $\lambda_t$  evaluates to  $\infty$ . Assume that the set of transitions  $X \subseteq T$  is enabled in a vanishing marking  $\mu$ . Then, the probability of firing transition  $t$  in  $\mu$  is given by  $w_t(\mu)/(\sum_{y \in X} w_y(\mu))$ . If a marking-dependent weight specification is not needed, the definition of  $w$  can be reduced to  $\forall t \in T, w_t \in \mathbb{R}^+$ .

The SRN components described so far define a trivariate discrete-parameter stochastic process:  $\{(\mu_n, \tau_n, \theta_n), n \in \mathbb{N}\}$ .  $\mu_n$  is the  $n$ -th marking encountered,  $\tau_n \in T$  is the  $n$ -th transition to fire (marking  $\mu_{n+1}$  is obtained by firing transition  $\tau_n$  in  $\mu_n$ ), and  $\theta_n \geq 0$  is the time at which it fires ( $\theta_i \geq \theta_{i-1}$ ). It is also possible to define a continuous-time process describing the marking at time  $\theta$ ,  $\{\mu(\theta), \theta \geq 0\}$ , which is completely determined given  $\{(\mu_n, \tau_n, \theta_n), n \in \mathbb{N}\}$ :  $\mu(\theta) = \mu_{\sup\{n:\theta_n \leq \theta\}}$ . This process describes only the evolution with respect to the tangible markings, that is,  $Pr\{\mu(\theta) \text{ is vanishing}\} = 0$ .

The last component of an SRN specification defines the measures to be computed:

- $M = \{(\rho_1, r_1, \phi_1), \dots, (\rho_{|M|}, r_{|M|}, \phi_{|M|})\}$  is a finite set of measures, each specifying the computation of a single real value. A measure  $(\rho, r, \phi) \in M$  has three components. The first and second components specify a reward structure over the underlying stochastic process  $\{(\mu_n, \tau_n, \theta_n), n \in \mathbb{N}\}$ .  $\rho : \mathbb{N}^{|P|} \rightarrow \mathbb{R}$  is a reward rate:  $\rho(\mu)$  is the rate at which reward is accumulated when the marking is  $\mu$ .  $\forall t \in T, r_t : \mathbb{N}^{|P|} \rightarrow \mathbb{R}$  is a reward impulse:  $r_t(\mu)$  is the instantaneous reward gained when firing transition  $t$  while in marking  $\mu$ . Often, a marking-dependent reward impulse specification is not needed and the definition of  $r$  can be simplified accordingly. The reward structure specified by  $\rho$  and  $r$  over  $\{(\mu_n, \tau_n, \theta_n), n \in \mathbb{N}\}$  defines a new stochastic process  $\{Y(\theta), \theta \geq 0\}$ , describing the reward accumulated by the SRN up to time  $\theta$ :

$$Y(\theta) = \int_0^\theta \rho(\mu(u)) du + \sum_{n:\theta_n \leq \theta} r_{\tau_n}(\mu_n)$$

The third component of a measure specification,  $\phi$ , is a function that computes a single real value from the stochastic process  $\{Y(\theta), \theta \geq 0\}$ . If  $\mathcal{R}$  is the set of real-valued stochastic processes with index over the naturals, then  $\phi : \mathcal{R} \rightarrow \mathbb{R}$ . The generality of this definition is best illustrated by showing the wide range of measures the triplet  $(\rho, r, \phi)$  can capture (in some SRNs, some of these measures might be infinite):

- Expected number of transition firings up to time  $\theta$ : this is simply  $E[Y(\theta)]$  when all reward rates are zero and all reward impulses are one.

- Expected time-averaged reward up to time  $\theta$ :  $E \left[ \frac{Y(\theta)}{\theta} \right]$ .
- Expected instantaneous reward rate at time  $\theta$ :  $E \left[ \lim_{\delta \rightarrow 0} \frac{Y(\theta+\delta) - Y(\theta)}{\delta} \right]$ .
- Expected accumulated reward rate in steady-state:  $E \left[ \lim_{\theta \rightarrow \infty} Y(\theta) \right]$ .
- Mean time to absorption: this is a particular case of the previous measure, obtained by setting the reward rate of transient and absorbing states to one and zero, respectively, and all reward impulses to zero.
- Expected instantaneous reward rate in steady-state:  $E \left[ \lim_{\theta \rightarrow \infty} \lim_{\delta \rightarrow 0} \frac{Y(\theta+\delta) - Y(\theta)}{\delta} \right]$ , which is also the same as the expected time-average reward in steady-state:  $E \left[ \lim_{\theta \rightarrow \infty} \frac{Y(\theta)}{\theta} \right]$ .
- Supremum reward rate (assume that all reward impulses are zero):  
 $\sup_{\theta \geq 0} \left\{ v : v \in \mathbb{R} \wedge Pr \left\{ \lim_{\delta \rightarrow 0} \frac{Y(\theta+\delta) - Y(\theta)}{\delta} = v \right\} > 0 \right\}$ .  
This quantity can be expressed more simply using the stochastic process  $\{(\mu_n), n \in \mathbb{N}\}$ :  $\sup_{n \geq 0} \left\{ \rho(\mu) : Pr\{\mu^{[n]} = \mu\} > 0 \right\}$ .

Our intention is to define parametric SRNs. This can be accomplished by allowing each component of an SRN to depend on a set of parameters  $\nu = (\nu_1, \dots, \nu_m) \in \mathbb{R}^m$ :

$$\mathbf{A}(\nu) = \left\{ P(\nu), T(\nu), D^-(\nu), D^+(\nu), D^o(\nu), e(\nu), >(\nu), \mu_0(\nu), \lambda(\nu), w(\nu), M(\nu) \right\}$$

Once the parameters  $\nu$  are fixed, a simple (non-parametric) SRN is obtained.

The underlying stochastic process can be solved analytically to compute the probability of being in each tangible marking  $\mu$  at time  $\theta$ ,  $\pi_\mu(\theta)$ , or in steady state,  $\pi_\mu$ . It is also possible to directly compute the cumulative time spent in each tangible marking  $\mu$  during the interval  $[0, \theta)$ ,  $\int_0^\theta \pi_\mu(\tau) d\tau$ . All the measures described in this paper are expressed as expectations using reward rates only and they can be easily computed as a linear combination of the values of these probabilities or cumulative times.

### 3 Analysis of a producer-consumer tasking system

Consider a computer system where data items produced by  $N_p$  producers are consumed by  $N_c$  consumers. The exchange of items between the  $N_p$  producer tasks and the  $N_c$  consumer tasks is performed using one additional buffer task. A pseudo-Ada description of this system appears in Figure 1 [7].

The buffer task stores the incoming items into array Slots, having  $N_s$  positions, and it uses the integer variable FullSlots to keep track of the number of non-empty slots. Producer tasks cannot pass items to the buffer task when FullSlots is equal to  $N_s$  and consumer tasks cannot retrieve items from the buffer task when FullSlots is equal to 0. The number of produced items cannot then exceed the number of consumed items plus  $N_s$ . A larger value for  $N_s$  can only attenuate the effect of temporary increases in the production or consumption rates, but these are equal in the long run.

The mechanism by which two Ada tasks synchronize and exchange data is the “rendezvous”. Whenever a producer task has an item ready to pass, it issues an “entry call” to the buffer task (line 16). If the buffer task accepts this entry call, the rendezvous takes place, FullSlots is

```

01  $N_p$  : constant := number of producers;
02  $N_c$  : constant := number of consumers;
03  $N_s$  : constant := number of buffer slots;

04 task Buffer is
05     entry Put( Item : in data );
06     entry Get( Item : out data );
07 end Buffer;

08 task type Producer;
09 task type Consumer;

10 Producers : array ( 1 ..  $N_p$  ) of Producer;
11 Consumers : array ( 1 ..  $N_c$  ) of Consumer;

12 task body Producer is
13     Item : data;
14 begin
15     loop
16         Buffer.Put( Item );
17         statements Sp;
18     end loop;
19 end Producer;

20 task body Consumer is
21     Item : data;
22 begin
23     loop
24         Buffer.Get( Item );
25         statements Sc;
26     end loop;
27 end Consumer;

28 task body Buffer is
29     Slots : array ( 1 ..  $N_s$  ) of data;
30     FullSlots : Natural := 0;
31 begin
32     loop
33         select
34             when EnablePut =>
35                 accept Put( Item : in data ) do
36                     FullSlots := FullSlots + 1;
37                     Slots( FullSlots ) := Item;
38                 end Put;
39             or
40             when EnableGet =>
41                 accept Get( Item : out data ) do
42                     Item := Slots( FullSlots );
43                     FullSlots := FullSlots - 1;
44                 end Get;
45             end select;
46         statements Sb;
47     end loop;
48 end Buffer;

```

Figure 1: Pseudo-Ada description of the producer-consumer system.

incremented, and the item is copied into array Slots (lines 35-37); similarly, a rendezvous with a consumer (lines 41-43) decrements FullSlots by one.

