Dependable and Secure Systems – Dependability

Master of Science in Embedded Computing Systems

Quantitative Dependability Analysis with Stochastic Activity Networks: the Möbius Tool

April 2016

Andrea Domenici

DII, Università di Pisa

Andrea.Domenici@iet.unipi.it

Outline

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

STOCHASTIC ACTIVITY NETWORKS

Stochastic Activity Networks (1)

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 + . . .

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**).

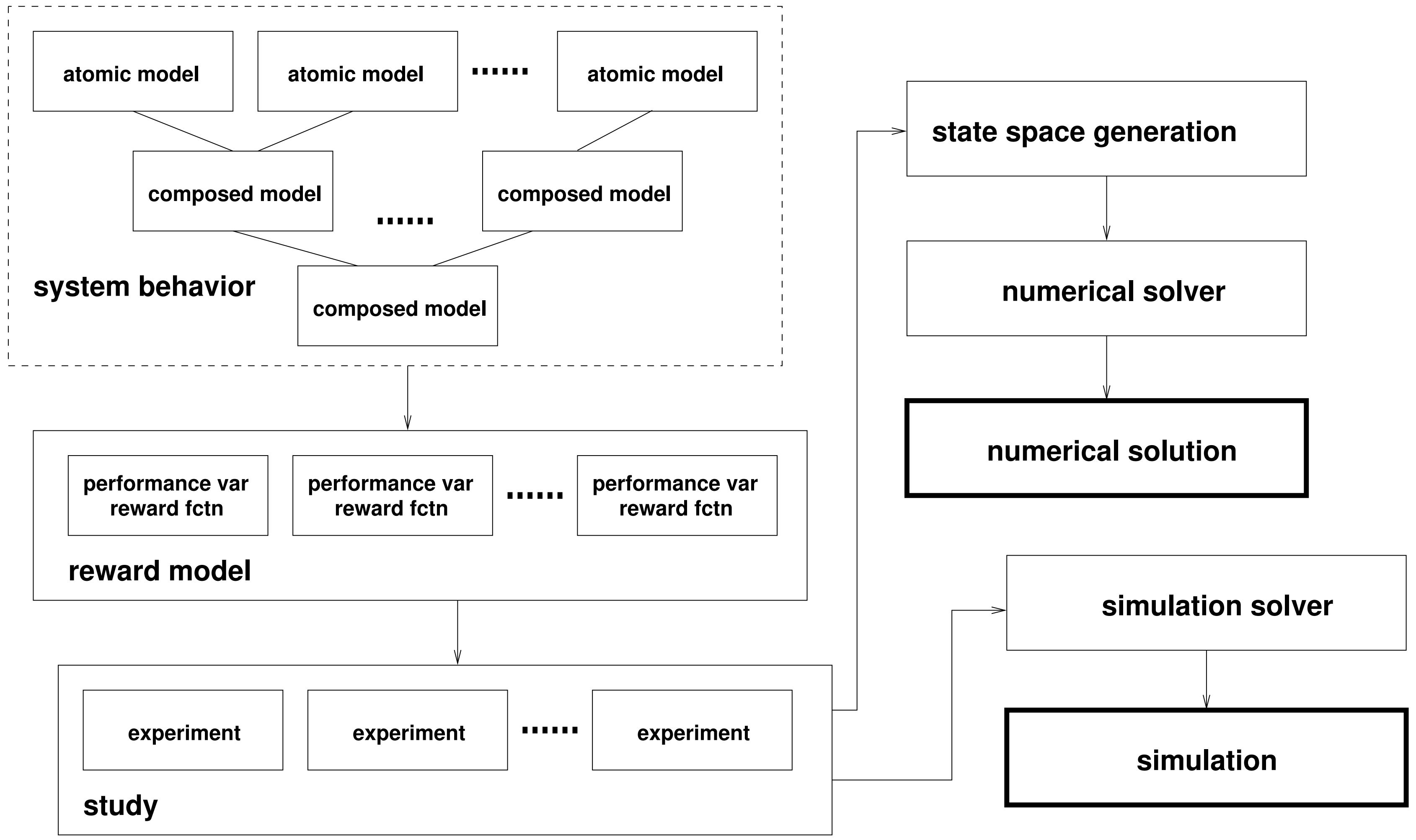
Stochastic transition outcome: Alternative results of an activity can be specified as mutually exclusive **cases** associated with the activity.

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).

The Möbius analysis process



The Möbius tool

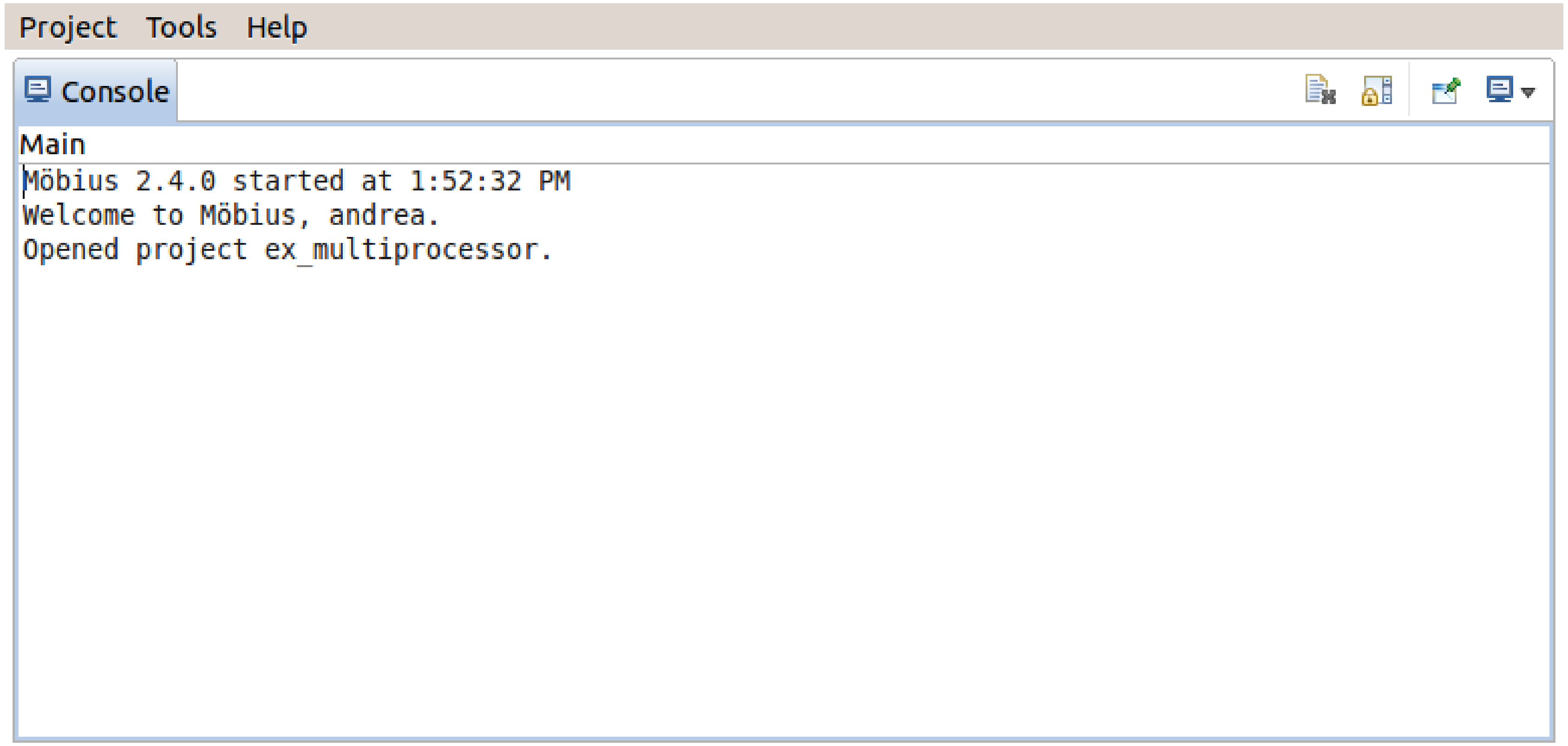
The Möbius environment provides:

- **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)

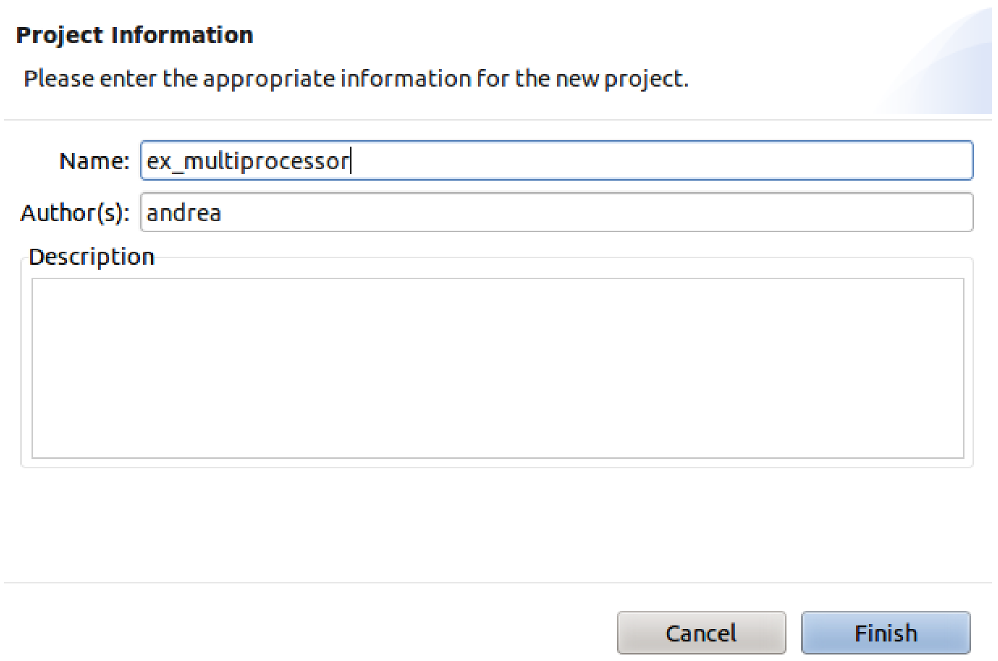


```
Möbius 2.4.0 started at 1:52:32 PM
Welcome to Möbius, andrea.
Opened project ex_multiprocessor.
```

