Session 2: Stochastic Activity Network Concepts

Performability Engineering Research Group
Information Trust Institute
Department of Electrical and Computer Engineering and
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

http://www.perform.csl.uiuc.edu

Session Goals

- Understand the workings and limitations of stochastic Petri nets.
- Understand the basics of stochastic activity networks, and how they work.
- Become familiar with looking at and understanding small SAN models.
- Understand reward structures and reward variables.
- Understand composed models and what they are appropriate for.
- Be prepared to build your first SAN model.

Session Outline

- Stochastic Petri nets
  - Places, tokens, input / output arcs, transitions
  - Readers / Writers example
- Stochastic activity networks
  - Input / output gates, cases, instantaneous and timed activities
  - Marking dependent behavior, well-specified, general distributions
  - Simple database server model
- Reward variables
  - Reward structures
  - Reward variable classification
  - Predicate / function implementation in *Mobius*
- Fault-tolerant computer example
- Composed models
  - Fault-tolerant computer revisited

Introduction

Stochastic activity networks, or SANs, are a convenient, graphical, high-level language for describing system behavior. SANs are useful in capturing the stochastic (or random) behavior of a system.

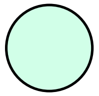
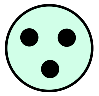
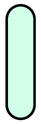
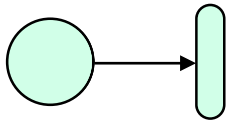
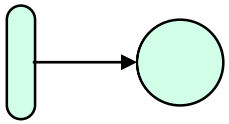
Examples:

- The amount of time a program takes to execute can be computed precisely if all factors are known, but this is nearly impossible and sometimes useless. At a more abstract level, we can approximate the running time by a random variable.
- Fault arrivals almost always must be modeled by a random process.

We begin by describing a subset of SANs: stochastic Petri nets.

Stochastic Petri Net Review

One of the simplest high-level modeling formalisms is called *stochastic Petri nets*. A stochastic Petri net is composed of the following components:

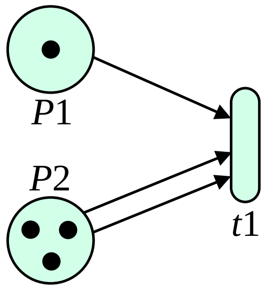
- Places:  which contain tokens, and are like variables
- tokens:  which are the "value" or "state" of a place
- transitions:  which change the number of tokens in places
- input arcs:  which connect places to transitions
- output arcs:  which connect transitions to places

Firing Rules for SPNs

A stochastic Petri net (SPN) executes according to the following rules:

- A transition is said to be *enabled* if for each place connected by input arcs, the number of tokens in the place is ≥ the number of input arcs connecting the place and the transition.

Example:

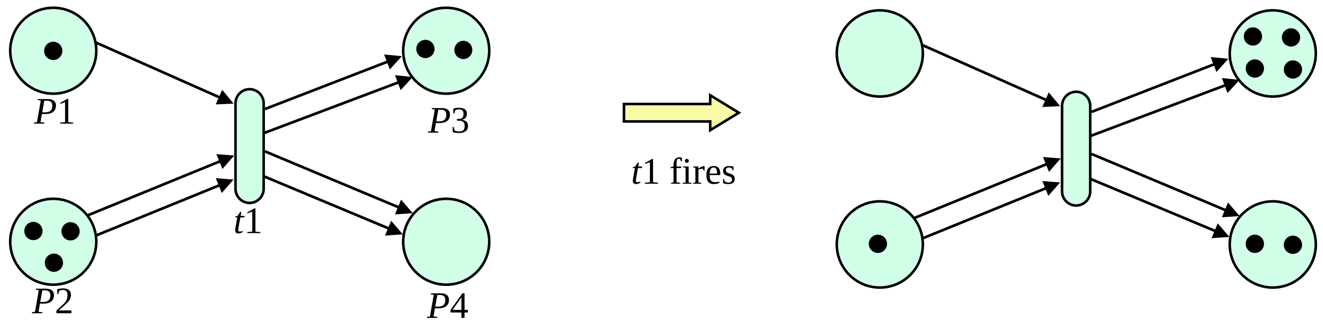


Transition *t*1 is enabled.

Firing Rules, cont.

- A transition may *fire* if it is enabled. (More about this later.)
- If a transition fires, for each input arc, a token is removed from the corresponding place, and for each output arc, a token is added to the corresponding place.

Example:



Note: tokens are not necessarily conserved when a transition fires.

Specification of Stochastic Behavior of an SPN

- A stochastic Petri net is made from a Petri net by
  - Assigning an exponentially distributed time to all transitions.
  - Time represents the “delay” between enabling and firing of a transition.
  - Transitions “execute” in parallel with independent delay distributions.

- Since the minimum of multiple independent exponentials is itself exponential, time between transition firings is exponential.

- If a transition *t* becomes enabled, and before *t* fires, some other transition fires and changes the state of the SPN such that *t* is no longer enabled, then *t* *aborts*, that is, *t* will not fire.

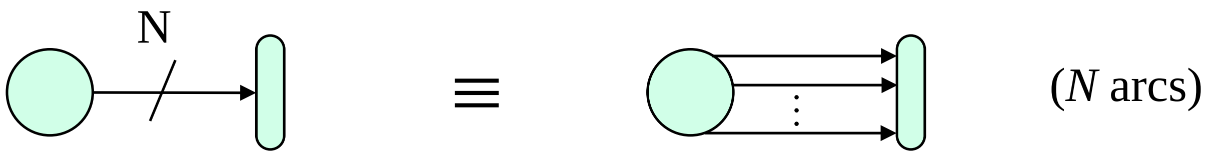
- Since the exponential distribution is memoryless, one can say that transitions that remain enabled continue or restart, as is convenient, without changing the behavior of the network.