Each “entry” (lines 05 and 06) has an associated queue, where tasks making an entry call wait for a rendezvous. The presence of “guards” EnablePut and EnableGet (lines 34 and 40) inhibits the rendezvous at the guarded entry if the boolean condition is false (the guard is “closed”). Table I describes the value assumed by these boolean predicates based on three factors (presence of tasks in each of the two queues and value of variable FullSlots), for the five different policies discussed. When the buffer task is ready to rendezvous, the following cases can arise:

- Only one guard is open, but its associated queue is empty. This happens only when all the slots are full and no consumer is waiting, or all the slots are empty and no producer is waiting. A rendezvous cannot take place right away; the buffer task waits until a task joins the queue with the open guard.
- Both guards are open, but their associated queues are empty. A rendezvous cannot take place; the buffer task waits for the first task to join any of the two queues.
- Only one guard is open and its associated queue contains at least one task. A rendezvous with the first task in that queue takes place immediately.
- Both guards are open and their associated queues both contain at least one task. A rendezvous with either the first producer or the first consumer in their respective queues takes place immediately.

Only the fourth case requires a choice between a rendezvous with a producer and a rendezvous with a consumer. In its definition, Ada makes no guarantee about which queue is actually selected. If a particular selection policy is desired, it can be enforced by modifying the guard predicates so that, when no queue is empty, exactly one guard is open.

In Table I, five different policies are presented:

- Nondeterministic (ND): nondeterministically select either a producer or a consumer, with uniform probability. This can be accomplished, for example, using a pseudo-random number generator. It is also possible to remember the selection made the previous time in this situation and toggle the selection; this is likely to be faster, but it introduces a correlation in the sequence of selections.
- Producer First (PF): select a producer.
- Consumer First (CF): select a consumer.
- Proportional (PR): nondeterministically select either a producer or a consumer, but, instead of using uniform probability for producers and consumers, use a probability split proportional to the number of empty and full slots, respectively. This bias tends to keep the number of empty and full slots more balanced, which is intuitively a good idea.
- Threshold (TH): choose a producer if more slots are empty than full; choose a consumer if more slots are full than empty; choose either with uniform probability if exactly half

of the slots are full. This policy tries to achieve the same goal as the previous one, but deterministically. When exactly half of the slots are full, the behavior is the same as in the ND policy; for simplicity, we assume  $N_s$  to be odd, so this case cannot arise.

Table I  
Values for Boolean Predicates EnablePut and EnableGet.

Policy	Condition			Value returned	
	Producers Waiting	Consumers Waiting	Value of FullSlots	EnablePut	EnableGet
X	X	X	0	true	false
X			$N_s$	false	true
X	no	no	$1..N_s - 1$	true	true
X	yes	no		true	X
X	no	yes		X	true
ND	yes	yes	$1..N_s - 1$	$\alpha$	not $\alpha$
PF	yes	yes	$1..N_s - 1$	true	false
CF	yes	yes	$1..N_s - 1$	false	true
PR	yes	yes	$1..N_s - 1$	$\beta$	not $\beta$
TH	yes	yes	$1.. \lfloor N_s - 1 \rfloor / 2$	true	false
			$\lceil N_s + 1 \rceil / 2 .. N_s - 1$	false	true
			$N_s / 2$ ( $N_s$ even)	$\alpha$	not $\alpha$
Note: X means that the value is not relevant. $\alpha$ is a boolean random variable with $Pr\{\alpha = true\} = 1/2$ . $\beta$ is a boolean random variable with $Pr\{\beta = true\} = (N_s - Fullslots)/N_s$ .					

So far, the description of the system has been focused on the software, but the actual timing behavior is determined also by the hardware architecture and by the allocation of tasks to processors. Three possibilities are considered:

- SINGLE: a classic single processor architecture, where all tasks share the same CPU.
- THREE: a three-processor architecture, one processor for the  $N_p$  producer tasks, another for the  $N_c$  consumer tasks, and the last one for the single buffer task.
- MANY: a one-processor-per-task architecture, with no processor sharing.

The actual number of processors in the actual system could probably be somewhere between 3 and  $N_p + N_c + 1$ , the single processor architecture is considered mainly for reference.

### 3.1 SRN model

The system just described is concisely modeled by the SRN in Figure 2. Tokens in places *Plocal*, *Clocal*, and *Blocal* represent tasks (of the appropriate type) executing the statements *Sp*, *Sc*, and *Sb*, respectively, while tokens in places *Pwait*, *Cwait*, and *Bwait* represent tasks waiting for a rendezvous at the *Put* or *Get* entries. The tokens in places *Empty* and *Full* count the number of empty and full slots, respectively.

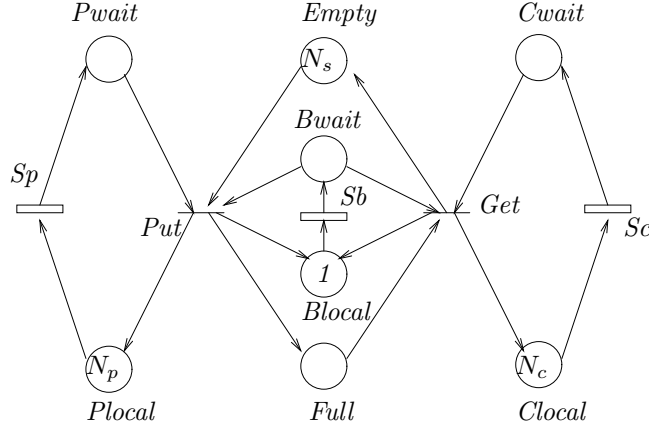


Figure 2: The SRN for the producer-consumer system.

Transitions  $Sp$ ,  $Sc$ , and  $Sb$  are assumed to be “black boxes” with an exponentially distributed time duration, but they could be changed into a more detailed phase-type expansion (using a “subnet”) if more information were available about the actual structure of the code. This would increase the size of the reachability graph, but it would also allow a more precise representation of the timing behavior in the case exponential distributions were not adequate.

Immediate transitions  $Put$  and  $Get$  correspond to the actions in the rendezvous (lines 34-38 and 40-44 in Figure 1, respectively), which are modeled as instantaneous, since the time spent for them is likely to be negligible compared to the other blocks of statements (if not, the SRN could be modified to represent these times durations explicitly).

The five different selection policies described in the previous section are obtained by defining the appropriate value for predicates EnabledPut and EnableGet. Exactly analogous to them, and even simpler to specify, are the “enabling functions”  $e_{Put}$  and  $e_{Get}$  associated with transitions  $Put$  and  $Get$ , respectively:

$$e_{Put} = \begin{cases} false & \text{if (policy = CF and } \#(Cwait) > 0 \text{ and } \#(Full) > 0) \\ & \text{or (policy = TH and } \#(Cwait) > 0 \text{ and } \#(Empty) > \#(Full)) \\ true & \text{otherwise} \end{cases}$$

$$e_{Get} = \begin{cases} false & \text{if (policy = PF and enabled(Put))} \\ & \text{or (policy = TH and } \#(Pwait) > 0 \text{ and } \#(Empty) < \#(Full)) \\ true & \text{otherwise} \end{cases}$$

In addition, the probabilistic choices in the the ND and PR policies (and TH, when  $N_s$  is even) can be specified by assigning weights  $w_{Put}$  and  $w_{Get}$  to the two transitions:

$$w_{Put} = \begin{cases} \#(Empty) & \text{if policy = TH} \\ 1 & \text{otherwise} \end{cases}$$

$$w_{Get} = \begin{cases} \#(Full) & \text{if policy = TH} \\ 1 & \text{otherwise} \end{cases}$$

The specification of the rates for the remaining three transitions completes the description of the SRN. These rates are related to the times required to execute the blocks of statements  $Sp$ ,  $Sc$ , and  $Sb$ , but also to the type of hardware architecture, since sharing the processor slows down the execution. Table II shows the firing rates used assuming perfect processor sharing with no context switch overhead, and assuming that the times required to execute blocks  $Sp$ ,  $Sc$ , and  $Sb$  for a task running on a processor in isolation (no sharing) are 0.003, 0.003, and 0.0005 seconds, respectively.

