The Mercury Scripting Language Cookbook

Danilo Oliveira

Abstract This document presents a series of modeling problems, and shows how the Mercury Scripting Language (MSL) makes easy to solve them. Our objective is to show the strong points of the language in different pratical contexts.

1 Introduction

MSL (*Mercury Scripting language*) is a language provided by the Mercury tool [1] [1] for creating and evaluating models. The scripts can be executed by command line interface (CLI), via graphical interface, or inside a Java program (as we will show later in this document). The main objective of the scripting language is to allow the use of Mercury evaluation engine with greater flexibility than the GUI provides. The scripting language enables, for example, using shell scripts to automate an evaluation workflow. The scripting language offers an additional advantage that is the increased support to hierarchical evaluation [2] [3]. Input parameters of any model can be defined as function of an output metric defined by another model, independent of the modeling formalism. The Mercury scripting language currently supports SPN (Stochastic Petri Net), RBD (Reliability Block Diagram), EFM (Energy Flow Model) and CTMC (Continuous Time Markov Chain) models.

We structured this material as a "*cookbook*", that provides a series of "*recipes*". Each recipe contains a practical example that emphasizes some capability of the language:

Danilo Oliveira
Centro de Informática, UFPE, e-mail: `dmo4@cin.ufpe.br`

[1] Available at: `http://www.modcs.org/?page_id=1630`

- **Recipe #1 - Hierarchical modelling + composite metrics** — In this example we show how to structure a set of models hierarchically, and how to declare a metric that is function of another metric;
- **Recipe #2 - Experiments and sensitivity analysis** — In this example we show how to perform quick experiments with the *for* loop, and how to perform sensitivity analysis with the *percentage difference*, and *design of experiments* techniques;
- **Recipe #3 - Reliability block diagram with variable number of blocks** — This example shows how to define a RBD model that does not have a fixed number of blocks, and how to change this model;
- **Recipe #4 - Energy Flow model + availability model** — This example shows how to link an energy flow model with a dependability model in order to compute certain metrics. This linkage turns possible to see how the dependability parameters impacts the energy flow model metrics;
- **Recipe #5 - Using phase type distributions in a Stochastic Petri Net Model** — In this example we see how to use a phase-type distribution (e.g.: hypoexponential, hyperexponential, Erlang) as delay on a SPN model;
- **Recipe #6 - Sensitivity analysis on a performability model** — In this example we illustrates how to compose a performability model by composing a dependability and a performance model. Then, we show how to measure the impact of the dependability parameters on performance metrics;
- **Recipe #7 - Executing a script inside a Java program** — In this example we show how to run scripts from a Java program, and obtaining references to model and metrics programatically.

2 Recipes

2.1 Recipe #1 - Hierarchical modelling + composite metrics

In this recipe, we use as example the hierarchical model for the non-redundant cloud architecture presented in [4]. Figure 1 show the top level RBD model for the system. It is composed by a frontend server, a node server, a storage volume, and the video streaming service that runs on the VM deployed in the node. Except the storage volume components, all blocks are evaluated by another submodels. The frontend and the node servers are modeled by RBDs displayed on figures 2 and 3, respectively. The video streaming service is modeled by the CTMC presented in Figure 4.

