

ECE/CS 541

Computer System Analysis: Stochastic Activity Networks

Mohammad A. Nouredine
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

Fall 2018

Announcements and Reminders

- **Project presentations on December 15**
 - Will try to start at 5:00 pm to finish early
- Homework 4 is out and due next week
- Submit papers on the 17th via EasyChair
 - Will send the link soon
 - You will get 3 *anonymous* reviews
 - I wonder who the reviewers are!
- **ICES forms!!!**
 - Please show and fill up the ICES forms
 - Get 1 point of the participation credits

Outline for the next 2 Weeks

- **Today**

- Stochastic Activity Networks Intro
- SAN Examples
- Intro to output analysis

- **Tuesday**

- Output analysis

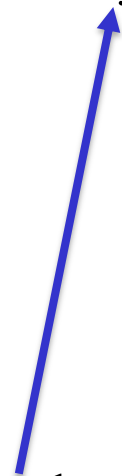
- **Thursday**

- TBA (More output analysis or Introduction to Game Theory)

Learning Objectives

- Or what is this course about?
- **At the start of the semester, you should have**
 - Basic programming skills (C++, Python, etc.)
 - Basic understanding of probability theory (ECE313 or equivalent)
- **At the end of the semester, you should be able to**
 - Understand different system modeling approaches
 - Combinatorial methods, state-space methods, etc.
 - Understand different model analysis methods
 - Analytic/numeric methods, simulation
 - Understand the basics of discrete event simulation
 - **Design simulation experiments and analyze their results**
 - Gain hands-on experience with different modeling and analysis tools

Project



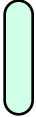
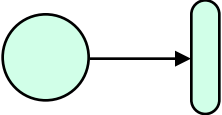
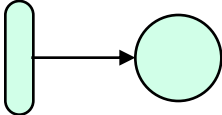


Today's Lecture

- Stochastic Activity Networks
 - Definitions and semantics
 - Dependent behavior, well-specified, general distributions
- Rep/Join semantics
- Example

Stochastic Petri Net Review

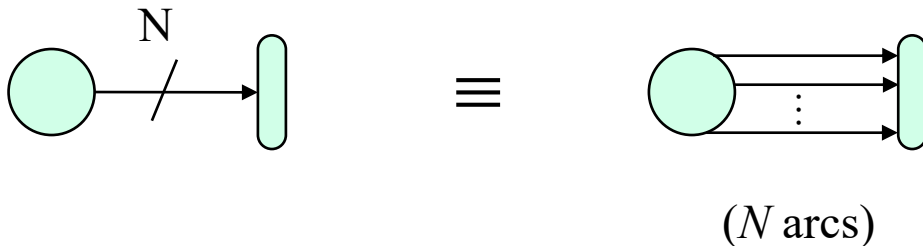
One of the simplest high-level modeling formalisms is called *stochastic Petri nets*. A stochastic Petri net is composed of the following components:

- Places:  which contain tokens, and are like variables
- tokens:  which are the “value” or “state” of a place
- transitions:  | (timed, untimed) change the #tokens in places
- input arcs:  which connect places to transitions
- output arcs:  which connect transitions to places

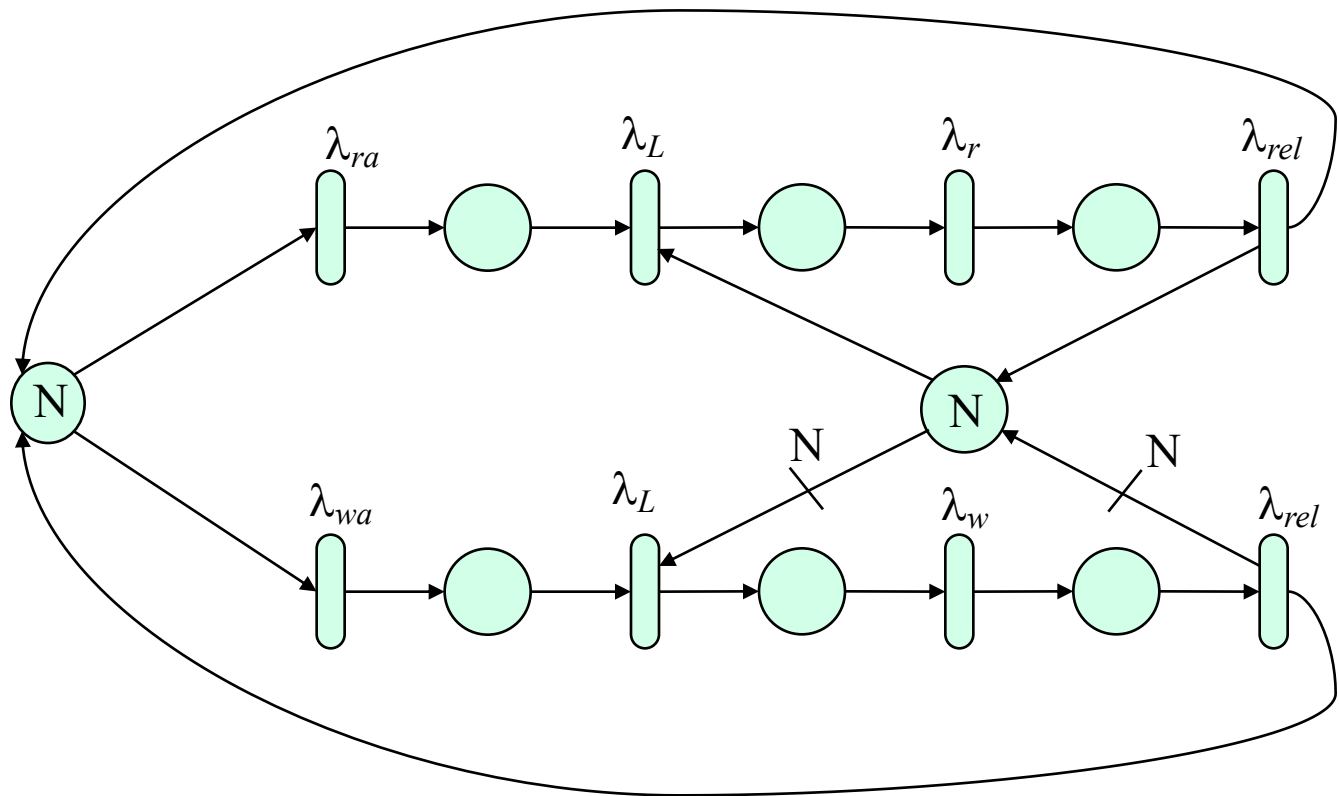
SPN Example: Readers/Writers Problem

- 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



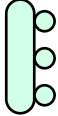

- SPNs are **much easier to read, write, modify, and debug** than Markov chains.
- SPN to Markov chain conversion can be automated to afford numerical solutions to Markov chains.
- Most SPN formalisms include a special type of arc called an *inhibitor arc*,
 - inhibit a transition if the connected place has “too many” tokens
- Some also include the identity (do nothing) function.
- 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

- We need more expressive modeling languages, with richer semantics
- Many extensions have been proposed
- You guessed it: We will examine **Stochastic Activity Networks**
- **Properties of SANs:**
- *General way* to specify when an activity (or a transition) is enabled
- *General way* to specify a completion (or firing) rule
- A way to represent *zero-timed* events
- Represent *probabilistic choices* upon activity completion
- *State dependent parameter* values
- General delay distributions on activities

SAN Symbols

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

Enabling Rules

- An input gate has two components (let \mathcal{S} be the set of possible states):

1. Enabling function or predicate:

$$f : \mathcal{S} \rightarrow \{\top, \perp\}$$

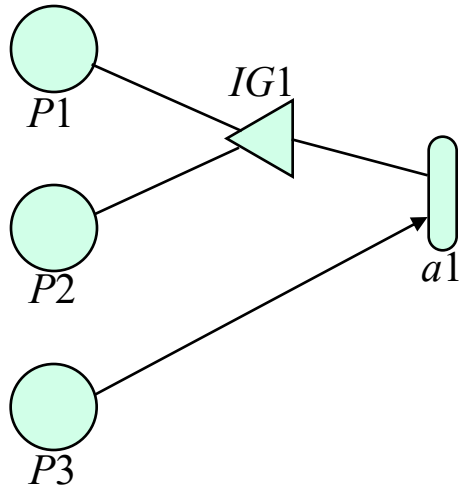
2. Input function: transition from state s to state s'

$$g : \mathcal{S} \rightarrow \mathcal{S}$$

- An activity is enabled if
 - For every connected input gate, the enabling predicate is true
 - For each input arc, the number of tokens in the connected place \geq the number of arcs
- Notation: $MARK(P) =$ number of tokens in place P

Example SAN Enabling Rule

Example:



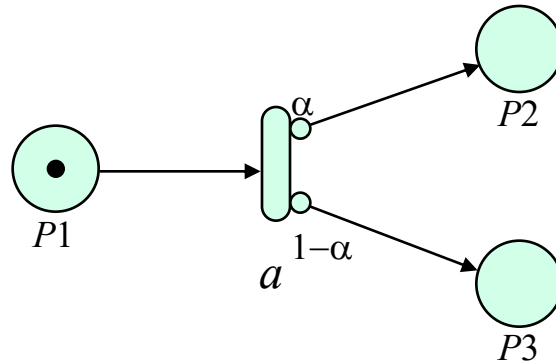
IG1 Predicate:

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

Activity $a1$ is enabled if $IG1$'s predicate is true and $MARK(P3) > 0$.