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)

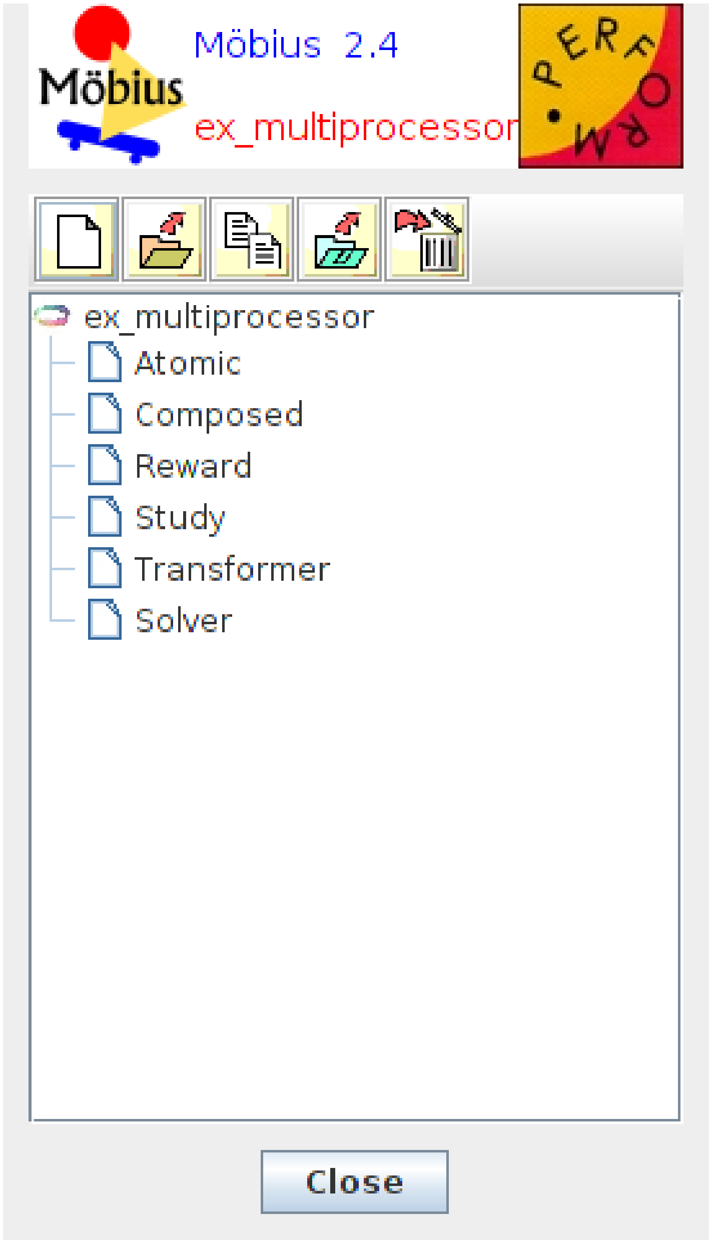

Project Information

Please enter the appropriate information for the new project.

Name: ex_multiprocessor

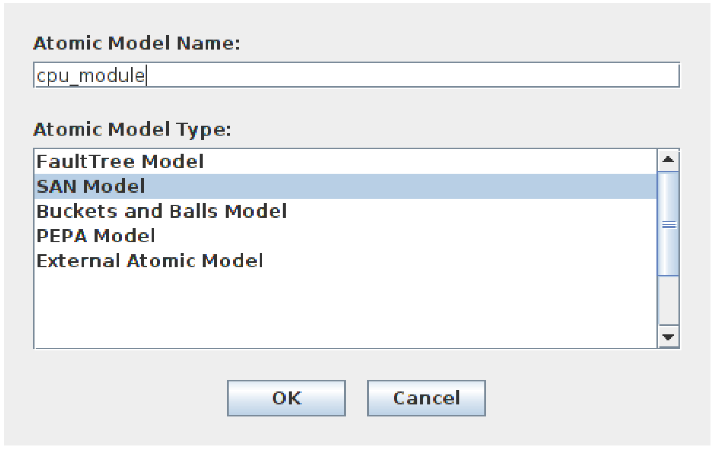
Author(s): andrea

Description



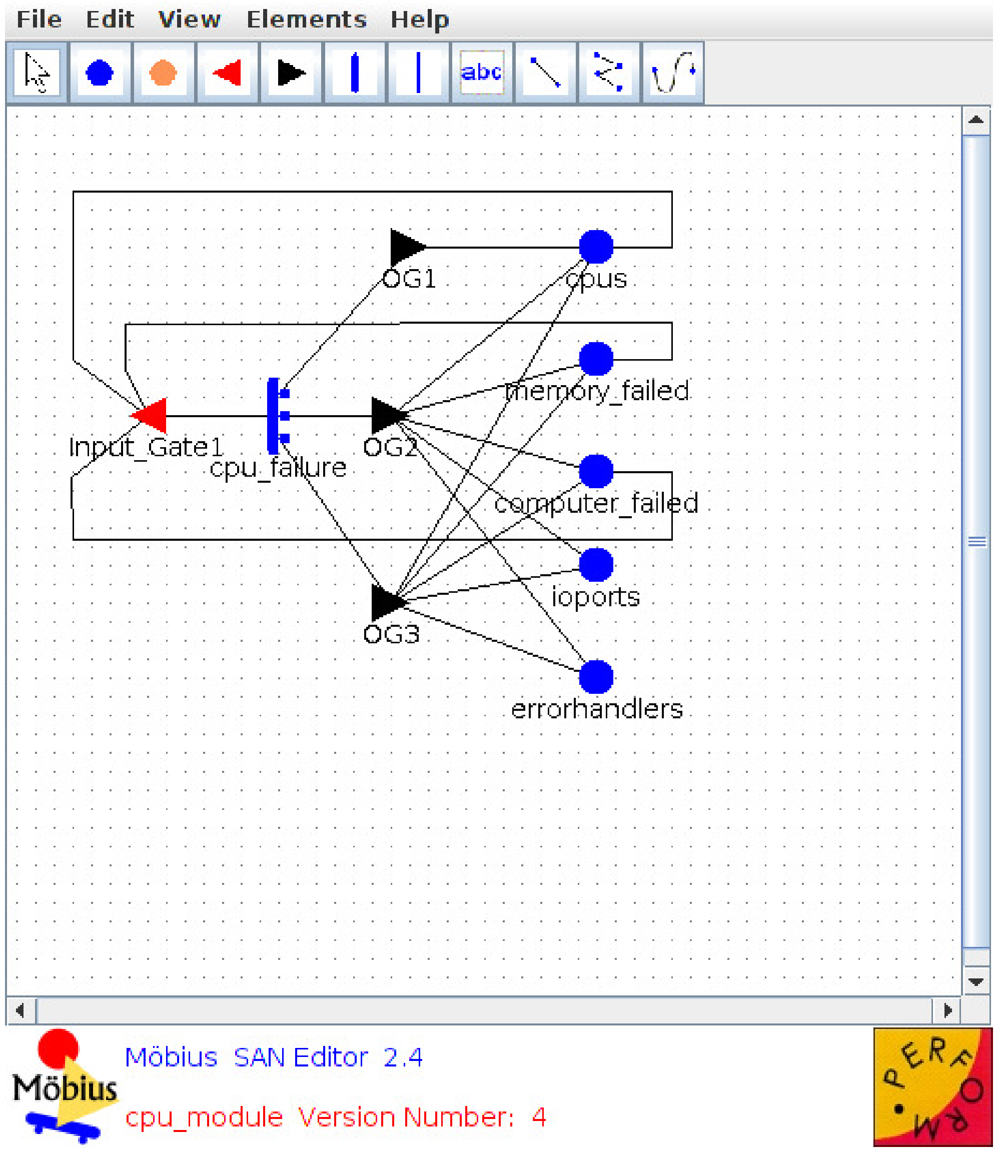
1. Type in project name.
2. The Project Editor pops up.

Creating an atomic model (1)



Atomic Model Name:
cpu_module

Atomic Model Type:
- FaultTree Model
- SAN Model
- Buckets and Balls Model
- PEPA Model
- External Atomic Model



Möbius SAN Editor 2.4

cpu_module Version Number: 4

1. [Project Editor → Atomic → 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.