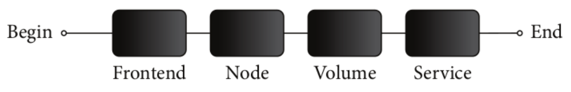


Fig. 1

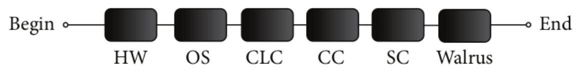


Fig. 2

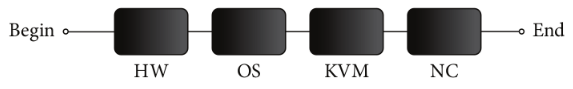


Fig. 3

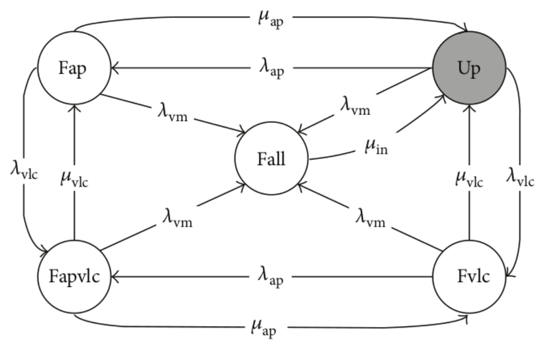


Fig. 4

The script for evaluating the system's availability is presented on Listing 1. On the preamble off the script, we set all the parameters for all models. After the preamble, we start declaring the models. Since the script is evaluated once, and there is no forward declarations, we must declare the submodels before the top level model. If a model A use the result of a metric defined by a model B as one of its parameters, the model B must be declared before the model A.

The models NRFrontend, and Node are simple RBDs composed by a set of blocks with exponential MTTF and MTTF. Each exponential block is declared using the "block" keyword followed by the MTTF and MTTR parameters enclosed by parenthesis. The value of a model parameter can be any numeric expression containing: parenthesis, arithmetic operators (*, +, /, -), number literals, variables, and functions. The availability metric is declared using the “metric” keword, followed by the metric identifer, and the metric type. In this case, we use the “availability” metric, that takes no parameters.

The model NRService is represented by a CTMC with five states, and in only one state (the “Up” state) the system is available. For CTMC availability models, there is three ways to compute the availability. The first is to annotate the “up states” with the “up” keyword, and use the “availability” metric. Alternatively, we can use CTMC expressions following the syntax described in the Mercury manual, by using the “ctmcExpression” metrics, that takes as parameter a string containing the expression. The “up” annotation and the “availability” metric is a shorthand for availability models, while the “ctmcMetric” is a more general way to compute CTMC metrics.

On the top level model, we create a RBD with a series arrangement of blocks. But, instead of using simple exponential blocks, we use hierarchical blocks, by using the “hierarchy” keyword. For availability models, we must define the “availability” parameter for each hierarchical block. We can use any numeric expression for this parameter. Using the “solve” function allow us to solve a metric defined by another model and, in this way, composing models hierarchically. The NRArchitecture model represents the top level model depicted in Figure 1. It defines three metrics. The metric named “av” computes the steady state availability for the RBD. The metrics named “uav” and “downtime” represents composite metrics, i.e., metrics that are defined as function of another metrics. A composite metric is declared by using a numeric expression enclosed by parentheses.