Probabilistic Choices

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



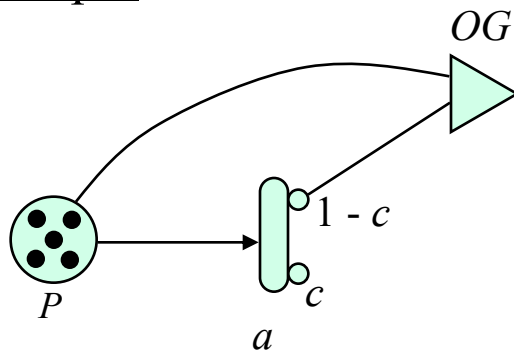
When activity a completes, a token is removed from place $P1$, and with probability α , a token is put into place $P2$, and with probability $1 - \alpha$, a token is put into place $P3$.

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.

Example



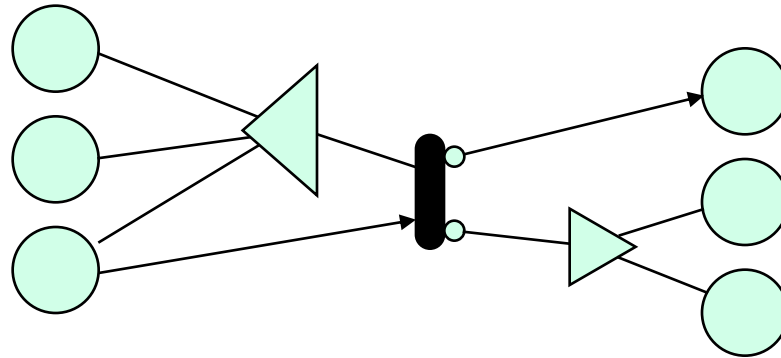
OG Function

$MARK(P) = 0;$

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.

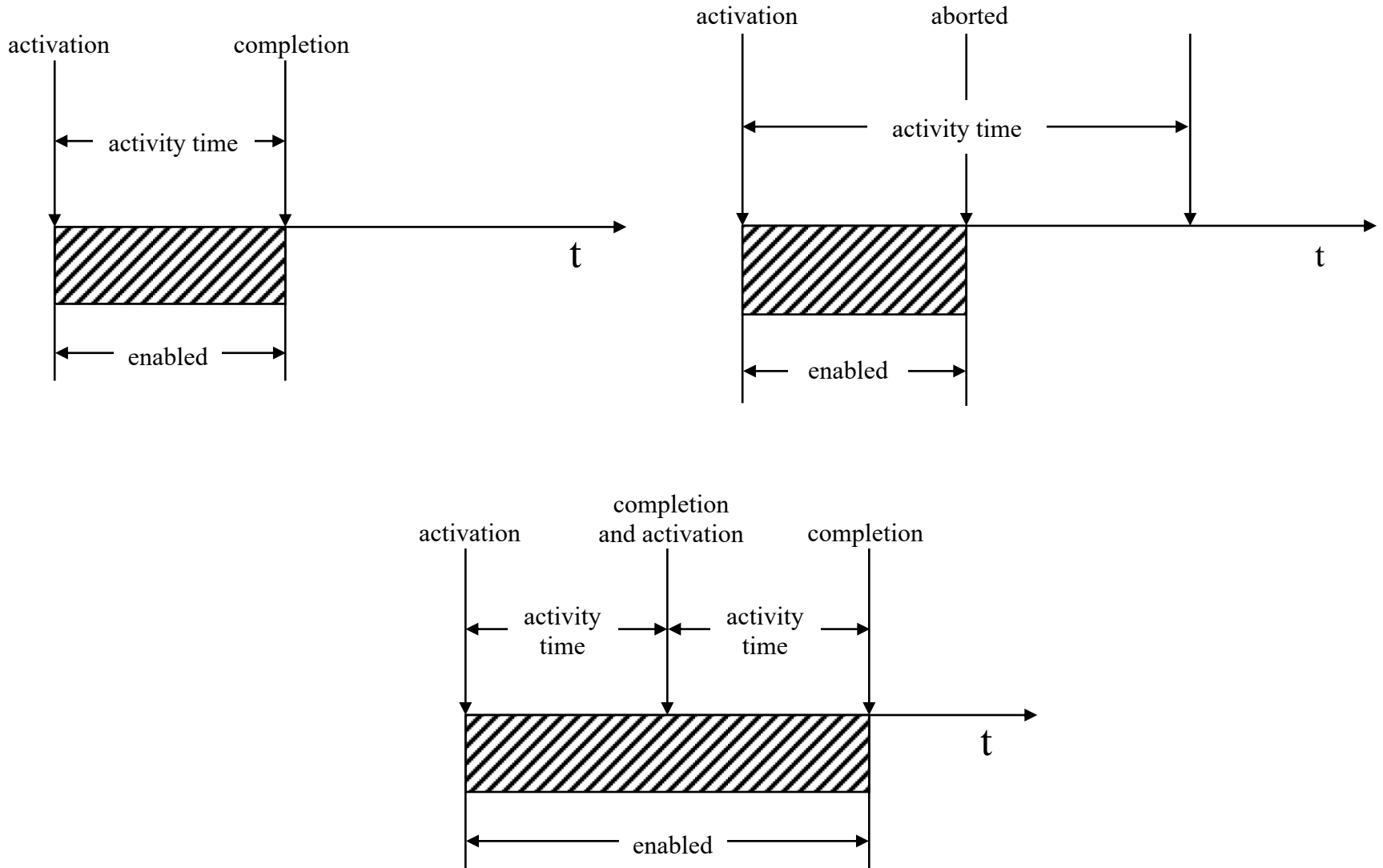


Instantaneous activities are useful when modeling events that have an effect on the state of the system, but happen in negligible time, with respect to other activities in the system, and the performance/dependability measures of interest.

SAN Terms

- *Activation:*
 - time at which an activity begins
- *Completion:*
 - time at which activity completes
- *Abort:*
 - time, after activation but before completion, when activity is no longer enabled
- *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.

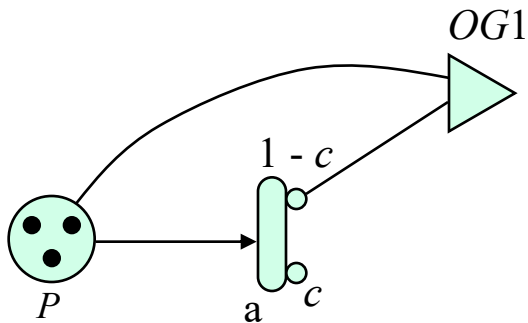
Ordering is important, since effect of actions can be marking-dependent.

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 does not crash the whole system) decreases after a failure.



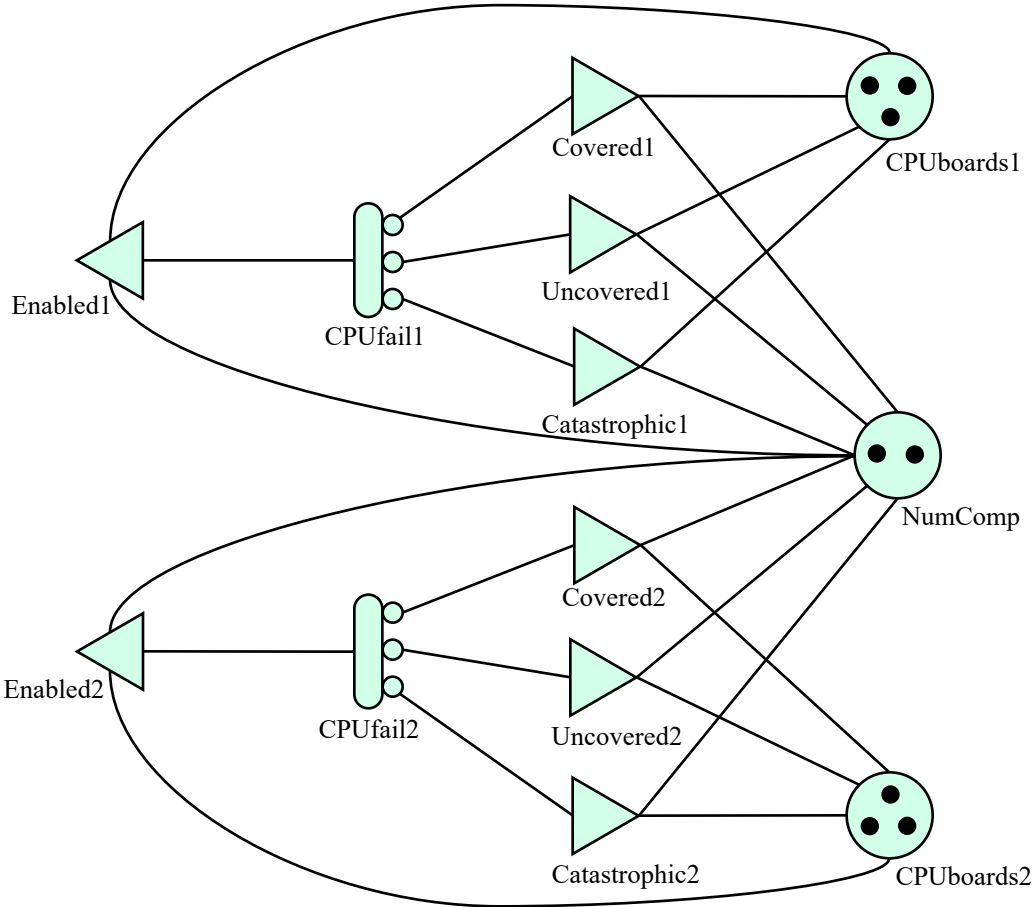
	a
case 1	$0.1 + 0.02 * \text{MARK}(P)$
case 2	$0.9 - 0.02 * \text{MARK}(P)$

Running 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,
- The system is operational if
 - at least 1 computer is operational.