Creating an atomic model (3)

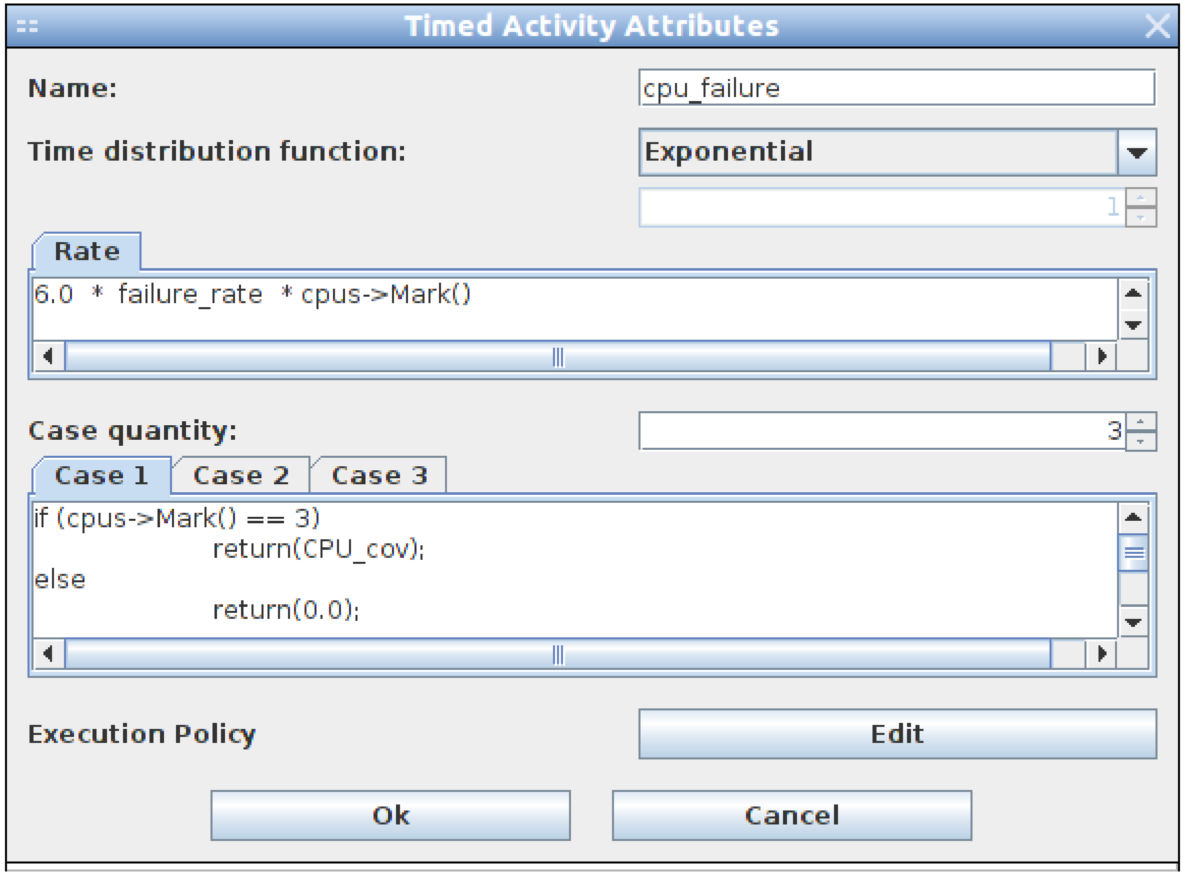
Time distribution rates, case probabilities, input and output functions, input predicates, and reward functions, are specified with C++ expressions or sequences of statements.

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



Name: cpu_failure

Time distribution function: Exponential

1

Rate

```
6.0  *  failure_rate  * cpus->Mark()
```

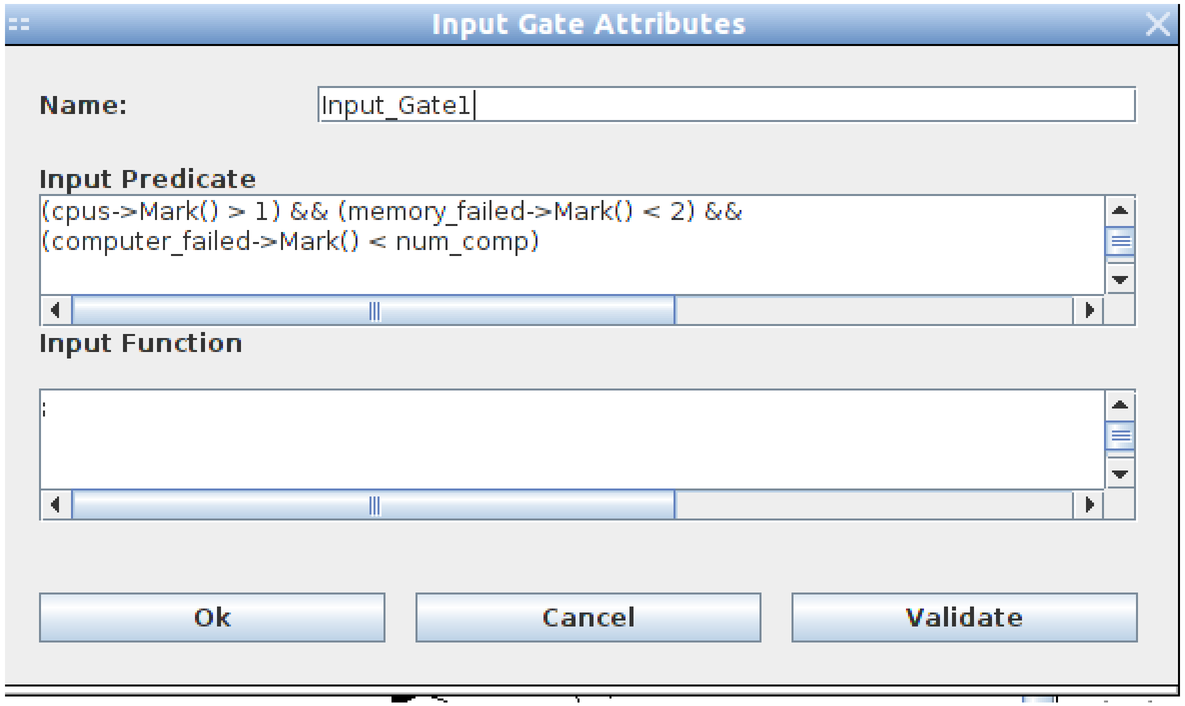
Case quantity: 3

Case 1 | Case 2 | Case 3

```
if (cpus->Mark() == 3)
                return(CPU_cov);
else
                return(0.0);
```

- 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



Name: Input_Gate1

Input Predicate

```
(cpus->Mark() > 1) && (memory_failed->Mark() < 2) &&
(computer_failed->Mark() < num_comp)
```

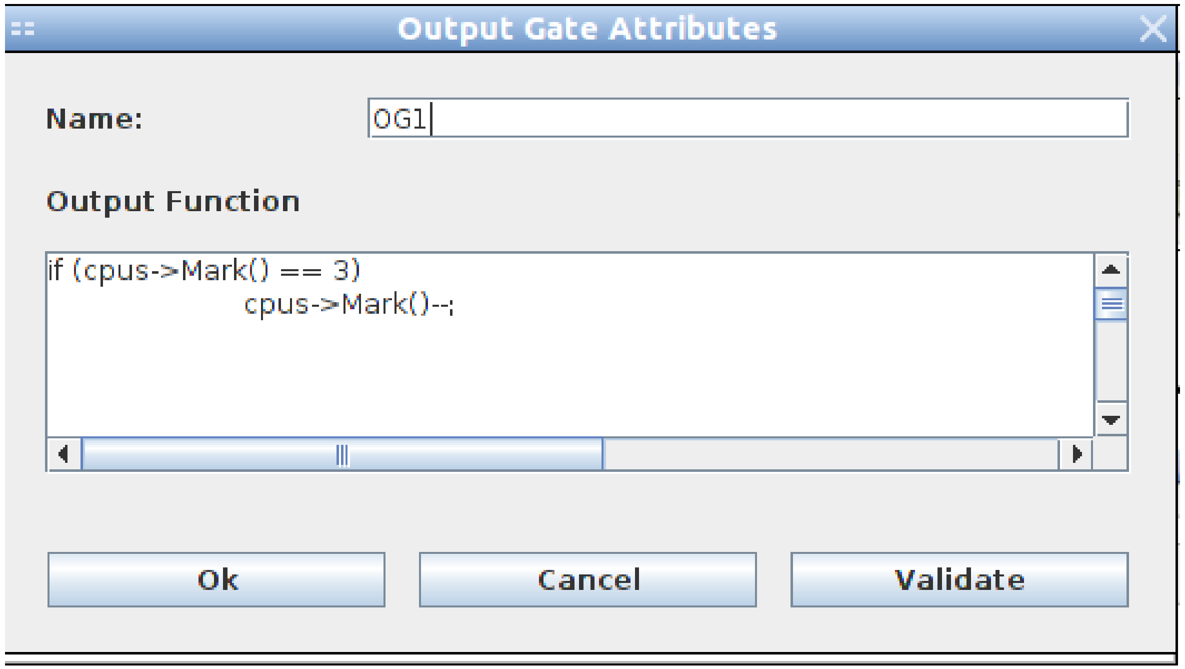
Input Function

```
;
```

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

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

Output gates



Name: OG1

Output Function

```
if (cpus->Mark() == 3)
                cpus->Mark()--;
```

- Specify the **Output function**.

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

Creating a composed model (1)

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).

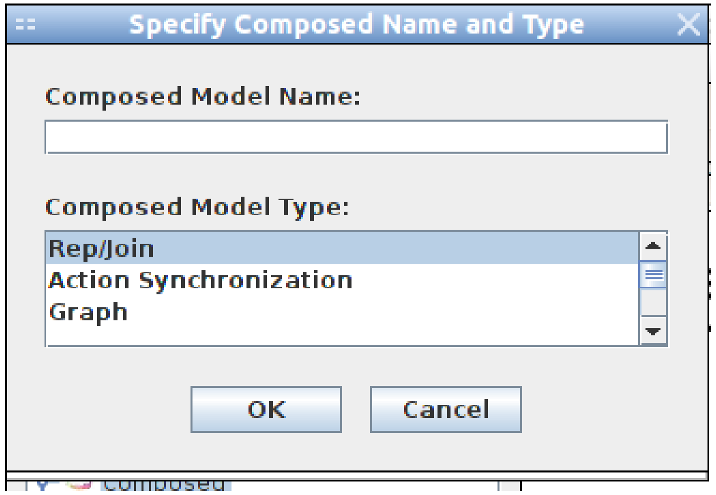
Submodels interact through **shared state variables**, i.e., shared places. When two or more submodels are composed, the user specifies which places are shared among them.

