دانشکده مهندسی کامپیوتر

عنوان درس:

طراحی نرم‌افزارهای اتکاءپذیر

(Dependable Software Design)

تحلیل با استفاده از شبکه‌های پاداش تصادفی

مدرس: محمد عبداللهی ازگمی

(Mohammad Abdollahi Azgomi)

azgomi@iust.ac.ir

Reference

- M.R. Lyu, *Software Fault Tolerance*, John Wiley & Sons (1995)
  - Chapter 6: Analyses using Stochastic Reward Nets (SRNs)
- **Outline:**
  - 1. Stochastic Reward Nets
  - 2. Fault Tolerant Software Models
  - 3. Dependencies in the SRN Models

1. 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).

- 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***.

1. Stochastic Reward Nets

- 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*.

1. Stochastic Reward Nets

- *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*.

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.
- A state of an SRN is actually a marking
  - labeled (#(P_1), #(P_2), …, #(P_n)) if there are n places in the net.

Measures

- **We label the set of all possible markings that can be reached in the net as Ω.**
- **These markings are subdivided into tangible markings Ω_T and vanishing markings Ω_V.**
- **For each tangible marking *i* in Ω_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.**

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 (بی‌نهایت)(جذب) or a given time**, and
  - **the distribution of accumulated reward** either until absorption or a given time.

Measures

- The *expected reward rate in steady state* can be computed using the steady state probability of being in each marking *i* for all $i \in \Omega_T$.
- For steady state distribution *i*, the expected reward rate is given by

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

Measures

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

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

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

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

Measures

- The *accumulated reward* in (0, *t*], *Y*(*t*), is denoted as

$$Y(t) = \int_0^t \mathcal{R}(u)du.$$

- The *expected accumulated reward in (0, t]* can be computed as

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

Measures

- The *expected accumulated reward until absorption*, labeled E[Y(∞)], can be computed as

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

Measures

- The *distribution of accumulated reward* is a measure of considerable interest.
- The distribution of accumulated reward until absorption is denoted 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.

2. 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*.

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.

Recovery Blocks

- 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.

Recovery Blocks

- 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*

Recovery Blocks

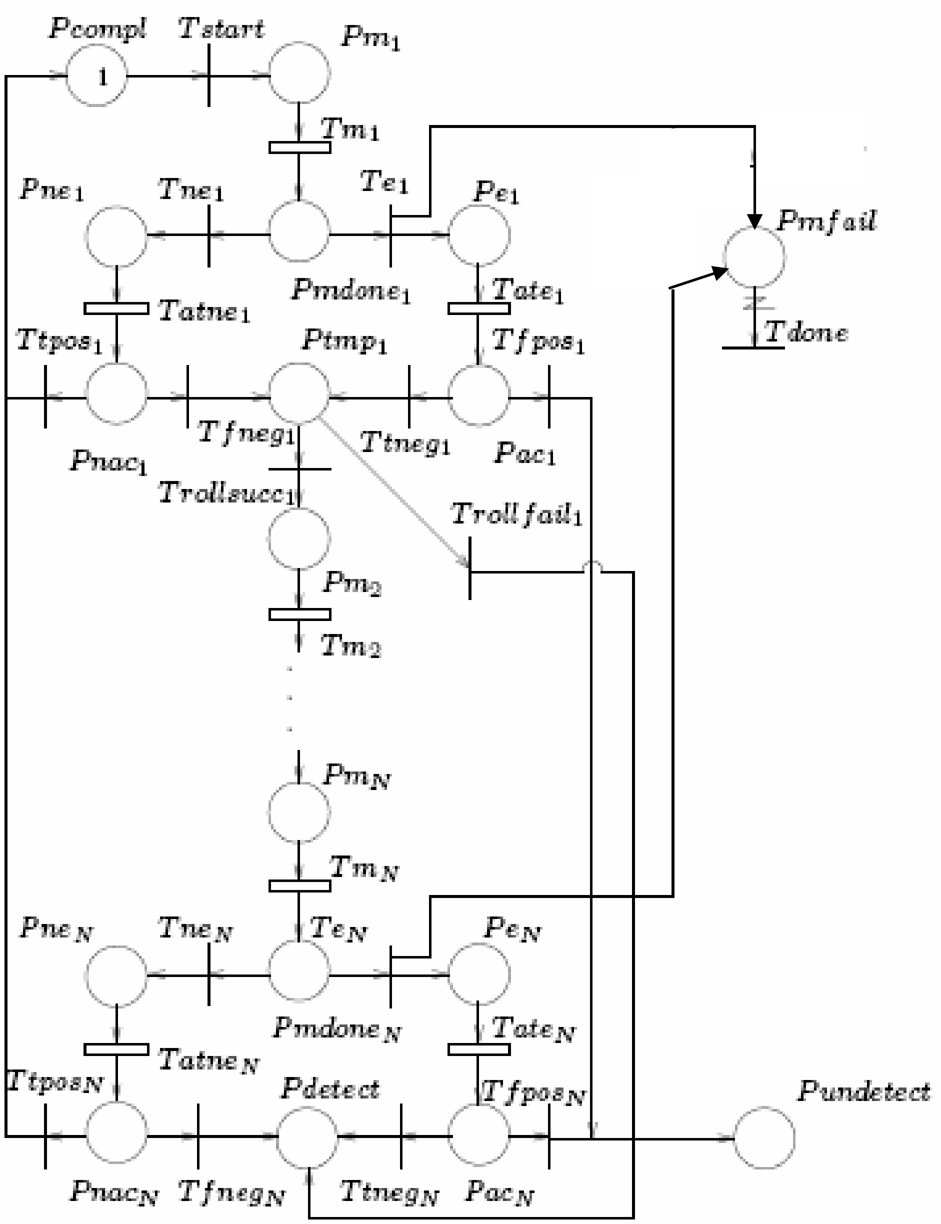
- The parameters required for a recovery block model are difficult to obtain.
- Following the categories of Pucci [Puc90, Puc92], events in the recovery blocks are classified into the following four types of events.