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

Representative SAN



Activity Case Probabilities and Input Gate Definition

<i>Activity</i>	<i>Case</i>	<i>Probability</i>
<i>CPUfail1</i>	<i>1</i>	<i>0.987</i>
	<i>2</i>	<i>0.005</i>
	<i>3</i>	<i>0.008</i>

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

Output Gate Definitions

<i>Gate</i>	<i>Definition</i>
<i>Covered1</i>	<u>Function</u> <i>if (MARK(CPUboards1) == 0)</i> <i>MARK(NumComp)--;</i>
<i>Uncovered1</i>	<u>Function</u> <i>MARK(CPUboards1) = 0;</i> <i>MARK(NumComp)--;</i>
<i>Catastrophic1</i>	<u>Function</u> <i>MARK(CPUboards1) = 0;</i> <i>MARK(NumComp) = 0;</i>

General Distributions

- SANs support many activity time distributions
 - Exponential, hyperexponential, deterministic, Weibull, etc.
- In addition, distribution parameters can be dependent on the state of the model
 - i.e., dependent on the marks in the places
- **Downside**
 - No equivalent CTMC can be generated for general distributions
- However, we can still perform **simulations** to obtain **meaningful results**
- Analytic solutions can still be possible under certain conditions

Reward Variables

- Okay we specified our system model and all of its parameters
- What comes next?
- Need to specify meaningful metrics to compute
 - E.g., performance, dependability, availability etc.
- Enter **reward variables**
- Examples:
 - Expected time until service,
 - Availability,
 - Number of misrouted packets in an interval of time,
 - Processor utilization,
 - Operational cost,
 - Length of downtime,
 - ...

Reward Structures

- There are two ways for us to *accumulate* rewards:
 - A model may be in a certain state or states for some period of time
 - This is called a **rate reward**
 - Example?
 - An activity may complete
 - This is called an **impulse reward**
 - Examples?
- At the end of time,
 - Reward variable = rate rewards + impulse rewards

Example

- We want to predict the profits of a certain cloud web hosting solution (think hosting your web service on Amazon EC2)
- Basically, when the service is up, your profits accumulate at a rate of \$N/hr.
- When system is running in degraded mode, the rate of profits drops to \$(N/6)/hr
- Finally, a repair operation has a fixed cost of \$K to your business

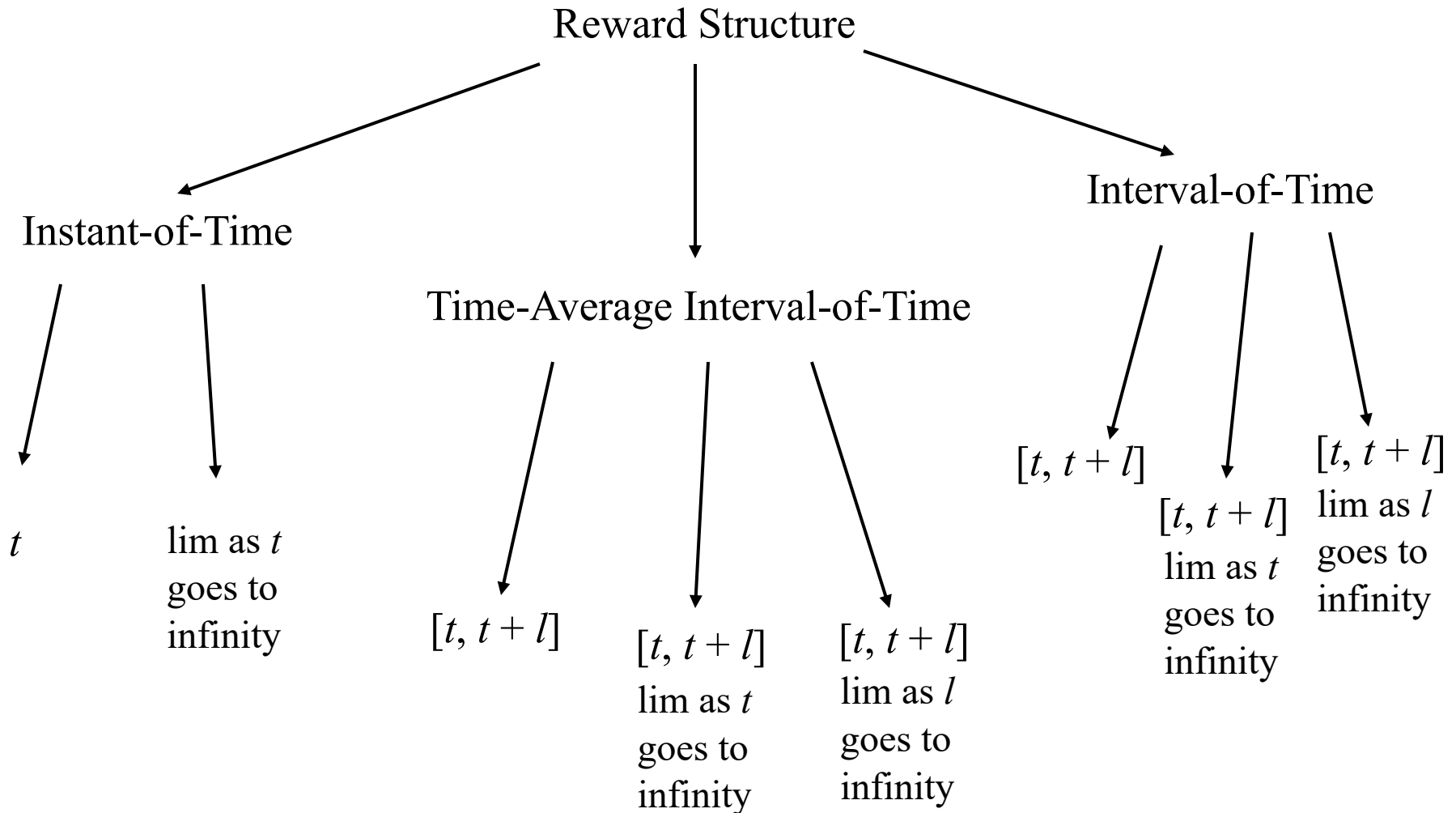
$$R(s) = \begin{cases} N & s \text{ is the fully functioning state} \\ \frac{N}{6} & s \text{ is the degraded operation state} \\ 0 & \textit{otherwise} \end{cases}$$

$$C(s) = \begin{cases} -K & s \text{ is when a repair is activated} \\ 0 & \textit{otherwise} \end{cases}$$

Types of Reward Variables

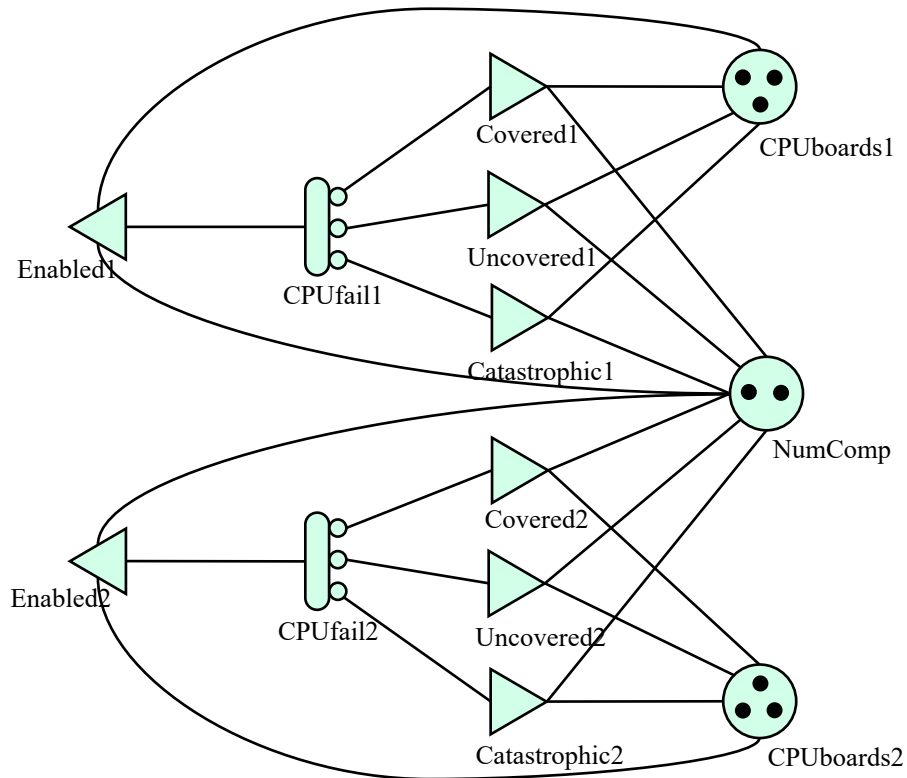
- By integrating the previous functions over an interval of time t , we can get our reward variable, i.e., the profit our business will make
- So basically, a reward variable is the sum of impulse and rate reward structures over a certain time interval
- Let $[t, t + l]$ be an interval of time over which the reward variable is defined
 - If $l = 0$, then we get an **instant of time** reward variable
 - If $l > 0$, then we get an **interval of time** reward variable
 - If $l > 0$, then dividing an interval of time variable by l gives us a **time averaged interval of time** reward variable

