

#### Dependable and Secure Systems – Dependability

Master of Science in Embedded Computing Systems

# Quantitative Dependability Analysis with Stochastic Activity Networks: the Möbius Tool

Andrea Domenici

DII, Università di Pisa

Andrea.Domenici@iet.unipi.it

- Stochastic Activity Networks
- The Möbius tool: Modeling
- The Möbius tool: Solving models
- An example
- A case study

The *Stochastic Activity Networks* (SAN) are a wide-ranging and complex extension to Petri Nets.

Petri Net = places + marking + transitions + enabling conditions + firing rules.

Stochastic Petri Net = PN + stochastic transition delay.

Stochastic Activity Network = SPN + stochastic transition outcome + user-defined enabling conditions + user-defined firing rules +  $\dots$ 

William H. Sanders and John F. Meyer, "Stochastic Activity Networks: formal definitions and concepts", in Lectures on formal methods and performance analysis: first EEF/Euro summer school on trends in computer science, 2002.

## **Stochastic Activity Networks (2)**

NOTE: SAN's have *activities* instead of PN transitions. The terms *activity*, *transition*, and *action* will be used interchangeably.

Activities may be *timed* or *instantaneous* (or *immediate*).

*Enabling conditions*: activities are enabled by user-defined **input predicates** associated with **input gates**.

An input predicate is a Boolean function of the net marking.

*Firing rules*: user defined functions specifying the next marking can be associated with input gates (**input functions**) and **output gates** (**output functions**).

Stochastic transition outcome: Alternative results of an activity can be specified as mutually eclusive **cases** associated with the activity.

Each case has a probability defined by a function of the marking (it may be a constant).

#### THE MÖBIUS TOOL: MODELING



August Ferdinand Möbius (1790, 1868).

A. F. Milius.

## The Möbius tool

The Möbius environment provides:

- **Graphical editor** to make (*atomic*) SAN models.
- Hierarchical composition of models.
- Reward models to define and compute *performance variables*, i.e., quantitative properties related to system performance or dependability.
- Numerical solution of Markov chain equations, if certain constraints on the model are satisfied.
- System simulation satisfying user-defined statistical parameters, such as confidence level and confidence interval (not covered in this seminar).

G. Clark *et al.*, "The Möbius modeling tool", in 9th Int. Workshop on Petri Nets and Performance Models, 2001.

The Möbius Manual, Version 2.4 Rev. 1,

https://www.mobius.illinois.edu/docs/MobiusManual.pdf

#### Creating a project (1)

Project Tools Help			
E Console	<b>.</b>	6	⊒ ▼
Main			
Möbius 2.4.0 started at 1:52:32 PM Welcome to Möbius, andrea. Opened project ex_multiprocessor.			

By default, projects are stored in a MobiusProject directory (you may change it with [Project  $\rightarrow$  Preferences]).

- 1. Start Möbius, e.g., with mobius &
- 2. [Project  $\rightarrow$  New]

#### **Creating a project (2)**

Project Information Please enter the appropriate information for the new project.	Mobius 2.4 Möbius ex_multiprocessor ex_multiprocessor ex_multiprocessor
Name: ex_multiprocessor Author(s): andrea Description	Atomic Composed Reward Study Transformer
Cancel Finish 1. Type in project name.	

## Creating an atomic model (1)

cpu_module		
Atomic Model Ty	/pe:	
FaultTree Mode	9	
SAN Model		
Buckets and Ba	lls Model	_
PEPA Model		
External Atomic	Model	

- 1. [Project Editor  $\rightarrow$  Atomic  $\rightarrow$  New], w/ right button.
- 2. Select "SAN Model" and enter name.
- 3. The SAN Editor pops up.



#### Creating an atomic model (2)

Use [Elements] menu or toolbar to create SAN elements:
 Places: Blue circles.

Timed activities: Thick bars.

Instantaneous activities: Thin bars.

Input gates: Left-pointing red triangles.

