

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 this Friday
- 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 week

- Today
 - SANs Rep/Join
 - Output Analysis
- Thursday
 - More output analysis

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

Today's Lecture

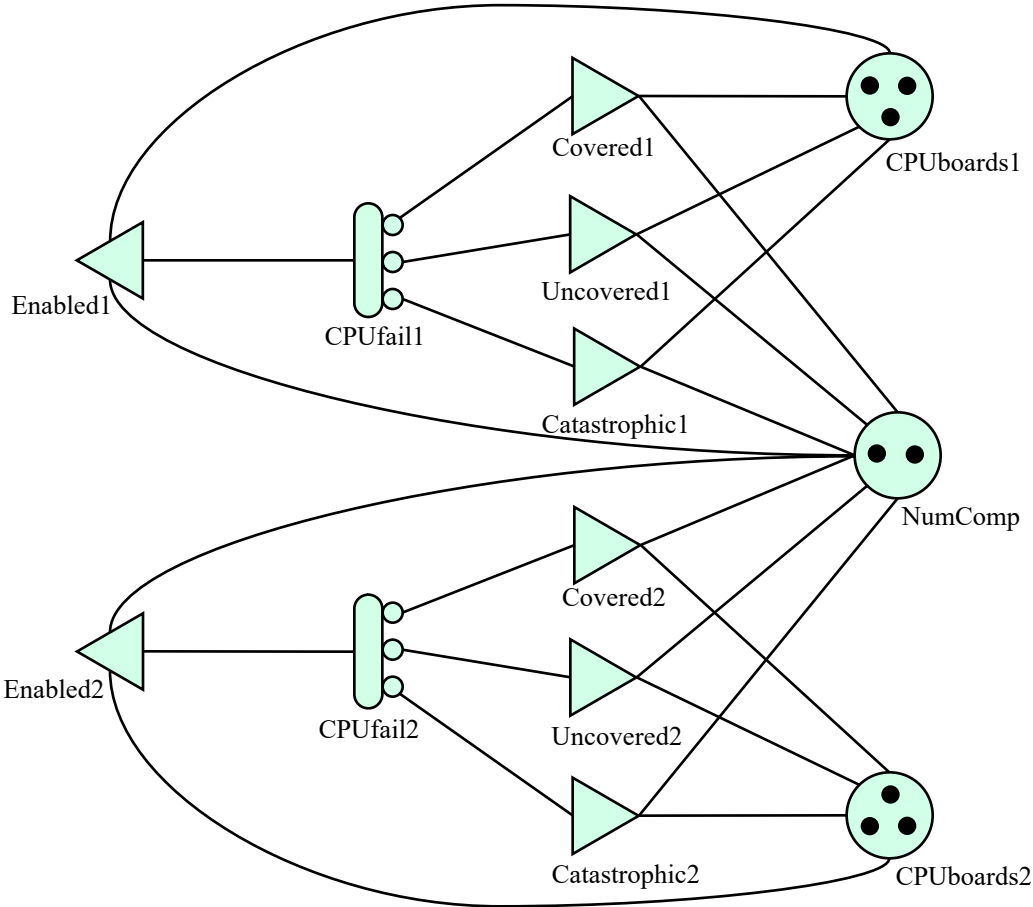
- Rep/Join semantics of Stochastic Activity Networks
- Example
- Output Analysis

Running Example

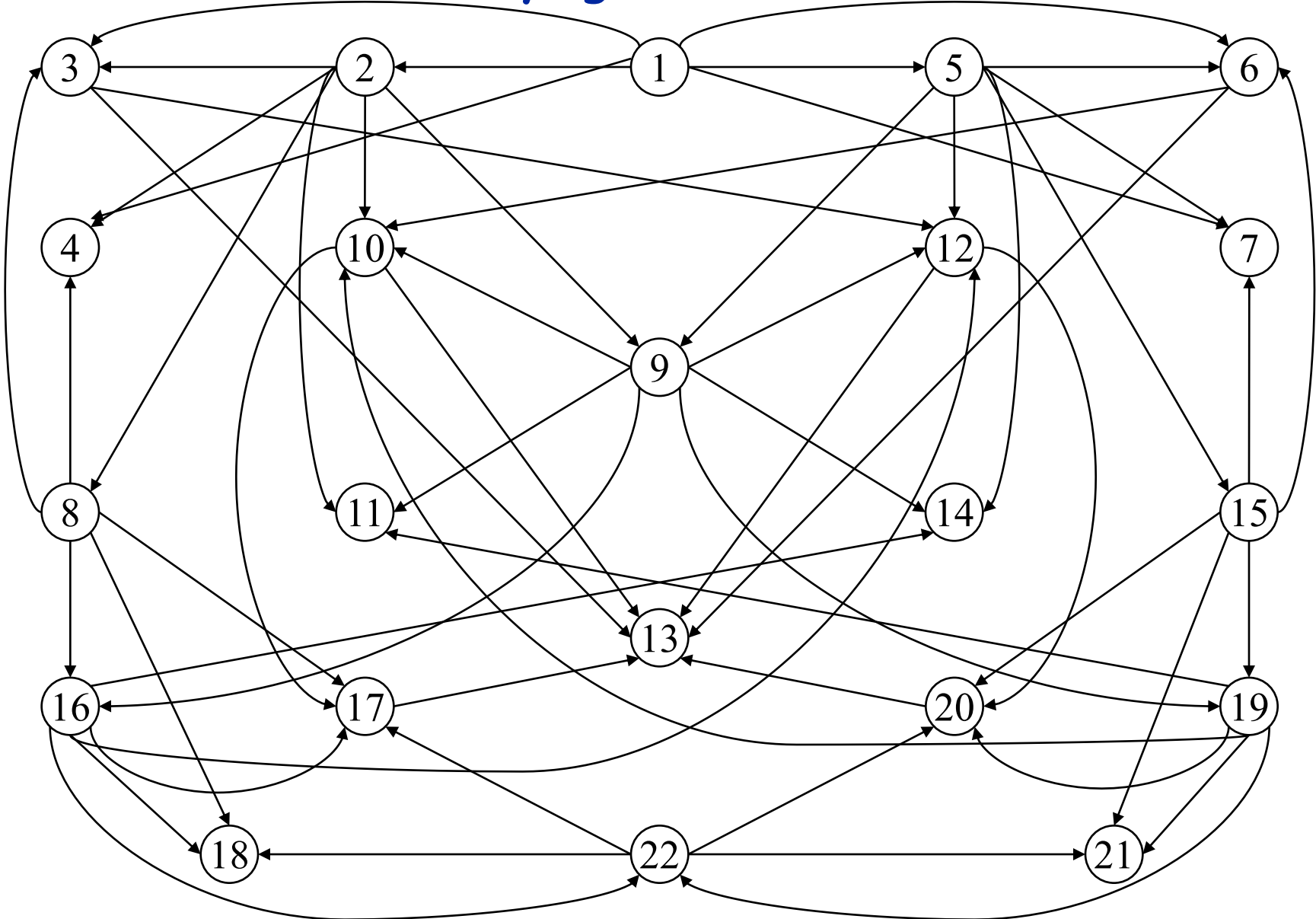
- A fault-tolerant computer system is made up of **two redundant computers**.
- Each computer is composed of **three redundant CPU boards**.
- A computer is operational if
 - at least 1 CPU board is operational,
- The system is operational if
 - at least 1 computer is operational.