Reward Variable Specification



Recall our Running Example

- We are interested in computing
 - Reliability
 - Number of board failures
 - Performability (performance + reliability)



Reward Variables for Running Example

<i>Reliability</i>	
	<u>Rate rewards</u> <i>Subnet = computer</i> <u>Predicate:</u> $MARK(NumComp) > 0$ <u>Function:</u> 1
	<u>Impulse reward</u> <i>none</i>
<i>NumBoardFailures</i>	
	<u>Rate reward</u> <i>none</i>
	<u>Impulse reward</u> <i>Subnet = computer</i> activity = CPUfail1, value = 1 activity = CPUfail2, value = 1

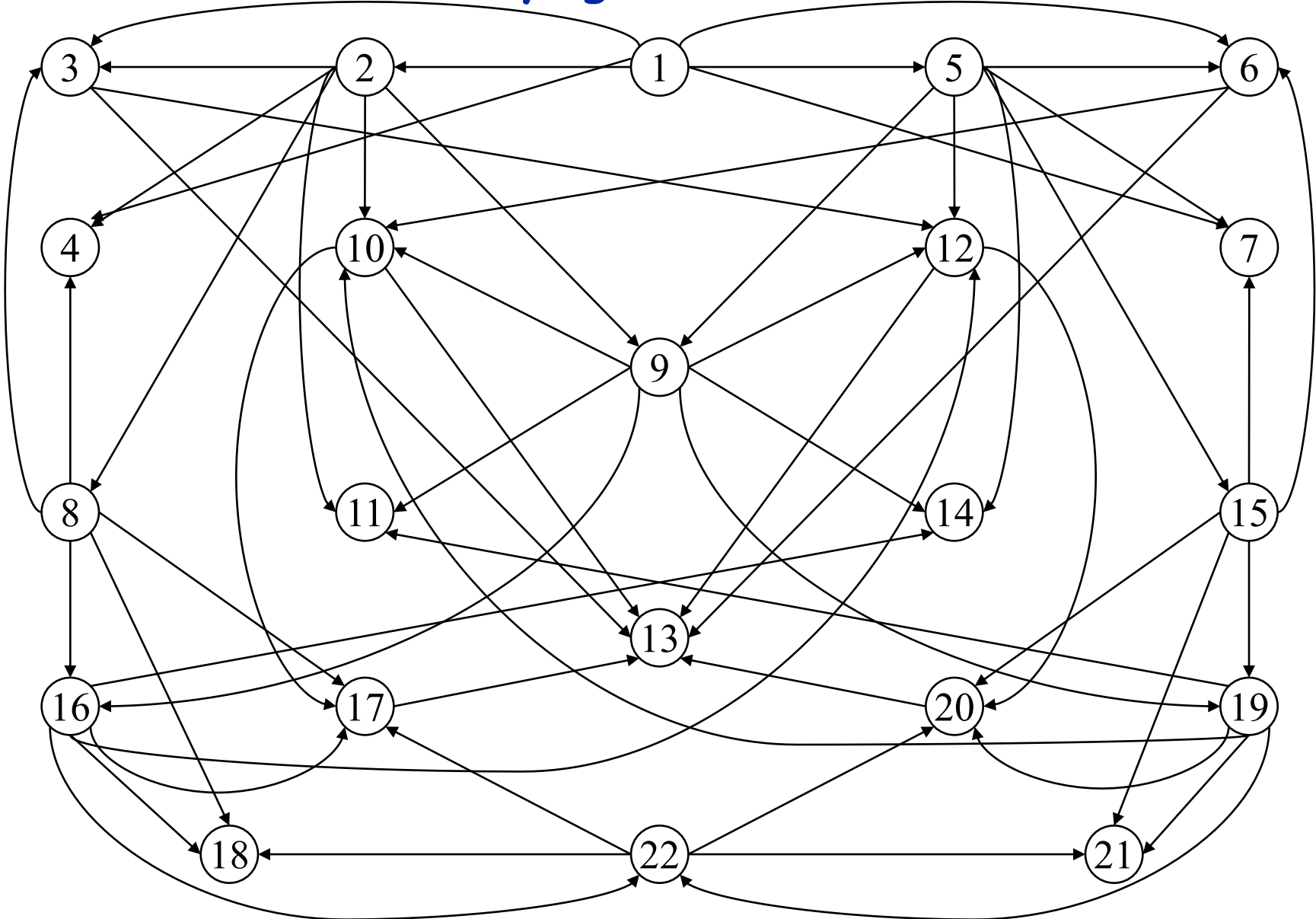
Reward Variables for Running Example

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

Generated State Space

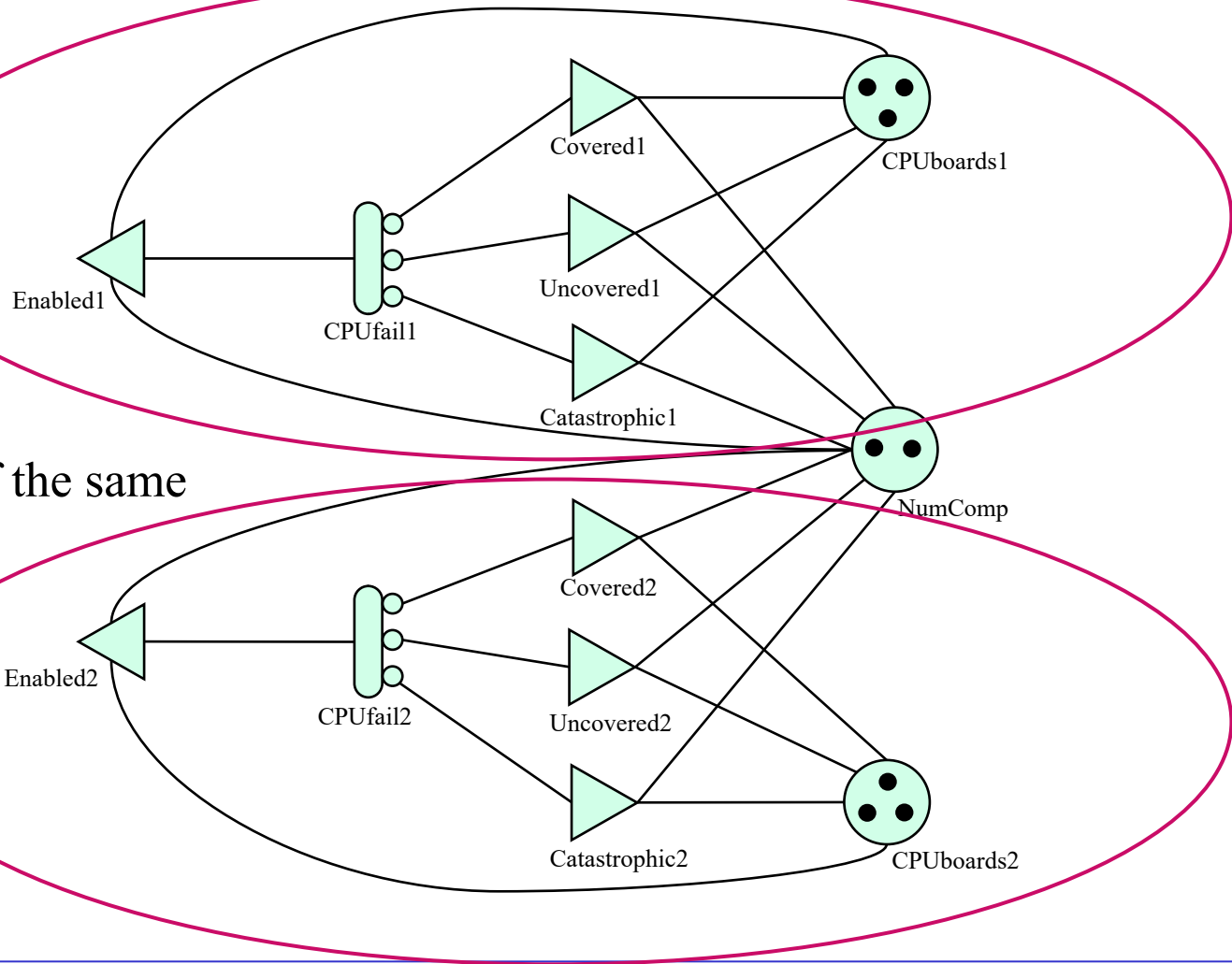
State No.	CPUboards1	CPUboards2	NumComp	(Next State, Rate)
1	3	3	2	(2, p1λ), (3, p2λ), (4, p3λ), (5, p1λ), (6, p2λ), (7, p3λ)
2	2	3	2	(8, p1λ), (3, p2λ), (4, p3λ), (9, p1λ), (10, p2λ), (11, p3λ)
3	0	3	1	(12, p1λ), (13, (p2+p3) λ)
4	0	3	0	
5	3	2	2	(9, p1λ), (12, p2λ), (14, p3λ), (15, p1λ), (6, p2λ), (7, p3λ)
6	3	0	1	(10, p1λ), (13, (p2+p3) λ)
7	3	0	0	
8	1	3	2	(3, (p1+p2) λ), (4, p3λ), (16, p1λ), (17, p2λ), (18, p3λ)
9	2	2	2	(16, p1λ), (12, p2λ), (14, p3λ), (19, p1λ), (10, p2λ), (11, p3λ)
10	2	0	1	(17, p1λ), (13, (p2+p3) λ)
11	2	0	0	
12	0	2	1	(20, p1λ), (13, (p2+p3) λ)
13	0	0	0	
14	0	2	0	
15	3	1	2	(19, p1λ), (20, p2λ), (21, p3λ), (6, (p1+p2) λ), (7, p3λ)
16	1	2	2	(12, (p1+p2) λ), (14, p3λ), (22, p1λ), (17, p2λ), (18, p3λ)
17	1	0	1	(13, λ)
18	1	0	0	
19	2	1	2	(22, p1λ), (20, p2λ), (21, p3λ), (10, (p1+p2) λ), (11, p3λ)
20	0	1	1	(13, λ)
21	0	1	0	
22	1	1	2	(20, (p1+p2) λ), (21, p3λ), (17, (p1+p2) λ), (18, p3λ)

