Module 4: Stochastic Activity Networks

William H. Sanders and David M. Nicol
Department of Electrical and Computer Engineering and
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
whs@crhc.uiuc.edu

http://www.crhc.uiuc.edu/PERFORM

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 *UltraSAN*
- 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.

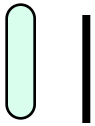
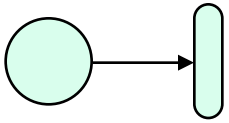
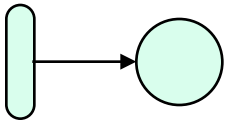
Examples:

- 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:

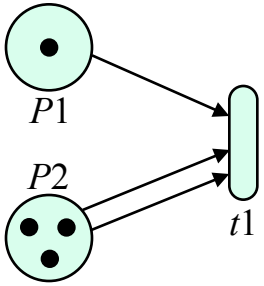
- Places:  which contain tokens, and are like variables
- tokens:  which are the "value" or "state" of a place
- transitions:  (timed,untimed) change the #tokens in places
- input arcs:  which connect places to transitions
- output arcs:  which connect transitions to places

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:

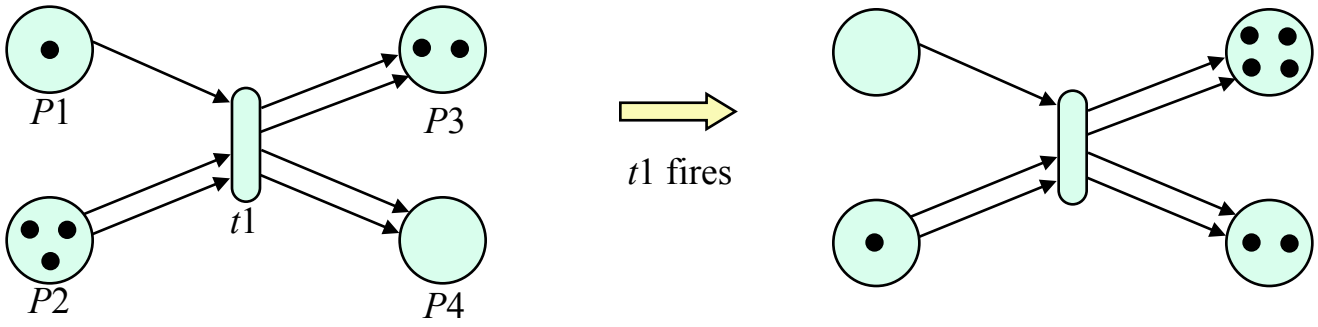


Transition $t1$ is enabled.

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 timed 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.
- Enabled immediate transitions are transient, state changes nondeterministically

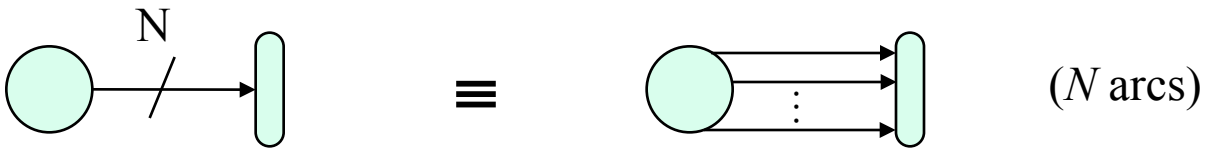
- Since the exponential distribution is memoryless, one can say that transitions that remain enabled continue or restart, as is convenient, without changing the behavior of the network.

SPN Example: Readers/Writers Problem

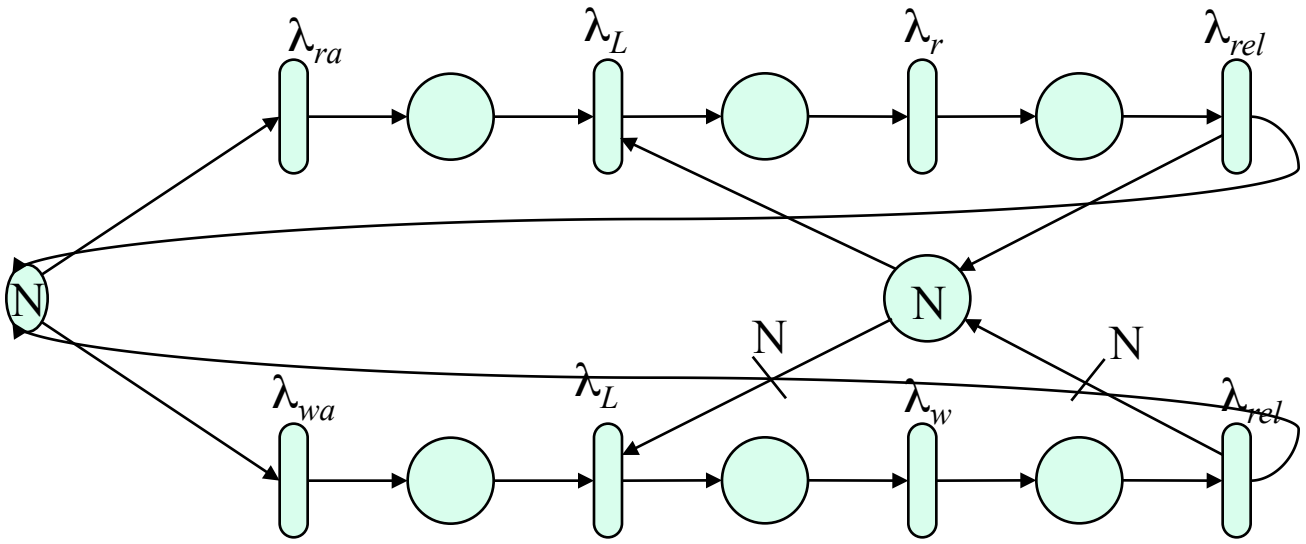
There are at most N requests in the system at a time. Read requests arrive at rate λ_{ra}, and write requests at rate λ_{wa}. Any number of readers may read from a file at a time, but only one writer may write at a time. A reader and writer may not access the file at the same time.

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 inhibits a transition if the connected place has "too many" tokens, 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

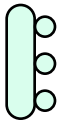
The need for more expressive modeling languages has led to several extensions to stochastic Petri nets. One extension that we will examine is called *stochastic activity networks*. Because there are a number of subtle distinctions relative to SPNs, stochastic activity networks use different words to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

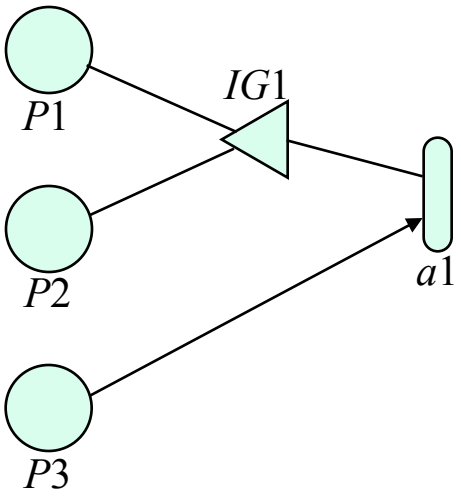
- enabling_function (state) → boolean; also called the *enabling predicate*
- input_function(state) → state; rule for changing the state of the model

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 ≥ number of arcs.

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

Example SAN Enabling Rule

Example:



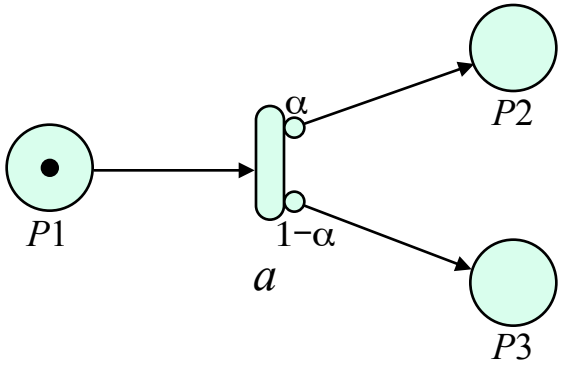
IG1 Predicate:

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

Activity $a1$ is enabled if $IG1$ predicate is true (1) and $MARK(P3) > 0$. (Note that "1" is used to denote true.)