- 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

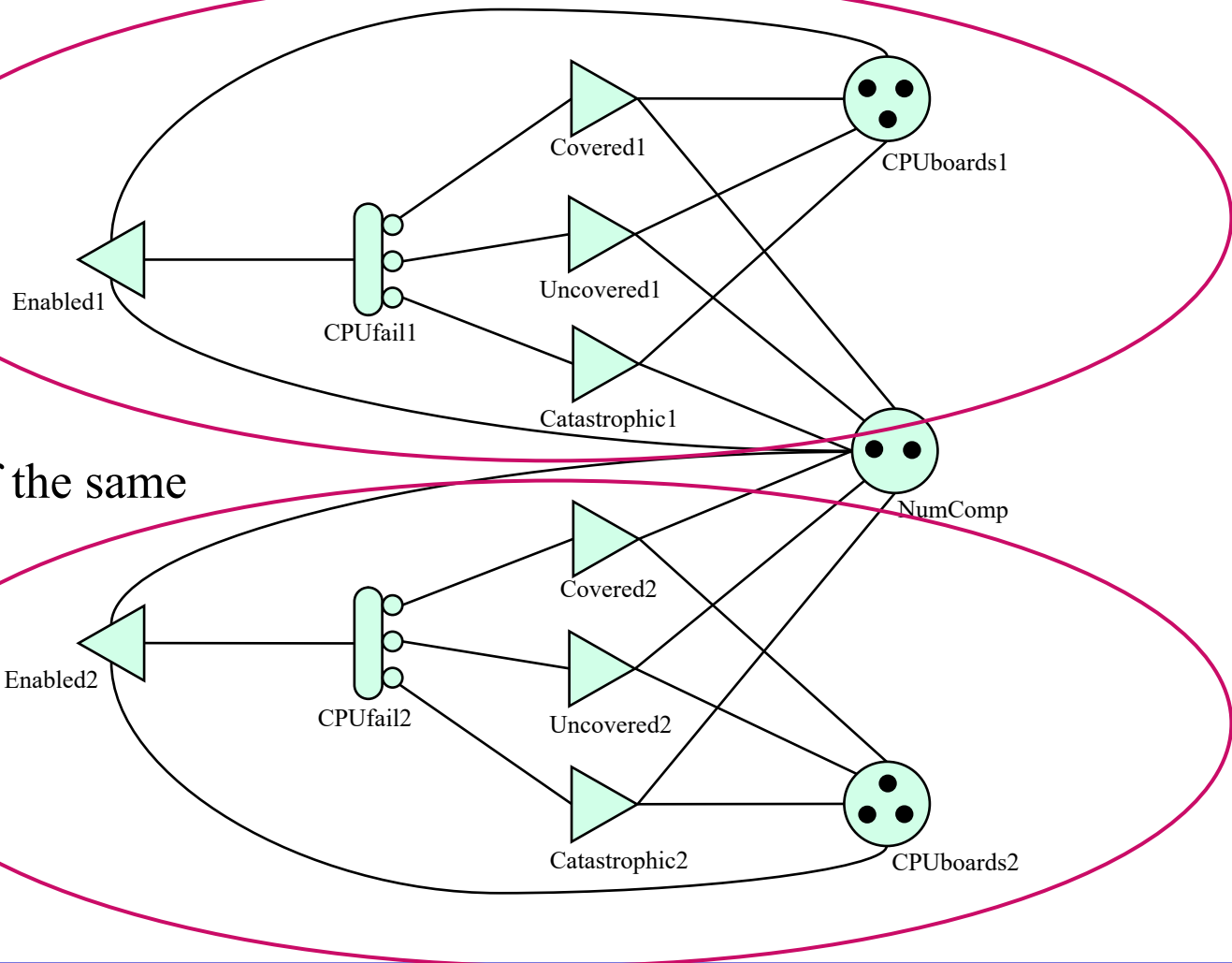


Underlying Markov Model



Model Composition

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



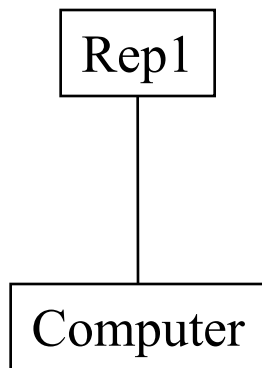
Mirrored copy of the same model

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.