1. *Variant i* produces correct output which the AT accepts.
2. *Variant i* produces correct output which the AT rejects.
3. *Variant i* produces incorrect output which the AT rejects.
4. *Variant i* produces incorrect output which the AT accepts.

Recovery Blocks

- 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.

Recovery Blocks



Transition	Trans. Priority
T_{done}	HIGH

Transition	Guard Function
T_{done}	$\#(P_{compl}) > 0$

Arc	Arc Multiplicity
$P_{mfail} \rightarrow T_{done}$	$max(\#(P_{mfail}),\ 1)$

Recovery Blocks

- The net is nearly *self-explanatory* (!).
- Place P_{compl} is the starting point of the RB.
  - compl => completion
- The firing of transition T_{start}, which places a token in place P_{m_1}, indicates that the recovery block has begun executing the next (or first in this case) dataset.
- A token in place P_{m_1} indicates that the primary variant in the recovery block has begun execution on the current dataset.

Recovery Blocks

- The firing of transition T_{m_1} corresponds to the completion of the execution of the primary variant.
- Transitions T_{ne_1} and T_{e_1} correspond to the events that the output produced by the variant are correct and incorrect respectively.
- Transition T_{ne_1} moves the token from place P_{mdone_1} to place P_{ne_1} indicating that the variant produced a correct output.

Recovery Blocks

- Transition T_{e_1} moves the token from place P_{mdone_1} to both places P_{e_1} and P_{mfail}.
- A token in place P_{e_1} indicates that the first variant produced an incorrect output.
- Place P_{mfail} counts the number of variants producing an incorrect result on the current dataset; this is needed to represent common-mode failures.

Recovery Blocks

- Transition T_{atne_1} represents the execution of the AT after the variant produces a correct output.
- The immediate transitions T_{tpos_1} and T_{fneg_1}, which correspond to a correct positive diagnosis by the AT and a false negative diagnosis by the AT respectively, are then enabled.
- Transition T_{ate_1} represents the execution of the AT after the variant produces an incorrect output.

Recovery Blocks

- The immediate transitions T_{tneg1} and T_{fpos1}, 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 $P_{undetect}$ indicating an undetected block failure.
- A true positive AT diagnosis causes the token to be moved to place P_{compl} indicating the block has completed execution on the current dataset.
- The block then begins operation on the next dataset.