Underlying Markov Model



Model Composition

- Let's take another look at our SAN model
- What can you notice in this model?



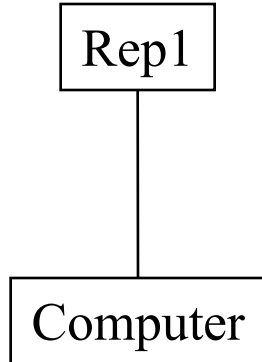
Mirrored copy of the same model

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.

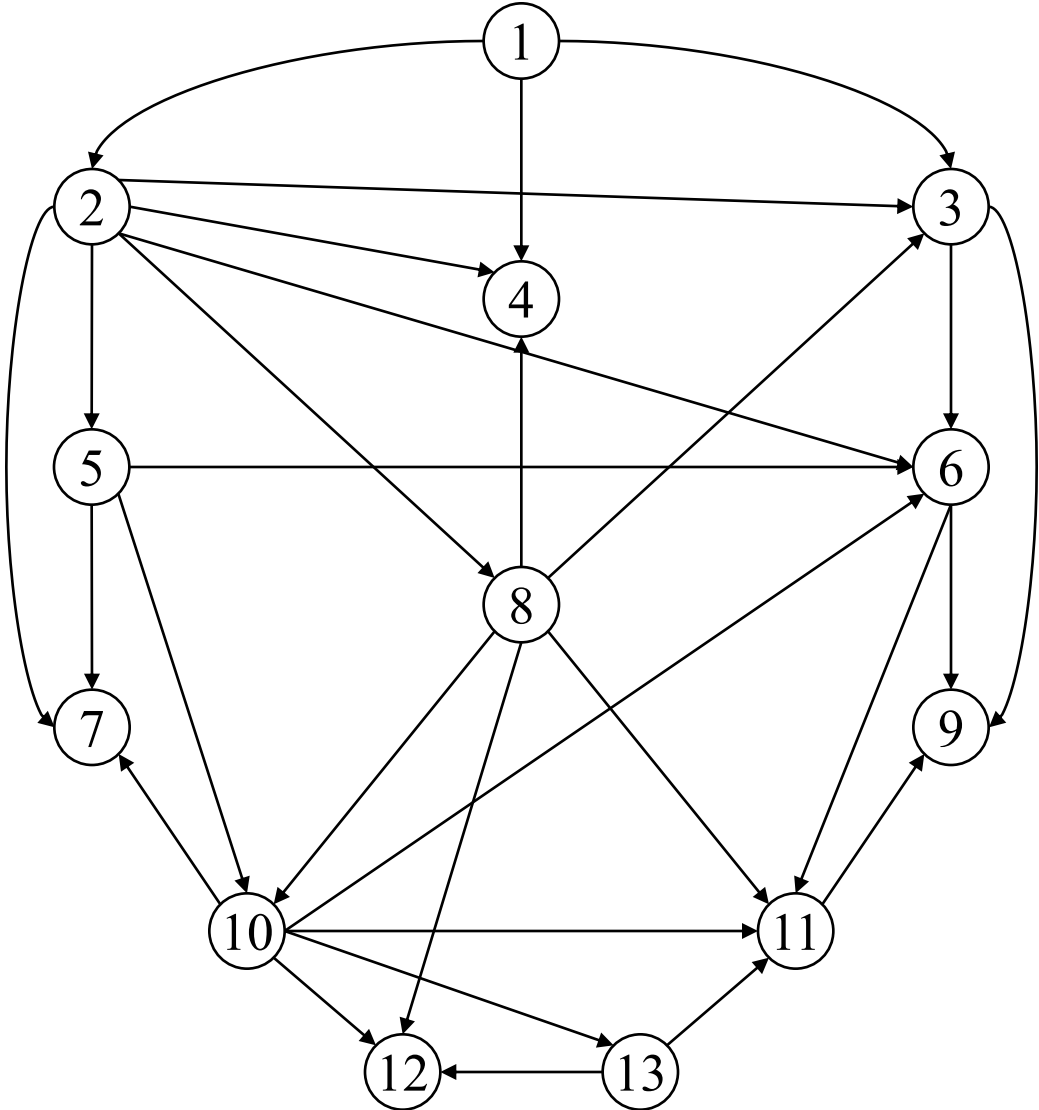
Composed Model for Computer Failure Model