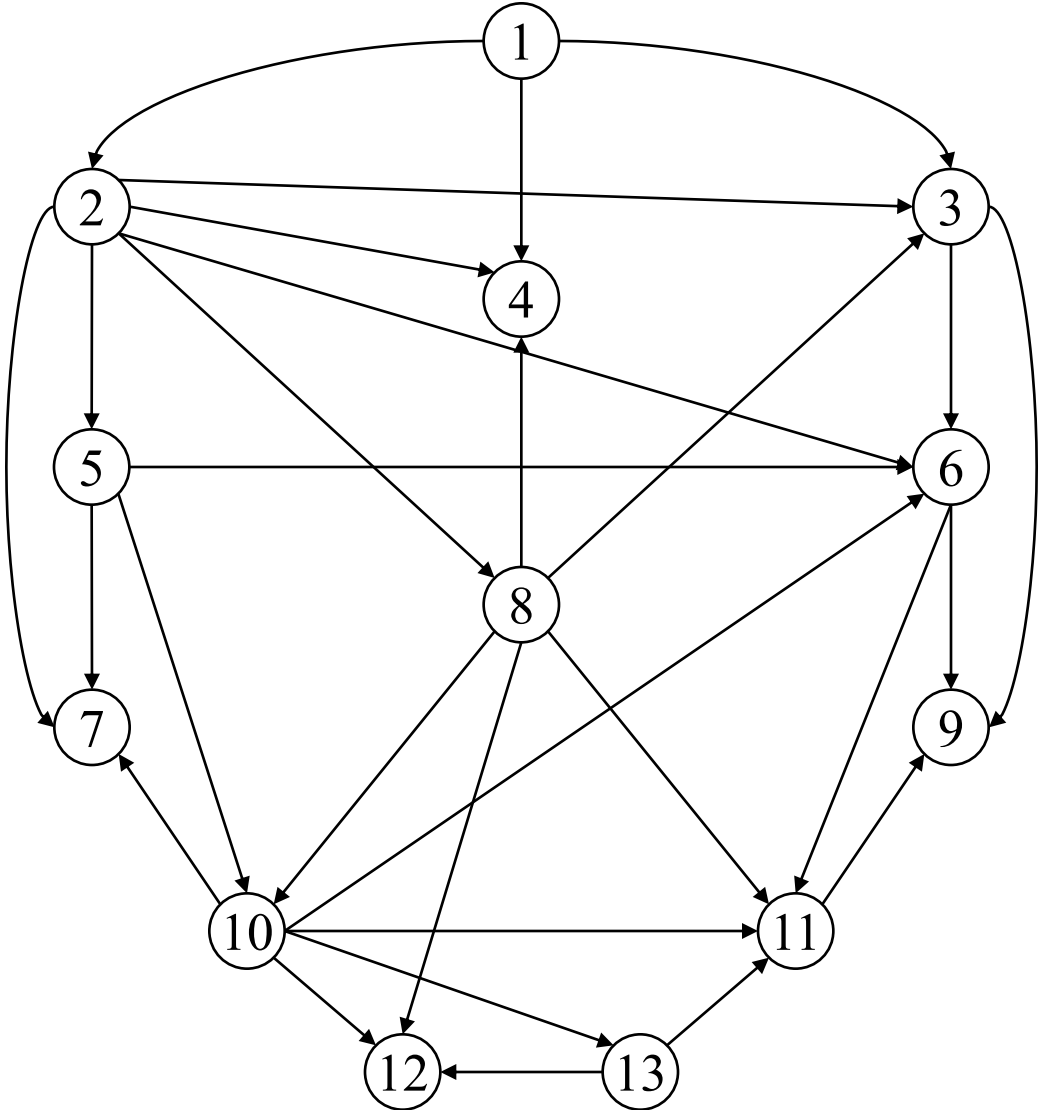
Composed Model for Computer Failure Model