Recovery Blocks

- If an error is discovered, represented by the firing of either T_{fneg_1} and T_{tneg_1}, the system initiates a rollback recovery action.
- Transition $T_{rollsucc_1}$ represents a successful rollback.
- Transition $T_{rollfail_1}$ represents an unsuccessful rollback resulting in an RB failure.
- The output arc from $T_{rollsucc_1}$ leads to P_{m2}, the starting place of the rst alternate variant, while the output arc from $T_{rollfail_1}$ leads to P_{detect} which represents a detected RB failure.

Recovery Blocks

- 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 T_{fneg_N} and T_{tneg_N} lead to place P_{detect}.

Recovery Blocks

- When the recovery block completes (the token is returned to place P_{m1}) then transition T_{done} fires and all tokens are removed from place P_{mfail} for the next execution of the recovery block.

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 (نتیجه غالب) 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.

N-Version Programming

- In Figure 6.2, an SRN model of an N-version programming system is shown…

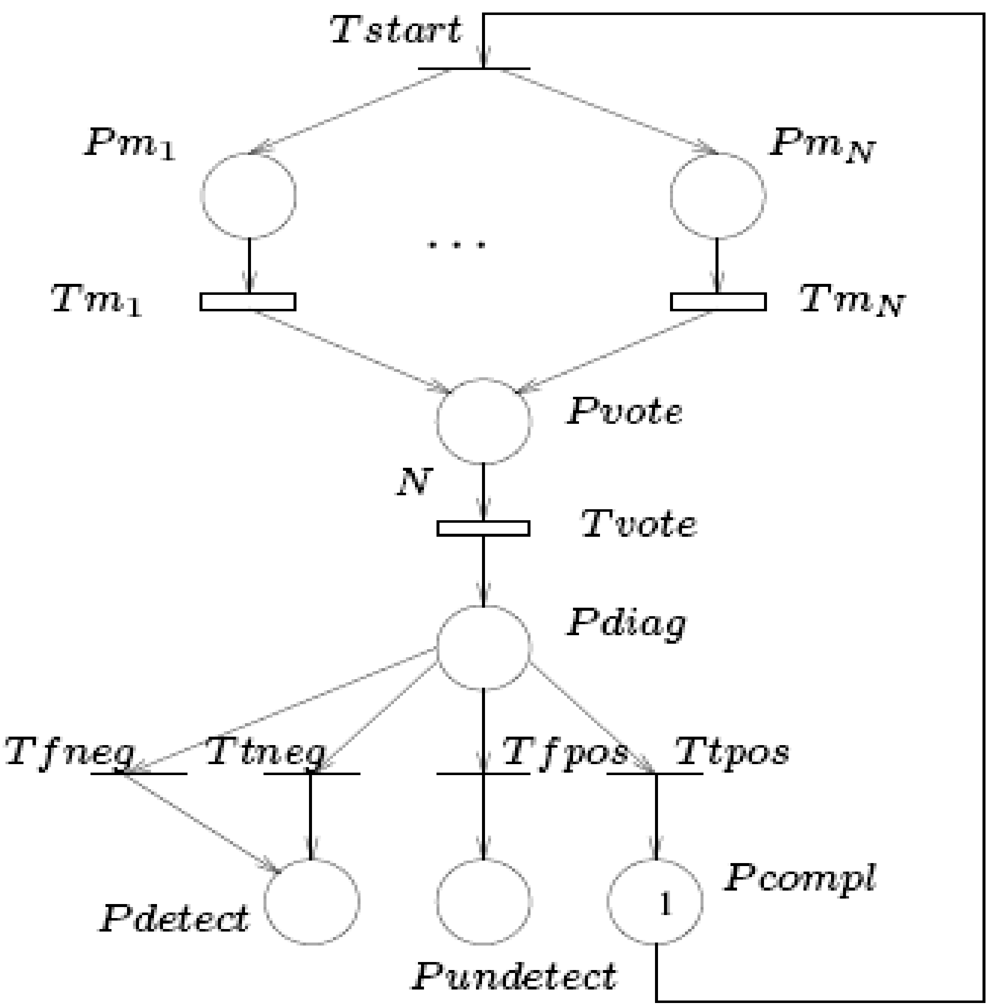


Figure 6.2 SRN model of N-version programming

N-Version Programming

- Initially, a single token is in place P_{compl}.
- The software block begins operating on the next input immediately with the firing of transition T_{start}.
- This transition places one token in each of the places P_{m_1}, P_{m_2}, ..., P_{m_N}, representing the fact that each variant begins operation on the provided input.
- Transitions T_{m_i} for $i \in 1, 2, \ldots, N$ represents the completion of execution of variant i.

N-Version Programming