Table II  
Rates for the Transitions of the Producer-Consumer SRN

Transition	Architecture	Firing rate ( $\text{sec}^{-1}$ )
$\lambda_{Sp}$	SINGLE	$\#(P_{local}) / (0.003(\#(P_{local}) + \#(C_{local}) + \#(B_{local})))$
	THREE	$1/0.003$
	MANY	$\#(P_{local}) / 0.003$
$\lambda_{Sc}$	SINGLE	$\#(C_{local}) / (0.003(\#(P_{local}) + \#(C_{local}) + \#(B_{local})))$
	THREE	$1/0.003$
	MANY	$\#(C_{local}) / 0.003$
$\lambda_{Sb}$	SINGLE	$1 / (0.0005(\#(P_{local}) + \#(C_{local}) + 1))$
	THREE	$1/0.0005$
	MANY	$1/0.0005$

Before concluding this section, it is useful to compute the number of markings generated by the SRN analysis, both to check the correctness of the model and to avoid attempting solutions that require excessive resources (memory in particular). For the parametric SRN of Figure 2, this number is a function of the parameters  $N_p$ ,  $N_c$ , and  $N_s$  (the policy and the architecture are also parameters, but they do not affect the number of markings). Table III shows how to count the exact number of vanishing and tangible markings using a case analysis. Since the number of markings grows as  $O(N_s N_p N_c)$ , it is possible to study the system for reasonably large values of these three parameters (SRNs with  $\approx 10^4$  markings can be normally analyzed in a few minutes on a workstation, but SRNs with  $\approx 10^5$  or even  $\approx 10^6$  markings can be solved in a matter of hours, if enough memory is available).

### 3.2 Performance analysis

The five policies defined earlier have a simple implementation in Ada. Even the ones requiring a pseudo-random number generator introduce only a small overhead compared to the number of statements likely to constitute the blocks  $Sp$ ,  $Sc$ , and  $Sb$ .

The selection of a policy among the five ones presented could then be based on the effect that these policies have on the performance of the system (in steady-state). Different aspects of the system behavior might be the most relevant in defining “performance”:

- Response time for producers, consumers, or both.
- Probability distribution of the number of producers blocked because all slots are full.
- Probability distribution of the number of consumers blocked because all slots are empty.

Table III  
Marking Count for the Producer-Consumer SRN

Contents of place				Markings	
<i>Empty</i>	<i>Pwait</i>	<i>Cwait</i>	<i>Bwait</i>	Type	Count
$0..N_s$	$0..N_p$	$0..N_c$	0	tangible	$(N_s + 1)(N_p + 1)(N_c + 1)$
$0..N_s$	0	0	1	tangible	$N_s + 1$
$0..N_s - 1$	0	$1..N_c$	1	vanishing	$N_s N_c$
$1..N_s$	$1..N_p$	0	1	vanishing	$N_s N_p$
$0..N_s$	$1..N_p$	$1..N_c$	1	vanishing	$(N_s + 1)N_p N_c$
$N_s$	0	$1..N_c$	1	tangible	$N_c$
0	$1..N_p$	0	1	tangible	$N_p$
tangible markings:		$(N_s + 1)((N_p + 1)(N_c + 1) + 1) + N_p + N_c$			
vanishing markings:		$(N_s + 1)N_p N_c + N_p(N_p + N_c)$			

- Throughput of the system, expressed as the number of items passed from any producer to any consumer in a unit of time.

For the purpose of this study, the throughput of the system,  $\tau$ , is used. In the SRN of Figure 2, the throughput of the producers,  $\tau_p$ , can be computed by defining the reward rate in marking  $\mu$  as  $\lambda_{Sp}(\mu)$ , the rate of transition  $Sp$ , and computing the expected reward rate in steady-state:

$$\tau_p = \sum_{\mu} \lambda_{Sp}(\mu) \pi_{\mu}$$

$\tau_c$  and  $\tau_b$  can be computed in a similar way, or by observing that  $\tau = \tau_p = \tau_c = \tau_b/2$ .

The value of  $\tau$  with the SINGLE architecture is the same independent of the policy adopted and of the number of slots,  $N_s$ , producers,  $N_p$ , or consumers,  $N_c$  (as long as none is zero):  $\tau = 142.857 \text{ sec}^{-1}$ . The reason is that the only processor is always busy, so  $\tau$  is simply the inverse of the total time spent to process each item: 0.003 seconds in the producer task, 0.0005 seconds in the buffer task after the rendezvous with a producer, plus another 0.0005 seconds after the rendezvous with a consumer, and finally 0.003 seconds in the consumer task, for a total of 0.007 seconds ( $1/0.007 = 142.857$ ).

With the THREE and MANY architectures,  $\tau$  is instead affected by the three parameters. Figures 3 and 4 plot  $\tau$  as a function of  $N_s$  for these two architectures using the ND policy in a balanced system ( $N_p = N_c$ ). The effect of different policies is minor compared to doubling  $N_p$  and  $N_c$ , so it is studied later in this section.

With the THREE architecture, the improvement due to the increase in  $N_s$  is sublinear. In addition, it is less noticeable for larger values of  $N_p = N_c$ , since, after a certain point, the processors for the producers and the consumers become the bottleneck. In this case, the limit for  $\tau$  is the inverse of the maximum of of the time spent on each item by each processor (0.003, 0.001, and 0.003 sec respectively):  $\lim_{N_p, N_c, N_s \rightarrow \infty} \tau = 1/0.003 \text{ sec}^{-1} = 333.333 \text{ sec}^{-1}$ . For example,  $\tau = 329.340 \text{ sec}^{-1}$  when  $N_p = N_c = 32$  and  $N_s = 19$  (not shown).

The improvement due to increasing  $N_s$  with the MANY architecture is even smaller. Furthermore, it appears that the system is saturated when  $N_p = N_c = 8$  and no appreciable improvement is achieved by increasing  $N_s$ . The reason is again to be found in the bottleneck,

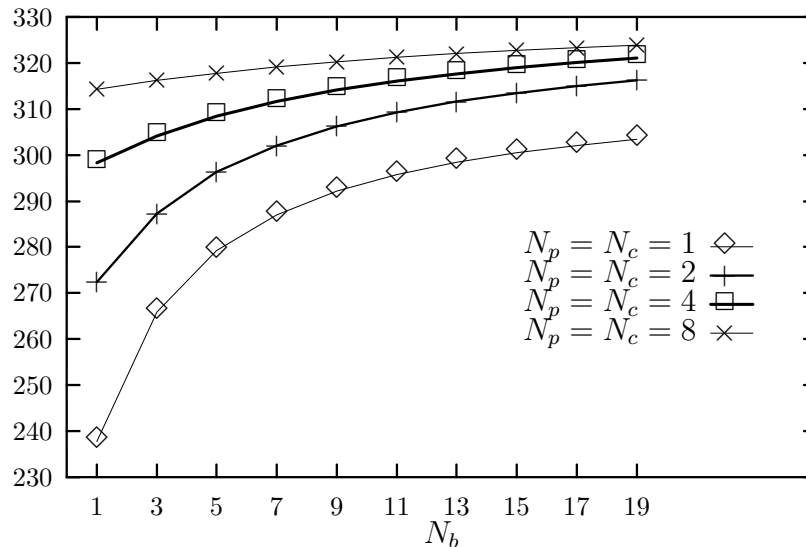


Figure 3:  $\tau$  (in  $\text{sec}^{-1}$ ) as a function of  $N_s$  (THREE architecture, ND policy).

this time the buffer task. The production (and consumption) rate could now be as high as  $N_p/0.003 \text{ sec}^{-1}$ , since each task has a dedicated processor, but the buffer task is involved in two rendezvous for each item, so that an upper bound on  $\tau$  is given by  $1/0.001 = 1000 \text{ sec}^{-1}$ . Since  $4/0.003 > 1000$ , it appears that the buffer task is already the bottleneck when  $N_p = N_c = 4$ , although, in this case, increasing the number of slots still has a visible effect on  $\tau$  (but the increase is smaller than when  $N_p = N_c = 1, 2$ ). To summarize this first part of the analysis:

- It is advantageous to increase the number of producers and consumers, *even if the total computational capacity remains constant* (architecture THREE). Depending on the nature of the system, though, this may not be possible, since the number of tasks could be dictated by external considerations (e.g., each producer task monitors a different sensor).
- Increasing the number of slots is always advantageous, but particularly so when only a few producer and consumer tasks are present.
- On a highly parallel architecture (MANY), the buffer task soon becomes a bottleneck. This points out a limitation of the Ada rendezvous. The buffer task must be introduced because, in Ada, a task performing an entry call must know the identity of the callee, so it is not possible to let a producer rendezvous directly with *any* consumer using a single entry call. This problem can be alleviated by having several buffer tasks and partitioning the producer and consumer tasks so that each buffer task serves only a subset of the producers and consumers. Partitioning, though, introduces a different kind of inefficiency. Producers associated to a buffer task having all the slots full sit idle, even if other buffer tasks may have some or even all the slots empty.