<i>Node</i>	<i>Reps</i>	<i>Common Places</i>
<i>Rep1</i>	<i>2</i>	<i>NumComp</i>

Shared place between the replicated models

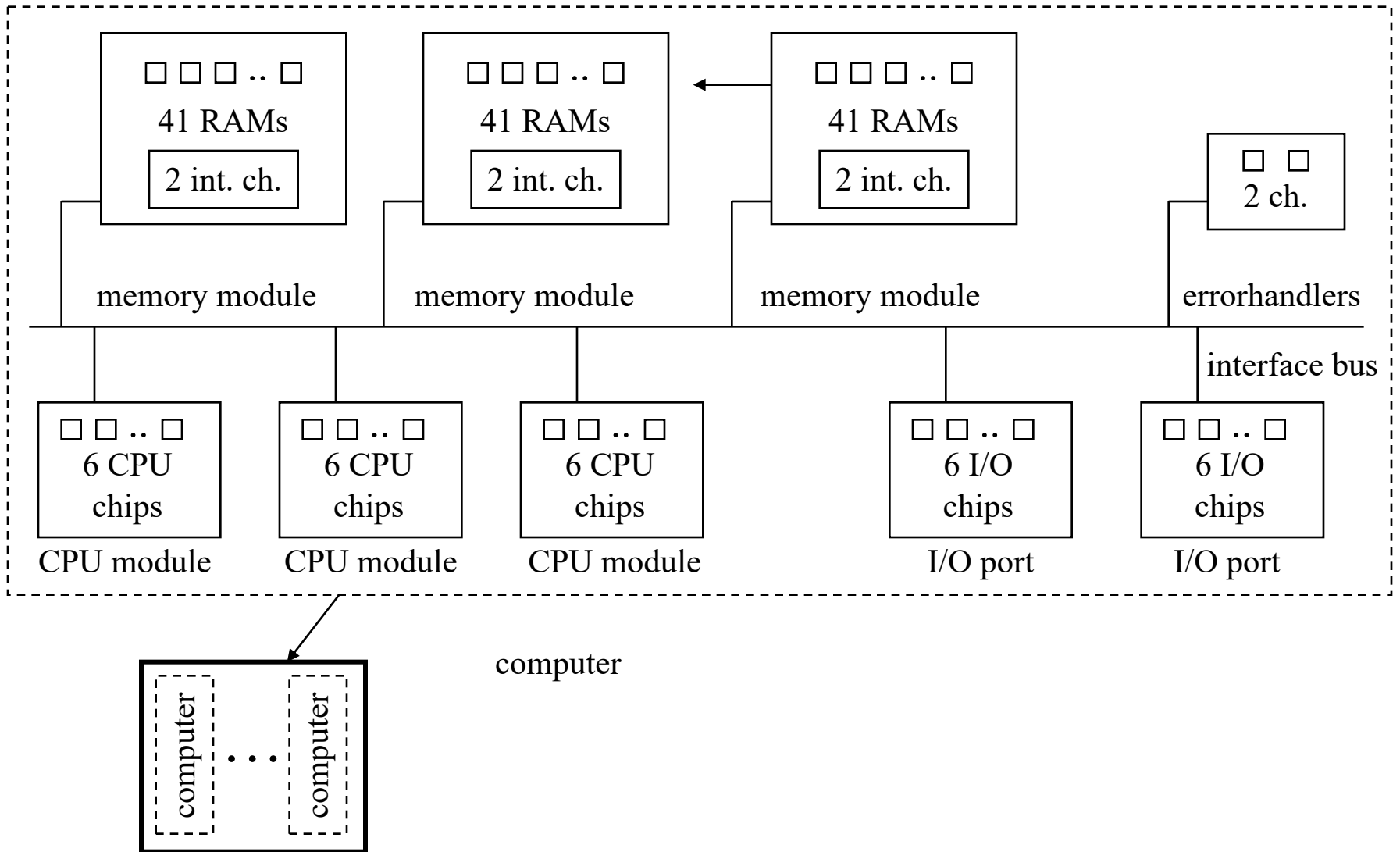
Markov Chain of Reduced Base Model



Fault-Tolerant Control Computer Example

- System consists of 2 computers
- Each computer consists of
 - 3 memory modules (2 must be operational)
 - 3 CPU units (2 must be operational)
 - 2 I/O ports (1 must be operational)
 - 2 error-handling chips (non-redundant)
- Each memory module consists of
 - 41 RAM chips (39 must be operational)
 - 2 interface chips (non-redundant)
- A CPU consists of 6 non-redundant chips
- An I/O port consists of 6 non-redundant chips
- 10 to 20 year operational life

Diagram of Fault-Tolerant Multiprocessor System



Definition of “Proper Operation”

- The system is operational if
 - at least one computer is operational
- A computer is operational if
 - all the modules are operational
- A memory module is operational if
 - at least 39 RAM chips and both interface chips are operational.
- A CPU unit is operational if
 - all 6 CPU chips are operational
- An I/O port is operational if
 - all 6 I/O chips are operational
- The error-handling unit is operational if
 - both error-handling chips are operational
- Failure rate per chip is 100 failures per 1 billion hours

Coverage

- This system could be modeled using combinatorial methods if we did not take coverage into account.
- *Coverage* is the chance that the failure of a chip will not cause the larger system to fail if sufficient redundancy exists. i.e., coverage is the probability that the fault is contained.

The coverage probabilities are given in the following table:

<i>Redundant Component</i>	<i>Fault Coverage Probability</i>
RAM Chip	0.998
Memory Module	0.95
CPU Unit	0.995
I/O Port	0.99
Computer	0.95

- For example, if a RAM chip fails, there is a 0.2% chance the memory module will fail even if sufficient redundancy exists.
- If the memory module fails, there is a 5% chance the computer will fail.
- If a computer fails, there is a 5% chance the system will fail.

Outline of Solution: List of SANs

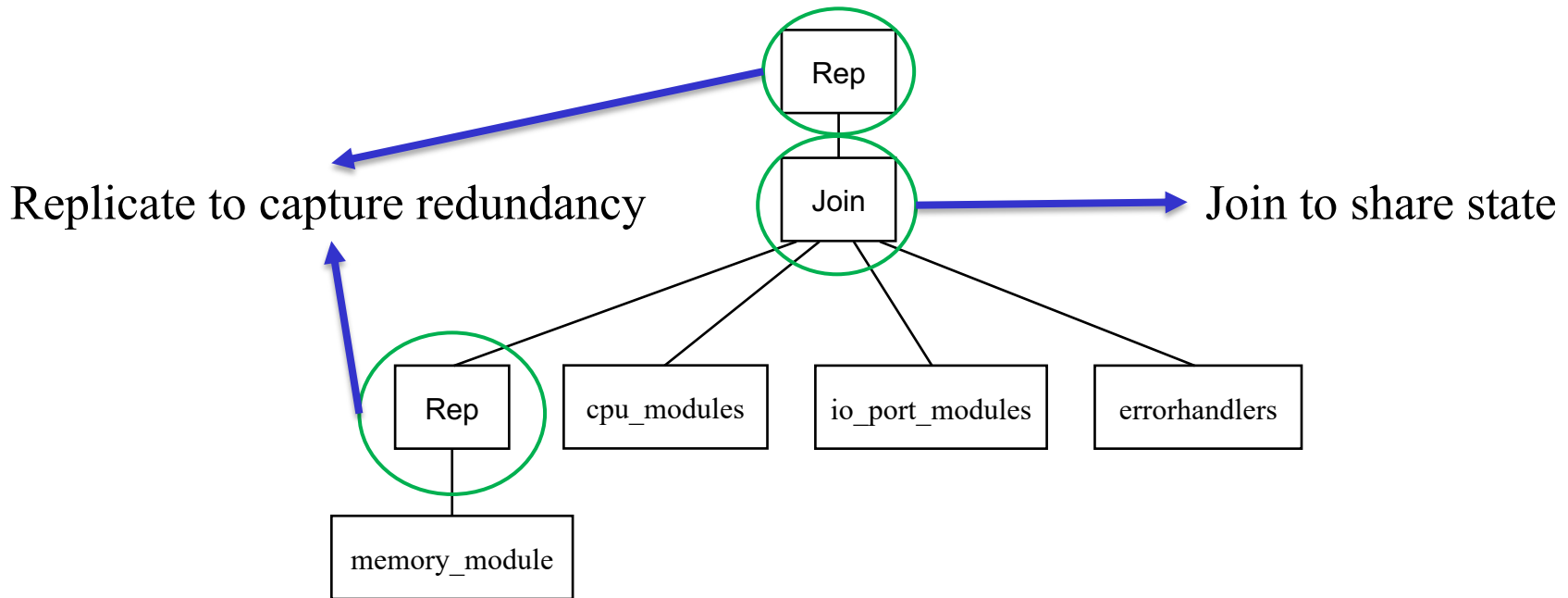
- The model is composed of four SANs:
 1. memory_module
 2. cpu_module
 3. errorhandlers
 4. io_port_module
- Each SAN models the behavior of the module in the event of a module component failure.

Tricks of the Trade

We would like to have the fewest number of states possible.

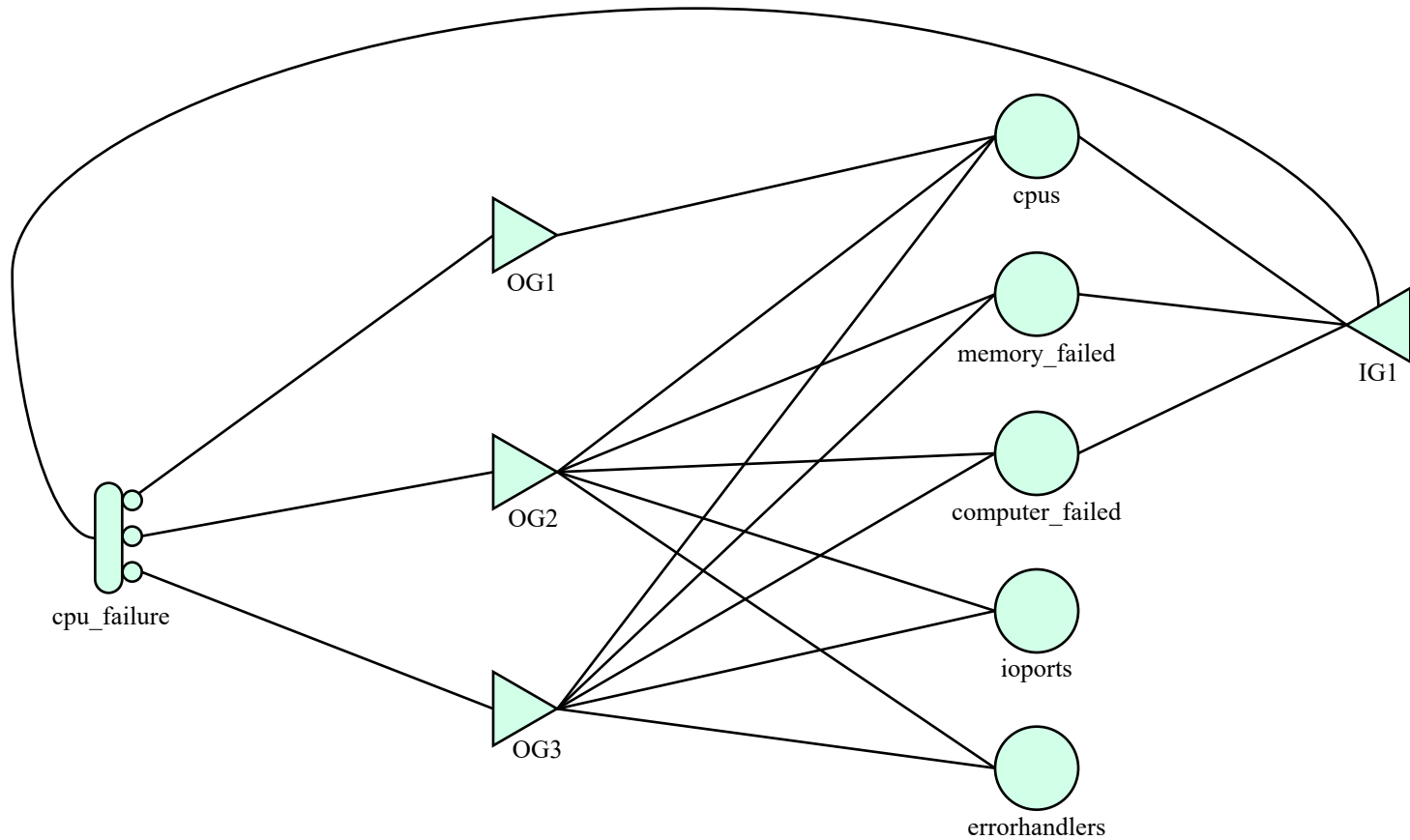
- We don't care *which* component failed or what particular failed state the model is in. Therefore, we lump all failure states into the same state.
- **We don't care which computer or which module is in what state.** Therefore, we make use of replication to further reduce the number of states.
- We use **marking-dependent rates to model RAM chip failure**, making use of the fact that the minimum of independent exponentials is an exponential.
- We use cases to denote coverage probabilities, and adjust the probabilities depending on the state of the system.