<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

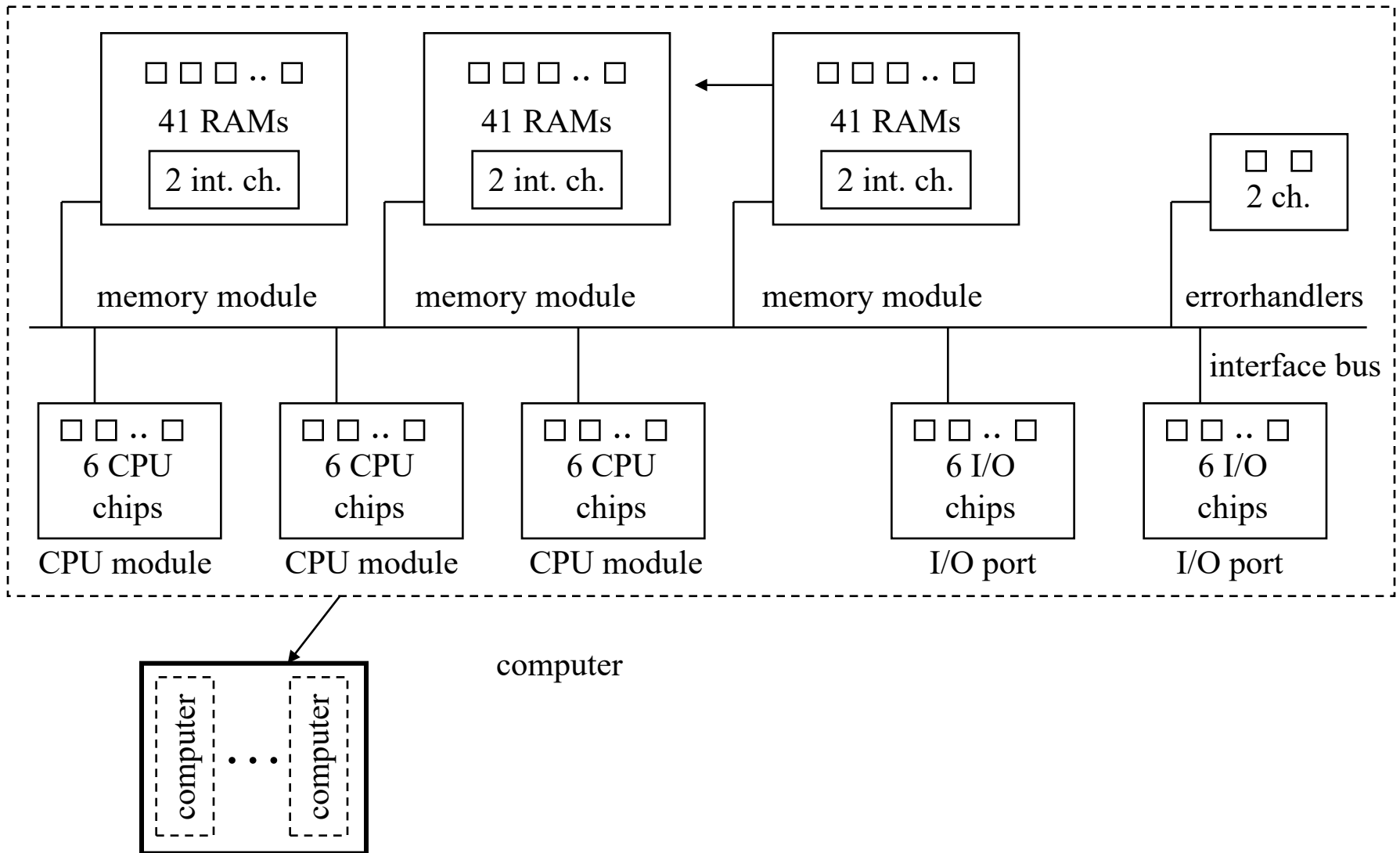
Markov Chain of Reduced Base Model



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

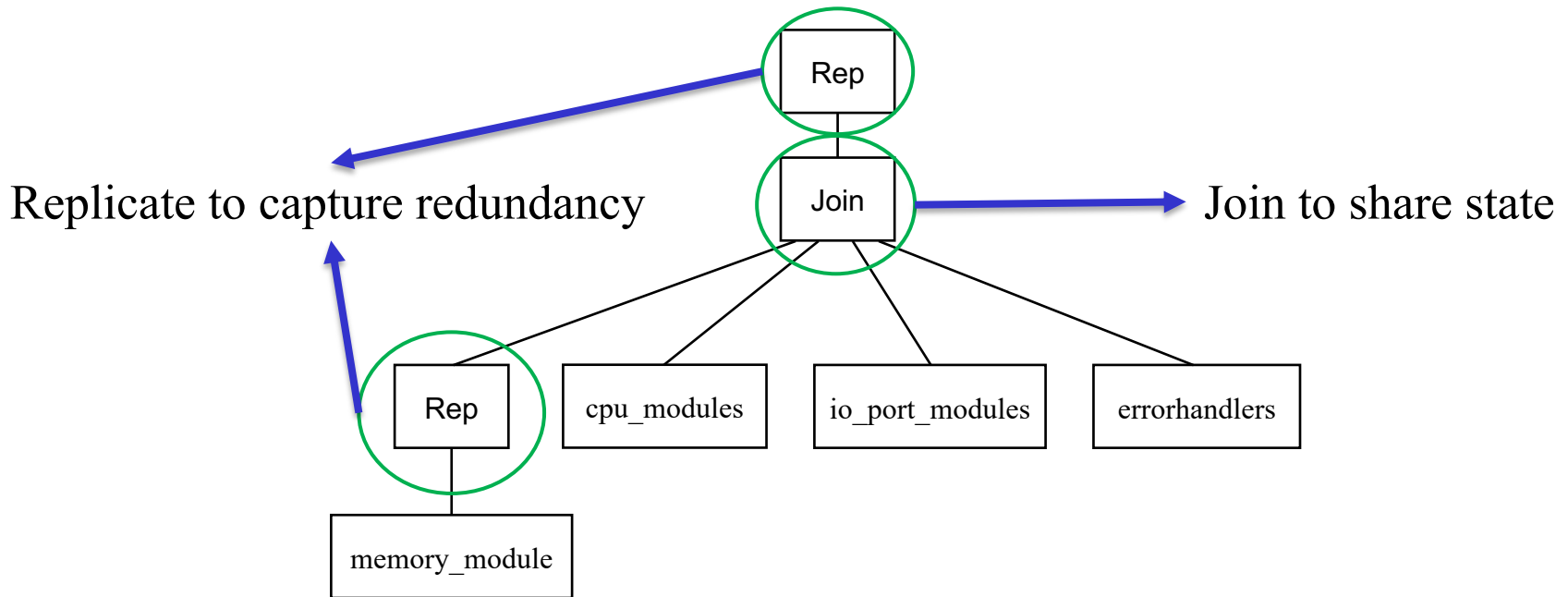
- The model is composed of four SANs:
 1. memory_module
 2. cpu_module
 3. errorhandlers
 4. io_port_module
- Each SAN models the behavior of the module in the event of a module component failure.