Output gates: Right-pointing black triangles.

- **Extended Places:** Orange circles, we are not using them in this seminar.
- Connect elements with segments, or connected segments, or splines.
- To draw a connection, left-click on source element, left-click at intermediate points (if any), left-click on target element.
- Respect the order: place to input gate, input gate to activity, activity to output gate, output gate to place.
- Left-click and drag to move. Use cursor to group elements.
- [Right-click  $\rightarrow$  Edit] to edit an element.

Time distribution rates, case probabilities, input and output functions, input predicates, and reward functions, are specified with C++ expressions or sequences of statements.

C++ code may refer to constants, **global variables**, and **state variables**.

Global variables are system parameters (e.g., failure rates, number of components, physical properties, ...). They can be accessed from any submodel, but they must be declared in each submodel that refers to them, with [SAN Editor  $\rightarrow$  Edit  $\rightarrow$  Edit Global Variables]. The global variables are initialized in the *Study* model.

State variables are the places, and the value of a state variable is the place marking. If Place is the name of a place, its marking can be accessed, both for reading or writing, with Place->Mark().

#### **Timed activities**

== Timed Activity	/ Attributes $ imes$
Name:	cpu_failure
Time distribution function:	Exponential 💌
	1 🔺
Rate	
6.0 * failure_rate * cpus->Mark()	<u> </u>
•	
Case quantity:	3-
Case 1 Case 2 Case 3	
if (cpus->Mark() == 3)	<u> </u>
else	
return(0.0);	<b>•</b>
Execution Policy	Edit
Ok	Cancel

- Choose Time distribution function: Use Exponential.
- Specify the Rate of the time distribution function.
- Enter Case quantity: number of alternative cases.
- For each case, specify its probability.

#### Input gates

	Input G	ate Attributes		×
Name:	Input_Gate1			
Input Predicate				
(cpus->Mark() > 1) (computer_failed->1	&& (memory_failed-> Mark() < num_comp)	>Mark() < 2) &&		
•				
Input Function				
:				
Ok		Cancel	Validate	
Ok		Cancel	Validate	

- Specify the Input predicate.
- Specify the **Input function**.

If no input function is required, type a single semicolon character.

	Output Gate	Attributes		×
Name:	0G1			
Output Function				
if (cpus->Mark() == 3) cpus->M	lark();			•
Ok	Can	cel	Validate	

#### ■ Specify the **Output function**.

If no output function is required, type a single semicolon character.

Atomic models can be aggregated into **composed models**, which in turn can be aggregated, so that a Möbius model is a tree of submodels, whose leaves are atomic SAN models (actually, atomic models can be built also with other formalisms, see the Möbius documentation).

Submodels interact through **shared state variables**, i.e., shared places. When two or more submodels are composed, the user specifies which places are shared among them.

Submodels are composed by means of the Join and Rep operators.

The Join operator is used to compose different models.

The *Rep* (*replication*) operator is used to compose copies of a same model.

## Creating a composed model (2)

compo					
Compo	sed Mod	lel Type	:		
Rep/Joi	n				
Action Graph	Synchro	nizatio	n		
	1			1	

- 1. [Project Editor  $\rightarrow$  Composed  $\rightarrow$  New], w/ right button.
- 2. Select "Rep/Join" and enter name.
- 3. The Rep/Join Editor pops up.



#### Creating a composed model (3)

- Use [Elements] menu or toolbar to composed model nodes: Rep nodes: Red bars.
   Join nodes: Blue bars.
   Submodel nodes: Black bars.
- Connect nodes with segments, or connected segments, or splines.
- To draw a connection, left-click on source element, left-click at intermediate points (if any), left-click on target element.
- Respect the order: composed node to component node.
- Rep nodes have only one child.
- Left-click and drag to move. Use cursor to group elements.
- [Right-click  $\rightarrow$  Edit] to edit an element.