Composed Model



<i>Node</i>	<i>Reps</i>	<i>Common Places</i>
<i>Rep1</i>	<i>3</i>	<i>computer_failed</i>
		<i>memory_failed</i>
<i>Rep2</i>	<i>2</i>	<i>computer_failed</i>

cpu_modules SAN



<i>Place</i>	<i>Marking</i>
cpus	3
ioports	2
errorhandlers	2
memory_failed	0
computer_failed	0

cpu_modules SAN, cont.

cpu_modules input gate predicates and functions:

<i>Gate</i>	<i>Enabling Predicate</i>	<i>Function</i>
<i>IG1</i>	$(MARK(cpus) > 1) \ \&\&$ $(MARK(memory_failed) < 2) \ \&\&$ $(MARK(computer_failed) < 2)$	<i>identity</i>

Only time we're interested in processor failure is when it hasn't already failed

cpu_modules activity time distributions:

<i>Activity</i>	<i>Distribution</i>
<i>cpu_failure</i>	$\text{expon}(0.0052596 * MARK(cpus))$

cpu_modules SAN, cont.

cpu_modules case probabilities for activities:

<i>Case</i>	<i>Probability</i>
module_cpu_failure	
1	<i>if (MARK(cpus) == 3)</i> <i> return(0.995);</i> <i>else</i> <i> return(0.0);</i>
2	<i>if (MARK(cpus) == 3)</i> <i> return(0.00475);</i> <i>else</i> <i> return(0.95);</i>
3	<i>if (MARK(cpus) == 3)</i> <i> return (0.00025);</i> <i>else</i> <i> return(0.05);</i>

- case 1: chip failure covered
- case 2: chip failure causes computer failure
- case 3: chip failure causes system (catastrophic) failure

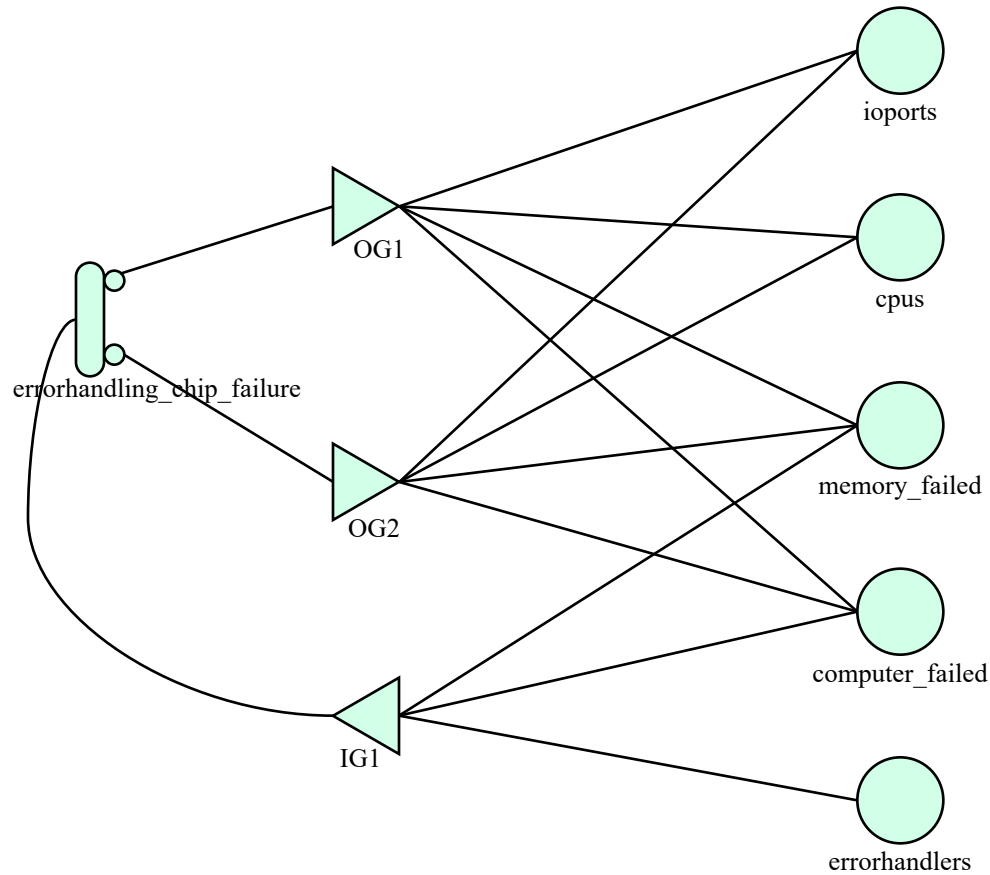
cpu_modules SAN, cont.

cpu_modules output gate functions:

<i>Gate</i>	<i>Function</i>
<i>OG1</i>	<i>if</i> (<i>MARK</i> (<i>cpus</i>) == 3) <i>MARK</i> (<i>cpus</i>) --;
<i>OG2</i>	<i>MARK</i> (<i>cpus</i>) = 0; <i>MARK</i> (<i>ioports</i>) = 0; <i>MARK</i> (<i>errorhandlers</i>) = 0; <i>MARK</i> (<i>memory_failed</i>) = 2; <i>MARK</i> (<i>computer_failed</i>) ++;
<i>OG3</i>	<i>MARK</i> (<i>cpus</i>) = 0; <i>MARK</i> (<i>ioports</i>) = 0; <i>MARK</i> (<i>errorhandlers</i>) = 0; <i>MARK</i> (<i>memory_failed</i>) = 2; <i>MARK</i> (<i>computer_failed</i>) = 2;

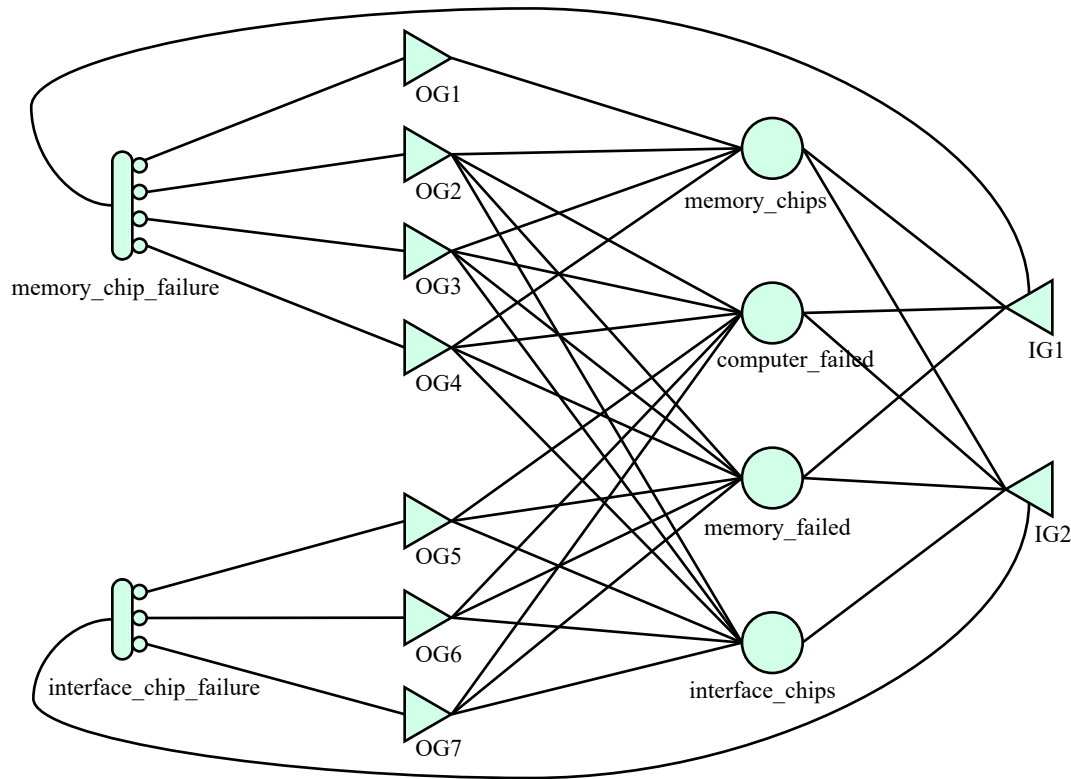
Different failures have different impacts on processor/system state

errorhandlers SAN



<i>Place</i>	<i>Marking</i>
errorhandlers	2
cpus	3
ioports	2
memory_failed	0
computer_failed	0