Tricks of the Trade

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

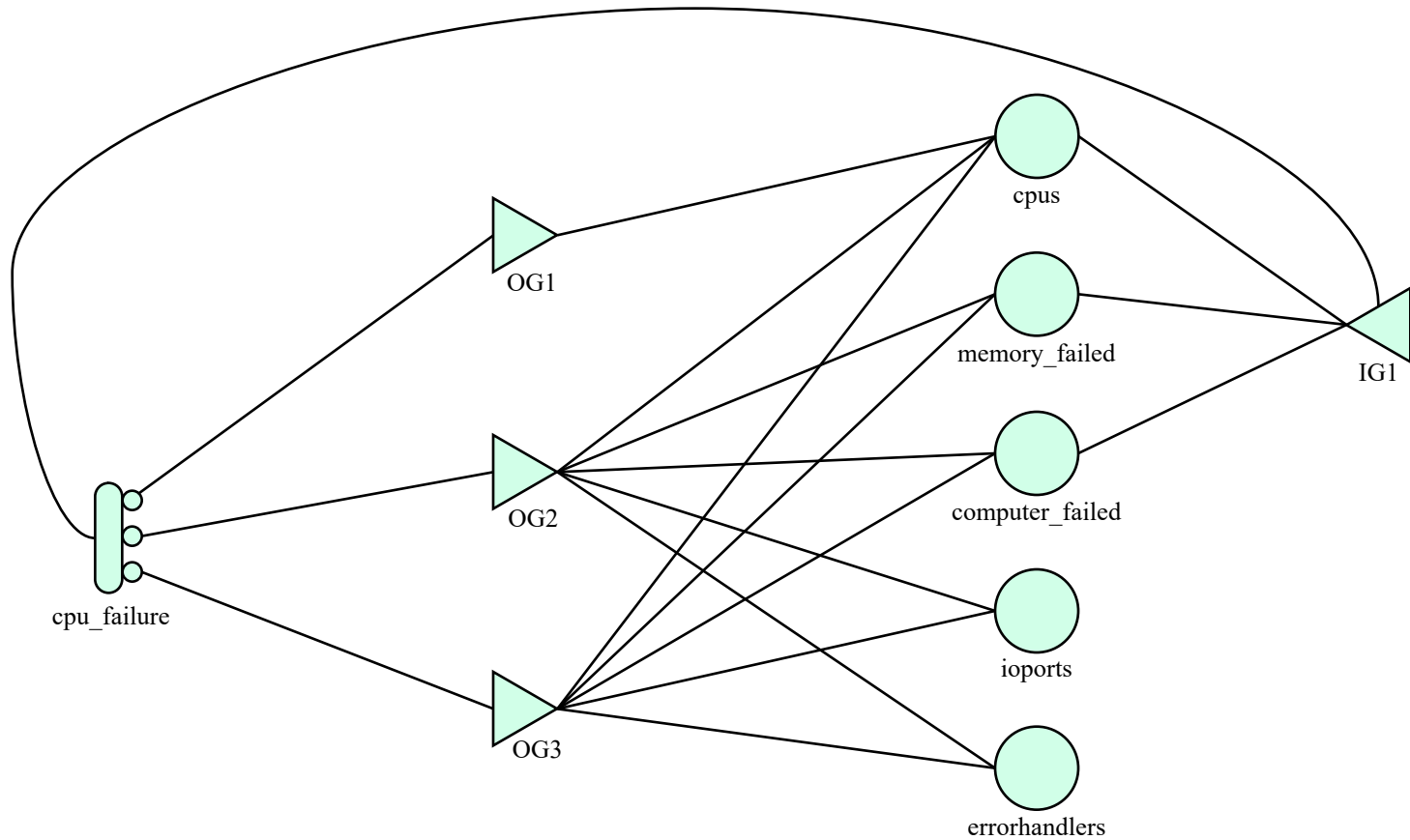
- We don't care *which* component failed or what particular failed state the model is in. Therefore, we lump all failure states into the same state.
- **We don't care which computer or which module is in what state.** Therefore, we make use of replication to further reduce the number of states.
- We 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