Cases

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



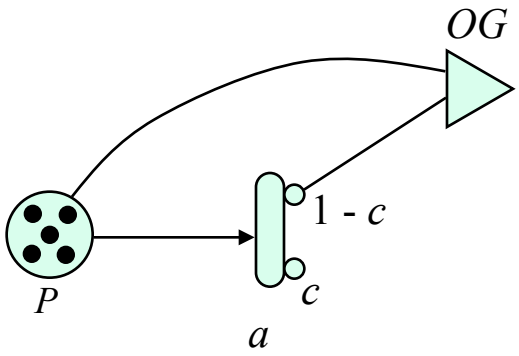
When activity *a* completes, a token is removed from place *P*1, and with probability α, a token is put into place *P*2, and with probability 1 - α, a token is put into place *P*3.

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.

Example

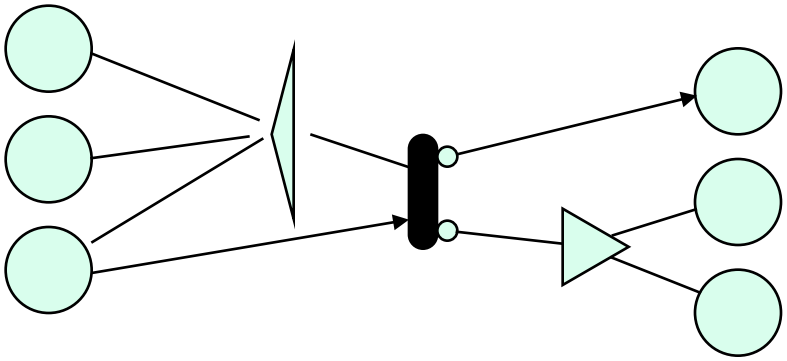


OG Function

```
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

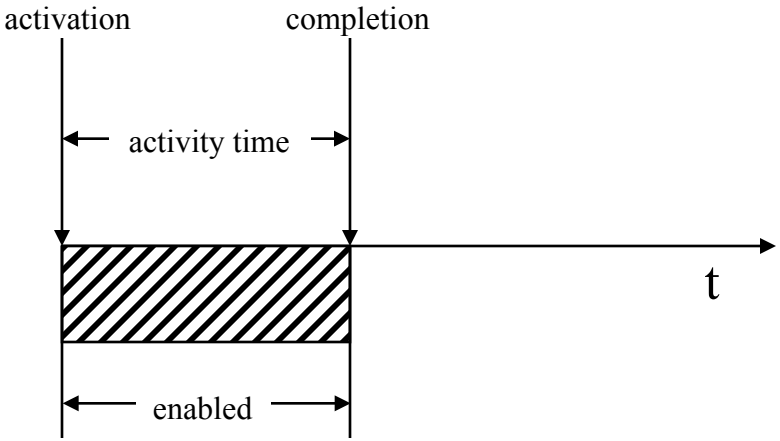
1. *activation* - time at which an activity begins

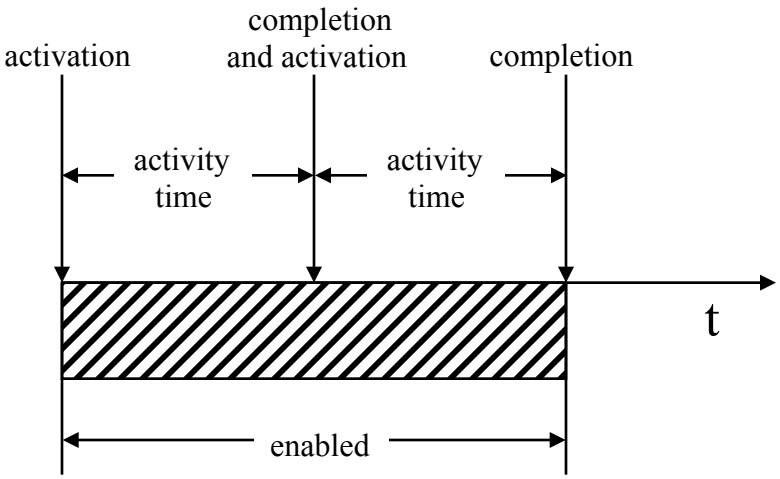
2. *completion* - time at which activity completes

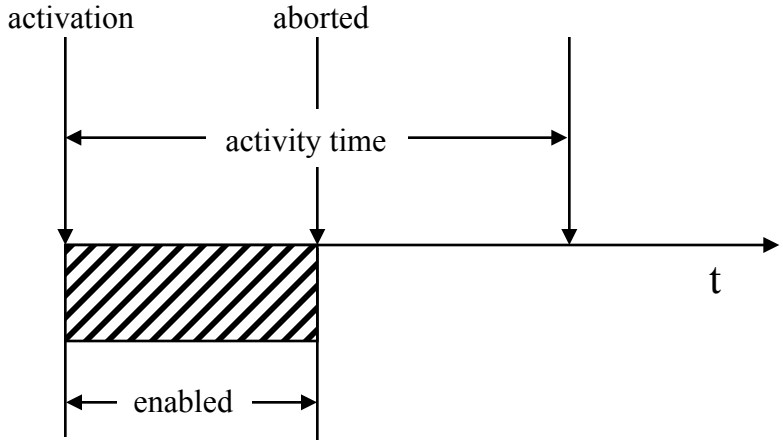
3. *abort* - time, after activation but before completion, when activity is no longer enabled

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.