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)

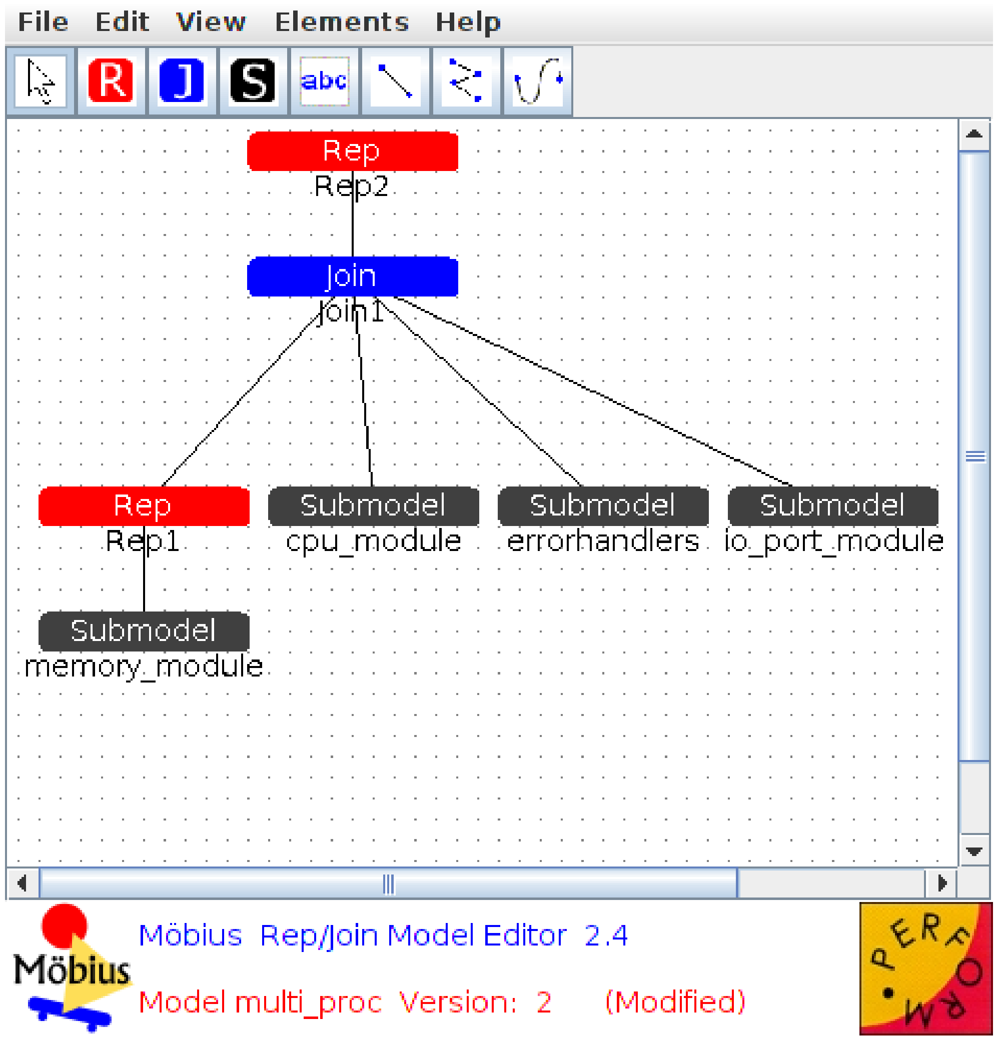

Specify Composed Name and Type

Composed Model Name:

Composed Model Type:

Rep/Join
Action Synchronization
Graph

OK
Cancel



Möbius Rep/Join Model Editor 2.4

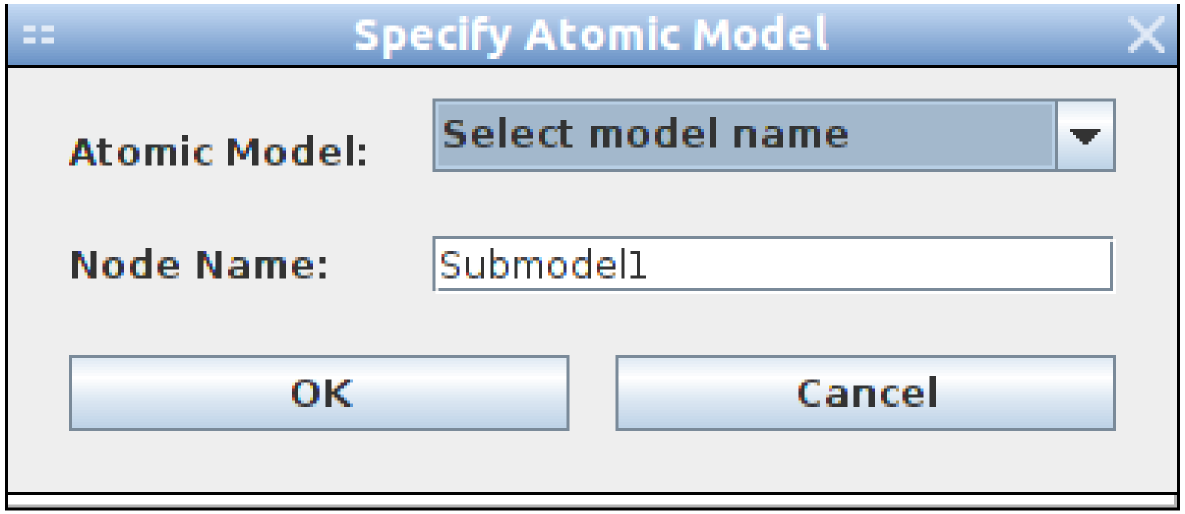
Model multi_proc Version: 2 (Modified)

1. [Project Editor → Composed → 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 compose 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.

Submodel node

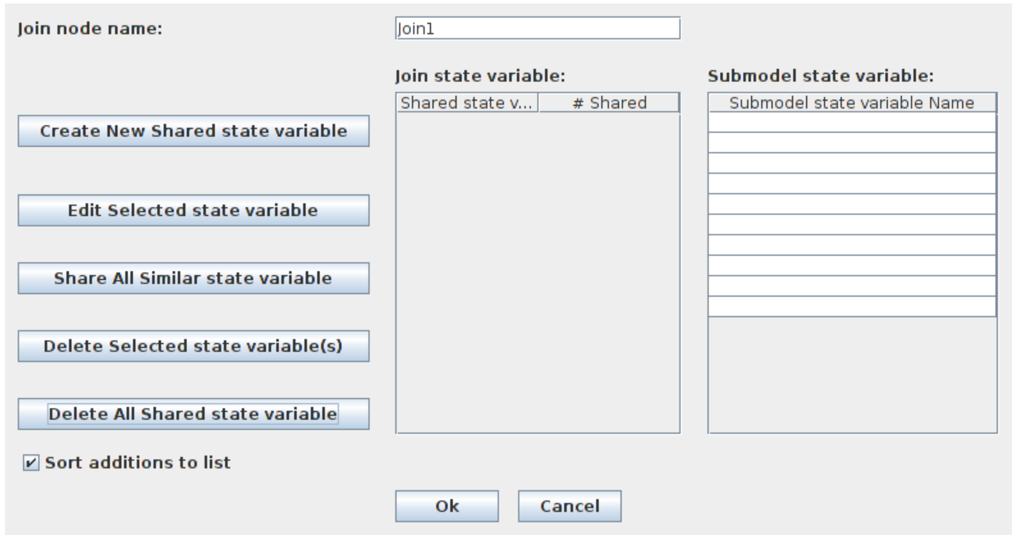


Atomic Model: Select model name

Node Name: Submodel1

- 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:

Shared state v...	# Shared

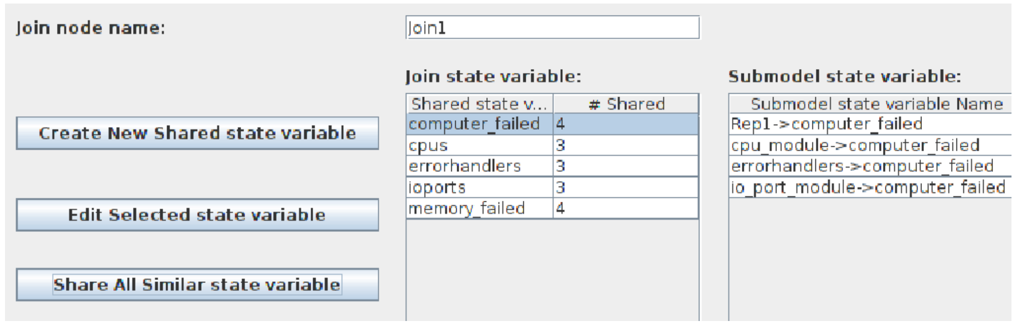
Submodel state variable:

Submodel state variable Name

Create New Shared state variable
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 variable:

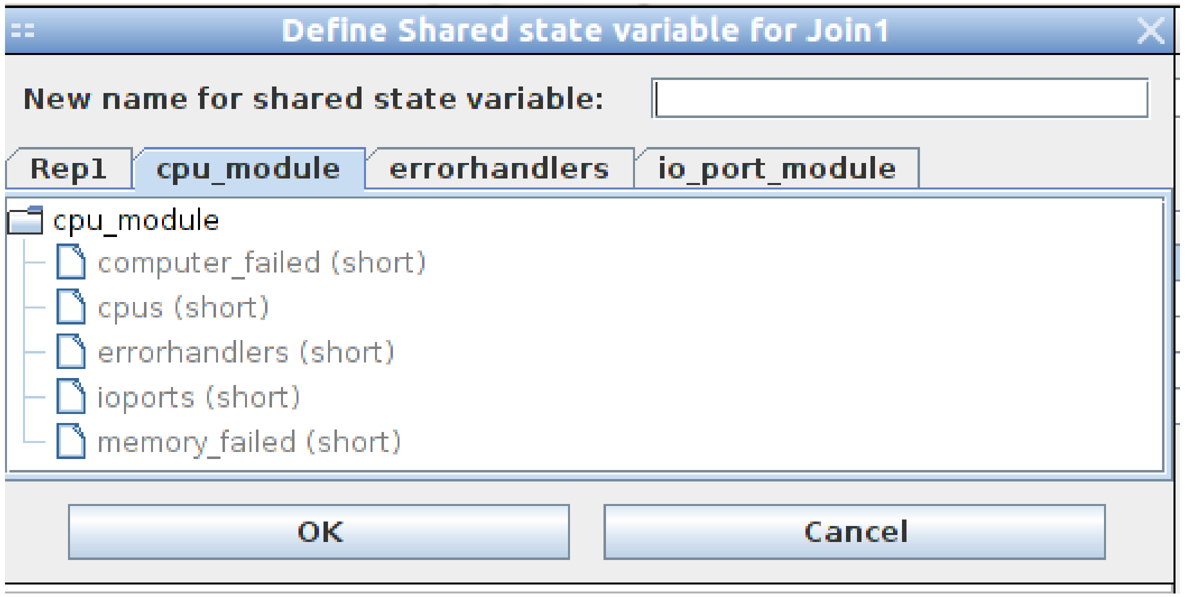
Shared state v...	# Shared
computer_failed	4
cpus	3
errorhandlers	3
ioports	3
memory_failed	4

Submodel state variable:

Submodel state variable Name
Rep1->computer_failed
cpu_module->computer_failed
errorhandlers->computer_failed
io_port_module->computer_failed

Create New Shared state variable
Edit Selected state variable
Share All Similar state variable

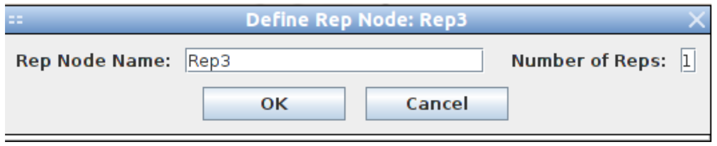
Join node (2)

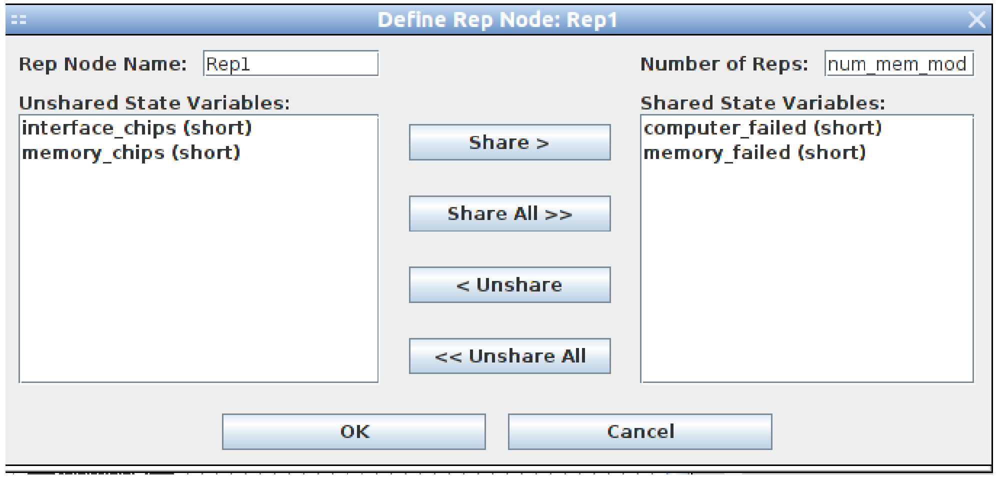


If places with different names must be shared:

- 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





Unshared State Variables:

- interface_chips (short)
- memory_chips (short)

Shared State Variables:

- computer_failed (short)
- memory_failed (short)

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.

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)

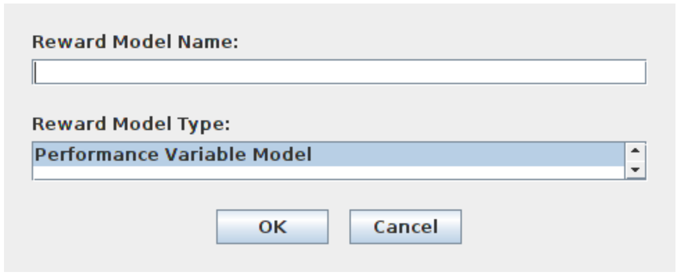
Variable class:

- rate reward: depend on state variables;
- impulse reward: depend on actions.

Variable type (set in the *Time* tab):

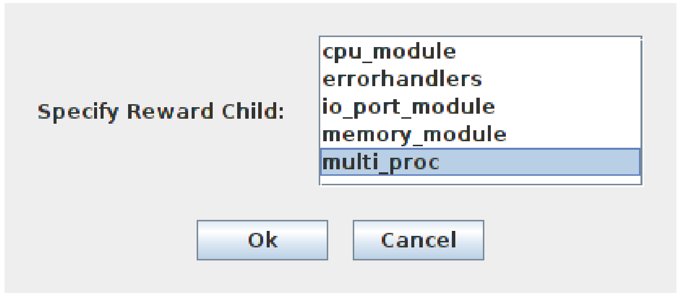
- instant of time: evaluated at specific points in time;
- interval of time: sum of the values of the RF over a time interval, each value weighted by the length of time it is returned by the RF. E.g., if a rate reward PV v of type *instant of time* is defined by a RF f returning 10 for 3 s and 20 for 2 s, then $v = 10 \cdot 3/5 + 20 \cdot 2/5 = 14$.
- time averaged: interval of time result, divided by the length of time for the interval. With w a time averaged PV and f as in the example above, $w = 14/5$.
- steady state: evaluated after transient phase (or, as $t \to \inf$).

Creating a reward model (3)



Reward Model Name:

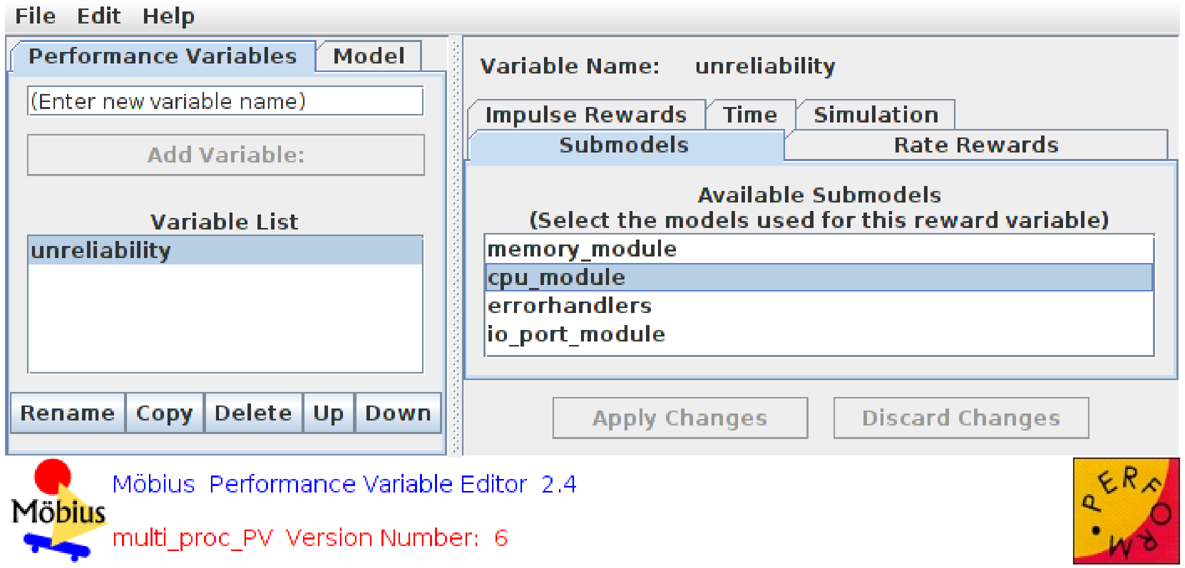
Reward Model Type: Performance Variable Model



Specify Reward Child:

cpu_module
errorhandlers
io_port_module
memory_module
multi_proc

1. [Project Editor → Reward → New], w/ right button.
2. Enter reward model name.
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**.