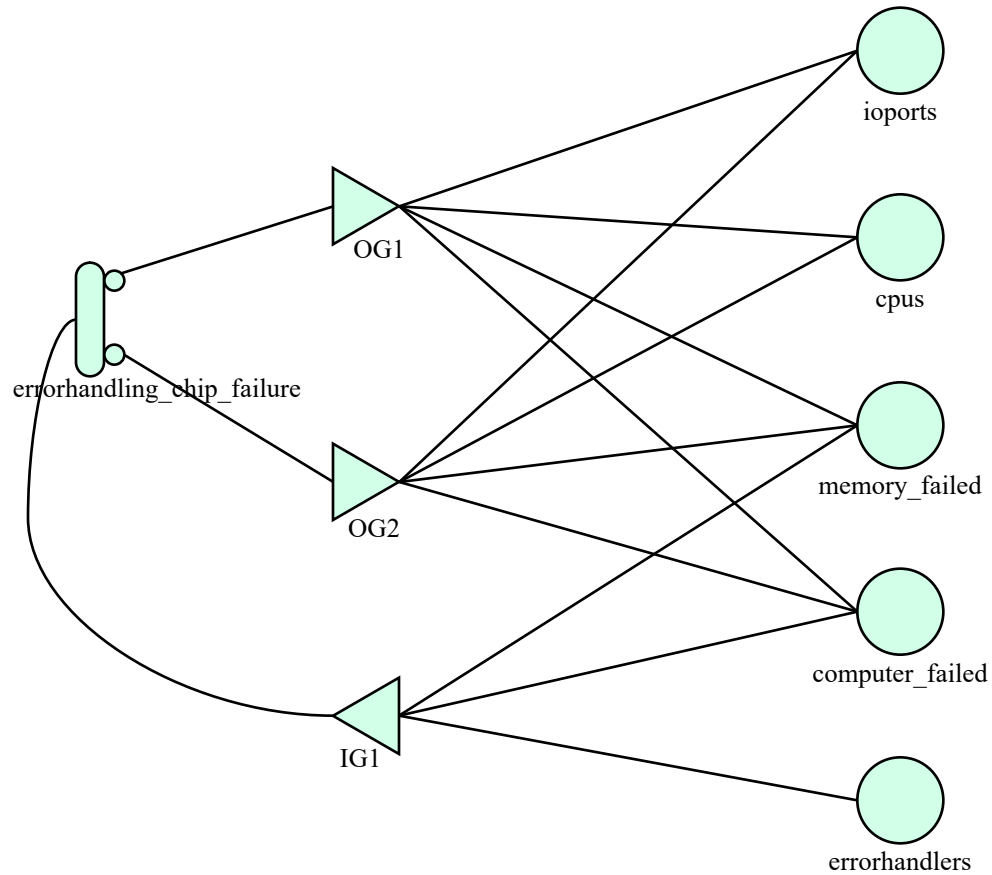
cpu_modules SAN, cont.

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;

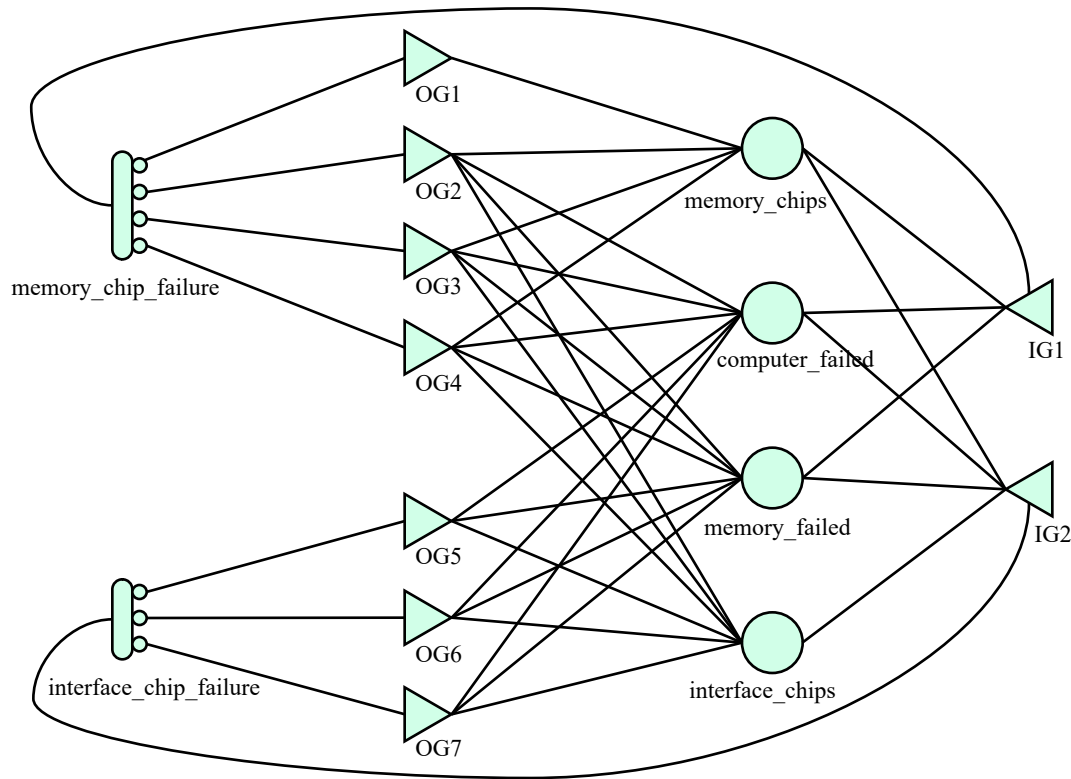
Different failures have different impacts on processor/system state