SPN Example: Readers/Writers Problem

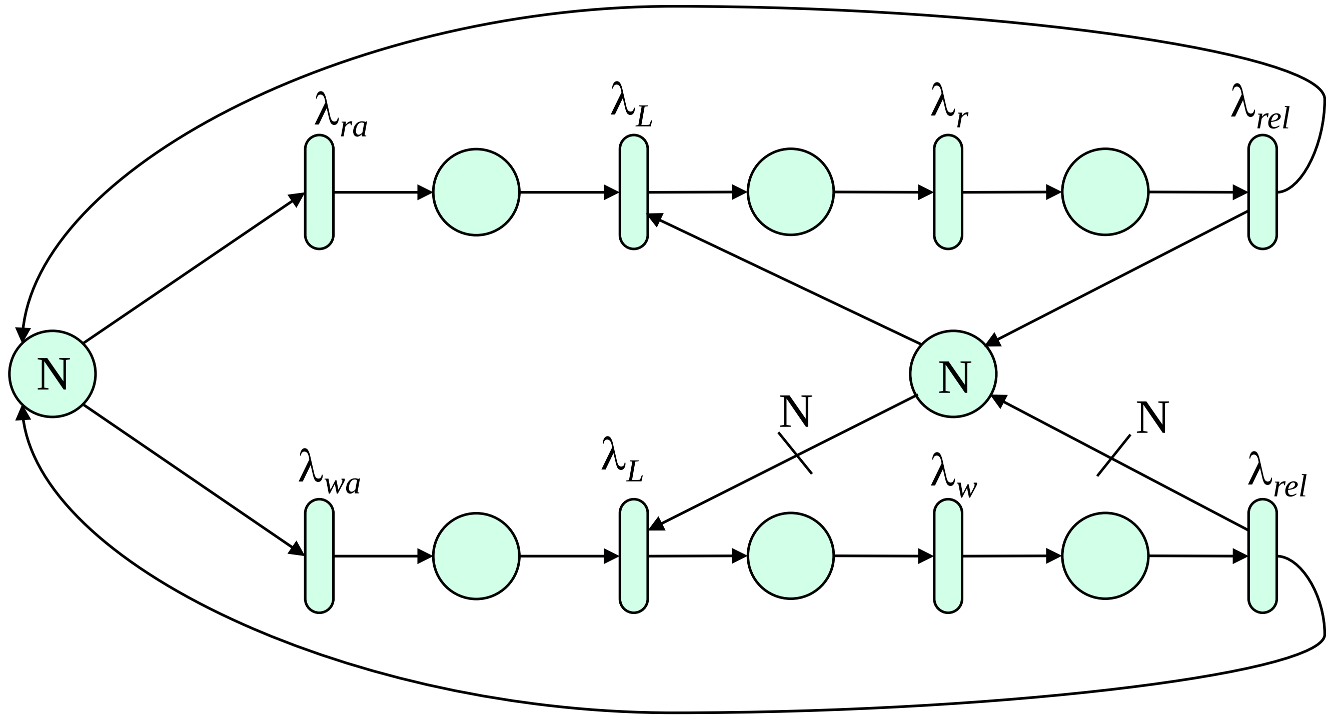
There are at most N requests in the system at a time. Read requests arrive at rate λ_{ra}, and write requests at rate λ_{wa}. Any number of readers may read from a file at a time, but only one writer may write at a time. A reader and writer may not access the file at the same time.

Locks are obtained with rate λ_L (for both read and write locks); reads and writes are performed at rates λ_r and λ_w respectively. Locks are released at rate λ_{rel}.

Note:



SPN Representation of Reader/Writers Problem



Notes on SPNs

- SPNs are much easier to read, write, modify, and debug than Markov chains.
- SPN to Markov chain conversion can be automated to afford numerical solutions to Markov chains.
- Most SPN formalisms include a special type of arc called an *inhibitor arc*, which enables the SPN if there are zero tokens in the associated place, and the identity (do nothing) function. Example: modify SPN to give writes priority.
- Limited in their expressive power: may only perform +, -, >, and test-for-zero operations.
- These very limited operations make it very difficult to model complex interactions.
- Simplicity allows for certain analysis, e.g., a network protocol modeled by an SPN may detect deadlock (if inhibitor arcs are not used).
- More general and flexible formalisms are needed to represent real systems.

Stochastic Activity Networks

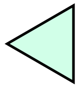
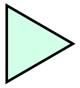
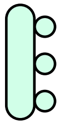
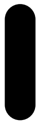
The need for more expressive modeling languages has led to several extensions to stochastic Petri nets. One extension that we will examine is called *stochastic activity networks*. Because there are a number of subtle distinctions relative to SPNs, stochastic activity networks use different words to describe ideas similar to those of SPNs.

Stochastic activity networks have the following properties:

- A general way to specify that an activity (transition) is enabled
- A general way to specify a completion (firing) rule
- A way to represent zero-timed events
- A way to represent probabilistic choices upon activity completion
- State-dependent parameter values
- General delay distributions on activities

SAN Symbols

Stochastic activity networks (hereafter SANs) have four new symbols in addition to those of SPNs:

- Input gate:  used to define complex enabling predicates and completion functions
- Output gate:  used to define complex completion functions
- Cases:  (small circles on activities) used to specify probabilistic choices
- Instantaneous activities:  used to specify zero-timed events

SAN Enabling Rules

An input gate has two components:

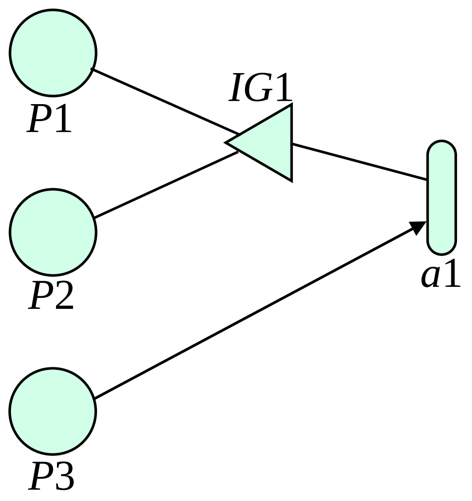
- enabling_function (state) $\rightarrow$ boolean; also called the *enabling predicate*
- input_function(state) $\rightarrow$ state; rule for changing the state of the model

An activity is *enabled* if for every connected input gate, the enabling predicate is true, and for each input arc, the number of tokens in the connected place $\geq$ number of arcs.