Variable List: unreliability

Variable Name: unreliability

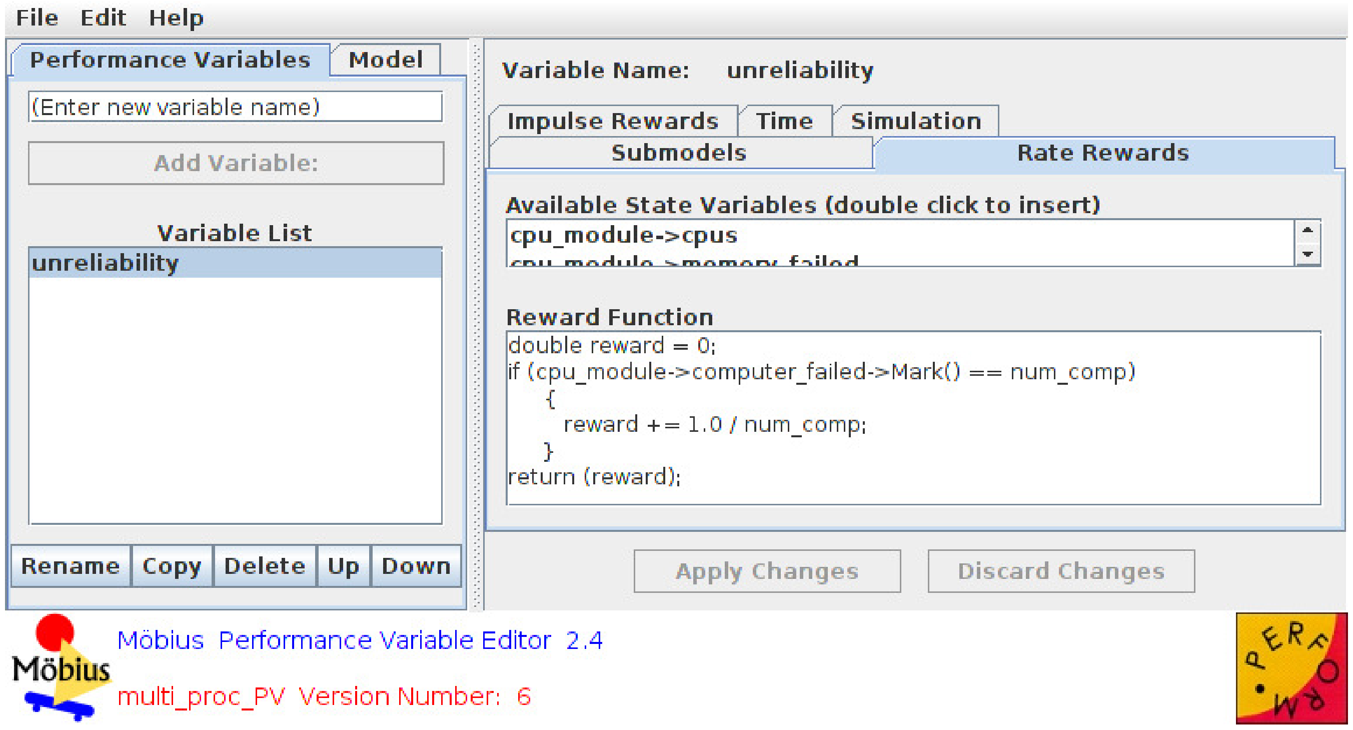
Available Submodels
(Select the models used for this reward variable)

memory_module
cpu_module
errorhandlers
io_port_module

Möbius Performance Variable Editor 2.4

multi_proc_PV Version Number: 6

Creating a reward model (4)



Variable Name: unreliability

Available State Variables (double click to insert)

```
cpu_module->cpus
cpu_module->memory_failed
```

multi_proc_PV Version Number: 6

Reward Function

```
double reward = 0;
if (cpu_module->computer_failed->Mark() == num_comp)
    {
      reward += 1.0 / num_comp;
    }
return (reward);
```

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.

Creating a study (1)

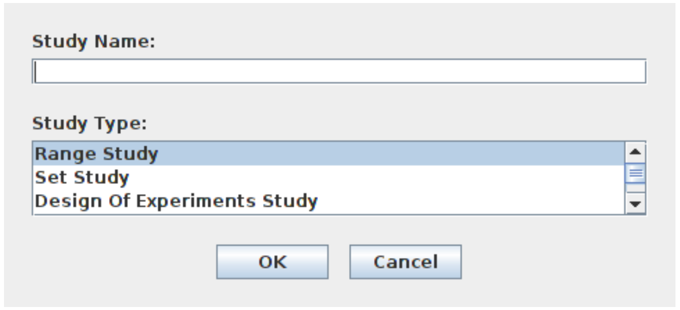
A **study** defines sets of values that will be assigned to each global variable.

An **experiment** is 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.

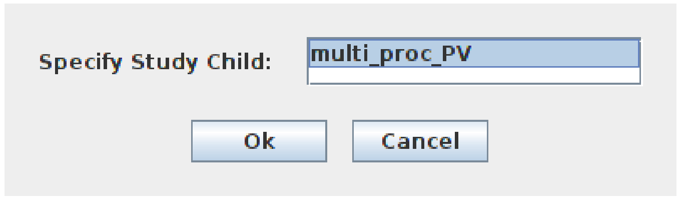
In a range study, for each variable the user may specify a single fixed value, or different kinds of value ranges (see the documentation).

Creating a study (2)

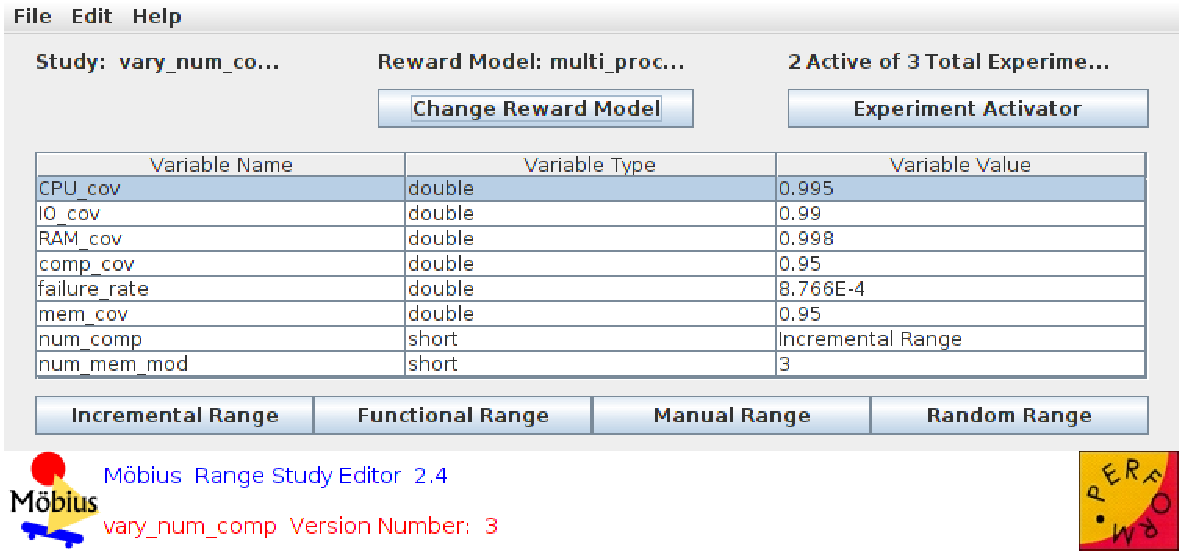

Study Name:

Study Type:

Range Study
Set Study
Design Of Experiments Study


Specify Study Child: multi_proc_PV

1. [Project Editor → Study → 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.



Study: vary_num_co... Reward Model: multi_proc... 2 Active of 3 Total Experime...

Variable Name	Variable Type	Variable Value
CPU_cov	double	0.995
IO_cov	double	0.99
RAM_cov	double	0.998
comp_cov	double	0.95
failure_rate	double	8.766E-4
mem_cov	double	0.95
num_comp	short	Incremental Range
num_mem_mod	short	3

Möbius Range Study Editor 2.4

vary_num_comp Version Number: 3

THE MÖBIUS TOOL: SOLVING MODELS

Andrej Andreevič Markov (1856, 1922).

Numerical solution of Möbius models

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).

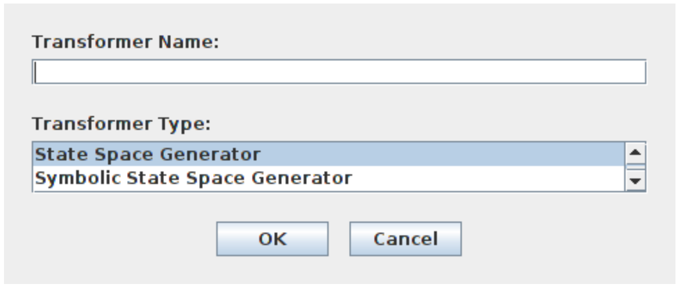
Transformation into a CTMC is possible if:

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**.

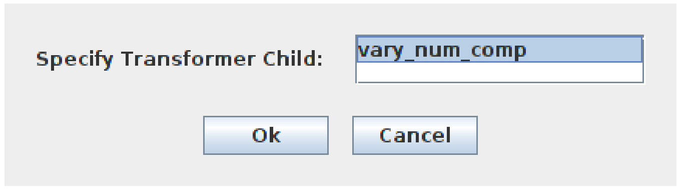
Two types of transformers are available: (flat) **state space generator** and **symbolic state space generator** (see documentation).