Considering now the effect of the selection policies, it is immediately apparent that there is no absolute “optimal” policy. Figure 5 shows  $\tau$  as a function of  $N_s$  in an unbalanced system

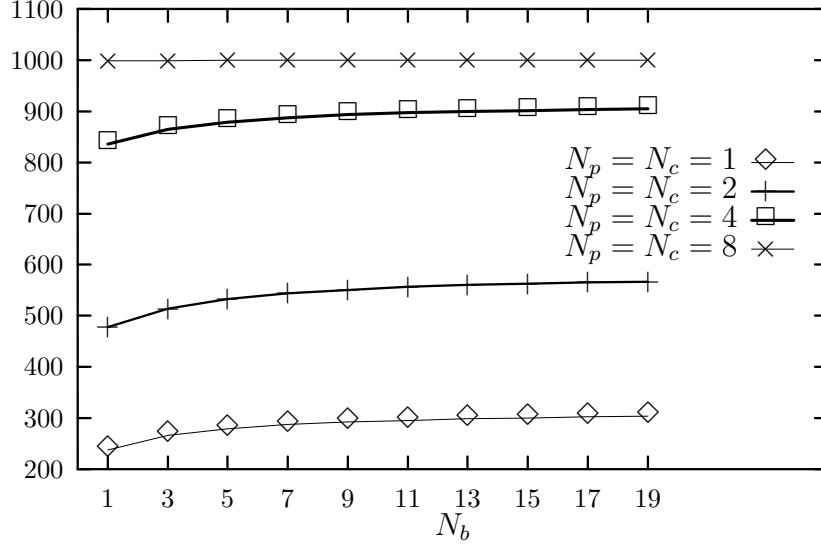


Figure 4:  $\tau$  (in  $\text{sec}^{-1}$ ) as a function of  $N_s$  (MANY architecture, ND policy).

( $N_p = 4, N_c = 2$ ), with the MANY architecture, for the five policies. The ability of producers to produce is higher than the ability of consumers to consume, hence giving precedence to consumers (CF policy) tends to restore the balance and is the optimal choice, while the PF policy increases the unbalance, resulting in the worst throughput. The plot for  $N_p = 2, N_c = 4$  (not shown) is exactly the same as the one for  $N_p = 4, N_c = 2$ , with the exception that the labels for the PF and CF policies are reversed: the PF policy is now optimal while the CF policy is the worst, and the others achieve the same value (the ND, PR, and TH policies are “symmetric”, so it should not be surprising that they result in the same throughput when  $N_p = 4, N_c = 2$  and when  $N_p = 2, N_c = 4$ ).

The TH policy is the second best in either case, followed by the PR and ND policies, in that order. This can be justified by observing that the PR policy is a (not so useful) compromise between the TH policy, which deterministically tries to achieve balance in the number of used and free slots, and the ND policy, which completely ignores the status of the slots.

The same reasoning explains the effect of the five policies in a balanced system where  $N_p = N_c = 4$  (Figure 6). The two asymmetrical policies, PF and CF, are equally poor, while the TH, PR, and ND policies are at the top, in that order.

The TH policy is a consistently good choice (nearly optimal in an unbalanced system, optimal in a balanced system). If the number of producers and consumers is subject to change during the deployment of the system, the TH policy is the best choice because of its easy implementation and predictable performance. For example, a system initially unbalanced in favor of consumers could suggest the adoption of the PF policy, but inefficiencies would arise if the situation had to be reversed later (the difference between the best and worst policy in Figure 5 is over 5%).

The plots for the PF and CF policy in Figure 6 appear to have a much slower rate of increase than the plots for the other three policies. This can be explained by considering

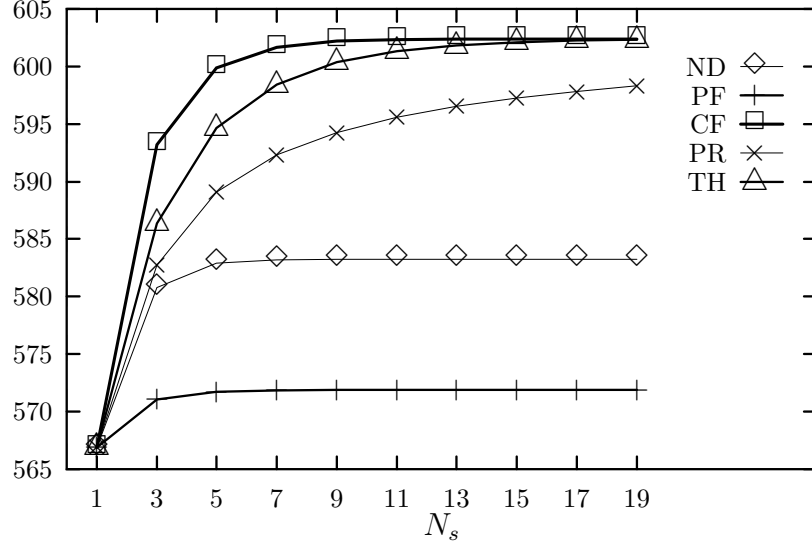


Figure 5:  $\tau$  (in  $\text{sec}^{-1}$ ) as a function of  $N_s$  (MANY architecture,  $N_p = 4$ ,  $N_c = 2$ ).

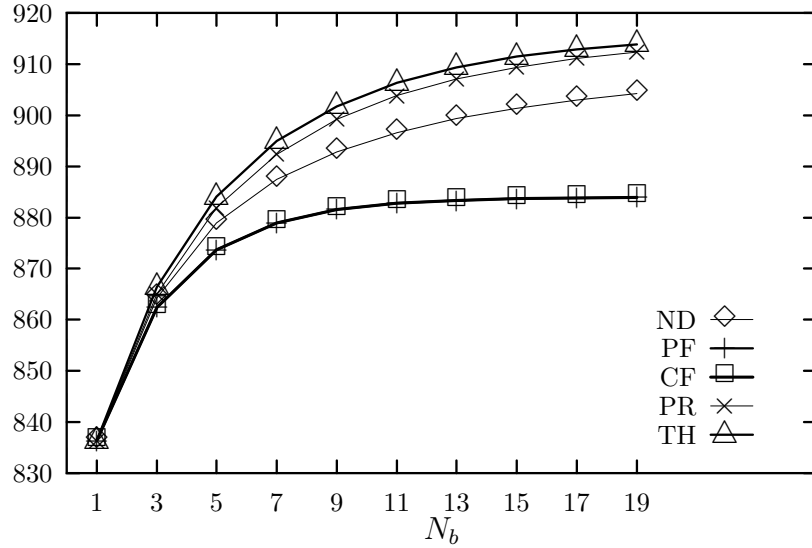


Figure 6:  $\tau$  (in  $\text{sec}^{-1}$ ) as a function of  $N_s$  (MANY architecture,  $N_p = 4$ ,  $N_c = 4$ ).

what happens with the PF policy when the system is not unbalanced toward the consumers ( $N_p \geq N_c$ ) and the buffer task is the bottleneck ( $N_c/0.003 > 1/0.001$ ). In this case, there are often both producers and consumers waiting whenever the buffer is ready to rendezvous. The PF policy, though, chooses a producer whenever possible, that is, whenever there is at least one empty slot. The effect of this behavior, in the limit, is to let the number of full slots alternate between  $N_s$  (choose a consumer) and  $N_s - 1$  (choose a producer), with a negative

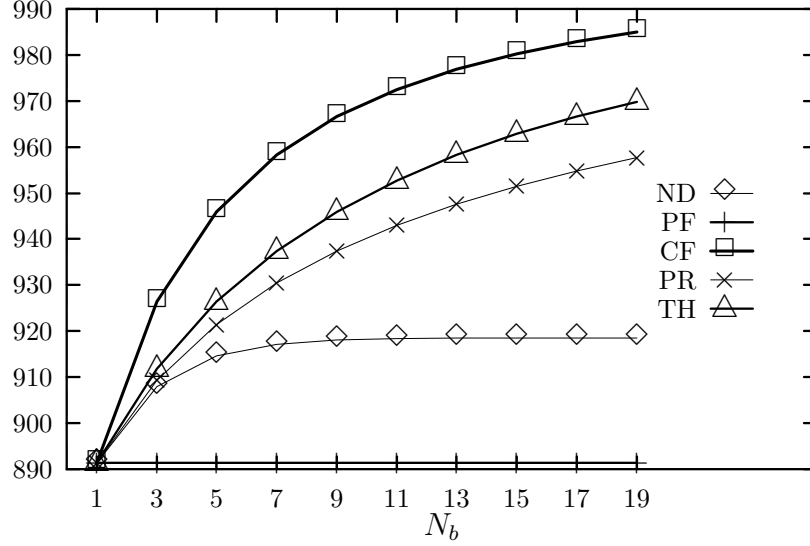


Figure 7:  $\tau$  (in  $\text{sec}^{-1}$ ) as a function of  $N_s$  (MANY architecture,  $N_p = 8$ ,  $N_c = 4$ ).

effect: the system might as well have just a single slot ( $N_s = 1$ ), since the “window” between the number of produced and consumed items is effectively restricted to one most of the time.