- When all variants have completed, place P_{vote} will contain N tokens, enabling transition T_{vote}.
- Transition T_{vote} represents the execution of the voting mechanism.
- When voting is complete, a single token is moved to place P_{diag} 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 T_{tneg}, or a false positive, represented by the firing of transition T_{fpos}.

N-Version Programming

- 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 T_{tpos}, or a false negative, represented by the firing of transition T_{fneg}.
- If the voting result is negative, either transition T_{tneg} or transition T_{fneg} fire, moving the token to place P_{detect}.
  - **This represents a detected error.**

N-Version Programming

- If the voting result is a false positive, transition T_{fpos} fires, moving the token to place $P_{undetect}$, **indicating an undetected error.**

- If the voting result is a true positive, transition T_{tpos} fires, moving the token to place P_{compl}, indicating the software block as successfully completed execution on the current dataset.

- Once the token is returned to place P_{compl}, the N-version programming block begins operating on the next input.

N Self-Checking Programming

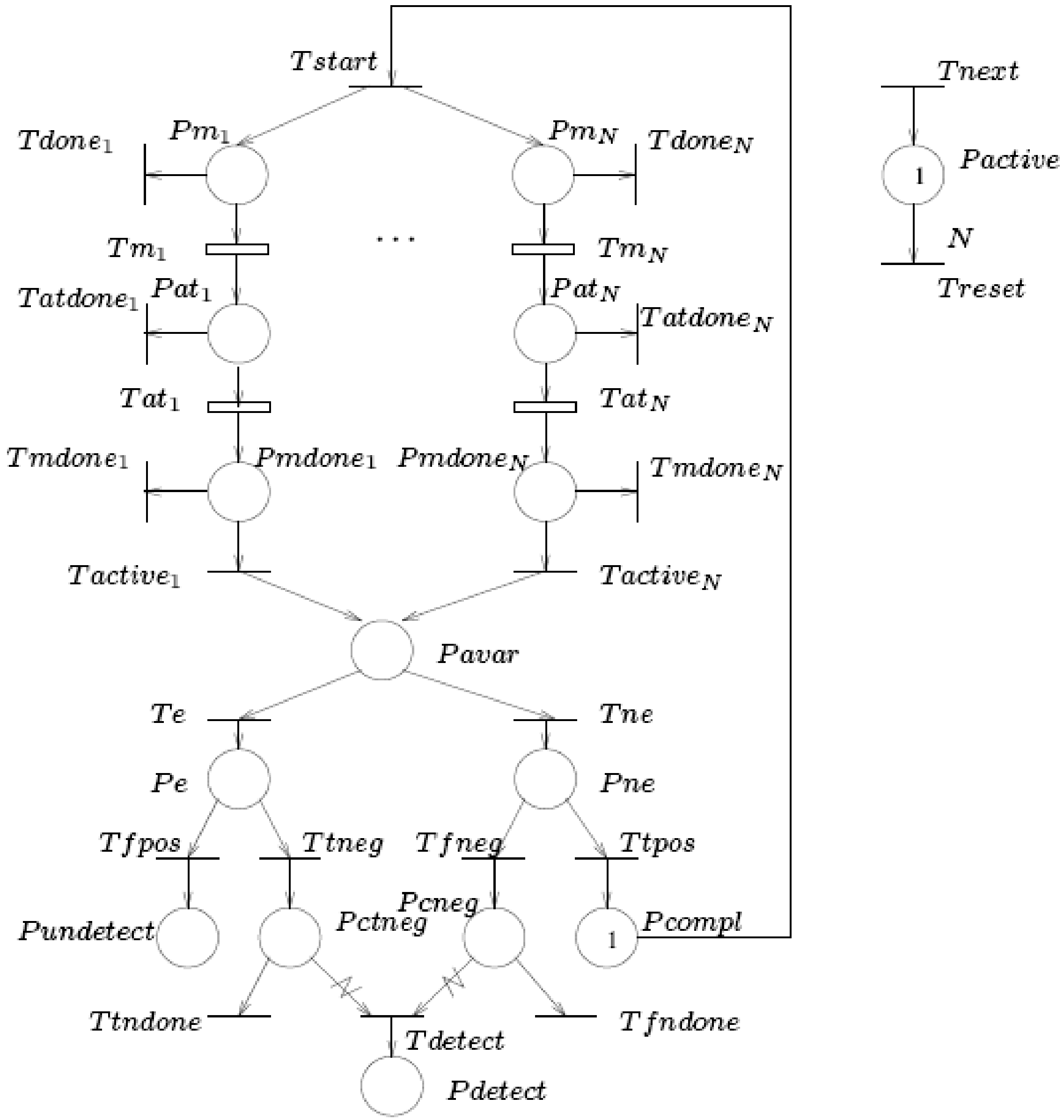
- In N *self-checking programming (NSCP)*, all variants operate in parallel on the same input.
- There are two possible decision mechanisms [Lap90] using this technique:
  - The first possibility is that each variant has its own **acceptance test.**
  - The second possibility is that each pair of variants is compared using a **comparison algorithm** associated with the pair.