Atomic Model:	Select model name	
Node Name:	Submodel1	
ок	Cancel	

- Select the SAN model from the drop-down list.
- Enter the name of the node.

# Join node (1)

Join node name:	Join1	
	Join state variable:	Submodel state variable:
Create New Shared state variable	Shared state v # Shared	Submodel state variable Name
Edit Selected state variable		
Share All Similar state variable		
Delete Selected state variable(s)		
Delete All Shared state variable		
Sort additions to list		
	Ok Cancel	

#### Easy way: Click **Share All Similar state variable** (*sic*):

All state variables (places) with the same name, type and initial value in the children of the join node are shared.

Join node name:	Join1		
	Join state varia	ble:	Submodel state variable:
	Shared state v	# Shared	Submodel state variable Name
Create New Shared state variable	computer_failed	4	Rep1->computer_failed
create new shared state variable	cpus	3	cpu_module->computer_failed
	errorhandlers	3	errorhandlers->computer_failed
	ioports	3	io_port_module->computer_failed
Edit Selected state variable	memory_failed	4	
Share All Similar state variable			

# Join node (2)

$\sim$ Define Shared state variable for Join1 $ imes$				
New name for shared	state variable:			
Rep1 cpu_module	errorhandlers	io_port_module		
📑 cpu_module				
🗕 🗋 computer_failed (sł	- 🗋 computer_failed (short)			
— 🗋 cpus (short)	– 🗋 cpus (short)			
– 🗋 errorhandlers (sho	<ul> <li>         —          Ö errorhandlers (short)         </li> </ul>			
– 🗋 ioports (short)				
memory_failed (sho	ort)			
ок		Cancel		

If places with different names must be shared:

- Click Create New Shared state variable.
- In the Define Shared state variable dialog, select from each child one variable name (there is one tab for each child, with the list of available variables).
- Enter a name for the shared variable, equivalent to the selected child variables.

#### **Rep node**

	Define Rep Node: Rep3	Х
Rep Node Name:	Rep3	Number of Reps: 1
	OK Cancel	

- 1. Enter node name and number of replicas.
- 2. In the new dialog window, select from the left-side list the variables to be shared among the replicas.

	Define Rep Node: Rep1	×
Rep Node Name: Repl	]	Number of Reps: num_mem_mod
Unshared State Variables: interface_chips (short) memory_chips (short)	Share >	Shared State Variables: computer_failed (short) memory_failed (short)
	Share All >>	
	< Unshare	
	<< Unshare All	
ок	Ca	ancel

#### Creating a reward model (1)

A **reward model** is a set of **performance variables** that describe system properties.

A *performance variable* (PV) is computed by performing certain operations (e.g., averaging) on the set of values returned by an associated **reward function** (RF) in the course of simulation or numerical solution.

A **rate reward** is a RF that is evaluated at an instant of time, used to measure properties related to the time during which the system remains in some state.

An **impulse reward** is a RF that is evaluated when an activity completes, used to measure properties related to the number of times that some activity completes.

The values of the RF for a PV can be evaluated at specified times (**Instant of time** PV's), accumulated over a specified interval of time (**Interval of time** PV's), averaged over a specified interval of time (**Time averaged interval of time** PV's), or evaluated when the system has reached a steady state (**Steady state** PV's).

#### Creating a reward model (2)

Reward Model Name: Reward Model Type: Performance Variable Model	cpu_module errorhandlers io_port_module memory_module multi_proc
OK Cancel	Ok Cancel

- 1. [Project Editor  $\rightarrow$  Reward  $\rightarrow$  New], w/ right button.
- 2. Enter name.

A. Domenici

- 3. In the **Select Reward Child** dialog, choose the relevant submodel (usually the top-level one).
- 4. In the **Performance Variable Editor**, enter name of PV, click **Add Variable**.