Finally, we create the “main” block, that is the section of the script where we can set parameters, evaluate models, and print the computed results using the “print” and “println” functions. These functions accepts any numeric or string expression as argument, and outputs its values in the console. Using the “..” operator we can concatenate string with numeric values and presenting a more readable output.

```
lambda_ap = 1/788.4;
mu_ap = 1;
lambda_vlc = 1/336;
mu_vlc = 1;
lambda_vm = 1/2880;
mu_vm = 1;
mu_in = 1/0.019166;

mttf_hw = 8760;
mttr_hw = 100/60;
mttf_os = 2895;
mttr_os = 1;
mttf_kvm = 2990;
mttr_kvm = 1;
mttf_nc = 788.4;
mttr_nc = 1;
mttf_clc = 788.4;
mttr_clc = 1;
mttf_cc = 2788.4;
```

```
mttr_cc = 1;
mttf_walrus = 2788.4;
mttr_walrus = 1;

mttf_volume = 100000;
mttr_volume = 1;

RBD NRFrontend{
  block hw( MTTF = mttf_hw, MTTR = mttr_hw );
  block os( MTTF = mttf_os, MTTR = mttr_os );
  block cc( MTTF = mttf_cc, MTTR = mttr_cc );
  block clc( MTTF = mttf_clc, MTTR = mttr_clc );
  block walrus( MTTF = mttf_walrus, MTTR = mttr_walrus );

  series s1(hw, os, clc, cc, walrus);

  top s1;

  metric av = availability;
}

RBD Node{
  block hw( MTTF = mttf_hw, MTTR = mttr_hw );
  block os( MTTF = mttf_os, MTTR = mttr_os );
  block kvm( MTTF = mttf_kvm, MTTR = mttr_kvm );
  block nc( MTTF = mttf_nc, MTTR = mttr_nc );

  series s1(hw, os, kvm, nc);

  top s1;

  metric av = availability;
}

CTMC NRService{

  state fap;
  state Up up;
  state fapvlc;
  state fvlc;
  state fall;

  transition fap -> Up( rate = mu_ap);
  transition fap -> fapvlc( rate = lambda_vlc);
  transition fap -> fall( rate = lambda_vm);
  transition Up -> fap( rate = lambda_ap);
  transition Up -> fvlc( rate = lambda_vlc);
  transition Up -> fall( rate = lambda_vm);
  transition fapvlc -> fap( rate = mu_vlc);
  transition fapvlc -> fvlc( rate = mu_ap);
  transition fapvlc -> fall( rate = lambda_vm);
  transition fvlc -> Up( rate = mu_vlc);
  transition fvlc -> fapvlc( rate = lambda_ap);
  transition fvlc -> fall( rate = lambda_vm);
  transition fall -> Up( rate = mu_in);

  metric av = availability;
  metric av2 = ctmcExpression( expression = "P{Up}" );
}

RBD NRArchitecture{
  hierarchy frontend(
    availability=solve(NRFrontend, av)
  );

  hierarchy node(
    availability=solve(Node, av)
  );

  block volume( MTTF = mttf_volume, MTTR = mttr_volume );

  hierarchy service(
    availability=solve(NRService, av)
  );

  series s1( frontend, node, volume, service);

  top s1;

  metric av = availability;
  metric uav( 1 - av );
  metric downtime( uav * 365 * 24 );
}

main{
  a = solve( NRArchitecture, av );
  println("Availability: " .. a);

  u = solve( NRArchitecture, uav );
  println( "Unavailability: " .. u);

  d = solve( NRArchitecture, downtime );
  println("Annual downtime: " .. d);
}
```

Listing 1 Script for a cloud video streaming service model

2.2 Recipe #2 - Experiments and sensitivity analysis

In this recipe, we will use the model defined in the previous recipe. One of the greatest strengths of the scripting language is the ease to change a model parameter and observe how the metrics reacts by this change. By modifying the contents of a variable that is used as a model parameter (using the assign operator "="), and re-solving the metric of interest using the "solve" function, we will get a new result that corresponds to the updated model. The relying on this feature, the scripting language provides a convenient way to perform experiments, i.e., outputting the result of a metric by changing a parameter over a list, and provide powerful functions to perform sensitive analysis.

2.2.1 Performing experiments using the "for" loop

To observe how a metrics reacts by changing a parameter over a list of values, we use the "for" loop available in the scripting language. Listing 2 shows an example. In this example, we change the "mttf_hw" parameter over the list enclosed by square brackets. For each iteration, we solve the "av" metric of the NRArchitecture model. Then, we print the parameter value, followed by a comma, and the metric result. By outputting the result this way, we can create a comma separated value (CSV) file that will be used by producing a chart by another tool (e.g. R, gnuplot, Excel, etc.)

In Figure 5 we show how to run the script in the command line interface. We run the Mercury program by typing "java -jar mercury.jar" on the terminal. Without any command line arguments, the graphical interface will be displayed. To evaluate the script, we must pass the "-cli" argument, followed by the file name of the script. Any "print" or "println" output will be sent to the console. To save the results in a file, we can use the redirect operator ">" of the shell.