Figure 7 is even more dramatic, showing no appreciable increase at all for the PF policy. The values of  $\tau$  for a system with the MANY architecture,  $N_p = 8$ ,  $N_c = 4$ , and  $N_s = 125$  (not shown) confirm this observation. The difference between the optimal CF policy and the TH policy is minimal ( $994.3 \text{ sec}^{-1}$  and  $994.1 \text{ sec}^{-1}$ , respectively), while the PF policy lags seriously behind ( $891.4 \text{ sec}^{-1}$ ). Even more illuminating is the inspection of the probabilities  $\Pi_e$  that all slots are empty and  $\Pi_f$  that all slots are full:

$$\Pi_e = \sum_{\mu: \#(Empty, \mu) = N_s} \pi_\mu = \begin{cases} < 10^{-12} & \text{(ND, PF, PR, and TH policies)} \\ 0.13929526 & \text{(CF policy)} \end{cases}$$

$$\Pi_f = \sum_{\mu: \#(Full, \mu) = N_s} \pi_\mu = \begin{cases} 0.275976 & \text{(ND policy)} \\ 0.553947 & \text{(PF policy)} \\ 0.000003 & \text{(CF policy)} \\ 0.010870 & \text{(PR policy)} \\ 0.000499 & \text{(TH policy)} \end{cases}$$

These probabilities should be kept small, since, when all slots are full (empty), no rendezvous can take place with a producer (consumer), thus increasing the probability that the buffer task, which is the real bottleneck, remains idle. While  $\Pi_e$  is numerically negligible only for the non-optimal policies,  $\Pi_f$  is the most relevant quantity to observe, since  $N_p > N_c$ ;  $\Pi_f$  is small for both the CF and TH policies (about 170 times smaller for CF than for TH, although this difference affects the actual throughput  $\tau$  only marginally), but it is definitely high for the PF policy, resulting in a considerably smaller value of  $\tau$ .

The average number  $\nu_f$  of full slots also confirms this behavior:

$$\nu_f = \sum_{\mu} \#(Full, \mu) \pi_{\mu} = \begin{cases} 123.22 & \text{(ND policy)} \\ 124.55 & \text{(PF policy)} \\ 9.66 & \text{(CF policy)} \\ 100.72 & \text{(PR policy)} \\ 71.40 & \text{(TH policy)} \end{cases}$$

The TH policy does indeed achieve the value of  $\nu_f$  closest to  $N_s/2 = 62.5$ , but is still sub-optimal, suggesting that the ideal value for  $\nu_f$  is actually a function of  $N_p$  and  $N_c$  as well.

## 4 Fault-tolerant software

Design diversity as a means of achieving fault-tolerance in software has been suggested by several authors. Fault-tolerant software using this method include N-version programming [4] and recovery blocks [22]. The former uses voting on the results of various versions for error detection and the latter uses an acceptance test (AT) and rollback recovery. While these are the two major approaches to software fault-tolerance, several hybrid methods have also been proposed [22, 24].

In this section, we analyze the recovery block (RB) scheme; a fault-tolerant software construct that uses design diversity [22]. It consists of a primary module, one or more alternate modules and an AT. The primary and the alternate modules are based on different algorithms for the same problem and may be implemented by different programmers. On a given set of data inputs, the primary is executed first and the results are checked using the AT. Should the AT fail to accept the results, the alternate modules are invoked in succession until one is found to produce results that are accepted by the test or until all of them fail to satisfy the AT. In the latter case, the RB is said to have failed on this input data set. The pseudocode for a RB with a primary module and  $m$  alternate modules is shown below:

```

ensure acceptance test
  by primary module
  else by alternate module 1
  else by alternate module 2
  ...
  else by alternate module m
  else error

```

Probabilistic models of RBs have been considered by several authors [3, 9, 25]. Discrete time Markov chains (DTMCs) have been used to derive measures like the probability of RB failure or the number of inputs (correctly) processed until RB failure; continuous time Markov chains (CTMCs) have been used to derive time-based measures like the mean time to failure (MTTF) or the (un)reliability of the RB. We note that if we are able to analyze a CTMC for transient cumulative measures besides transient instantaneous measures, we can derive both the above types of measures using a CTMC, i.e., we do not need to resort to two different formalisms depending on the measures desired.

Pucci [21] points out some of the difficulties in estimating the parameters used in earlier models. He classifies events occurring in a RB into four distinct categories based on the behavior of the alternate modules and the AT. Four different events can occur:

- (1) Module  $i$  produces correct results which the AT accepts.
- (2) Module  $i$  produces correct results which the AT rejects.
- (3) Module  $i$  produces incorrect results which the AT rejects.
- (4) Module  $i$  produces incorrect results which the AT accepts.

It is easier to estimate parameters corresponding to these events. We consider a similar event classification in the model presented in the next section.

## 4.1 SRN model

In this section, we present a general SRN model for the recovery block scheme. The primary module is indexed by 0 and the alternate modules are indexed 1 through  $m$ . The execution time of module  $i$  is assumed to be exponentially distributed with mean  $1/\lambda_{Tm_i}$  and that of the AT is exponentially distributed with mean  $1/\lambda_{Tatne_i} = 1/\lambda_{Tate_i}$ . The probability that module  $i$  produces incorrect output is  $\rho_i$ . We assume that the AT fails to detect erroneous module output with probability  $p_e$ . This probability corresponds to event (4) mentioned above. We assume that this event is not catastrophic, unlike the assumption used by Pucci [21]. However, it is easy to change our model to make this event catastrophic. The AT might raise a false alarm with probability  $p_f$ , which corresponds to event (2) above. We assume that this event does not result in subsequent rejection of results from the other alternate modules, unlike as assumed by Pucci [21]. This assumption can easily be changed in the model. We let  $p_c$  be the probability that recovery following a failure to satisfy the AT is successful. We must realize that all the above probabilities pertaining to any module  $i$  are conditional probabilities, conditioned upon the fact that module  $i$  is actually invoked and that the previous  $i - 1$  modules have failed. Thus, the correlation between the software modules is automatically accounted for by the conditional nature of these probabilities.

The SRN model of a recovery block is shown in Figure 8. The net is nearly self-explanatory. Place  $Pm_0$  is the starting point of the RB. The firing of transition  $Tm_0$  corresponds to the completion of the execution of the primary module. Transitions  $Tne_0$  and  $Te_0$  correspond to the events that the results produced by the module are correct and incorrect respectively and have weights  $1 - \rho_i$  and  $\rho_i$ , respectively. Transition  $Tatne_0$  represents the execution of the AT after the module produces correct results. The immediate transitions  $Ts_0$  and  $Tfa_0$ , which correspond to events (1) and (2) mentioned above, are then enabled. The weights of these two transitions are given by  $1 - p_f$  and  $p_f$  respectively. Transition  $Tate_0$  represents the execution of the AT after the module produces incorrect results. The immediate transitions  $Tse_0$  and  $Tee_0$ , which correspond to events (3) and (4) mentioned above, are then enabled. The weights of these two transitions are  $1 - p_e$  and  $p_e$  respectively. Once an error is discovered, represented by the firing of either  $Tfa_0$  and  $Tse_0$ , the system initiates a recovery action. Transition  $Tsr_0$  represents a successful recovery after a failure of the AT and transition  $Tfl_0$  represents an unsuccessful recovery, thus resulting in the RB failure. The corresponding weights of these

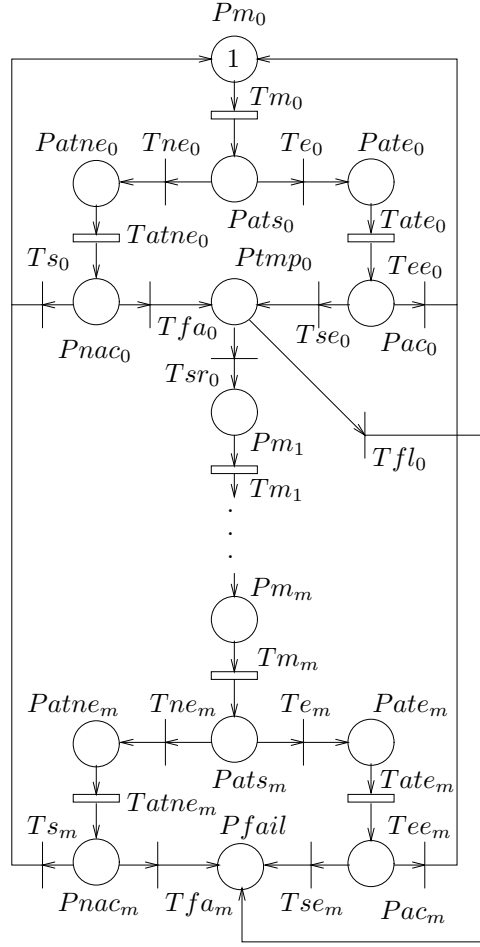


Figure 8: SRN model for a recovery block.

two transitions are  $p_c$  and  $1 - p_c$  respectively. The output arc from  $Tsr_0$  leads to  $Pm_1$ , the starting place of the first alternate module, while the output arc from  $Tfl_0$  leads to  $Pfail$  which represents the RB failure.