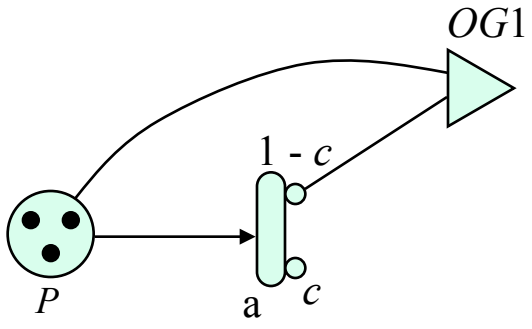
Ordering is important, since effect of actions can be marking-dependent.

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



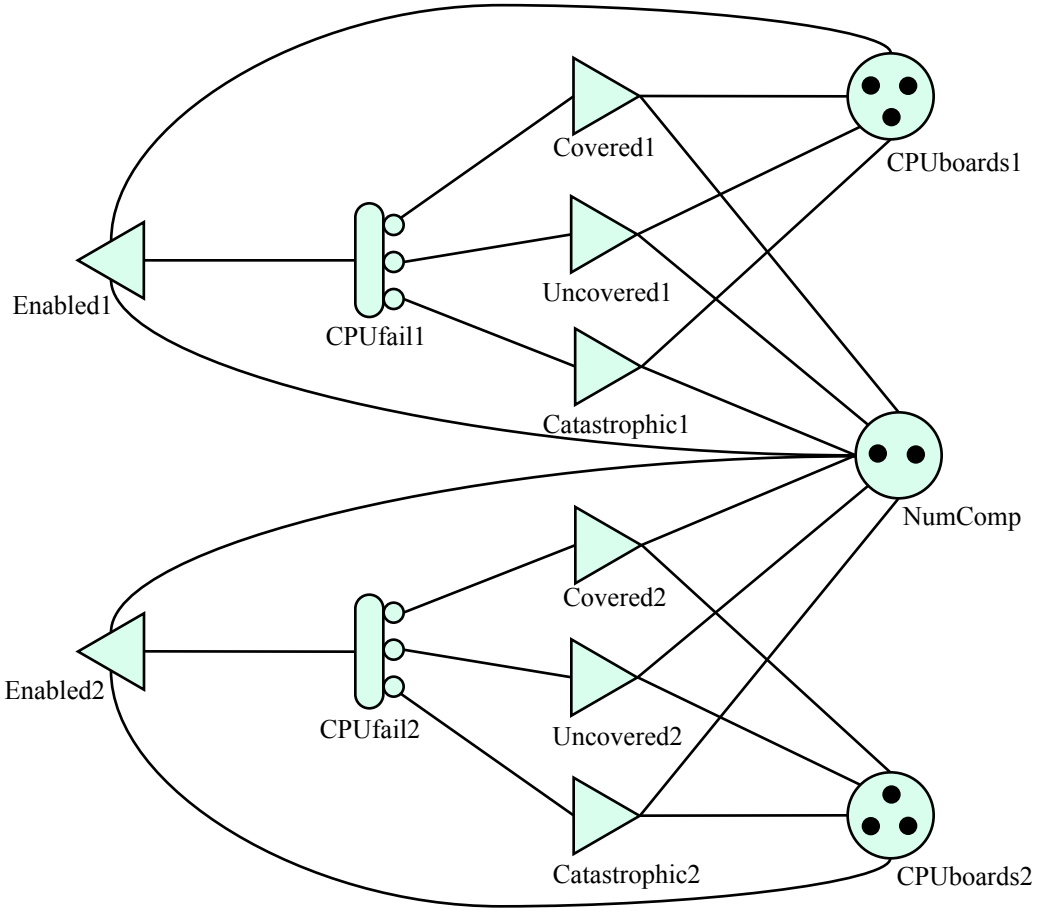
	a
case 1	0.1 + 0.02 * MARK(P)
case 2	0.9 – 0.02 * MARK(P)

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	Predicate *MARK(CPUboards1 > 0) && MARK(NumComp) > 0*
	Function *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;*

General Delay Distributions, cont.

- SANs (and their implementation in *UltraSAN*) support many activity time distributions, including:
  - Exponential
  - Hyperexponential
  - Deterministic
  - Weibull
  - Conditional Weibull
  - Normal
  - Erlang
  - Gamma
  - Beta
  - Uniform
  - Binomial
  - 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 *UltraSAN* 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.