We use the notation $MARK(P)$ to denote the number of tokens in place P.

Example SAN Enabling Rule

Example:



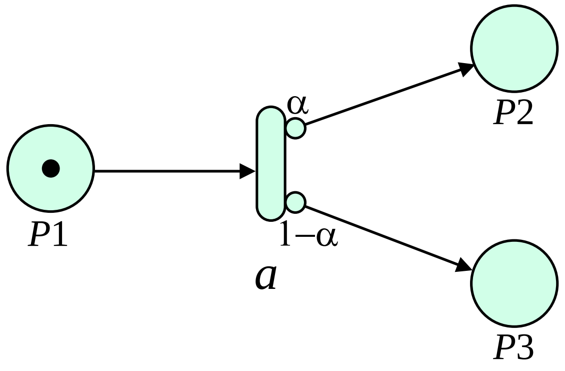
IG1 Predicate:

```
        if((MARK(P1)>0 && MARK(P2)==0)||
              (MARK(P1)==0 && MARK(P2)>0))
                    return 1;
        else return 0;
```

Activity *a*1 is enabled if *IG*1 predicate is true (1) and *MARK*(*P*3) > 0.
(Note that in *Mobius*, “1” is used to denote true.)

Cases

Cases represent a probabilistic choice of an action to take when an activity completes.



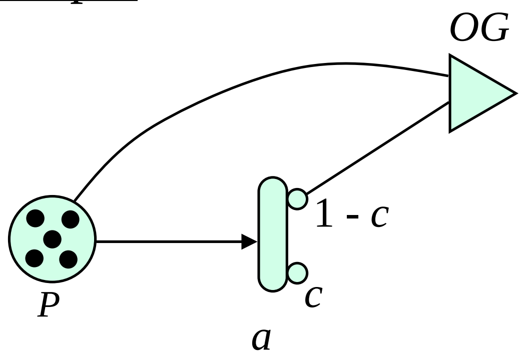
When activity *a* completes, a token is removed from place *P*1, and with probability α, a token is put into place *P*2, and with probability 1 - α, a token is put into place *P*3.

Note: cases are numbered, starting with 1, from top to bottom.

Output Gates

When an activity completes, an output gate allows for a more general change in the state of the system. This output gate function is usually expressed using pseudo-C code.

<u>Example</u>

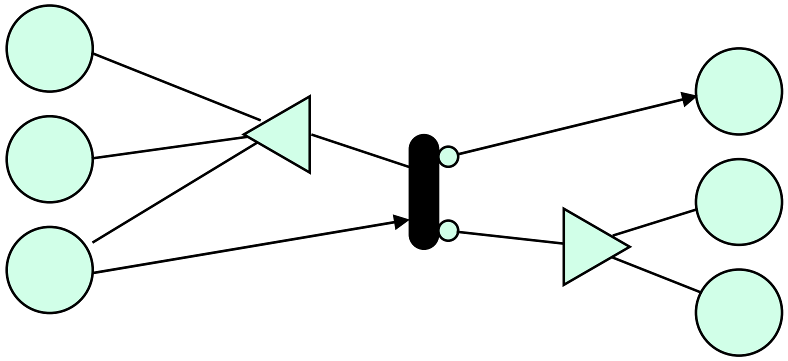


<u>OG Function</u>

```
MARK(P) = 0;
```

Instantaneous Activities

Another important feature of SANs is the instantaneous activity. An *instantaneous activity* is like a normal activity except that it completes in zero time after it becomes enabled. Instantaneous activities can be used with input gates, output gates, and cases.



Instantaneous activities are useful when modeling events that have an effect on the state of the system, but happen in negligible time, with respect to other activities in the system, and the performance/dependability measures of interest.

SAN Terms

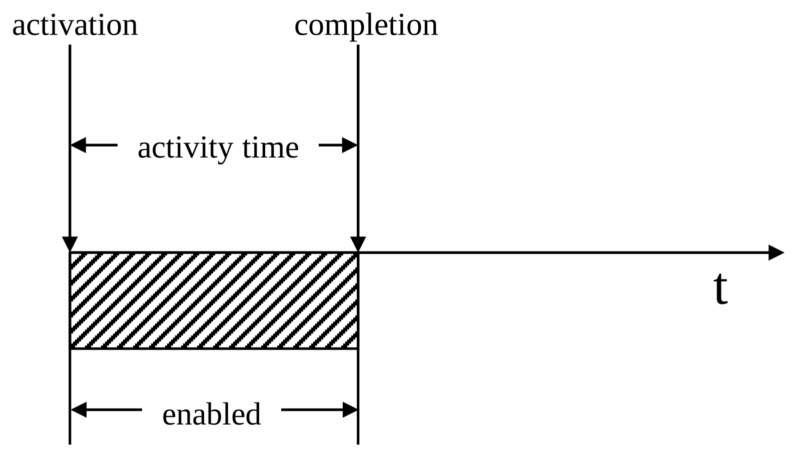
1. *activation* - time at which an activity begins