errorhandlers SAN



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

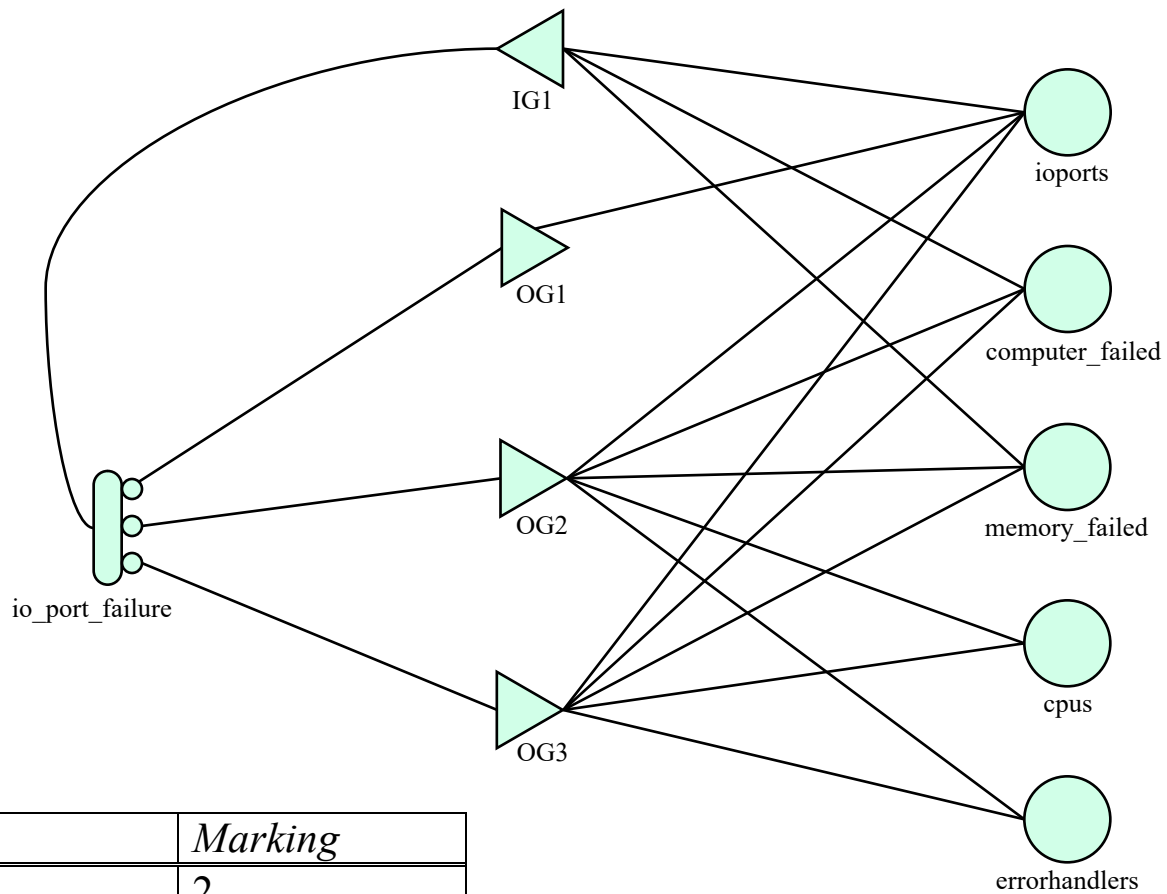
memory_module SAN



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

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

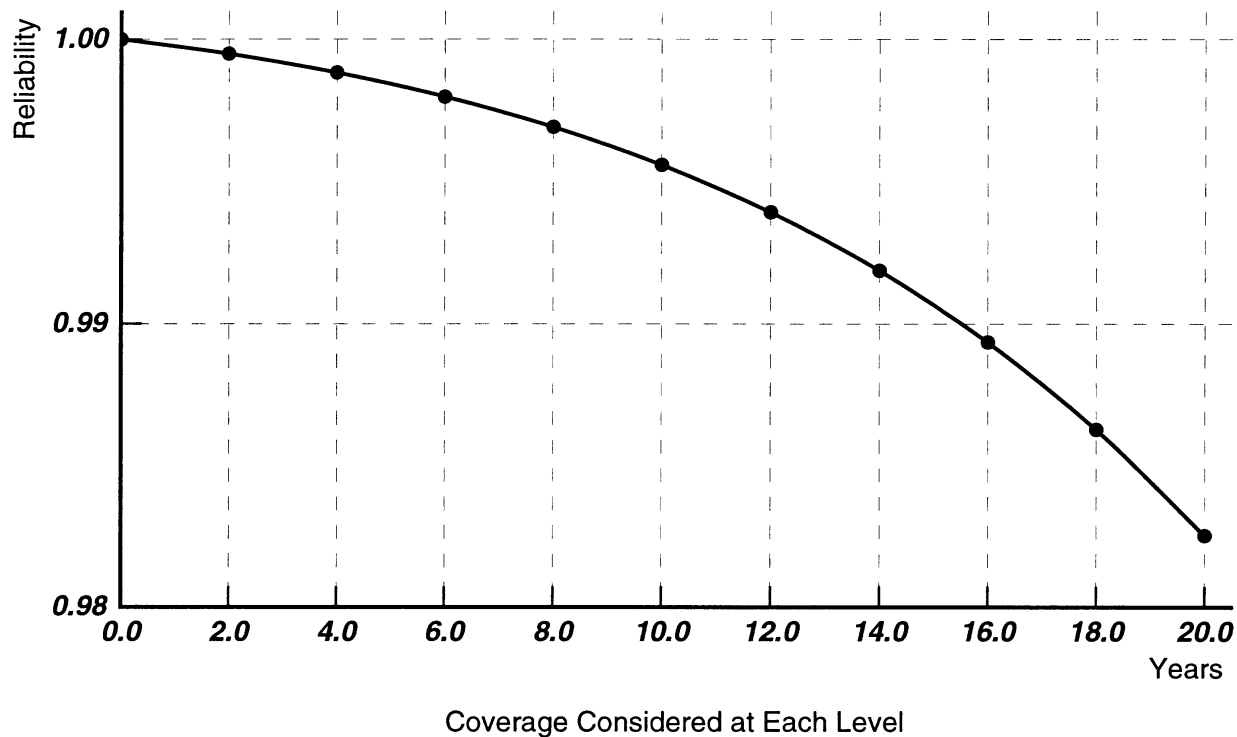
io_port_modules SAN



<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>

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

Motivating Example

- Let's say you wrote an amazing piece of code, call it
 - `my_amazing_code(x)`
- You now set out to measure the performance of your code
- What's the typical way to achieve that?