N Self-Checking Programming

- 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 (به‌طور مستقل، طبق تنوع طراحی حاصل شده باشند).
- 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**.

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.



Tstart
Tnext
Pactive
1
N
Treset
Pm_1
Pm_N
$Tdone_1$
$Tdone_N$
. . .
Tm_1
Tm_N
$Tatdone_1$
Pat_1
Pat_N
$Tatdone_N$
Tat_1
Tat_N
$Tmdone_1$
$Pmdone_1$
$Pmdone_N$
$Tmdone_N$
$Tactive_1$
$Tactive_N$
Pavar
Te
Tne
Pe
Pne
Tfpos
Ttneg
Tfneg
Ttpos
Pcneg
Pundetect
Pctneg
Pcompl
1
Ttndone
Tdetect
Tfndone
Pdetect

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
$Pctneg \rightarrow Tdetect$	$\#(Pctneg)$
$Pcfneg \rightarrow Tdetect$	$\#(Pcfneg)$
$Pctneg \rightarrow Ttndone$	$max(\#(Pctneg), 1)$
$Pcfneg \rightarrow Tfndone$	$max(\#(Pcfneg), 1)$

NSCP with Acceptance Test

- Initially, there is a single token in both places P_{compl} 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 *Pm1*, *Pm2*, ..., *PmN*, representing the fact that each variant begins operation on the provided input.

NSCP with Acceptance Test

- Transitions *Tmi* for $i \in$ 1, 2, …, *N* represents the execution of each variant *i*.
- As each variant completes execution, a token is placed in *Pati* (for variant *i*).
- The self-checking procedure (acceptance test execution) is modeled by transition *Tati*.
- When the acceptance test completes, the token is moved from place *Pati* to place *Pmdonei* (for variant *i*).

NSCP with Acceptance Test

- 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 *Pmdonei* enabling transition *Tactivei*, where i is the number of tokens in place *Pactive*.
- The firing of transition *Tactivei* moves the token to place *Pavar*.

NSCP with Acceptance Test

- 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.

NSCP with Acceptance Test

- 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.

NSCP with Acceptance Test

- 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.

NSCP with Acceptance Test

- 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.

NSCP with Acceptance Test

- 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.