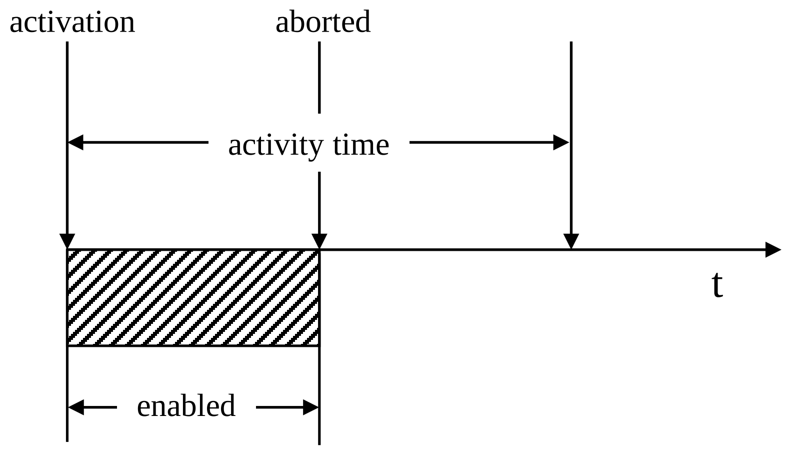
2. *completion* - time at which activity completes

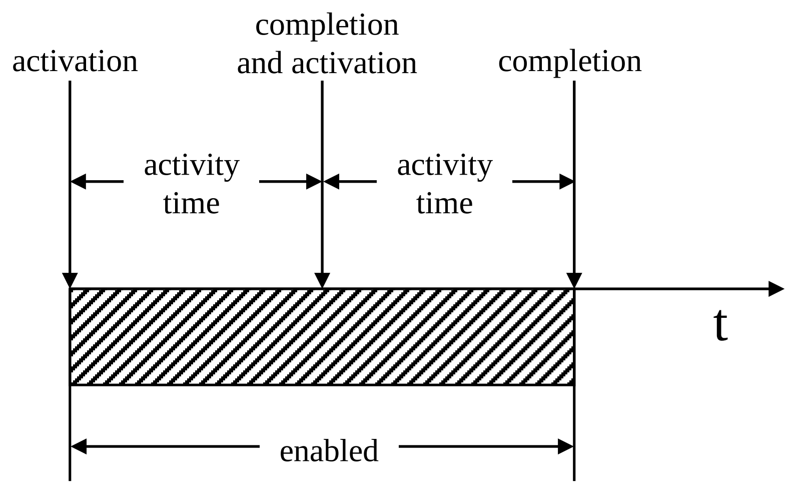
3. *abort* - time, after activation but before completion, when activity is no longer enabled

4. *active* - the time after an activity has been activated but before it completes or aborts.

Illustration of SAN Terms







Completion Rules

When an activity *completes*, the following events take place (in the order listed), possibly changing the marking of the network:

1. If the activity has cases, a case is (probabilistically) chosen.
2. The functions of all the connected input gates are executed (in an unspecified order).
3. Tokens are removed from places connected by input arcs.
4. The functions of all the output gates connected to the chosen case are executed (in an unspecified order).
5. Tokens are added to places connected by output arcs connected to the chosen case.

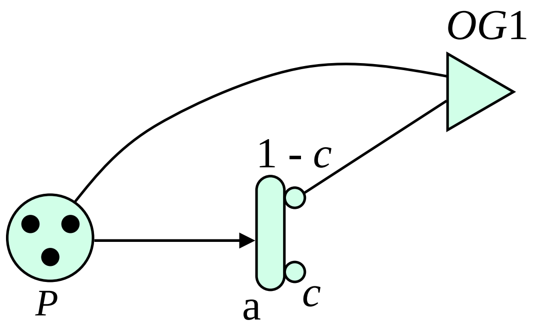
Ordering is important, since effect of actions can be marking-dependent.

Marking Dependent Behavior

Virtually every parameter may be any function of the state of the model. Examples of these are

- rates of exponential activities
- parameters of other activity distributions
- case probabilities

An example of this usefulness is a model of three redundant computers where the coverage (probability that a single computer crashing crashes the whole system) increases after a failure.

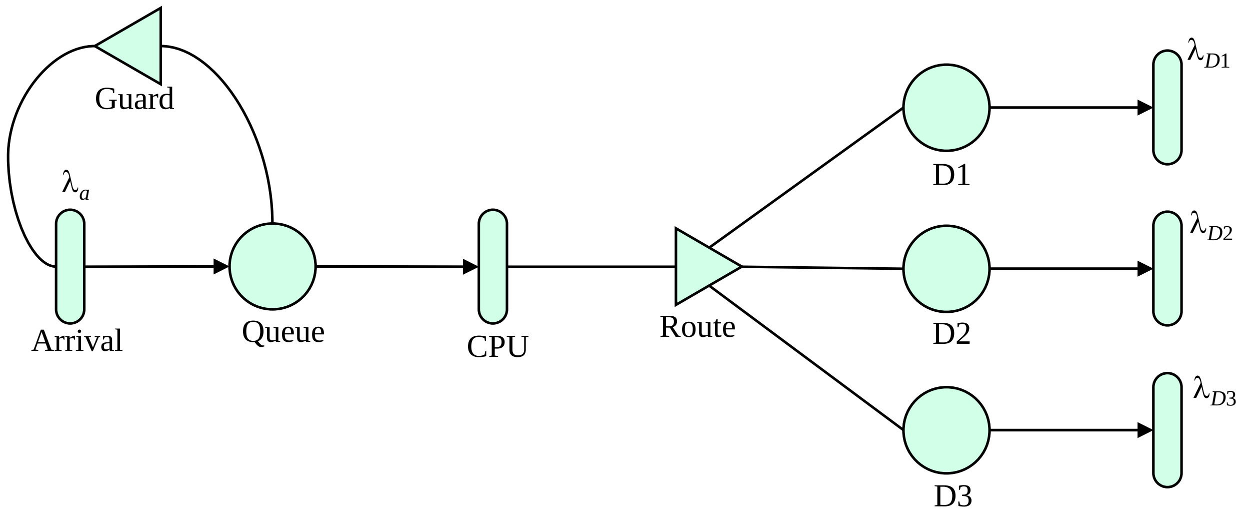


	a
case 1	0.1 + 0.02 * MARK(P)
case 2	0.9 – 0.02 * MARK(P)