The alternate modules are similarly modeled by the other places and transitions indexed from 1 to  $m$ . The structure of the last module is slightly different, since the failure of the last module automatically results in a system failure. Thus, the output arcs from transitions  $Tfa_m$  and  $Tse_m$  lead to place  $Pfail$ .

We can compute the mean time to recovery block failure or the distribution of time to recovery block failure (its complement is the reliability of recovery block). For this purpose, we assign reward rate 1 to all markings in which there is no token in place  $Pfail$ ; all other markings are assigned a reward rate equal to zero. If we now compute the expected accumulated reward until system failure, we obtain the *MTTF*:

$$MTTF = \sum_{\mu: \#(Pfail, \mu) = 0} \int_0^{\infty} \pi_{\mu}(\tau) d\tau$$

By computing the expected reward rate at time  $\theta$ , we obtain instead RB reliability at time  $\theta$ :

$$R(\theta) = \sum_{\mu: \#(P_{fail}, \mu)=0} \pi_{\mu}(\theta)$$

The unreliability  $UR(\theta)$ , or probability of being failed by time  $\theta$ , is simply given by  $1 - R(\theta)$ .

We can also compute the number of number of data sets processed until system failure,  $N$ , or until a specified time  $\theta$ ,  $N(\theta)$ . To compute these quantities, we assign the rate of transition  $Tm_0$ ,  $\lambda_{Tm_0}(\mu)$ , as the reward of marking  $\mu$ . The expected accumulated reward until system failure or by time  $\theta$  yield respectively  $N$  and  $N(\theta)$ .

$$N = \sum_{\mu} \lambda_{Tm_0}(\mu) \int_0^{\infty} \pi_{\mu}(\tau) d\tau$$

$$N(\theta) = \sum_{\mu} \lambda_{Tm_0}(\mu) \int_0^{\theta} \pi_{\mu}(\tau) d\tau$$

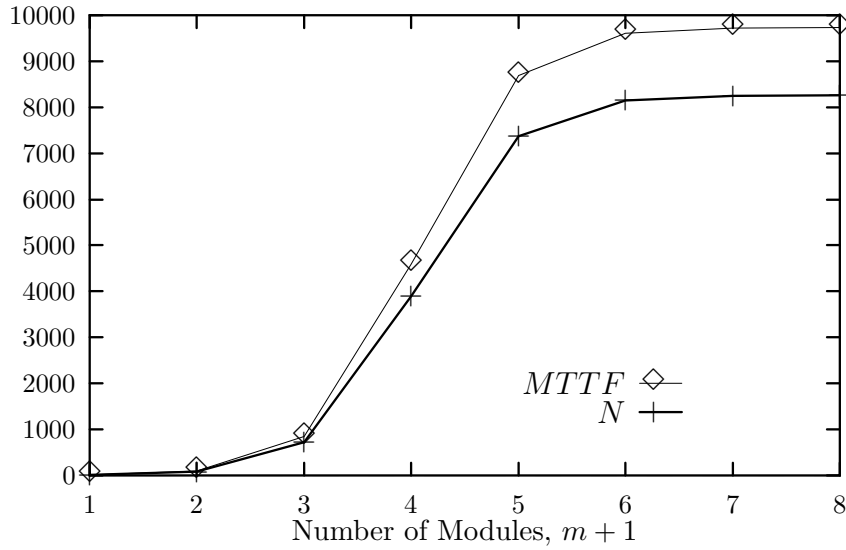


Figure 9:  $MTTF$  and  $N$  for the RB as function of the number of modules.

The  $MTTF$  and  $N$ , the expected number of inputs processed until system failure, as a function of the number of modules available in the system (including the primary module) are shown in Figure 9. We assume that the execution rate of the primary module is  $\lambda_{Tm_0} = 1 \text{ min}^{-1}$ . For each alternate module, we assume that the execution rate is three-quarters that of the previous module, i.e.,  $\lambda_{Tm_i} = 0.75\lambda_{Tm_{i-1}}$ . This is a reasonable assumption, since we would tend to use the fastest module as the primary module. The execution rate of the AT is  $\lambda_{Tate_i} = \lambda_{Tatne_i} = 100 \text{ min}^{-1}$ . The probabilities are  $\forall i, \rho_i = 0.1, p_e = 0.01, p_f = 0.01$ , and  $p_c = 0.999$ . Figure 9 shows how an increase in the number of alternate modules causes an increase in  $MTTF$  and  $N$ . Further, it is interesting to note that the greatest benefit of

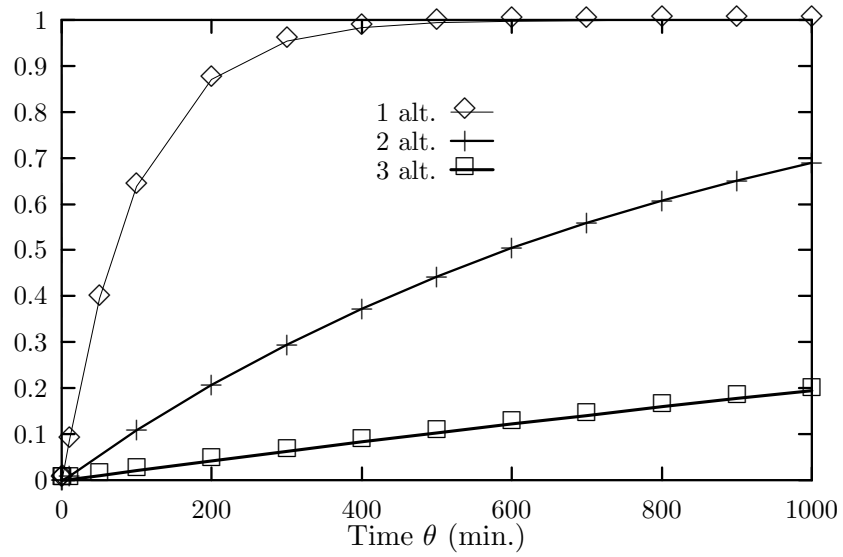


Figure 10: Failure probability for the RB as a function of time.

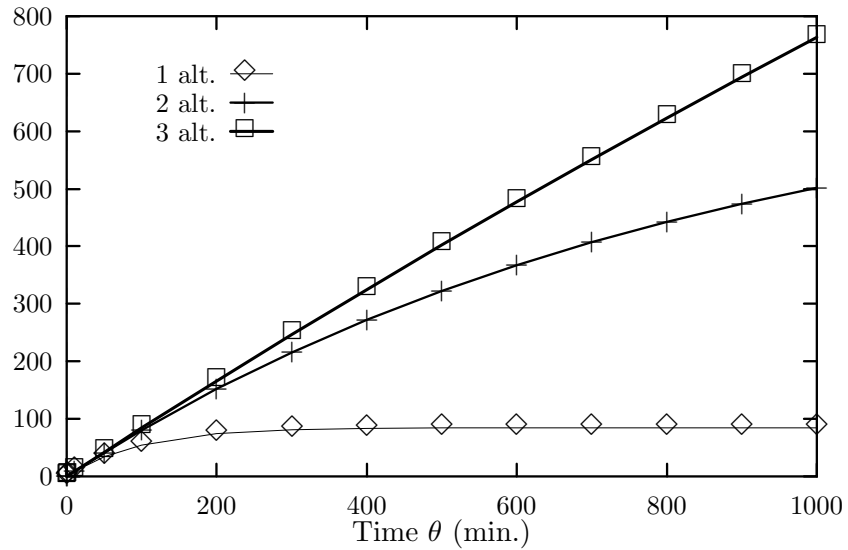


Figure 11: Number of inputs processed by the RB as a function of time.

increasing the number of alternate modules is between 1 and 5. Beyond five alternate modules, the additional benefit is quite small.