Performance Variables Model	Variable Name: unreliability				
(Enter new variable name)	Impulse Rewards Time	Simulation			
Add Variable:	Submodels Rate Rewards				
Variable List	Availal (Select the models u	ole Submodels sed for this reward variable)			
unreliability	memory_module				
	cpu_module				
	errorhandlers io_port_module				
Rename Copy Delete Up Down	Apply Changes	Discard Changes			

### Creating a reward model (3)

#### File Edit Help

Performance Variables Model	Variable Name: unreliability			
(Enter new variable name)	Impulse Rewards Time Simulation			
Add Variable:	Submodels Rate Rewards			
	Available State Variables (double click to insert)			
Variable List unreliability	cpu_module->cpus			
	Reward Function			
	<pre>double reward = 0; if (cpu_module-&gt;computer_failed-&gt;Mark() == num_comp) { reward += 1.0 / num_comp; } return (reward);</pre>			
Rename Copy Delete Up Down	Apply Changes Discard Changes			
Möbius Performance Variabl	e Editor 2.4			

For each submodel on which a rate reward PV is computed:

- 1. Select the submodel in the **Submodel** tab.
- 2. In the Rate Rewards tab, enter the Reward Function.

Similar procedure for an impulse reward PV.

A **study** defines sets of values that will be assigned to each global variable.

An **experiment** one possible assignment of values from a study. E.g., if we have two global variables x and y and a study specifies two values for x (say,  $\{1,2\}$ ) and three for y (say,  $\{0,0.5,1\}$ ), then six experiments are possible.

In a **range study**, experiments are generated for all possible combinations of variable values, while in a **set study** only user-defined combinations are used.

In a range study, for each variable the user may specify a single fixed value, or different kinds of value ranges (see the documentation).

## Creating a study (2)

Study Name:	
Study Type: Range Study	Specify Study Child- multi_proc_PV
Design Of Experiments Study	
OK Cancel	Ok Cancel

- 1. [Project Editor  $\rightarrow$  Study  $\rightarrow$  New], w/ right button.
- 2. Select study type and enter name.
- 3. In the Select Study Child dialog, choose a reward model.
- 4. In the **Range Study Editor**, for each variable enter a value or select range. Enter range parameters in pop-up dialogs.

		Change Reward Model		Experiment Activator		
	Change Reward					
Variable Name	Variable	Variable Type		Variable Value		
CPU cov	double	double		0.995		
IO cov	double	double		0.99		
RAM cov	double	double		0.998		
comp cov	double	double		0.95		
failure rate	double	double double		8.766E-4 0.95		
mem cov	double					
num comp	short	ort	Incremen	Incremental Range		
num_mem_mod	short	short				
Incremental Range	Functional Range	Manual	Range	Random Range		

#### THE MÖBIUS TOOL: SOLVING MODELS



Andrej Andreevič Markov (1856, 1922).

Any Möbius model can be simulated, but it is possible to compute its performance variables in closed form, if the model can be transformed into a *continuous time Markov chain* (CTMC).

Transformation into a CTMC is possible if:

- 1.(a) All timed actions are exponential ( $\lambda e^{-\lambda t}$ ), or
  - (b) All timed actions are deterministic or exponentially distributed, at most one deterministic action is enabled at any time, and the delay of deterministic actions does not depend on the state.
- 2. The model must begin in a stable state, with no immediate actions.

In order to solve a model, its **state space** must be generated by a **transformer**.

Two types of transformers are available: (flat) **state space generator** and **symbolic state space generator** (see documentation).

#### **State space generation**

Transformer Name:		
Transformer Type:		
State Space Generator	Specify Transformer Child:	vary_num_comp
Symbolic State Space Generator		
OK Cancel	Ok	Cancel

- 1. [Project Editor  $\rightarrow$  Transformer  $\rightarrow$  New], w/ right button.
- 2. Select transformer type and enter name.
- 3. In the **Select Transformer Child** dialog, choose a study.
- 4. In the **State Space Generator** window, change parameters as needed (see documentation) and click **Start State Space Generation**.