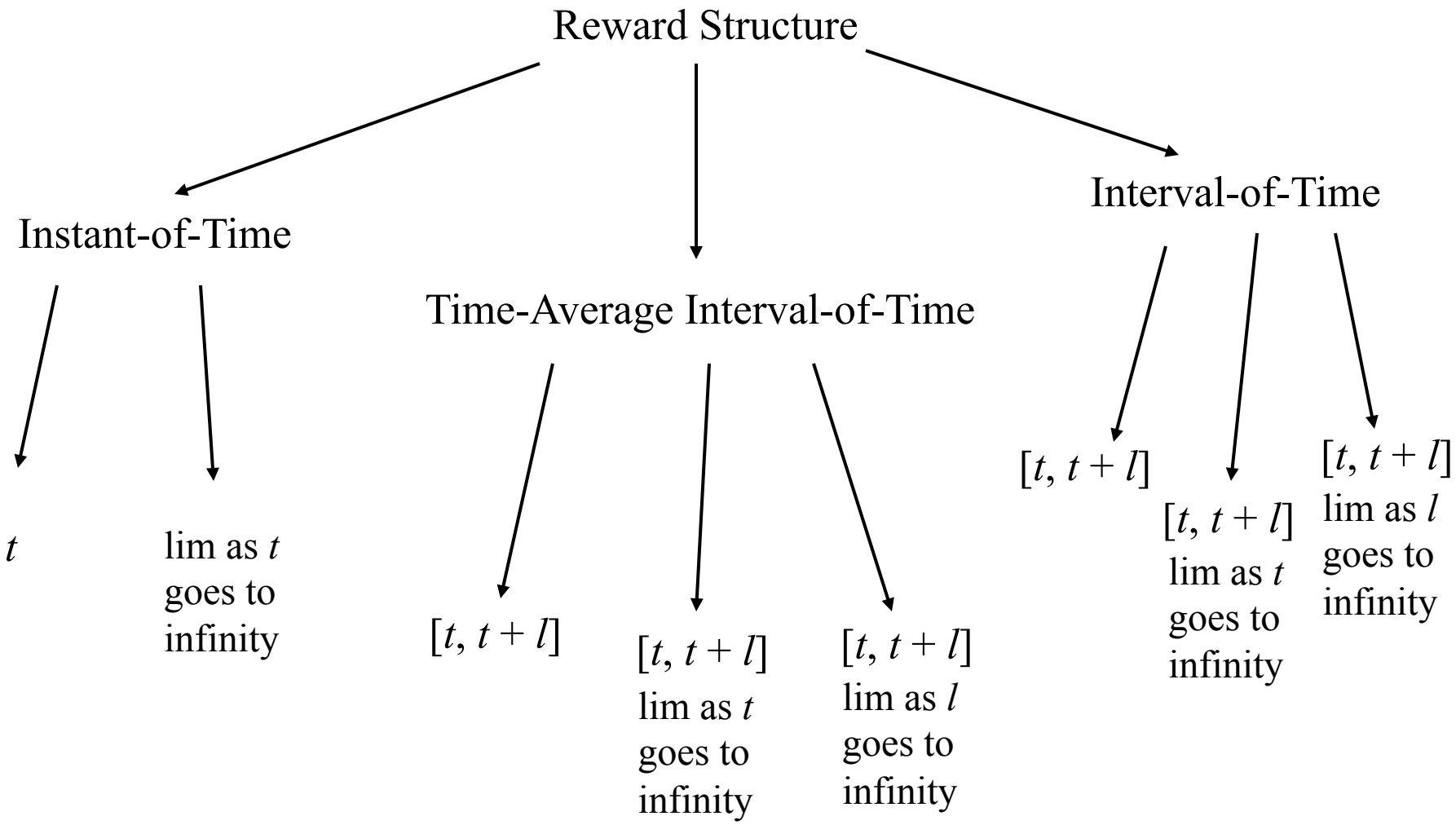
Reward Variables

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 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

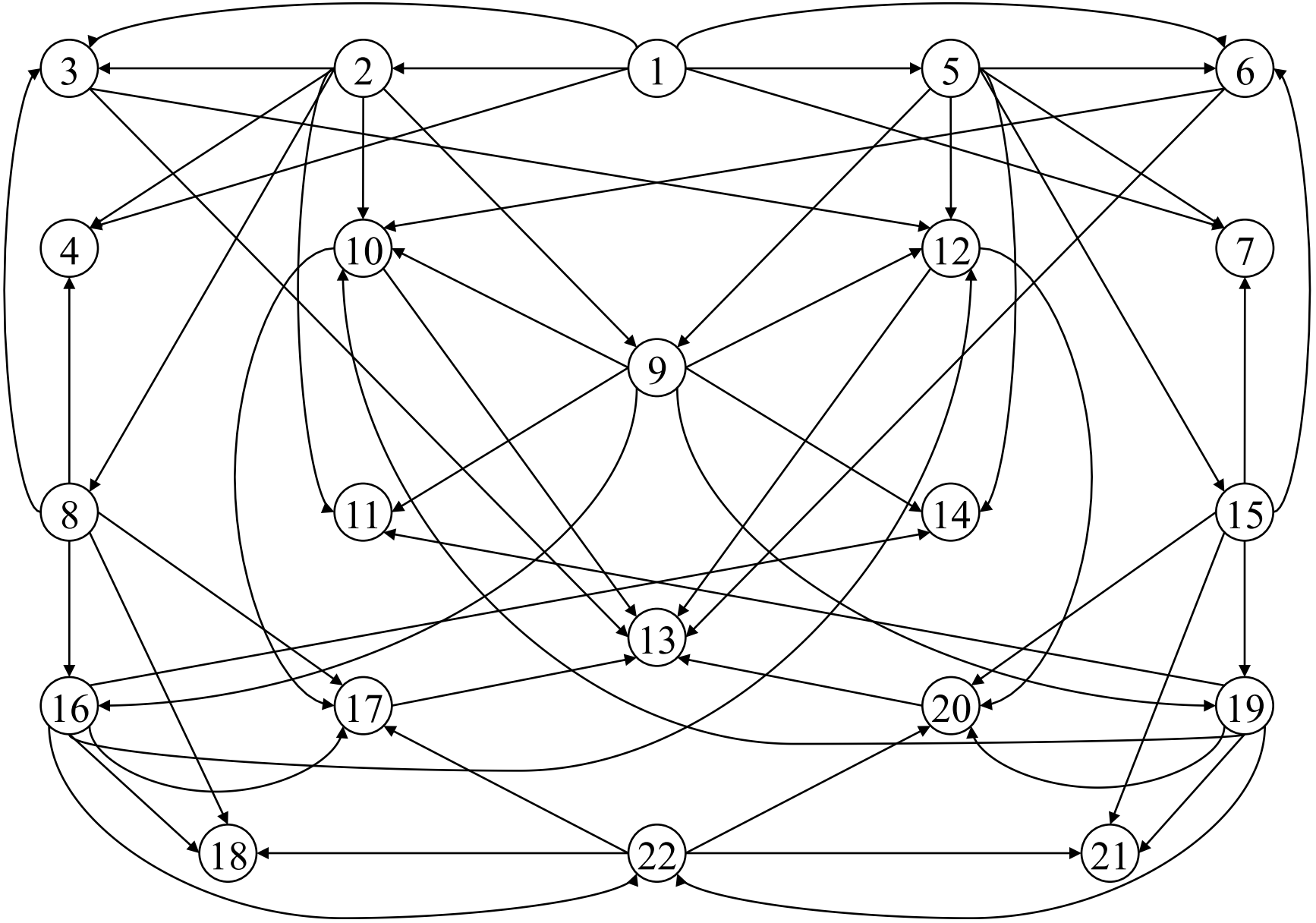
Reward Variables for Computer Failure Model

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

State Space (Generated by *UltraSAN*)

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+p2λ),(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 (State Transition Rates Not Shown)



Model Composition

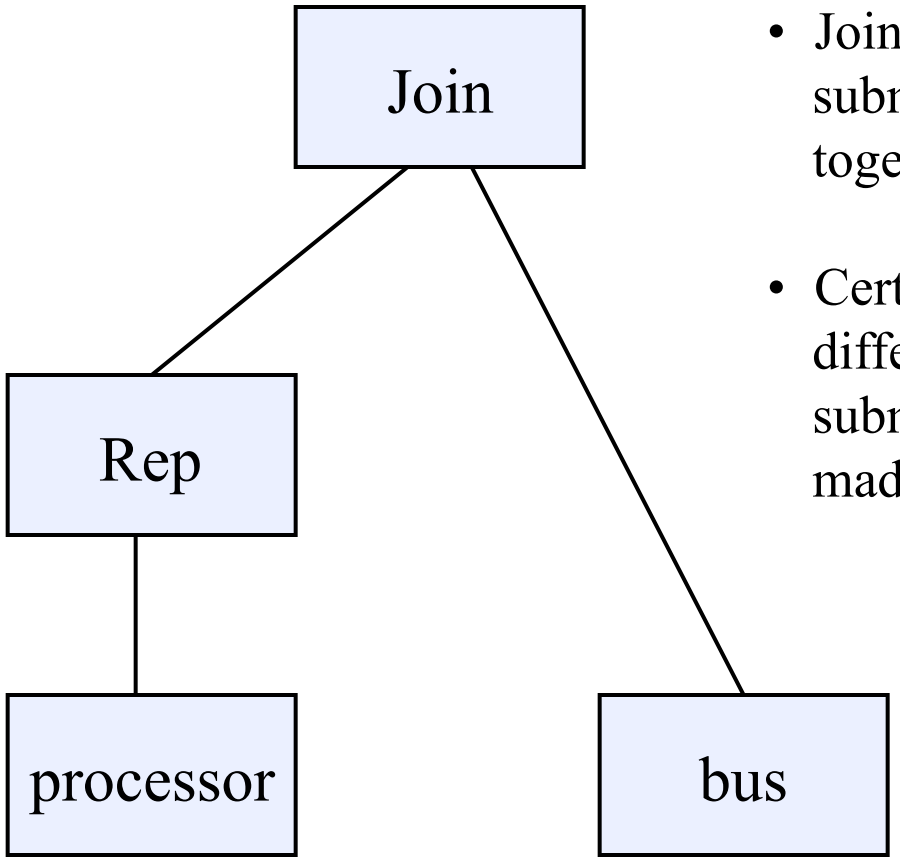
A composed model is a way of connecting different SANs together to form a larger model.

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

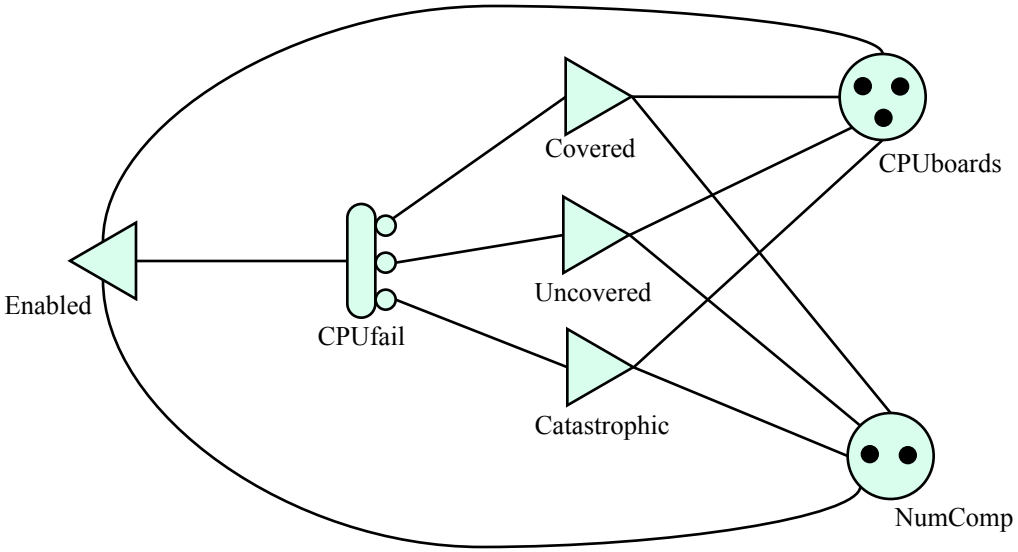
- Replicate submodel a certain number of times