Example Problem

- A database server is composed of a compute server and three file servers, and can queue up to N_c requests at a time (including the one in service).
- Requests arrive at rate λ_a and spend on average $1/\lambda_{\mathrm{CPU}}$ time at the compute server being processed.
- The request is then forwarded to the file server that has the fewest outstanding requests.
- Requests are processed at a rate of λ_{D1}, λ_{D2}, and λ_{D3} for file servers D1, D2, and D3 respectively.
- File server buffers may hold at most N_f requests (including requests in service); if all buffers are full, the request is discarded.

SAN Representation of Example Database Problem



Gate Functions for SAN

Gate	*Definition*

Guard	Predicate
	  MARK(Queue) < GLOBAL_S(Nc)
	Function
	  ;

Gate	*Definition*

Route	Function
	if (MARK(D1) == GLOBAL_S(Nf) &&
	  MARK(D2) == GLOBAL_S(Nf) &&
	  MARK(D3) == GLOBAL_S(Nf))
	    return;
	if (MARK(D1) < MARK(D2)) {
	  if (MARK(D1) < MARK(D3)) {
	    MARK(D1)++;
	  } else {
	    MARK(D3)++;
	  }
	} else {
	  if (MARK(D2) < MARK(D3)) {
	    MARK(D2)++;
	  } else {
	    MARK(D3)++;
	  }
	}

General Delay Distributions

Frequently, an exponential delay distribution is an acceptable approximation, but when more accuracy is required (and available), you may want to model using different delay distributions.

Example: The life of a rechargeable battery is often modeled using a Weibull distribution.

$$F_X(t) = 1 - e^{-\left(\frac{t}{\beta}\right)^{\alpha}},$$

where parameters α and β are found through measurement.

Recall: If an activity becomes disabled, when it later becomes enabled, it will start over.

General Delay Distributions, cont.

- SANs (and their implementation in *Mobius*) support many activity time distributions, including:

• Exponential	• Erlang
• Hyperexponential	• Gamma
• Deterministic	• Beta
• Weibull	• Uniform
• Conditional Weibull	• Binomial
• Normal	• Negative Binomial

- All distribution parameters can be marking-dependent
- The obvious implication of general delay distributions is that there is no conversion to a CTMC. Hence, no solutions to CTMCs are applicable. However, simulation is still possible.
- Analytical/numerical solution is possible for certain mixes of exponential and deterministic activities. See the *Mobius* manual for details.
- See [Kececioglu 91], for example, for appropriate use of some of these distributions.

Reward Variables

Reward variables are a way of measuring performance- or dependability-related characteristics about a model.

Examples:

- Expected time until service
- System availability
- Number of misrouted packets in an interval of time
- Processor utilization
- Length of downtime
- Operational cost
- Module or system reliability

Reward Structures

Reward may be "accumulated" two different ways:

- A model may be in a certain state or states for some period of time, for example, "CPU idle" states. This is called a *rate reward*.
- An activity may complete. This is called an *impulse reward*.

The reward variable is the sum of the rate reward and the impulse reward structures.

Reward Structure Example

A web server failure model is used to predict profits. When the web server is fully operational, profits accumulate at \$$N$/hour. In a degraded mode, profits accumulate at $\frac{1}{6}N$/hour. Repairs cost \$$K$.

$$R(m) = \begin{cases} N & m \text{ is a fully functioning marking} \\ \frac{1}{6}N & m \text{ is a degraded-mode marking} \\ 0 & \text{otherwise} \end{cases}$$

$$C(a) = \begin{cases} -K & a \text{ is an activity representing repair} \\ 0 & \text{otherwise} \end{cases}$$

By carefully integrating the reward structure from 0 to t, we get the profit at time t. This is an example of an "interval-of-time" variable.

Reward Variables

A *reward variable* is the sum of the impulse and rate reward structures over a certain time.