Study Name:	vary_num_comp		Browse
Experiment List:	Experiment_1	\$	Experiment Activator
Run Name:	Results		Man and a second se
Build Type:	Optimize	-	
Trace Level:	Level 0: None	~	
Hash Value:	0.5		
	Flag Absorbing Stat Place Comments in	es Output	Edit Comments
	Start State Space G	eneration	

Finally, a **solver** must be generated.

Several different types of solvers are available.

Two main classes are transient and steady-state solvers.

*Transient* solvers compute solutions related to specified times or periods.

*Steady-state* solvers compute limit solutions as time approaches infinity.

#### Numerical solver generation (2)

Solver Name:	
Solver Type:	
Mobius Simulator	
Simple Connection Model	Specify Solver Child: 339
Transient Solver	
OK Cancel	Ok Cancel

- 1. [Project Editor  $\rightarrow$  Solver  $\rightarrow$  New], w/ right button.
- 2. Select solver type and enter name.
- 3. In the Select Solver Child dialog, choose a state space.
- 4. In the **Solver** window, change parameters as needed (see documentation) and click **Solve**.

ate Space Name:	ssg		
ccuracy		9	
erbosity		0	
utput File Name	Results		
ebug File Name			
ate Vector File Name	3		
	🗌 Plot Complementary Distribu	tion	
	Run In The Background		
	Place Comments in Output		

#### **AN EXAMPLE**

#### A simple computer system

- A computer is *idle*, *busy*, or *failed*;
- jobs arrive at a rate  $\alpha$ ;
- **I** jobs are completed at a rate  $\beta$ ;
- the computer fails at rate  $\lambda_i$  when idle;
- the computer fails at rate  $\lambda_w$  when busy.

We want to calculate the reliability after twenty-four hours of operation.



#### The atomic model (1)



A composed model is not needed.

The initial marking is 1 for *idle*, 0 for the other places.

All activities have exponential time distribution.

activity	gate	predicate	function
arrival	l_arr	idle->Mark() == 1	idle->Mark() = 0;
	O_arr		busy->Mark() = 1;
idle_fail	I_idle_fail	idle->Mark() == 1	idle->Mark() = 0;
	O_idle_fail		<pre>failed-&gt;Mark() = 1;</pre>
completed	I_comp	busy->Mark() == 1	busy->Mark() = 0;
	O_comp		idle->Mark() = 1;
busy_fail	I_busy_fail	busy->Mark() == 1	busy->Mark() = 0;
	O_busy_fail		failed->Mark() = 1;

#### The reward and study models

We define a PV *reliability* of type *instant* of *time*, with the following *rate reward* function:

```
if (computer->failed->Mark() == 0)
    return 1;
```

In the *Time* tab, we set a Start Time of 24.0.

Study: study_1	Reward Model: reliab 5		Active of 5 Total Experiments		
	Cha	nge Reward Moo	lel	Ехр	eriment Activator
Variable Name		Variab	e Type		Variable Value
alpha		double		1000.0	
beta		double		10000.0	
lambda_i		double		1.0E-7	
lambda_w		double		Incremer	ital Range
Incremental Ran	Fund	tional Range	Manual Ra	nge	Random Range

In the study, we let  $\lambda_w$  vary from  $1 \times 10^{-6}$  to  $5 \times 10^{-6}$ .

NOTE: These values are just an example and are not meant to be realistic.

After generating the state space and the solver, we run the five experiments.

The results are written in files stored in
<project\_name>/Solver/<solver\_name>/.