- Hold certain places common to all replicas

Rationale

There are many good reasons for using composed models.

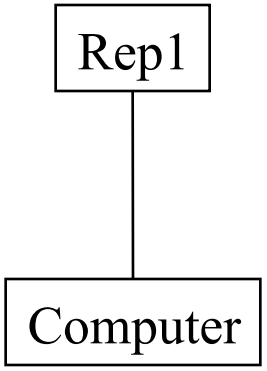
- Building highly reliable systems usually involves redundancy. The replicate operation models redundancy in a natural way.
- Systems are usually built in a modular way. Replicates and Joins are usually good for connecting together similar and different modules.
- Tools can take advantage of something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

Computer Failure Model Revisited: Single *computer* Model



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

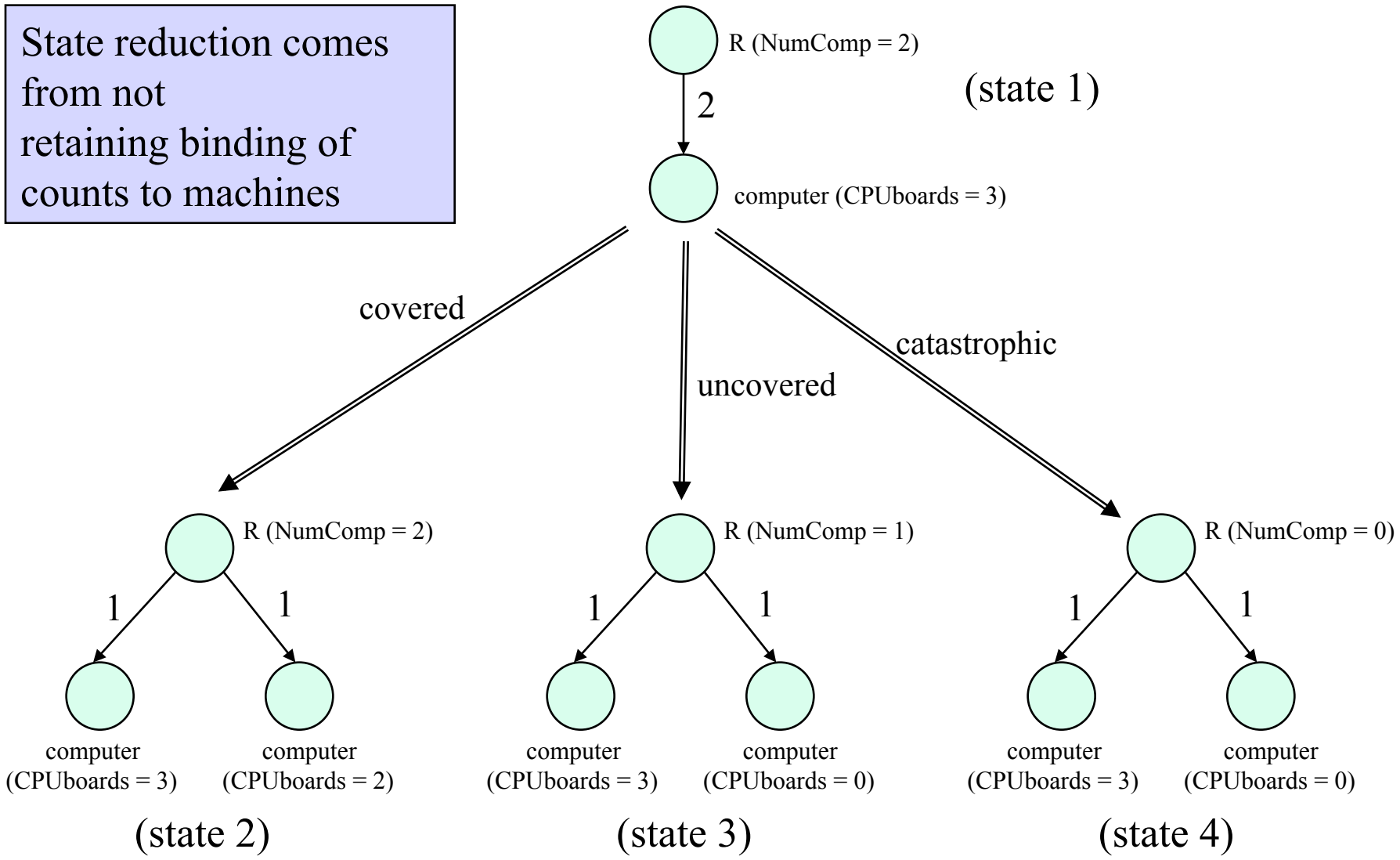
Composed Model for Computer Failure Model



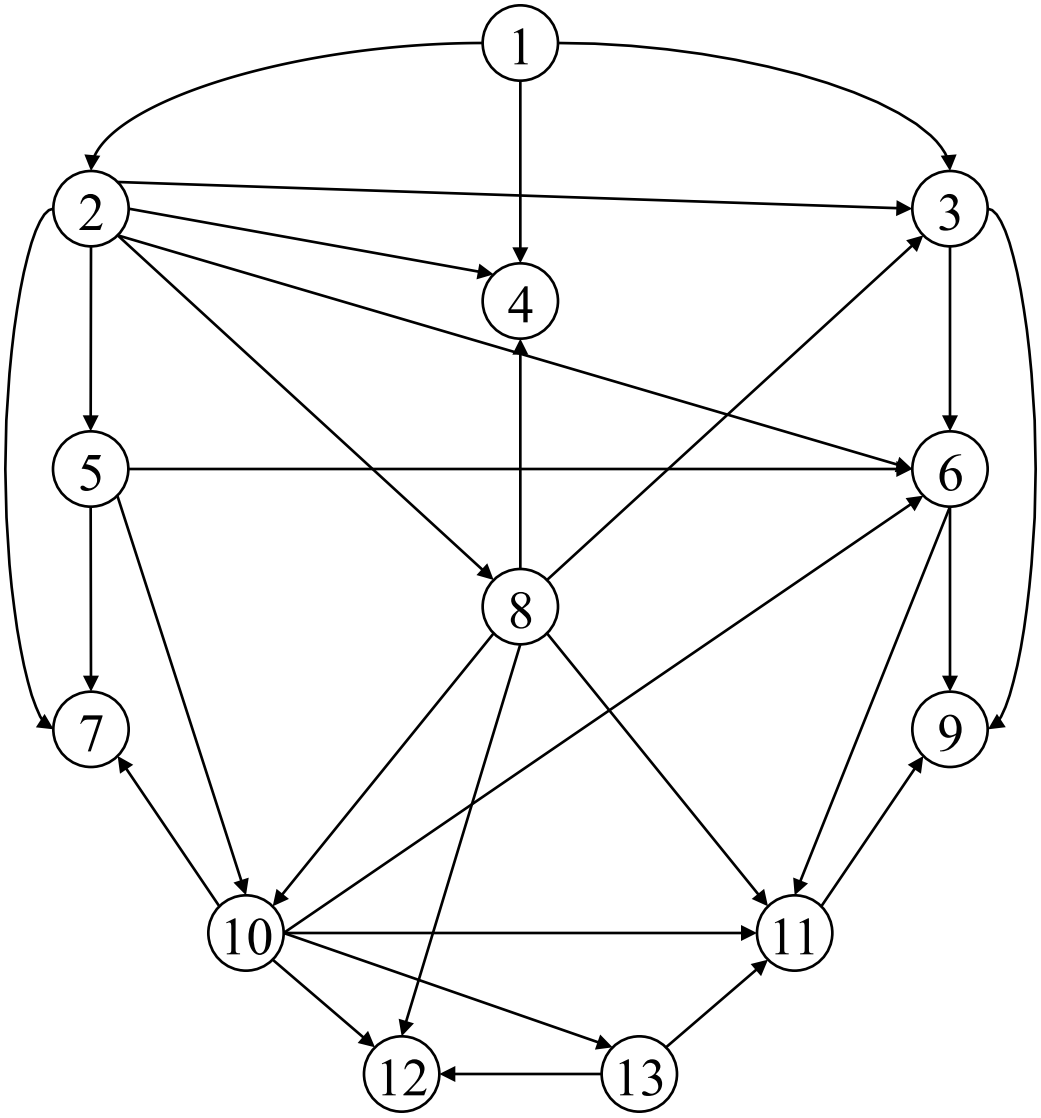
Node	*Reps*	*Common Places*
Rep1	*2*	*NumComp*

Example Reduced Base Model States and Transitions