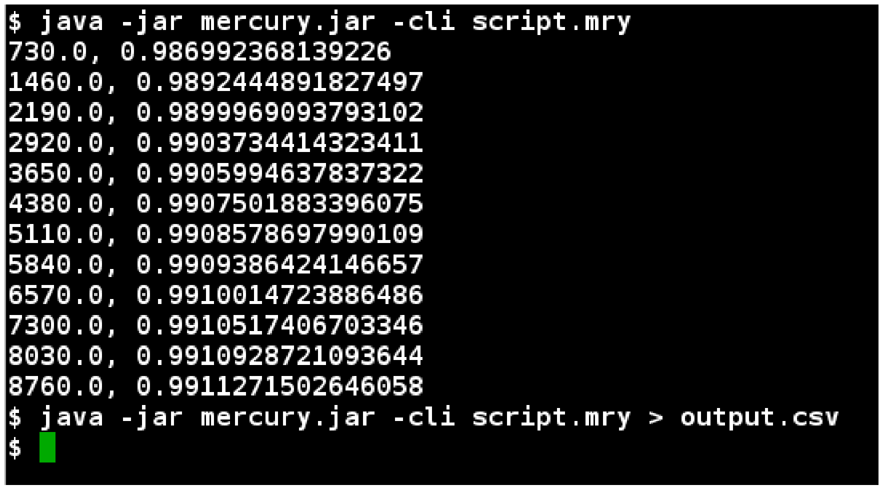
```
main{
  for mttf_hw in [ 730, 1460, 2190, 2920, 3650, 4380, 5110, 5840,
     6570, 7300, 8030, 8760 ]{
    a = solve( NRArchitecture, av );
    println(mttf_hw .. ", " .. a);
  }
}
```

Listing 2 Performing an experiment with the for loop

2.2.2 Sensitivity analysis

The language has two predefined functions for sensitivity analysis: percentage difference, and design of experiments. The technique of sensitivity analysis by means of percentage difference consists in changing one parameter over a list of values, while holding the other parameters fixed, and calculating the percentage difference in the output metric considered. We perform this step for each parameter into our list, and sort them from the highest difference to the lowest. The formula for obtaining the percentage difference is [5]:

$$SI = \frac{D_{max} - D_{min}}{D_{max}}$$



```
$ java -jar mercury.jar -cli script.mry
730.0, 0.986992368139226
1460.0, 0.9892444891827497
2190.0, 0.9899969093793102
2920.0, 0.9903734414323411
3650.0, 0.9905994637837322
4380.0, 0.9907501883396075
5110.0, 0.9908578697990109
5840.0, 0.9909386424146657
6570.0, 0.9910014723886486
7300.0, 0.9910517406703346
8030.0, 0.9910928721093644
8760.0, 0.9911271502646058
$ java -jar mercury.jar -cli script.mry > output.csv
$
```

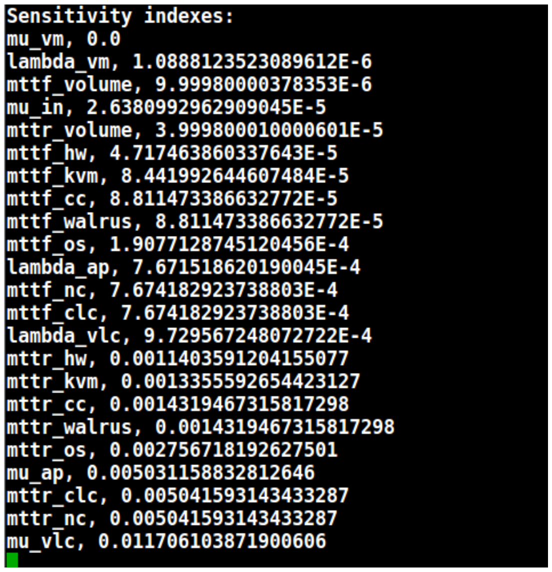
Fig. 5

, where SI is the sensitivity index for the selected parameter, D_{max} is the maximum value of the output metric, and D_{min} is the minimum value.