The distribution of the time to failure for the RB with 1, 2, and 3 alternate modules is plotted in Figure 10. From the figure we can see that the probability that the RB has failed by any given time  $\theta$  decreases with increase in the number of alternate modules. The number of inputs processed by time  $\theta$ ,  $N(\theta)$ , for the RB with 1, 2, and 3 alternate modules is shown

in Figure 11. The value of  $N(\theta)$  levels off beyond  $\theta = 400$  min for the system with 1 alternate, since the RB is very likely to have failed by that time.

## 4.2 Extensions

### 4.2.1 Clustering in the input data stream

The failure points in the input space for the RB tend to occur in clusters [2]. The sequence of input values to the RB tend to change slowly with time, thus, given a failure of the primary module for a given input, there is a greater likelihood of it failing for subsequent inputs. This clustering behavior in the input data stream should be taken into account. Csenki [9] considers a discrete time Markov model of a RB with failure clustering. He assumes that the system has a primary module and a single alternate. Given that the primary module has failed for a particular input, the number of subsequent inputs for which the module fails is assumed to be a random variable  $\xi$ . The length of this additional sequence is however upper-bounded by a fixed value  $\sigma$ . Thus, we can define the probabilities  $p_i = Pr\{\xi = i\}$  where  $0 \leq i \leq \sigma$ .

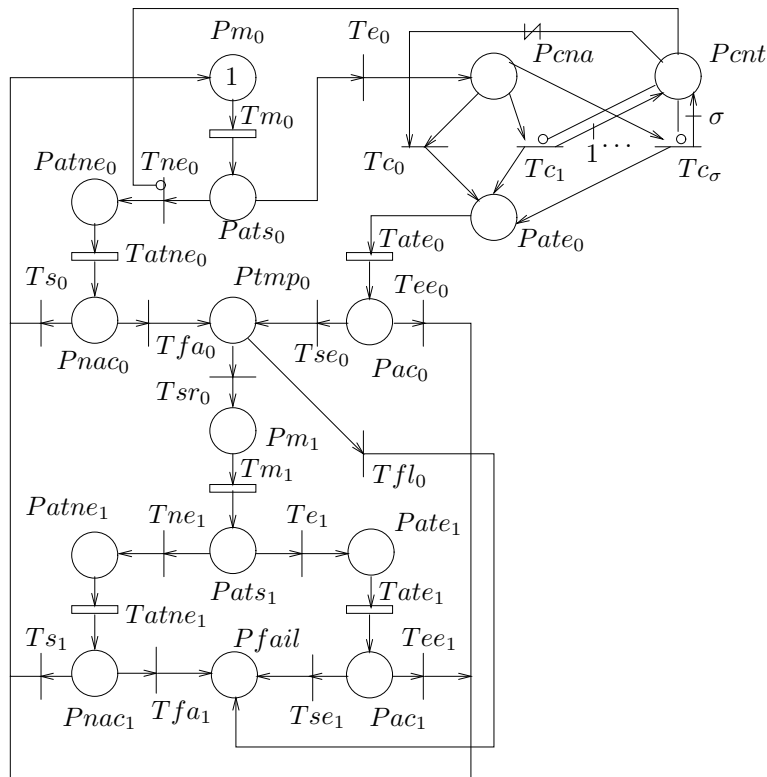


Figure 12: SRN model for a RB with clustered failures.

A SRN model for the RB with clustered failures and  $m = 1$  is shown in Figure 12. We assume that the system has one primary module and a single alternate. The structure of this SRN is similar to that of the original SRN in Figure 8. The additional places  $Pna$  and  $Pcnt$  together with the transitions  $Tc_0, Tc_1, \dots, Tc_\sigma$ , model the clustering in the input space.

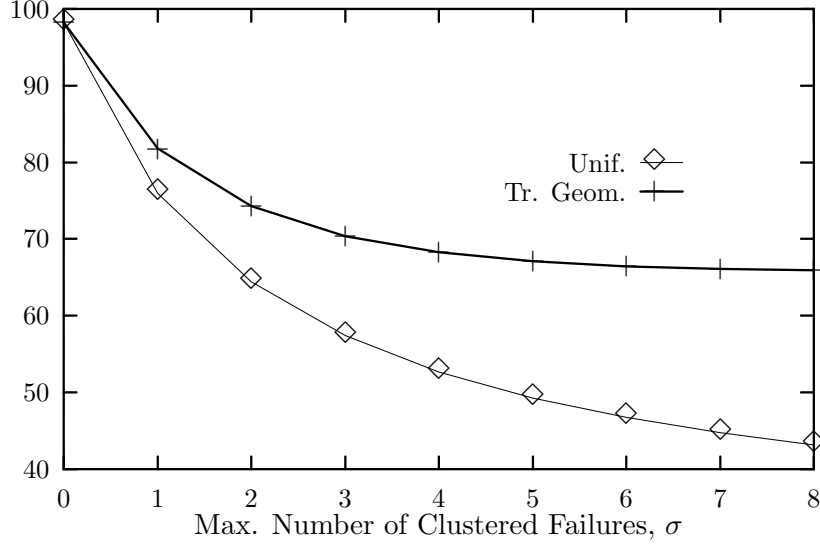


Figure 13: *MTTF* for the RB with clustered failures as a function of  $\sigma$ .

Transitions  $T_{c_0}, \dots, T_{c_\sigma}$  correspond to the events where the size of the input cluster is given by  $0, \dots, \sigma$  respectively. When the first datum of the input sequence that causes a clustered failure is encountered, it causes a failure of the primary module. Thus, the immediate transition  $T_{e_0}$  fires, depositing a token in place  $Pcna$ . Then immediate transitions  $T_{c_0}, \dots, T_{c_\sigma}$  become enabled. The weights of these transitions are given by  $p_0, \dots, p_\sigma$  respectively. Whenever transition  $T_{c_i}$  fires, representing the fact that the primary module will fail for the next  $i$ ,  $0 \leq i \leq \sigma$ , successive inputs,  $i$  tokens are deposited in place  $Pcnt$ , due to the multiple output arcs from transitions  $T_{c_i}$  to place  $Pcnt$ . At this point, transitions  $T_{ne_0}, T_{c_1}, \dots, T_{c_\sigma}$  are disabled by the inhibitor arcs from  $Pcnt$ . Thereafter, for the next  $i$  times that  $T_{m_0}$  fires, transition  $T_{e_0}$  will fire depositing a token in  $Pcna$ . Transition  $T_{c_0}$  is now used to remove a token from  $Pcnt$  and starting the usual RB sequence corresponding to the case where the first module generates an erroneous output (token in  $Pate_0$ ). This firing sequence continues until  $Pcnt$  is empty. To achieve this behavior, the input arc from  $Pcnt$  to  $T_{c_0}$  has multiplicity 1 if  $\#(Pcnt) \geq 1$  and 0 otherwise.

The *MTTF* as a function of  $\sigma$  is plotted in Figure 13 (the case with  $\sigma = 0$  corresponds to the RB with no clustered failures). Two different distributions are considered for  $p_i$ , uniform and truncated geometric. For the uniform distribution,  $\forall i, 0 \leq i \leq \sigma, p_i = 1/(\sigma + 1)$ . In this case, given that a failure has occurred, the probability that the next  $i$  inputs also result in a failure of the primary is the same for all  $i$  (a pessimistic assumption). For the truncated geometric distribution,  $\forall i, 0 \leq i \leq \sigma, p_i = p(1 - p)^i / (1 - (1 - p)^{\sigma+1})$ , where  $0 < p < 1$  (in Figure 13, we set  $p = 0.5$ ). The probability that the failure cluster has size  $i$  tapers off as  $i$  increases. This is more realistic than the uniform distribution. Clustered failures have a negative impact on the reliability of the recovery block. The effect is larger with the uniform distribution than with the truncated geometric distribution.