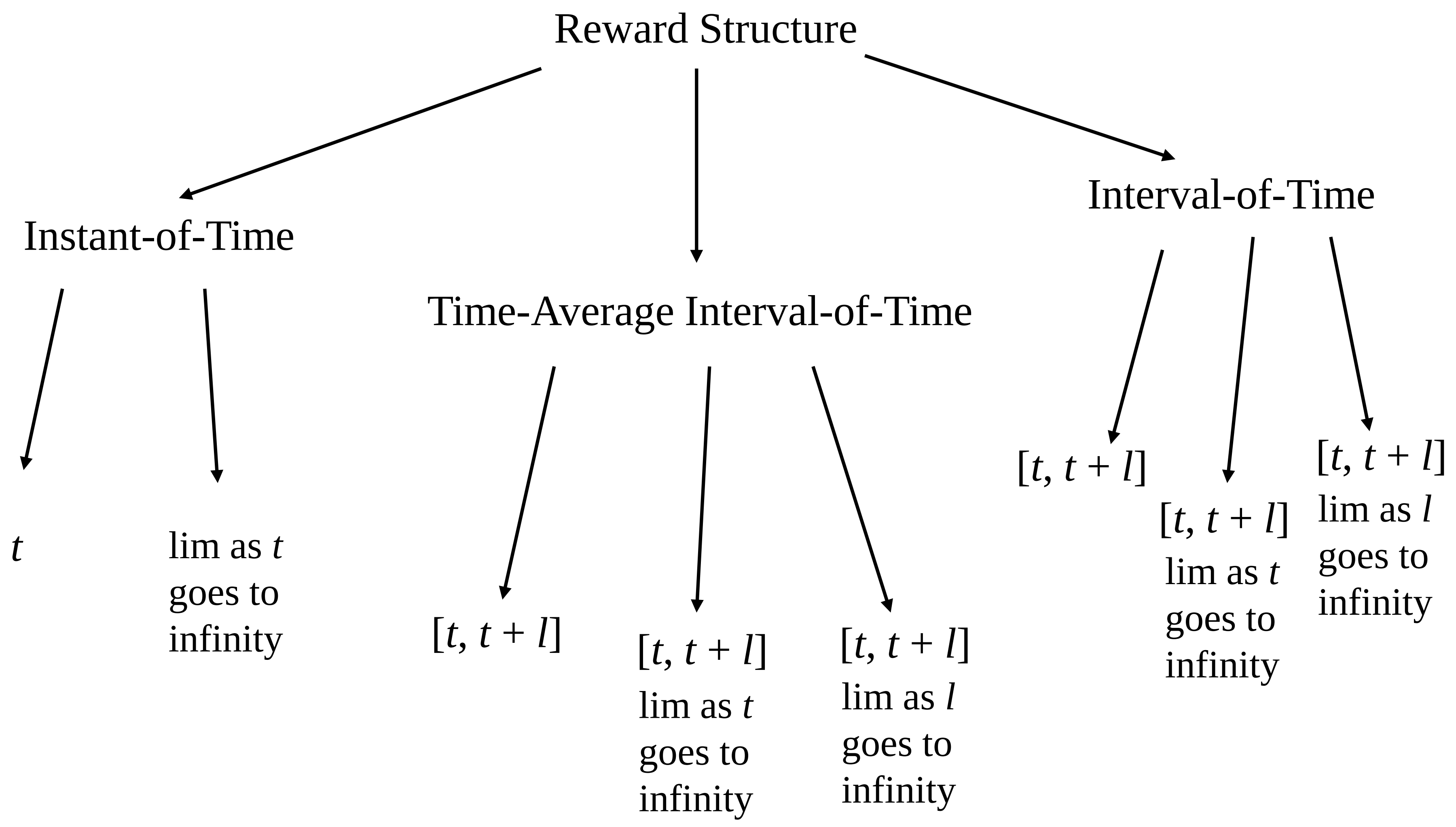
Let $[t, t + l]$ be the interval of time defined for a reward variable:

– If l is 0, then the reward variable is called an *instant-of-time* reward variable.

– If $l > 0$, then the reward variable is called an *interval-of-time* reward variable.

– If $l > 0$, then dividing an interval-of-time reward variable by l gives a *time-averaged interval-of-time* reward variable.

Reward Variable Specification


Reward Structure
Instant-of-Time
Time-Average Interval-of-Time
Interval-of-Time
t
lim as t goes to infinity
$[t, t + l]$
$[t, t + l]$ lim as t goes to infinity
$[t, t + l]$ lim as l goes to infinity
$[t, t + l]$
$[t, t + l]$ lim as t goes to infinity
$[t, t + l]$ lim as l goes to infinity

Reward Variables are Random Variables

Note that since the behavior of a SAN is a stochastic process, then a reward variable is a measure defined on the stochastic process, and therefore a reward variable is a random variable.

A tool can solve for the reward variables, but solving for the distribution in many cases can be difficult. It is often much simpler to solve for the mean or variance of the reward variable, especially when using numerical techniques.

Example reward variables:

- $A(0,t)$ - Fraction of time the system delivers proper service during $[0,t]$. Hard to compute.
- $E[A(0,t)]$ - Expected value of $A(0,t)$. Easier to compute.

Specifying Reward Variables in *Mobius*

- When specifying a rate portion of a reward structure in *Mobius*, you must define a predicate and function.
  - predicate: while true (i.e., integer greater than 0 in C), accumulate the reward
  - function: the value (i.e., double in C) to accumulate
- Note that both the predicate and function may be any C statement or expression.

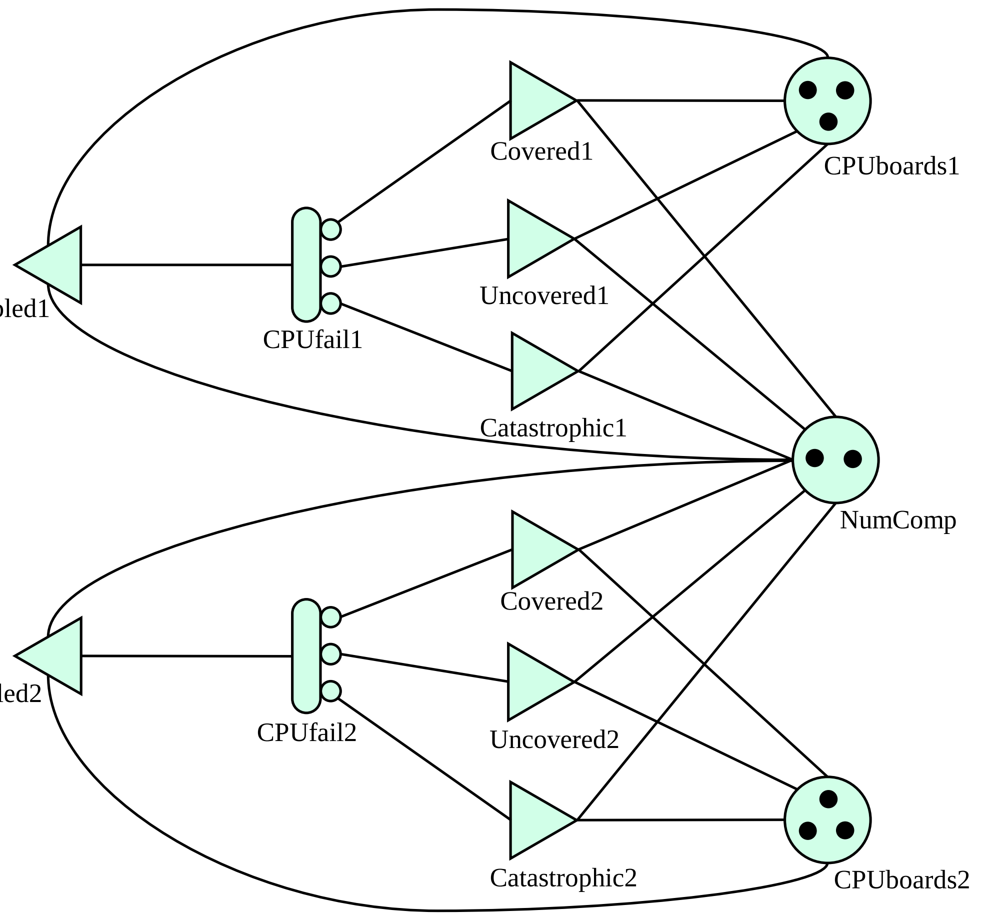
Queue Length	
	Rate rewards
	subnet = database
	Predicate:
	1
	Function:
	MARK(Queue)
	Impulse reward
	none