State reduction comes from not retaining binding of counts to machines



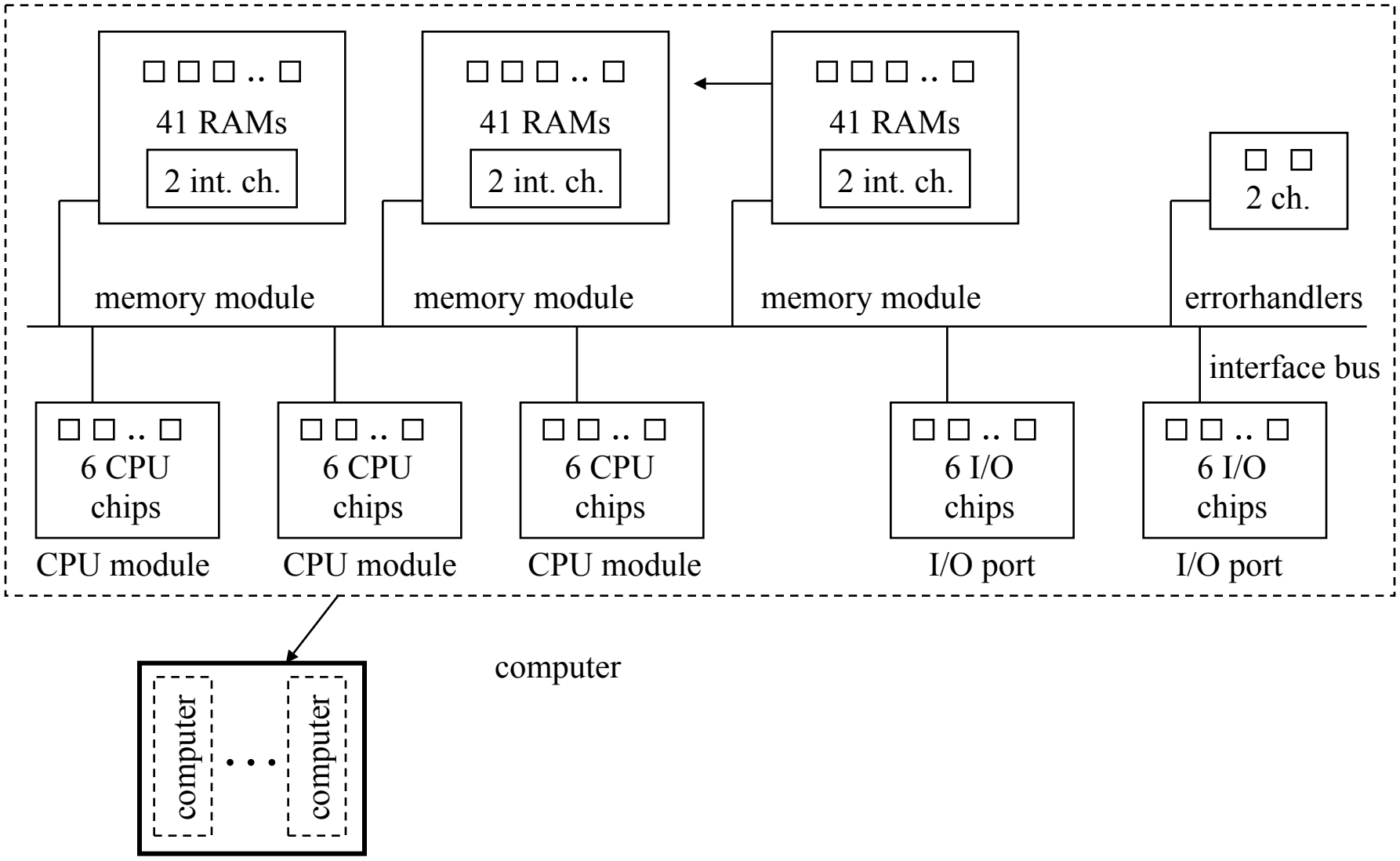
Markov Chain of Reduced Base Model (State Transition Rates not Shown)



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:

Redundant Component	*Fault Coverage Probability*
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

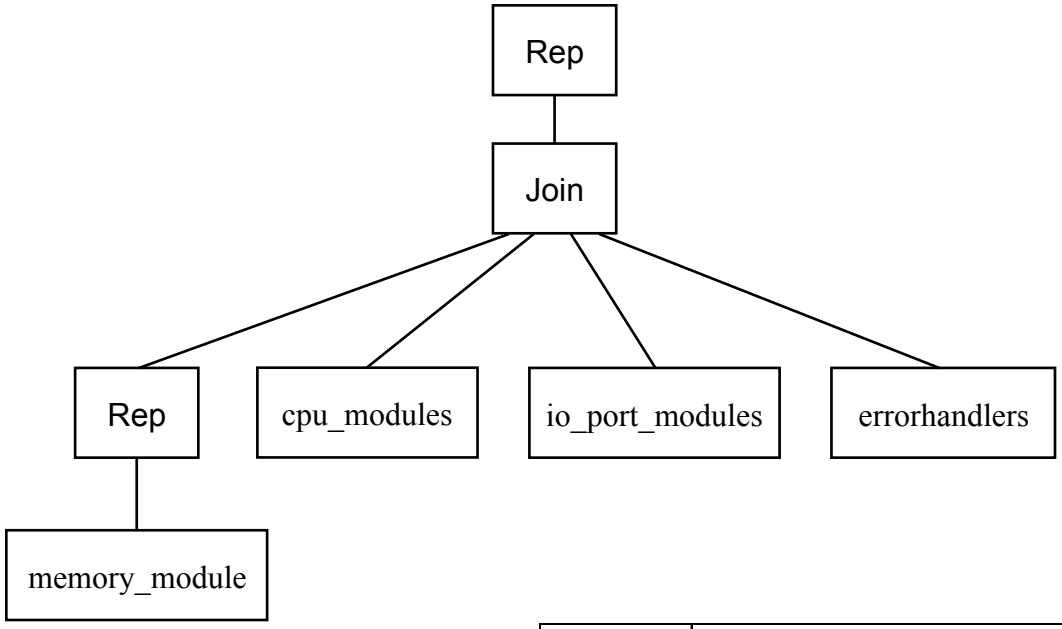
- Each SAN models the behavior of the module in the event of a module component failure.

Tricks of the Trade

Since we intend to solve this model analytically, we want 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.

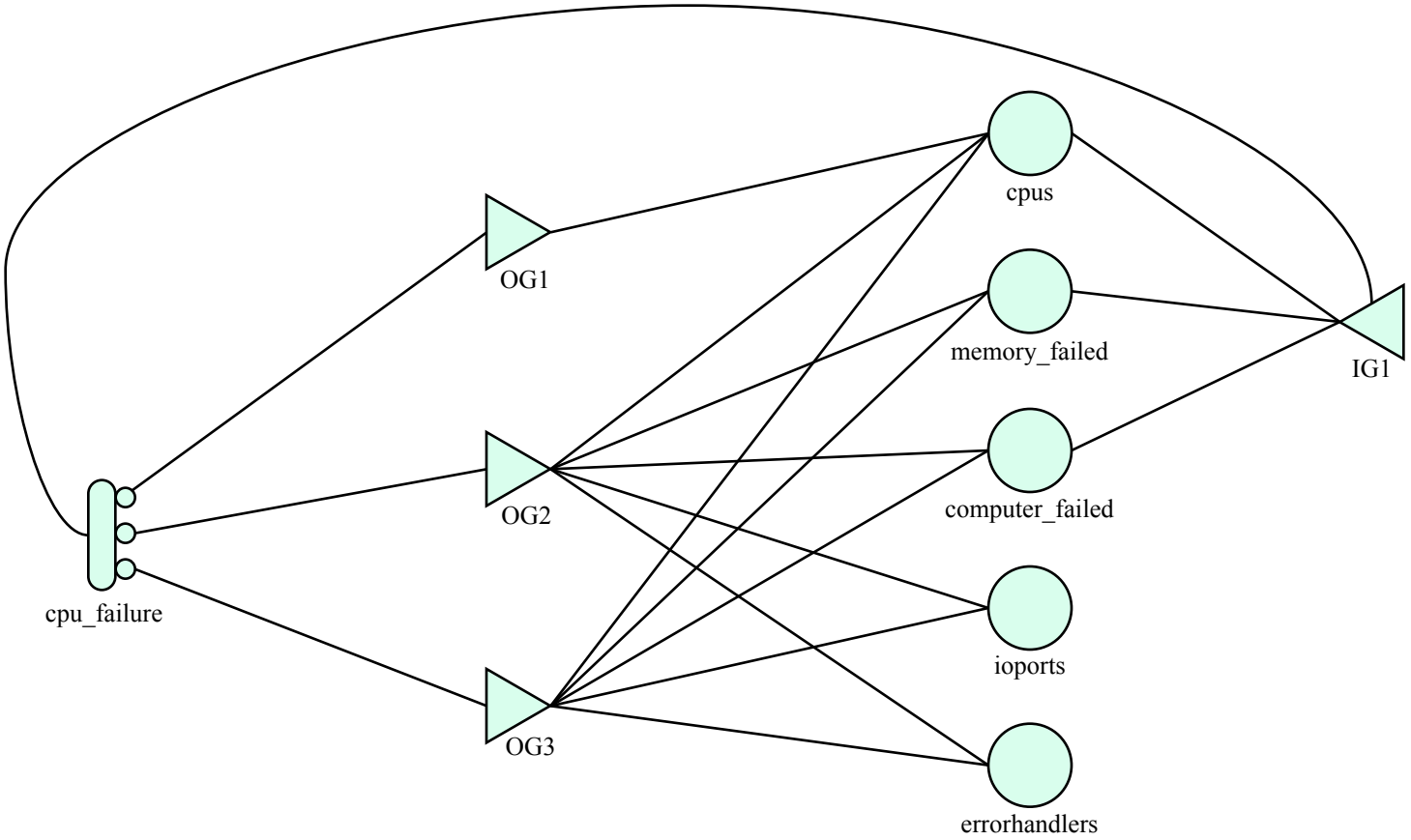
Composed Model



Node	*Reps*	*Common Places*
Rep1	*3*	*computer_failed*
		memory_failed
Rep2	*2*	*computer_failed*

Node	*Common Places*				
Join1	*Subtree*	1	2	3	4
	computer_failed	•	•	•	•