$\lambda_w$	reliability	
	mean	variance
$1.00000000000000000 \times 10^{-6}$	$9.999956 \times 10^{-6}$	$4.363599 \times 10^{-6}$
$2.0000000000000000 \times 10^{-6}$	$9.999935 \times 10^{-6}$	$6.545373  imes 10^{-6}$
$3.0000000000000000 \times 10^{-6}$	$9.999913 \times 10^{-6}$	$8.727125 \times 10^{-6}$
4.000000000000000000000000000000000000	$9.999891 \times 10^{-6}$	$1.090887 \times 10^{-5}$
$4.99999999999999996 \times 10^{-6}$	$9.999869 \times 10^{-6}$	$1.309061 \times 10^{-5}$

# A fault-tolerant multiprocessor system (1)

From the Möbius Manual, Version 2.4 Rev. 1, https://www.mobius.illinois.edu/docs/MobiusManual.pdf (**Please don't cheat**: Try to do it yourself before reading the solution in the manual).

- A system is made of *N* computers.
- A **computer** is made of:
- 2 + 1 CPU's (2 in service, 1 spare);
- 2+1 memory modules;
- 1+1 I/O ports;
- 2 error-handling chips.
- A CPU is made of 6 chips.
- A memory module is made of:
- 39 + 2 RAM chips;
- 2 interface chips.

An **I/O port** is made of 6 chips.

### A fault-tolerant multiprocessor system (2)

The **system** is operational if at least **1 computer** is operational.

- A computer is operational if at least
- 2 CPU's,
- 2 memory modules,
- 1 I/O port,
- 2 error-handling chips are operational.

#### A memory module is operational if at least

- 39 RAM chips,
- 2 interface chips are operational.

If a spare is available, a failed component may be replaced with a given **fault coverage** probability.

The fault coverage probabilities for each type of component are:

- chips (*chip\_cvg*): 0.998
- memory modules (*mem\_cvg*): 0.950
- CPU's (*cpu\_cvg*): 0.995
- I/O ports (*io\_cvg*): 0.990
- computers (*comp\_cvg*): 0.950

The failure rate for chips (*lambda\_chip*) is 0.0008766 failures per year.

component	subcomponents		number	coverage		
	needed	spare	type		parameter	value
system	1	<b>1</b> - N	computer	N	comp_cvg	0.950
computer	2	1	CPU	3	cpu_cvg	0.995
	2	1	memory	3	mem_cvg	0.950
	1	1	I/O port	2	io_cvg	0.990
	2	0	EH chip	2		
CPU	6	0	cpu chip	6	chip_cvg	0.998
memory	39	2	ram chip	41	chip_cvg	0.998
	2	0	intf chip	2		
I/O port	6	0	I/O chip	6		

A possible modeling strategy:

- The system is the composition (by *rep*) of *N* computers;
- a computer is the composition (by *join*) of its subsystems;
- each subsystem is modeled as a SAN submodel (except for memories, see below);
- for each subsystem, we model its behavior wrt failures.

All submodels have a similar structure:

- A timed activity whose completion represents the failure of the corresponding subsystem;
- the timed activity is assumed to have an exponential time distribution;
- the timed activity is enabled if the subsystem is (as yet) operational and at least one computer is operational;
- the timed activity has a number of cases representing possible outcomes of the failure;
- the case probabilities depend on the availability of spares and on fault coverage probabilities;
- some place markings represent the number of operational components, others represent the number of failed memory modules within a computer, and the number of failed computers in the system.

The submodel for memory modules has two failure transitions.

A failed computer could have many different configurations: e.g., it could have two failed CPU's and the other modules operational, or two failed memory modules and the other modules operational, or have failed CPU's and failed memories, and so on.

Each of these configuration produces different states, but all these states are equivalent as they all result in a failed computer.

The number of states can be reduced by *lumping* equivalent states, by assuming that all components of a computer are failed if the computer is not operational.

For example, if a computer fails because two of its CPU's have failed, we set to zero the number of operational memories, I/O ports, and error handlers.

- cpus: number of operational CPU's.
- errorhandlers: number of operational errorhandlers.
- ioports: number of operational I/O ports.
- computer\_failed: number of failed computers.
- memory\_failed: number of failed memory modules.

The failure activity (*cpu\_failure*) is enabled if the following conditions hold:

- At least 2 CPU's are operational;
- At least 2 memory modules are operational;
- At least 1 computer is operational.