State space generation

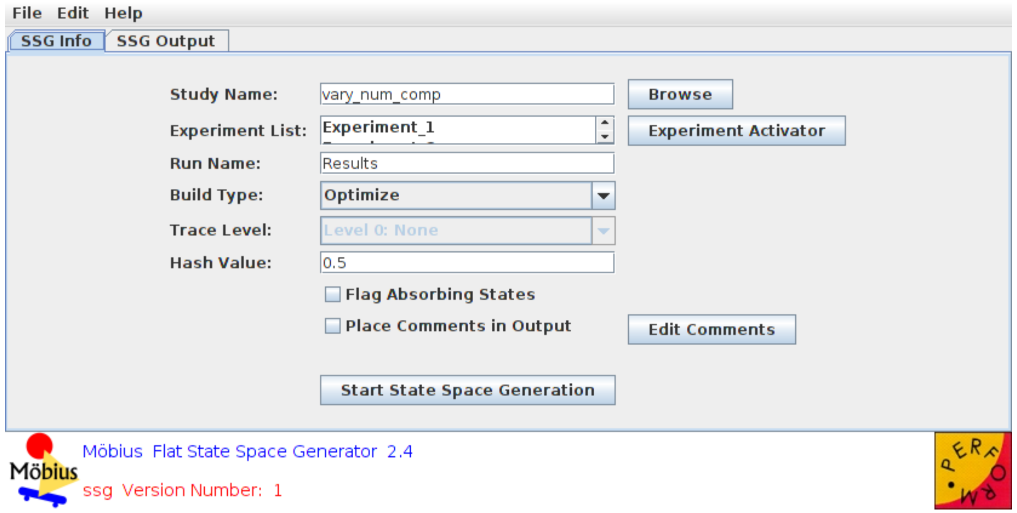

Transformer Name:

Transformer Type:

State Space Generator
Symbolic State Space Generator


Specify Transformer Child: vary_num_comp

1. [Project Editor → Transformer → 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
Experiment List:	Experiment_1
Run Name:	Results
Build Type:	Optimize
Trace Level:	Level 0: None
Hash Value:	0.5

☐ Flag Absorbing States
☐ Place Comments in Output

Möbius Flat State Space Generator 2.4
ssg Version Number: 1

Numerical solver generation (1)

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

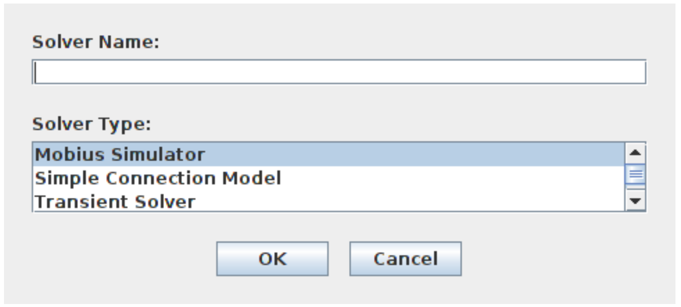
Several different types of solvers are available.

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.

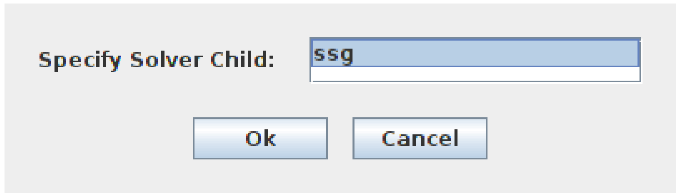
Numerical solver generation (2)



Solver Name:

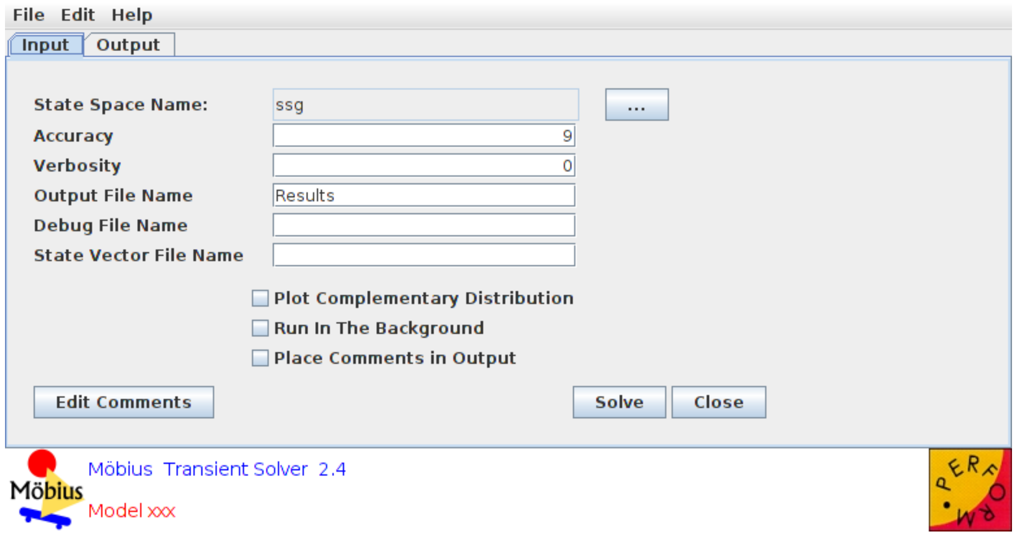
Solver Type:

- Mobius Simulator
- Simple Connection Model
- Transient Solver



Specify Solver Child: ssg

1. [Project Editor → Solver → 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**.



State Space Name:	ssg
Accuracy	9
Verbosity	0
Output File Name	Results
Debug File Name	
State Vector File Name	

- [ ] Plot Complementary Distribution
- [ ] Run In The Background
- [ ] Place Comments in Output

Möbius Transient Solver 2.4

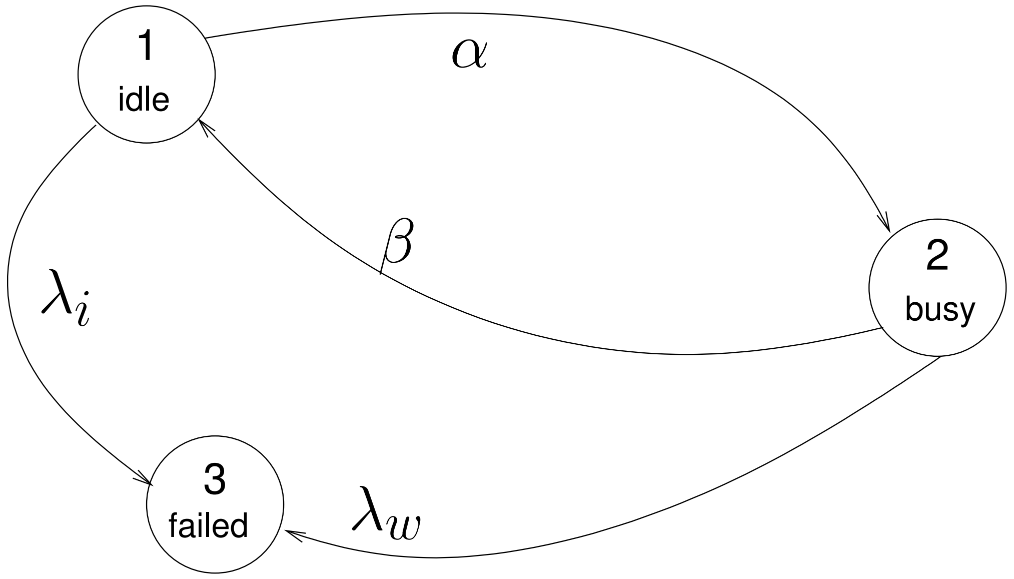
Model xxx

AN EXAMPLE

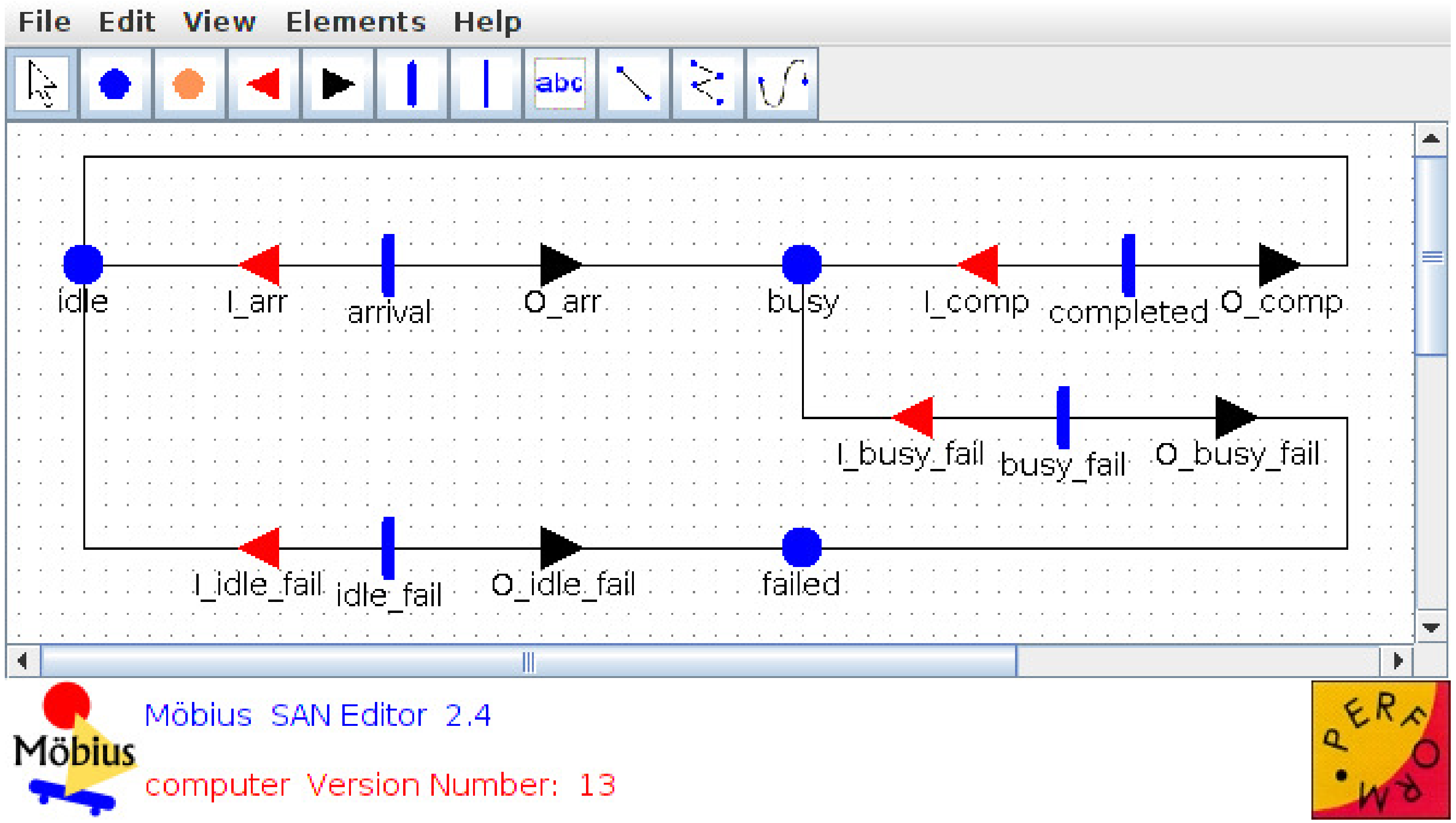
A simple computer system

- A computer is *idle*, *busy*, or *failed*;
- jobs arrive at a rate α;
- jobs are completed at a rate β;
- the computer fails at rate λ_i when idle;
- the computer fails at rate λ_b 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 atomic model (2)

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

All activities have exponential time distribution.