	memory_failed	•	•	•	•
	cpus		•	•	•
	errorhandlers		•	•	•
	ioports		•	•	•

cpu_modules SAN



Place	*Marking*
cpus	3
ioports	2
errorhandlers	2
memory_failed	0
computer_failed	0

cpu_modules SAN, cont.

cpu_modules input gate predicates and functions:

Gate	*Enabling Predicate*	*Function*
IG1	*(MARK(cpus) > 1) && (MARK(memory_failed) < 2) && (MARK(computer_failed) < 2)*	*identity*

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

cpu_modules activity time distributions:

Activity	*Distribution*
cpu_failure	expon(0.0052596 * MARK(cpus))

cpu_modules SAN, cont.

cpu_modules case probabilities for activities:

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

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

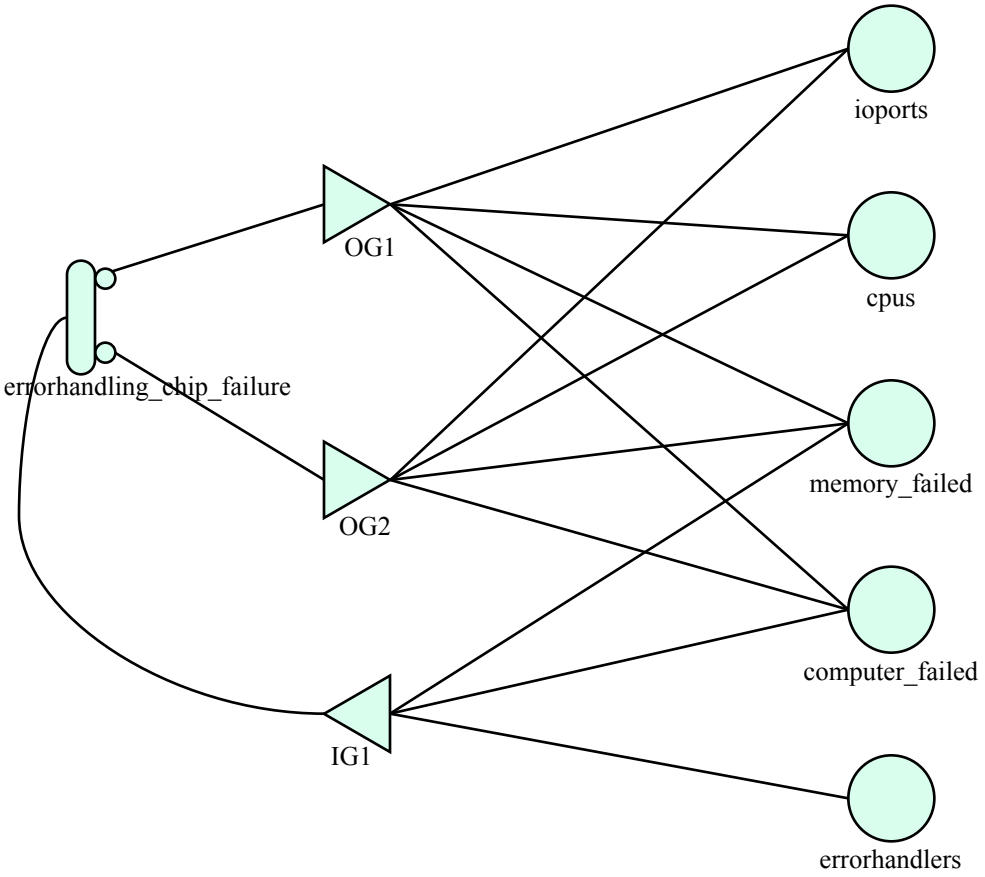
cpu_modules SAN, cont.

cpu_modules output gate functions:

Gate	*Function*