The failure rate is the chip failure rate times the number of chips times the number of operational CPU's (place *cpus*).

The possible outcomes are:

- 1. The CPU can be replaced: decrease the number of operational CPU's.
- 2. The CPU cannot be replaced, but the computer can: increase the number of failed computers, set to zero the number of operational components.
- 3. No replacement is possible: set the number of failed computers to N, set to zero the number of operational components.

case	probability	
	spare available	spare unavailable
CPU replaced	cpu_cvg	0
computer replaced	(1 - cpu_cvg) ⋅ comp_cvg	comp_cvg
no replacement	(1 - cpu_cvg	1 - comp_cvg
	- (1 - cpu_cvg) · comp_cvg)	

## The memory module submodel (1)

This submodel is more complex because a memory module has two sets of chips, one with spare chips and one without. So, it has two failure activities.

The (*interface\_chip\_failure*) activity is enabled if the following conditions hold:

- At least 2 chips are operational;
- At least 2 memory modules are operational;
- At least 1 computer is operational.

The failure rate is the chip failure rate times the number of interface chips.

The possible outcomes are:

- 1. Module replacement: increase the number of failed MM's; if 2 MM's are failed, increase the number of failed computers.
- 2. Computer replacement: if one MM has already failed and at least one computer is operational, set the number of failed computers to N, otherwise increase the number of failed computers.
- 3. No replacement: set the number of failed computers to N.

case	probability
memory module replaced	mem_cvg
computer replaced	(1 - mem_cvg) · comp_cvg
no replacement	(1 - mem_cvg) · (1 - comp_cvg)

## The memory module submodel (2)

The (*memory\_chip\_failure*) activity is enabled if the following conditions hold:

- At least 39 chips are operational;
- At least 2 memory modules are operational;
- At least 1 computer is operational.

The failure rate is the chip failure rate times the number of RAM chips.

The possible outcomes are:

- 1. Chip replacement: if spares are available, decrease the number of RAM chips.
- 2. Module replacement: increase the number of failed MM's; if one MM has already failed, increase the number of failed computers.
- 3. Computer replacement: if one MM has already failed and at least one computer is operational, set the number of failed computers to N, otherwise increase the number of failed computers.
- 4. No replacement: set the number of failed computers to N.

#### **Case probabilities of memory\_chip\_failure**

case	probability		
	spare available	spare unavailable	
1	ram_cvg	0	
2	(1 - ram_cvg) · mem_cvg	mem_cvg	
3	((1 - ram_cvg)	(1 - mem_cvg) · comp_cvg	
	· (1 - mem_cvg) · comp_cvg)		
4	((1 - ram_cvg)	(1 - mem_cvg) · (1 - comp_cvg)	
	· (1 - mem_cvg) · (1 - comp_cvg))		

Case 1: Chip replaced. Case 2: Memory module replaced. Case 3: Computer replaced. Case 4: No replacement.

The memory subsystem is the replication of three memory module submodels, sharing the numbers of failed computers and failed memories.

The computer subsystem is the join of the memory subsystem, and the CPU module, error handlers, and I/O port submodules, sharing all the places.

The multiprocessor system is the replication of N computer subsystems, sharing the number of failed computers.

We compute the *unreliability* of the multiprocessor at a given time (namely, after 20 years of operation).

The unreliability is a performance variable of type *instant of time*, whose reward function returns 1 when all computers have failed.

More precisely, the function returns 1/N, since it is computed once for each computer.

In this case, we observe the mean value of the unreliability as the number of computer varies.

This seminar is a (hopefully) user-friendly introduction to the Stochastic Activity Networks and their application to the quantitative analysis of dependability, using the Möbius environment.

Clearly, there is much more to say about these topics, both from the theoretical and the practical point of view. For example, the use of Möbius as a simulator has been ignored.

Readers are encouraged to explore the tool and the literature. The best starting point is the Möbius site:

https://www.mobius.illinois.edu/