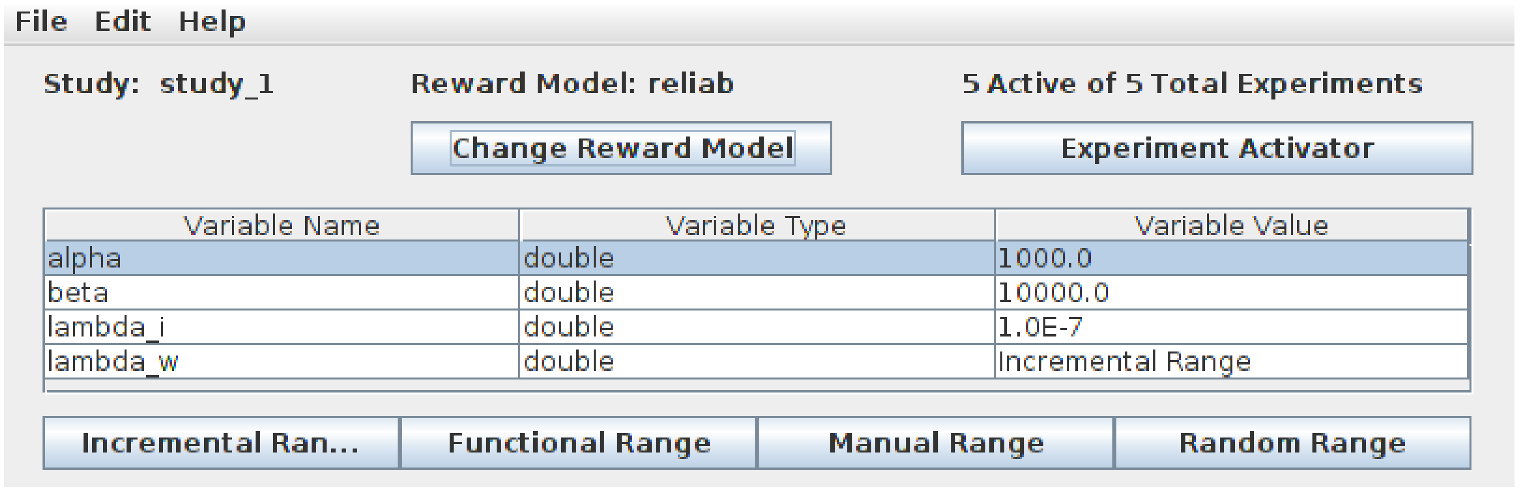
activity	gate	predicate	function
arrival	I_arr	`idle->Mark() == 1`	`idle->Mark() = 0;`
	O_arr		`busy->Mark() = 1;`
idle_fail	I_idle_fail	`idle->Mark() == 1`	`idle->Mark() = 0;`
	O_idle_fail		`failed->Mark() = 1;`
completed	I_comp	`busy->Mark() == 1`	`busy->Mark() = 0;`
	O_comp		`idle->Mark() = 1;`
busy_fail	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

Variable Name	Variable Type	Variable Value
alpha	double	1000.0
beta	double	10000.0
lambda_i	double	1.0E-7
lambda_w	double	Incremental Range

In the study, we let λ_b vary from 1×10^{-6} to 5×10^{-6}.

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

Results

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>/`.

λ_b	reliability	
	mean	variance
$1.0000000000000000 \times 10^{-6}$	9.999956×10^{-6}	4.363599×10^{-6}
$2.0000000000000000 \times 10^{-6}$	9.999935×10^{-6}	6.545373×10^{-6}
$3.0000000000000000 \times 10^{-6}$	9.999913×10^{-6}	8.727125×10^{-6}
$4.0000000000000000 \times 10^{-6}$	9.999891×10^{-6}	1.090887×10^{-5}
$4.9999999999999996 \times 10^{-6}$	9.999869×10^{-6}	1.309061×10^{-5}

A CASE STUDY

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.

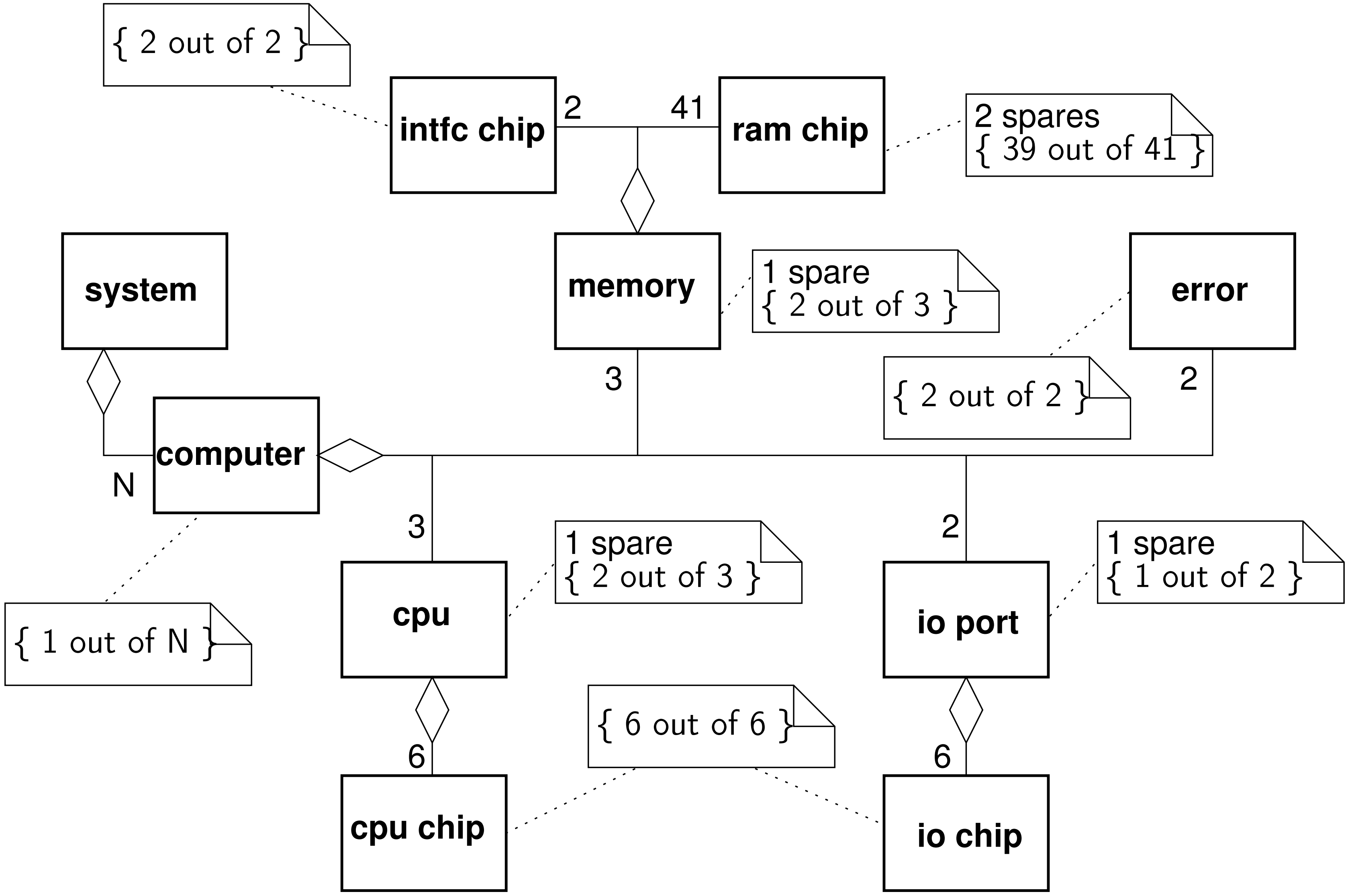
A **computer** is operational if at least
- 2 CPU's,
- 2 memory modules,
- 1 I/O port,
- 2 error-handling chips

are operational.

A **memory module** is operational if at least
- 39 RAM chips,
- 2 interface chips

are operational.

A fault-tolerant multiprocessor system (3)



Fault coverage and failure rate

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.

Summary

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

Modeling strategy

A possible modeling strategy:

- The system is the composition (by *rep*) of N computers;
- a computer is the composition (by *join*) of three atomic models and a replication submodel;
- the three atomic models represent the behavior wrt failures of CPU's, memories, I/O, and error handlers, respectively;
- The replication submodel groups three atomic models, one for each memory module.

Atomic models' structure

All submodels have a similar structure:

- A timed activity whose completion represents the failure of one component of the respective class;
- the timed activity is assumed to have an exponential time distribution;
- the timed activity is enabled if the computer hosting the component 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.

State lumping

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.

Shared places

- 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 CPU module submodel

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

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 probabilities

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 - cpu_cvg) · comp_cvg)	1 - comp_cvg
sum of probabilities	1	1

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 probabilities of interface_chip_failure

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 composed model

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.

The reward model, studies and experiments

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

The unreliabillity 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, using a transient solver.

Conclusions

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/