Motivating Example

Why this value?

```
#define N 1000
#define I_AM_AWSEOME 1

void my_amazing_code(int x) {};

int main(int argc, char **argv)
{
    struct timeval t1, t2;
    int i;
    int x;
    double ttime, elapsedtime;

    srand(time(NULL));
    for (i = 0; i < N; ++i) {
        x = rand();
        gettimeofday(&t1, NULL);
        my_amazing_code(x);
        gettimeofday(&t2, NULL);

        elapsedtime = (t2.tv_sec - t1.tv_sec) * 1000.0;
        elapsedtime += (t2.tv_usec - t1.tv_usec) / 1000.0;

        ttime += elapsedtime;
    }

    ttime = elapsedtime / (double)N;

    printf("My great program runs in %lf\n", ttime);
    printf("Submit paper with amazing result!\n");

    return I_AM_AWSEOME;
}
```


How did this paper get accepted?

Why is this a good estimator?

Goals

- We would like to run simulations to estimate a parameter of interest θ
 - e.g., availability, reliability, performance, average number of jobs in the system, etc.
- After obtaining our data from the simulation runs, we will use an estimate $\hat{\theta}$
 - But what is the precision of $\hat{\theta}$?
- We will discuss two measure of the precision of $\hat{\theta}$
 - The variance
 - The width of a confidence interval for $\hat{\theta}$
- Our eventual goal would then to be
 - Estimate the variance or the confidence interval
 - Figure out how many simulation runs do we need

Recall: Types of Simulation

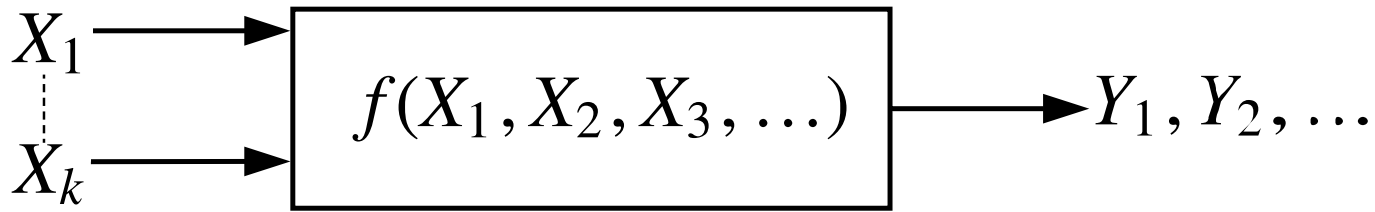
- We distinguished between two types of simulation approaches: terminating and non-terminating
- Terminating simulation  This class
 - Run for a specific duration T_E or until a specific event E happens
 - Start at time 0 under well-specified initial conditions
 - Example: Bank opens at 8:30 a.m., closes at 4:30 p.m.
 - We are only interested in these 480 minutes
- Non-terminating simulation
 - Run for a very long period of time
 - Goal is to study the long run (steady-state) properties of the system
 - Properties that are not influenced by the initial conditions
 - Example: network packet switching, queueing systems, hospital emergency rooms

Output Data

- The outputs of a model consist of one or more **random variables**
 - Why?

Output Data

- The outputs of a model consist of one or more **random variables**
 - Why?
 - We can look at a model as an input-output transformation of a given set of random variables (e.g. exponentially distributed inter-arrival times)



- Example: Consider an M/G/1 queueing example
 - Poisson arrival rate
 - Normal service time
- Suppose we are interested in the long-run mean queue length, i.e. L_Q
 - We run the simulation for 5000 time units, and we divide it into 5 equal sub-intervals of 1000 time units
 - Average the number of customers in the queue in each sub-interval

Stochastic Nature of Output Data

- We performed 3 independent replications, and we obtain the following

Batching Interval (minutes)	Batch, j	Replication		
		1, Y_{1j}	2, Y_{2j}	3, Y_{3j}
[0, 1000)	1	3.61	2.91	7.67
[1000, 2000)	2	3.21	9.00	19.53
[2000, 3000)	3	2.18	16.15	20.36
[3000, 4000)	4	6.92	24.53	8.11
[4000, 5000)	5	2.82	25.19	12.62
[0, 5000)		3.75	15.56	13.66

- Variability in stochastic simulation both
 - Within a single replication
 - Across different replications

Not independent!

Independent and identically distributed (i.i.d)

Current Setup

- We run k independent simulations and obtain k i.i.d. random variables

$$Y_1, Y_2, \dots, Y_k$$

- where,

$$E[Y_i] = \theta, \text{ and, } Var(Y_i) = \sigma^2$$

Quantity we want to estimate

- Our purpose is to devise an estimator $\hat{\theta}$ of θ
 - Evaluate how good of an estimator that is
 - Obtain insight into when to stop running simulations, i.e., how large must k be