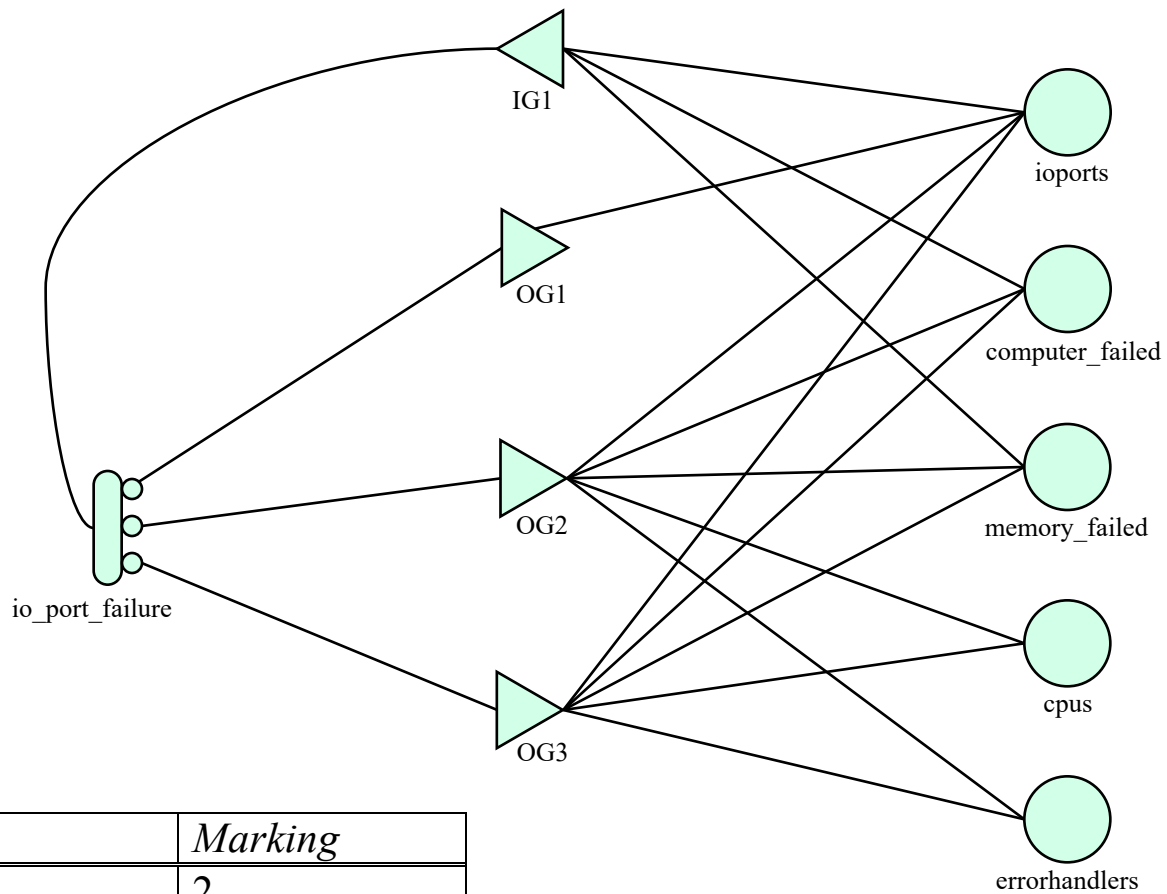
memory_module SAN



<i>Place</i>	<i>Marking</i>
<code>memory_chips</code>	41
<code>interface_chips</code>	2
<code>memory_failed</code>	0
<code>computer_failed</code>	0

Note: `memory_module` is replicated 3 times, `computer_failed` and `memory_failed` held in common.

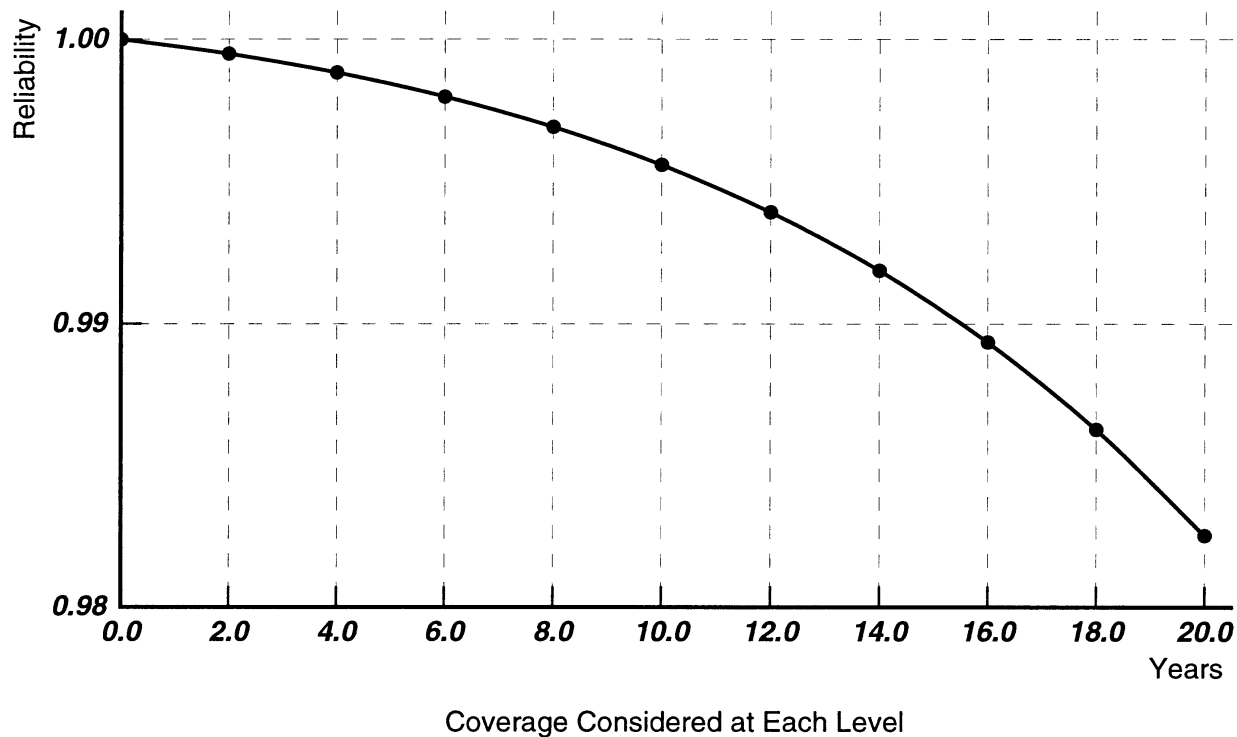
io_port_modules SAN



<i>Place</i>	<i>Marking</i>
ioports	2
cpus	3
errorhandlers	2
memory_failed	0
computer_failed	0

Model Solution

The modeled two-computer system with non-perfect coverage at all levels (i.e., the model as described), the state space contains 10,114 states. The 10 year mission reliability was computed to be .995579.



Impact of Coverage

- Coverage can have a large impact on reliability and state-space size. Various coverage schemes were evaluated with the following results.

<i>Design description</i>	<i>State-space size</i>	<i>Reliability (10-year mission time)</i>
<i>100% coverage at all levels</i>	<i>4278</i>	<i>0.999539</i>
<i>Nonperfect coverage considered at all levels</i>	<i>10114</i>	<i>0.995579</i>
<i>Nonperfect coverage considered at all levels, no spare memory module</i>	<i>1335</i>	<i>0.987646</i>
<i>Nonperfect coverage considered at all levels, no spare CPU module</i>	<i>3299</i>	<i>0.973325</i>
<i>Nonperfect coverage considered at all levels, no spare IO port</i>	<i>3299</i>	<i>0.985419</i>
<i>Nonperfect coverage considered at all levels, no spare memory module, CPU module, or IO port</i>	<i>511</i>	<i>0.935152</i>
<i>100% coverage at all levels, no spare memory module, CPU module, IO port, or RAM chips</i>	<i>6</i>	<i>0.702240</i>

Coming up Next: Output Analysis

- We have seen ways to generate random numbers
 - And evaluate their characteristics
- We have seen ways to generate random variates
 - Starting from the uniform distribution
- We have seen how to define our models in high-level language
- Okay, we run the simulation and **we get numbers**
 - But wait how do we **interpret those numbers**?
 - When do we **stop our simulations**?
 - How **confident** are we in our numbers?
- We will set out to answer those questions in the following lectures