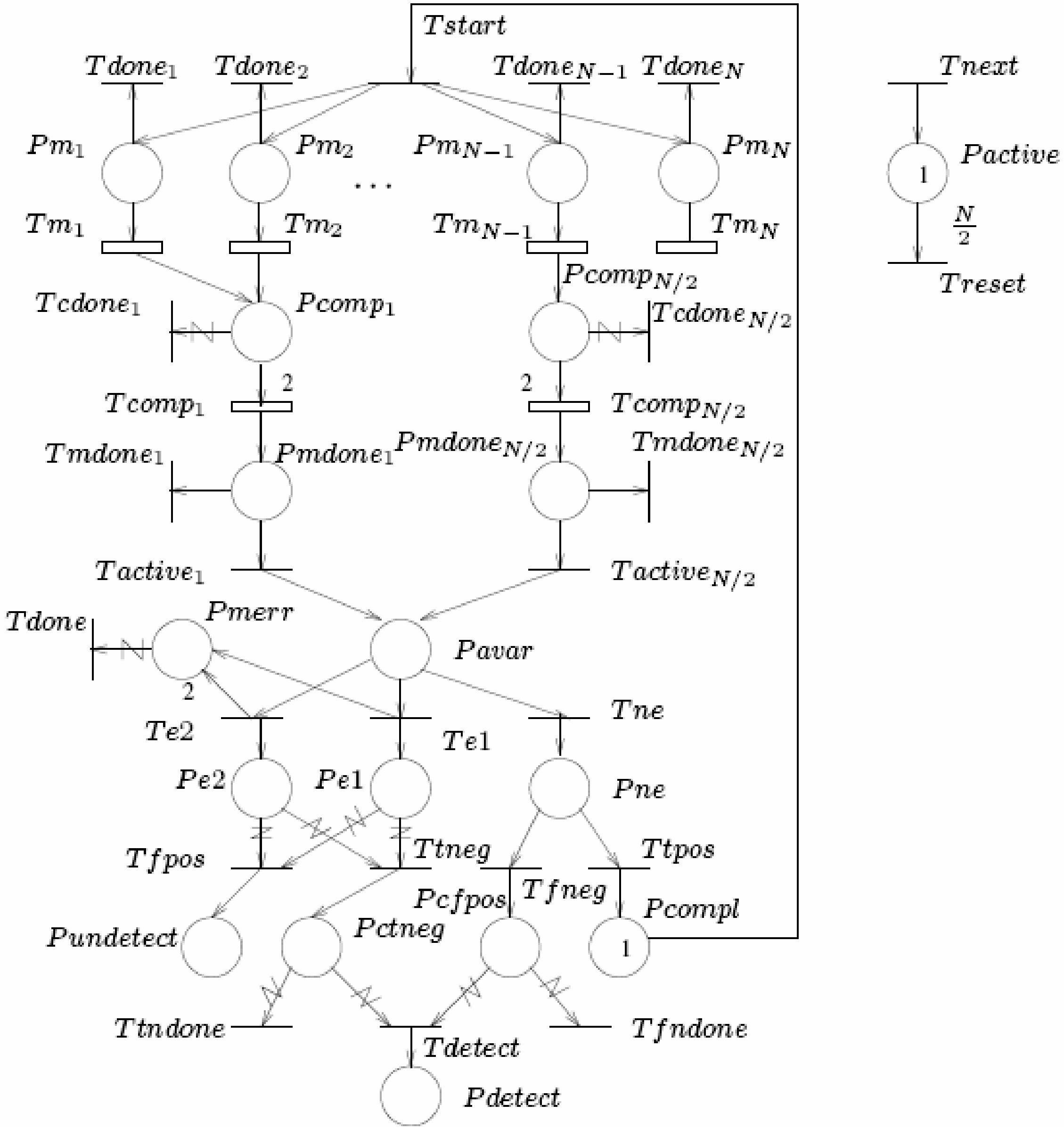
- If the diagnosis of the correct output is a true positive, transition *Ttpos* fires, moving the token from place *Pne* to place *Pcompl*.

NSCP with Acceptance Test

- A token in place *Pcompl* satises the guard function for transitions *Tdonei*, *Tatdonei*, 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.

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.