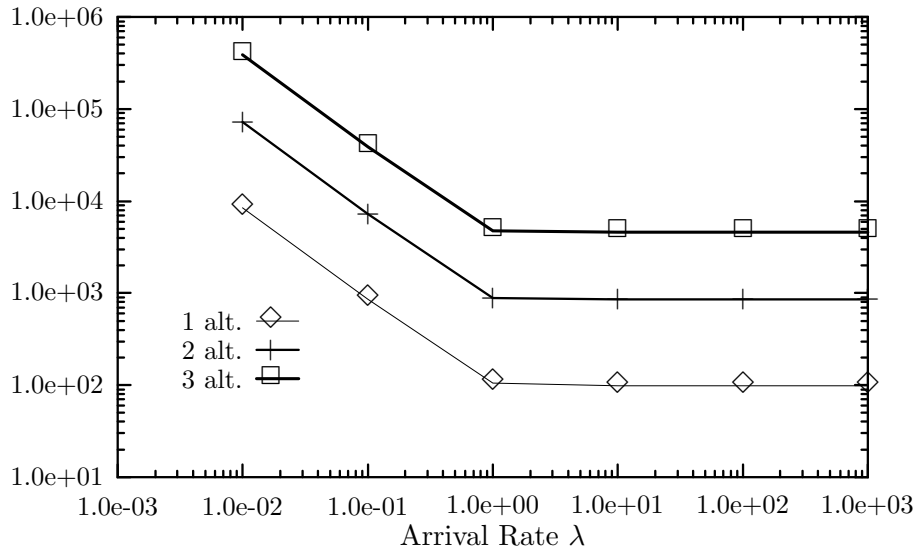


Figure 15:  $MTTF$  for a RB with input arrivals as a function of  $\lambda_{Tarr}$ .

process increases the time to failure. This is reflected in Figure 15, where the  $MTTF$  is plotted as a function of the arrival rate  $\lambda_{Tarr}$  for the RB with 1, 2, and 3 alternate modules, and  $N = 10$ . When  $\lambda_{Tarr}$  is very small, we notice that the  $MTTF$  is large. This is because there is a greater probability of the RB being unused for longer periods of time. As  $\lambda_{Tarr}$  increases, the  $MTTF$  approaches that of the basic system considered in the earlier section; in fact  $\lambda_{Tarr} = \infty$  corresponds to this system. This is understandable since, when  $\lambda_{Tarr}$  increases, there is a greater chance of finding an input datum waiting for processing when the RB completes processing an earlier input value.

### 4.3 Other extensions

The software recovery block is executed on some form of hardware platform. In the earlier sections, we have implicitly assumed that the processor(s) on which the RB is being executed is inherently fault free. In reality, hardware is subject to failures and can sometimes be repaired. Hence, any realistic model should take into account the behavior and characteristics of the underlying hardware such as processors and memory limitations. It is easy to extend our models to allow for the failure/repair behavior of the processor(s) or other hardware components. This will then allow us to carry out the combined evaluation of hardware and software.

## 5 Conclusions

In this paper, we presented two software modeling applications where SRNs can be effectively used to gain insight into a problem. The first application considers a producer-consumer tasking system with an intermediate buffer task, and studies how the performance is affected by different selection policies when multiple tasks are ready to synchronize.

The second application studies the reliability of a recovery block scheme. The initial model is incrementally augmented by considering the possibility of clustered failures or by taking into account the effective arrival rate of inputs to be processed by the system.

In either model, each quantity to be computed is defined in terms of either the expected value of a reward rate in steady-state or at a given time  $\theta$ , or as the expected value of the accumulated reward until absorption or until a given time  $\theta$ . This allows extreme flexibility while maintaining a rigorous formalization of these quantities.

## References

- [1] Ajmone Marsan, M., Balbo, G., and Conte, G. A class of Generalized Stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*. **2**, 2 (May 1984), 93–122.
- [2] Ammann, P. E., and Knight, J. C. Data diversity: An approach to software fault tolerance. In *Proc. Seventeenth Int. Symp. on Fault-Tolerant Computing*. IEEE Computer Society Press, Los Alamitos, CA, July 1987, pp. 122–126.
- [3] Arlat, J., Kanoun, K., and Laprie, J. C. Dependability evaluation of software fault-tolerance. In *Proc. Eighteenth Int. Symp. on Fault-Tolerant Computing*. IEEE Computer Society Press, Los Alamitos, CA, June 1988, pp. 142–147.
- [4] Avizienis, A. The N-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.* **SE-11**, 12 (Dec. 1985), 1491–1501.
- [5] Blakemore, A., and Schebella, G. Tools for analyzing dynamic properties of system and software designs. In *Proceedings of the XI IFIP Congress, San Francisco, CA*. Aug. 1989.
- [6] Chiola, G. A software package for the analysis of Generalized Stochastic Petri Net models. In *Proceedings of the International Workshop on Timed Petri Nets*. Torino, Italy, July 1985.
- [7] Ciardo, G. Analysis of large stochastic Petri net models. PhD thesis, Duke University, Durham, NC, USA, 1989.
- [8] Ciardo, G., Trivedi, K. S., and Muppala, J. SPNP: stochastic Petri net package. In *Proceedings of the Third International Workshop on Petri Nets and Performance Models (PNPM89)*. Kyoto, Japan, Dec. 1989.
- [9] Csenki, A. Recovery block reliability analysis with failure clustering. In *Proc. of IFIP Working Group 10.4 Int. Working Conf. on Dependable Computing for Critical Applications*. University of California, Santa Barbara, Aug. 1989.
- [10] Duda, A. Approximate performance analysis of parallel systems. In Iazeolla, G., Courtois, P. J., and Boxma, O. J. (Eds.), *Computer Performance and Reliability*. Elsevier Science Publishers, B. V. (North-Holland), Amsterdam, The Netherlands, 1988, pp. 189–202.

- [11] Heidelberg, P., and Trivedi, K. S. Queueing network models for parallel processing with asynchronous tasks. *IEEE Transactions on Computers*. **C-31**, 11 (Nov. 1982), 1099–1109.
- [12] Heidelberg, P., and Trivedi, K. S. Analytic queueing models for programs with internal concurrency. *IEEE Transactions on Computers*. **C-32**, 1 (Jan. 1983), 73–82.
- [13] Hsueh, M. C., Iyer, R. K., and Trivedi, K. S. Performability modeling based on real data: a case study. *IEEE Transactions on Computers*. (Apr. 1988).
- [14] Laprie, J. C. Dependability evaluation of software systems. *IEEE Trans. Softw. Eng.* **SE-10**, 6 (Nov. 1984), 701–714.
- [15] Leu, S.-W., Fernandez, E. B., and Khoshgoftaar, T. Fault-tolerant software reliability modeling using Petri net. *Microelectronics and Reliability*. **31**, 4 (1991), 645–667.
- [16] Mak, V. W., and Lundstrom, S. F. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*. **1**, 3 (July 1990), 257–270.
- [17] Musa, J. D., Iannino, A., and Okumoto, K. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, Singapore, 1987.
- [18] Peng, D., and Shin, K. G. Modeling of concurrent task execution in a distributed system for real-time control. *IEEE Transactions on Computers*. **C-36**, 4 (Apr. 1987), 500–516.
- [19] Peterson, J. L. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.
- [20] Petri, C. Kommunikation mit Automaten. PhD thesis, University of Bonn, Bonn, West Germany, 1962.
- [21] Pucci, G. On the modelling and testing of recovery block structures. In *Proc. Twentieth Int. Symp. on Fault Tolerant Computing*. Newcastle upon Tyne, UK, 1990, pp. 356–363.
- [22] Randell, B. System structure for software fault tolerance. *IEEE Trans. Softw. Eng.* **SE-1**, 2 (June 1975), 220–232.
- [23] Sanders, W. H., and Meyer, J. F. METASAN: a performability evaluation tool based on Stochastic Activity Networks. In *Proceedings of the ACM-IEEE Comp. Soc. Fall Joint Comp. Conf.*. Nov. 1986.
- [24] Scott, R. K., Gault, J. W., and McAllister, D. F. The consensus recovery block. In *Proc. Total System Reliability Symp.*. 1983, pp. 74–85.
- [25] Scott, R. K., Gault, J. W., and McAllister, D. F. Fault-tolerant software reliability modeling. *IEEE Trans. Softw. Eng.* **SE-13**, 5 (May 1987), 582–592.
- [26] Stansifer, R., and Marinescu, D. Petri net models of concurrent Ada programs. *Microelectronics and Reliability*. **31**, 4 (1991), 577–594.
- [27] Stone, H. S. *High-Performance Computer Architecture*. Addison-Wesley, Reading, Massachusetts, 1987.

- [28] Thomasian, A., and Bay, P. Analytic queueing network models for parallel processing of task systems. *IEEE Transactions on Computers*. **C-35**, (Dec. 1986), 1045–1054.
- [29] Trivedi, K. S. *Probability & Statistics with Reliability, Queueing, and Computer Science Applications*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1982.
- [30] Woodside, C. M. Throughput calculation for basic Stochastic Rendezvous Networks. *Performance Evaluation*. **9**, (1988/89), 143–160.
- [31] Woodside, C. M., et al. An active-server model for the performance of parallel programs written using rendezvous. *The Journal of Systems and Software*. **6**, 1 & 2 (May 1986), 125–131.