Fault-Tolerant Computer Failure Model Example

A fault-tolerant computer system is made up of two redundant computers. Each computer is composed of three redundant CPU boards. A computer is operational if at least 1 CPU board is operational, and the system is operational if at least 1 computer is operational.

CPU boards fail at a rate of $1/10^6$ hours, and there is a 0.5% chance that a board failure will cause a computer failure, and a 0.8% chance that a board will fail in a way that causes a catastrophic system failure.

SAN *computer* for Computer Failure Model



Activity Case Probabilities and Input Gate Definition

Activity	*Case*	*Probability*
CPUfail1	*1*	*0.987*
	2	*0.005*
	3	*0.008*

Gate	*Definition*
Enabled1	<u>Predicate</u> *MARK(CPUboards1 > 0) && MARK(NumComp) > 0*
	<u>Function</u> *MARK(CPUboards1)– –;*

Output Gate Definitions

Gate	*Definition*
Covered1	Function *if (MARK(CPUboards1) == 0)* *MARK(NumComp)--;*
Uncovered1	Function *MARK(CPUboards1) = 0;* *MARK(NumComp)--;*

Catastrophic1	Function
	MARK(CPUboards1) = 0;
	MARK(NumComp) = 0;

Reward Variables for Computer Failure Model

Reliability	
	Rate rewards *Subnet = computer* Predicate: *MARK(NumComp) > 0* Function: *1*
	Impulse reward *none*
NumBoardFailures	
	Rate reward *none*

	Impulse reward
	Subnet = computer
	activity = CPUfail1, value = 1
	activity = CPUfail2, value = 1

Reward Variables for Computer Failure Model

Performability	
	<u>Rate rewards</u> *Subnet = computer* <u>Predicate:</u> *1* <u>Function:</u> *MARK(NumComp)*
	<u>Impulse reward</u> *none*
NumBoards	
	<u>Rate reward</u> *Subnet = computer* <u>Predicate:</u> *1* <u>Function:</u> *MARK(CPUBboards1) + MARK(CPUboards2)*
	<u>Impulse reward</u> *none*

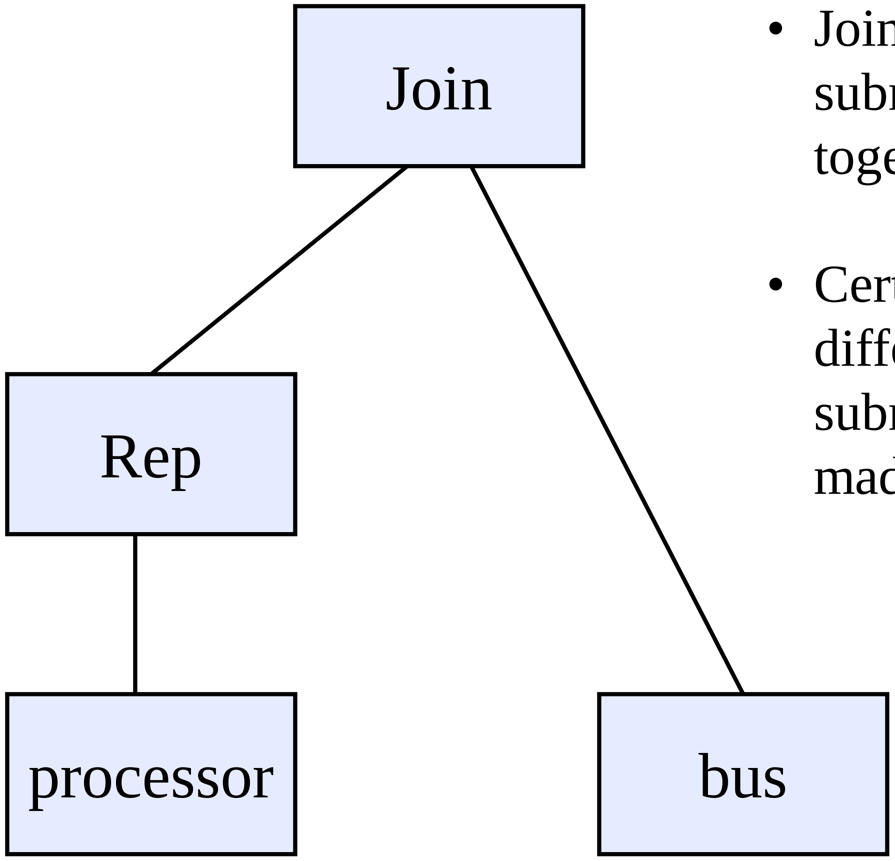
Model Composition

A composed model is a way of connecting different SANs together to form a larger model.