Tstart
Tdone1
Tdone2
TdoneN−1
TdoneN
Tnext
Pm1
Pm2
PmN−1
PmN
Pactive
1
N/2
Tm1
Tm2
TmN−1
TmN
Treset
PcompN/2
Tcdone1
Pcomp1
TcdoneN/2
2
Tcomp1
TcompN/2
Tmdone1
Pmdone1
PmdoneN/2
TmdoneN/2
Tactive1
TactiveN/2
Tdone
Pmerr
Pavar
2
Tne
Te2
Te1
Pe2
Pe1
Pne
Tfpos
Ttneg
Ttpos
Pcfpos
Tfneg
Pcompl
Pundetect
Pctneg
1
Ttndone
Tdetect
Tfndone
Pdetect

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

NSCP with Comparison Algorithm

- 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 *Pm1*, *Pm2*, ..., *PmN*, representing the fact that each variant begins operation on the provided input.

NSCP with Comparison Algorithm

- Transitions *Tmi* for i ∈ 1, 2, …, *N* represents the execution of each variant *i*.
- As each variant completes execution, a token is placed in $T_{comp\lceil i/2 \rceil}$ (for variant *i*).
- The self-checking procedure (comparison algorithm execution) is modeled by transition *Tcompi*.
- This transition is enabled only when both variants in a pair have completed execution (when two tokens are in place *Pcompi*).

NSCP with Comparison Algorithm

- When the comparison test completes, the token is moved from place *Pcompi* to place *Pmdonei* (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).

NSCP with Comparison Algorithm

- When the active variant pair completes its comparison algorithm, a token is in place *Pmdonei*, enabling transition *Tactivei*, where *i* is the number of tokens in place *Pactive*.
- The firing of transition *Tactivei* moves the token to place *Pavar*.
- This enables transitions *Te1*, *Te2*, and *Tne*.

NSCP with Comparison Algorithm

- 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.

NSCP with Comparison Algorithm

- 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.

NSCP with Comparison Algorithm

- 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.

NSCP with Comparison Algorithm

- 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.

NSCP with Comparison Algorithm

- 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.

NSCP with Comparison Algorithm

- 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.

NSCP with Comparison Algorithm

- 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*.

NSCP with Comparison Algorithm

- A token in place *Pcompl* satises the guard function for transitions *Tdonei* for i ∈ [1, *N*], transitions *Tcdonei* and *Tmdonei* for i ∈ [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.

3. 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:
  - ***Separate* or *common-mode***
    - A *common-mode fault* is a fault which occurs simultaneously in two or more redundant components.
  - ***Detected* or *undetected***

3. Dependencies in The SRN Models

- **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*.
- در ادامه تعریف خواهند شد...
- **Next, failures can either be *detected* or *undetected*.**

3. Dependencies in The SRN Models

- 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 (قابل قضاوت) definition of the immediate transition probabilities.

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 P_{detect}, for detected failures, and $P_{undetect}$, 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.

Detected versus Undetected Failures

- **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 $P_{undetect}$.
  - 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.

Detected versus Undetected Failures

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

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.

Common-Mode versus Separate Failures

- **Measurements have shown that software variants do not exhibit separate failures.**
  - توضیحی که چرا این طور است نداده و مرجعی هم برای اندازه‌گیریهای مورد نظر نداده است.
- Measurements provide a **probability mass function** $p_N(.)$ where $p_N(i)$ is the probability that i of the N variants produce incorrect output.
- If all variant failures are separate, then $p_N(.)$ is a **binomial** (دوجمله‌ای) probability mass function.

Common-Mode versus Separate Failures

- 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.**

Common-Mode versus Separate Failures

- 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.

Common-Mode versus Separate Failures

- 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 $p_N(.)$ each time a new dataset begins processing.

Common-Mode versus Separate Failures

- 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

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

Common-Mode versus Separate Failures

- 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.