OG1	*if (MARK(cpus) == 3)* *MARK(cpus) – –;*
OG2	*MARK(cpus) = 0;* *MARK(ioports) = 0;* *MARK(errorhandlers) = 0;* *MARK(memory_failed) = 2;* *MARK(computer_failed) ++;*
OG3	*MARK(cpus) = 0;* *MARK(ioports) = 0;* *MARK(errorhandlers) = 0;* *MARK(memory_failed) = 2;* *MARK(computer_failed) = 2;*

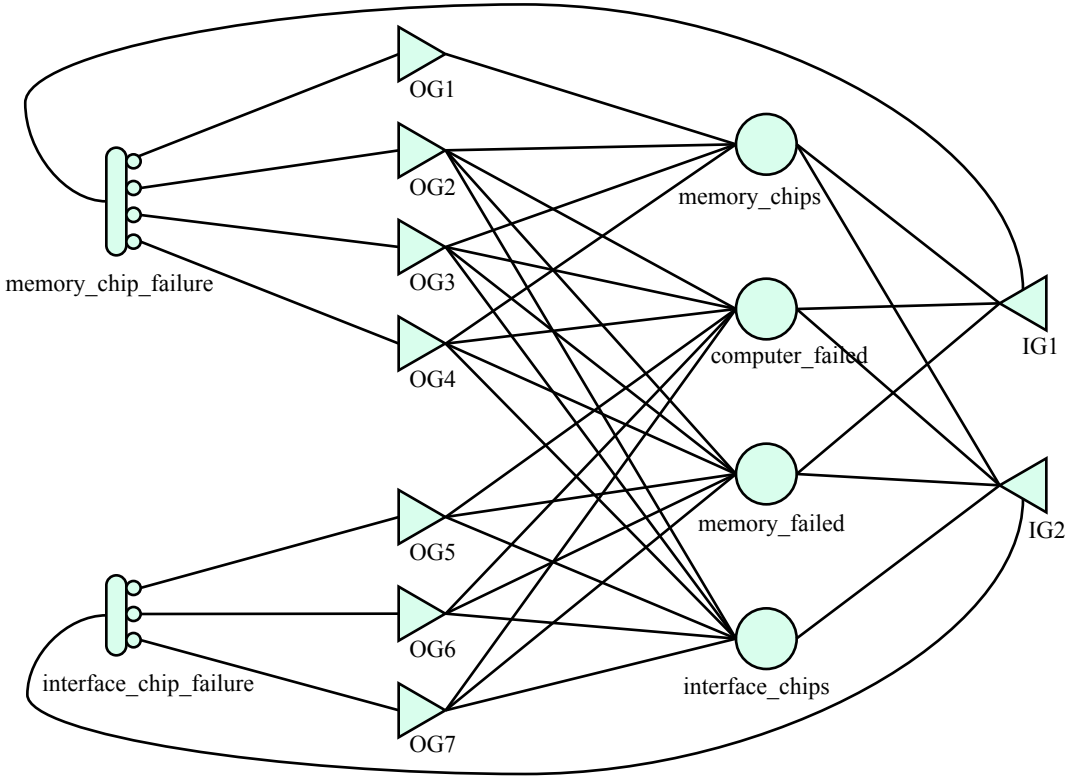
Different failures have different impacts on processor/system state

errorhandlers SAN



Place	*Marking*
errorhandlers	2
cpus	3
ioports	2
memory_failed	0
computer_failed	0

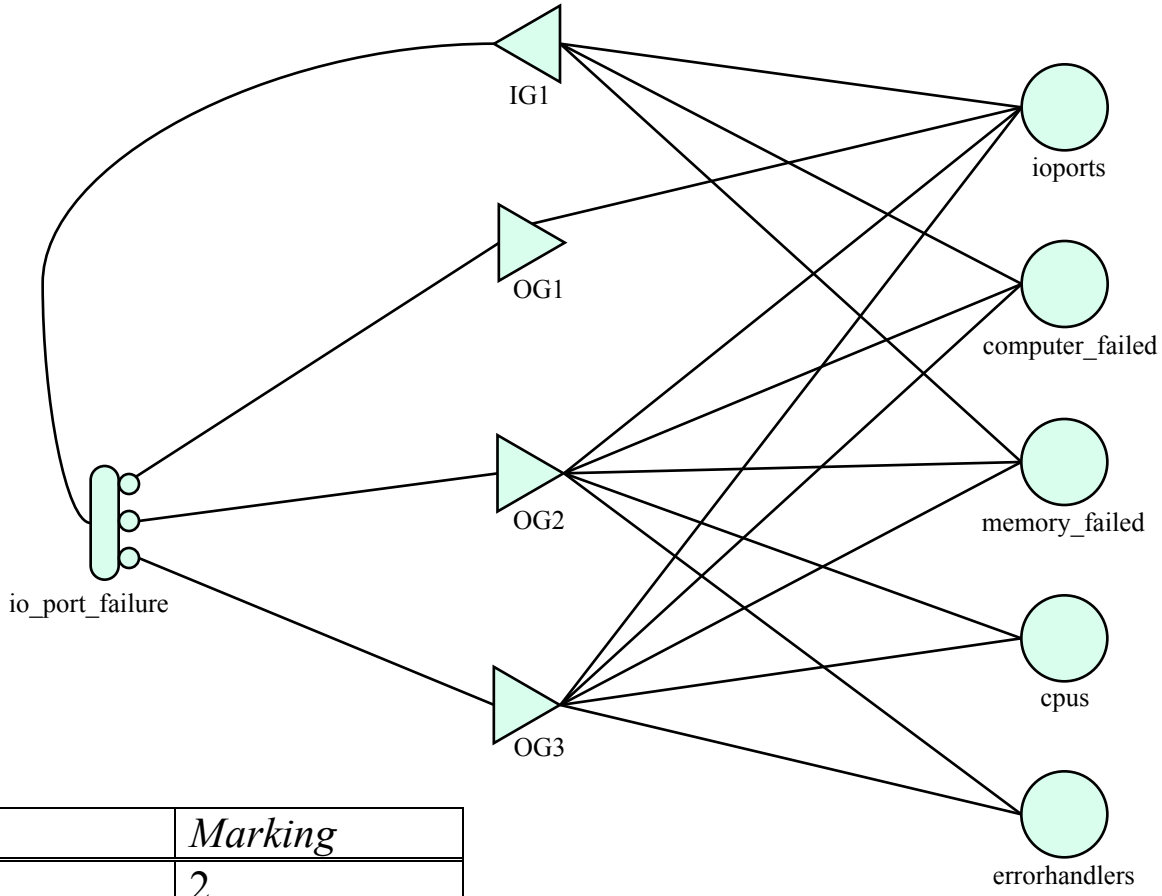
memory_module SAN



Place	*Marking*
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.

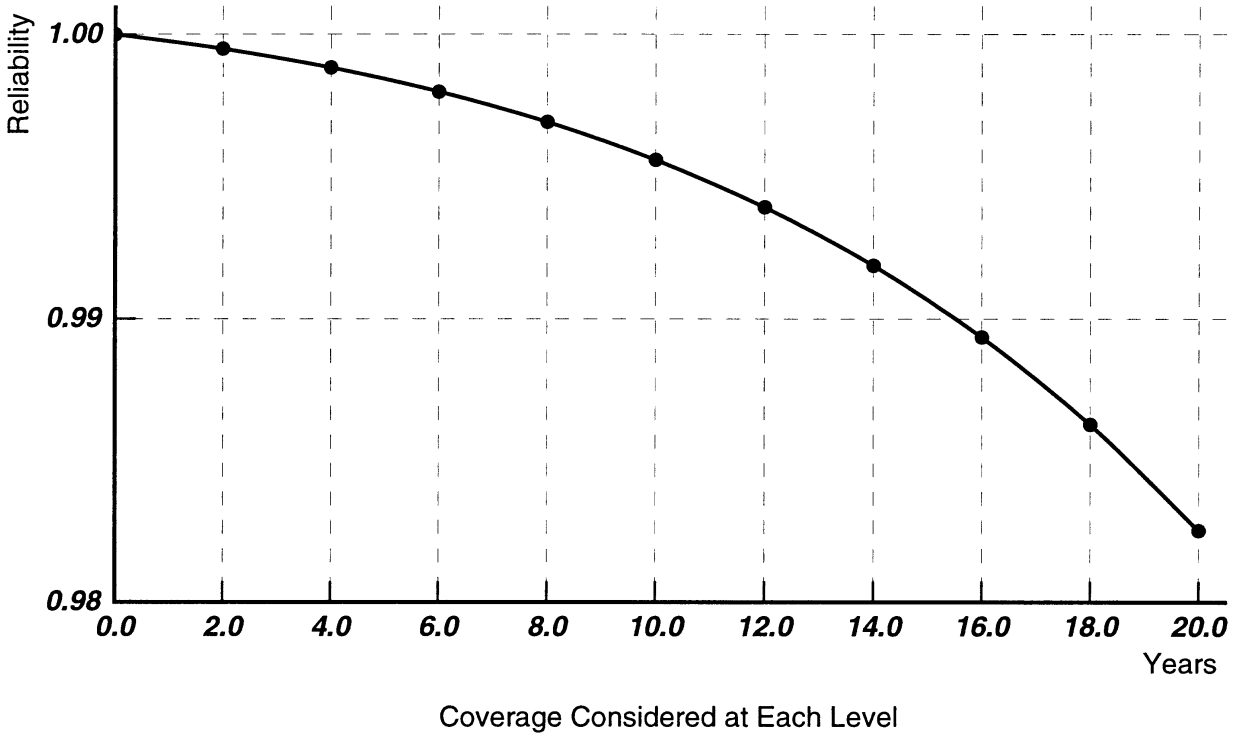
io_port_modules SAN



Place	*Marking*
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.

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