Model composition has two operations:

- Replicate: Combine 2 or more identical SANs and reward structures together, holding certain places common among the replicas.
- Join: Combine 2 or more different SANs and reward structures together, combining certain places to permit communication.

Composed Model Specification



- Join two or more submodels together
- Certain places in different submodels can be made common

- Replicate submodel a certain number of times
- Hold certain places common to all replicas

Rationale

There are many good reasons for using composed models.

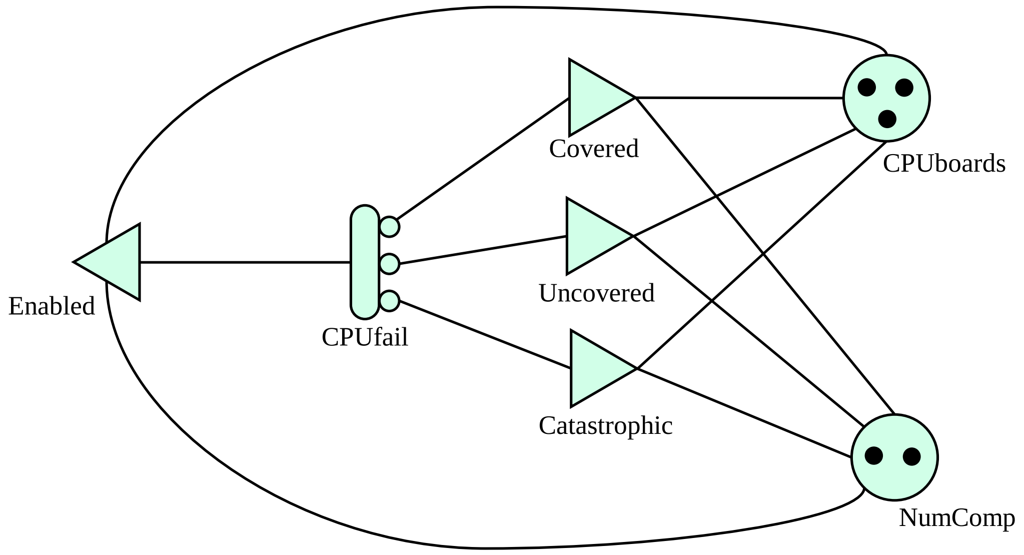
- Building highly reliable systems usually involves redundancy. The replicate operation models redundancy in a natural way.
- Systems are usually built in a modular way. Replicates and Joins are usually good for connecting together similar and different modules.
- Tools can take advantage of something called the *Strong Lumping Theorem* that allows a tool to generate a Markov process with a smaller state space (to be described in Session 7).

Rules for Building Composed Models

(as Implemented in *Mobius*)

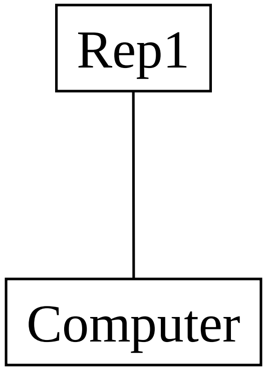
- Places that are joined together must have the same name and initial marking.
- Places that are common at a certain level of the tree must be common at all lower levels.
- Places that are common cannot be connected to the input side of an instantaneous activity.

Computer Failure Model Revisited: Single *computer* Model



(Note initial marking of NumComp is two since there will be two computers in the composed model.)

Composed Model for Computer Failure Model



Node	*Reps*	*Common Places*
Rep1	*2*	*NumComp*

Reward Variables for Composed Model

Reliability	
	Rate rewards *Subnet = computer* Predicate: *MARK(NumComp) > 0* Function: *0.5*
	Impulse reward *none*
NumBoardFailures	
	Rate reward *none*
	Impulse reward *Subnet = computer* activity = CPUfail, value = 1

Reward Variables for Composed Model

Performability	
	Rate rewards *Subnet = computer* Predicate: *1* Function: *MARK(NumComp) / 2.0*
	Impulse reward *none*
NumBoards	
	Rate reward *Subnet = computer* Predicate: *1* Function: *MARK(CPUboards)*
	Impulse reward *none*

Composed Model

How does adding an additional computer affect reliability?

- In the composed model, change number of replications to 3 and change various reward variables - easy (Use a global variable if you think suspect you may want to do this.)
- In “flat” model, add another computer - hard

In composed model, the number of states in the underlying Markov chain is much smaller, especially for large numbers of replications. (Details will be given in Session 7.)

Global Variables

- There may be some parameter to a model that may want to change (in our example, the level of redundancy). This can be implemented easily as global variables.
- Global variables come as two types: short (short integer) and double (double precision floating point). Use `name` and `name` to indicate global variable "name." (You do NOT need to declare these anywhere; *UltraSAN* does this automatically.)
- In *Mobius*, you may use global variables virtually anywhere you may type text.

Example: Use `num_comp` and `num_boards` to set the level of redundancy of computers and boards. Use this as the initial number of tokens, the number of replications, and when specifying reward variables. Later, you can change the levels of redundancy easily.

Summary

- Stochastic Petri nets
  - Places, tokens, input / output arcs
  - Readers / Writers example
- Stochastic activity networks
  - Input / output gates, cases, instantaneous activities
  - Marking dependent behavior, well-specified, general distributions
- Reward variables
  - Reward structures
  - Reward variable classification
  - Predicate / function implementation in *UltraSAN*
- Fault tolerant computer example
- Composed model
  - Fault tolerant computer revisited