Common-Mode versus Separate Failures

- 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

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

Common-Mode versus Separate Failures

- The probability both variants produce **incorrect** output is

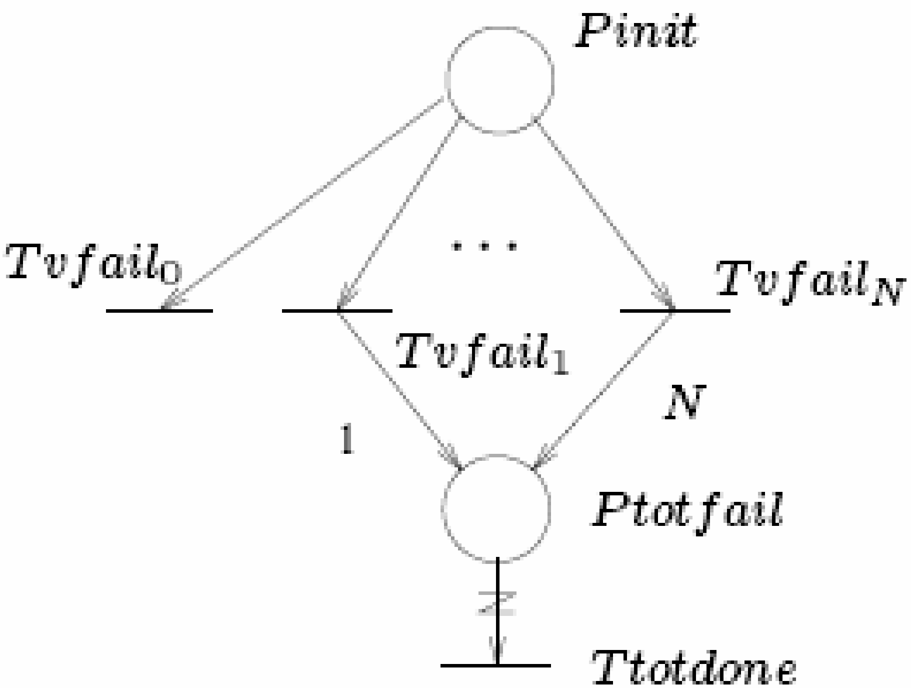
$$prob(\text{both variants fail}) = \frac{n_{totfail} - n_{fail}}{N - n_{vdone}} \times \frac{n_{totfail} - (n_{fail} + 1)}{N - (n_{vdone} + 1)}$$

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

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

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…



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

SRN Models with Common-Mode and Separate Failures

- A token arrives in transition P_{init} as a result of the firing of transition T_{start}.
- Transition *Tstart* (which is part of each previously developed SRN model) is modified to include another output arc associated with place P_{init}.
- When a token arrives in place P_{init}, the software block is ready to operate on a new dataset.
- Transitions $Tvfail_0$, $Tvfail_1$, … $Tvfail_N$ become enabled.

SRN Models with Common-Mode and Separate Failures

- The probability associated with each transition *Tvfaili* 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.

SRN Models with Common-Mode and Separate Failures

- 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.

Recovery Block SRN Model

- In the SRN model of the RB scheme, transitions T_{nei} and T_{ei} 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, #(P_{mfail}) produced incorrect results.

Recovery Block SRN Model

- The immediate transition probabilities for T_{ei} and T_{nei} incorporating common-mode failures are given by

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

and

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

NVP SRN Model

- In the SRN model of the NVP scheme, out of the N available variants $\#(P_{totfail})$ 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 $\#(P_{totfail}) < N/2$), 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.

NVP SRN Model

- 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.

NVP SRN Model

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

NVP SRN Model

- The variant probabilities used in the above equations are given by

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

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

NSCP with AT SRN Model

- In the SRN model of the NSCP with acceptance test, a token in place P_{avar} enables transitions T_e, which indicates an incorrect variant result, and transition T_{ne}, which indicates a correct variant result.
- The number of previously completed and diagnosed variants is $\#(P_{ctneg})+\#(P_{cfneg})$.
- The number of these variants which produced incorrect results is $\#(P_{ctneg})$.

NSCP with AT SRN Model

- The probability that the variant represented by a token in place P_{avar} produces an incorrect result is

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

and

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

NSCP with Comparison 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.

NSCP with Comparison SRN Model

- 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×(#(*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*).

NSCP with Comparison SRN Model

- The probability neither variant produces incorrect output is

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

NSCP with Comparison SRN Model

- The probability both variants produce incorrect output is

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

NSCP with Comparison SRN Model

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

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