The Listing 3 shows the usage of the *percentageDifference* function. This function has three mandatory parameters: i) *model_*, which defines the model that will be used in the analysis; ii) *metric_*, which defines the metric from the model that will be used; iii) *parameters*, which represents the list of parameters that will be used in the analysis, and its respective values. Each parameter is set with a list of values enclosed by square brackets. Optionally, the user can define only two values: a minimum and a maximum value, and set the *samplingPoints* parameters, that specifies the number of intermediate points that will be generated for each list. The function prints in the console the list of parameters and respective sensitivity indexes, ranked from least to most influential parameter. Figure 6 shows the output for the script displayed on Listing 3.

```
main{
    av = solve( NRArchitecture, av );
    println(av);

    percentageDifference(
        model_ = "NRArchitecture",
        metric_ = "av",
        samplingPoints = 5,

        parameters =  (
            lambda_ap = [ 1/2000, 1/788.4],
            mu_ap = [ 1/5, 1 ],
            lambda_vlc = [ 1/500, 1/336 ],
            mu_vlc = [ 1/5, 1],
            lambda_vm = [ 1/4000, 1/2880 ],
            mu_vm = [ 1, 5 ],
            mu_in = [ 1/0.019166, 1/0.1 ],
            mttf_hw = [ 8760, 10000 ],
            mttr_hw = [ 100/60, 400/60 ],
            mttf_os = [ 2895, 4000 ],
            mttr_os = [ 1, 5 ],
            mttf_kvm = [ 2990 ],
            mttr_kvm = [ 1, 5 ],
            mttf_nc = [ 788.4, 2000 ],
            mttr_nc = [ 1, 5 ],
            mttf_clc = [ 788.4, 2000 ],
            mttr_clc = [ 1, 5 ],
            mttf_cc = [ 2788.4, 4000],
            mttr_cc = [ 1, 5 ],
            mttf_walrus = [ 2788.4, 4000 ],
            mttr_walrus = [ 1, 5 ],
            mttf_volume = [ 100000 ],
            mttr_volume = [ 1, 5 ]
        ),

        output = (
            type = "swing",
            yLabel = "Steady-state availability",
            baselineValue = av
        )
    );
}
```

Listing 3 Performing sensitivity analysis with the percentage difference technique



```
Sensitivity indexes:
mu_vm, 0.0
lambda_vm, 1.0888123523089612E-6
mttf_volume, 9.99980000378353E-6
mu_in, 2.6380992962909045E-5
mttr_volume, 3.999800010000601E-5
mttf_hw, 4.717463860337643E-5
mttf_kvm, 8.441992644607484E-5
mttf_cc, 8.811473386632772E-5
mttf_walrus, 8.811473386632772E-5
mttf_os, 1.9077128745120456E-4
lambda_ap, 7.671518620190045E-4
mttf_nc, 7.674182923738803E-4
mttf_clc, 7.674182923738803E-4
lambda_vlc, 9.729567248072722E-4
mttr_hw, 0.0011403591204155077
mttr_kvm, 0.0013355592654423127
mttr_cc, 0.0014319467315817298
mttr_walrus, 0.0014319467315817298
mttr_os, 0.002756718192627501
mu_ap, 0.005031158832812646
mttr_clc, 0.005041593143433287
mttr_nc, 0.005041593143433287
mu_vlc, 0.011706103871900606
```

Fig. 6 Output of the *percentageDifference* function

The *output* optional parameter is used to produce charts for each parameter. This parameter has three sub-parameters: i) *type*, which defines the type of chart that will be produced; ii) *yLabel*, which defines the label of the y axis in a chart; iii) *baselineValue*, for displaying a horizontal line in the chart for comparing each point with a baseline value. Currently, the only chart type available is *"swing"*, that displays all charts inside a GUI window. In the next release, we plan to include support for generating R and Gnuplot scripts for producing charts. Figure 7 displays the generated charts for the script shown above.

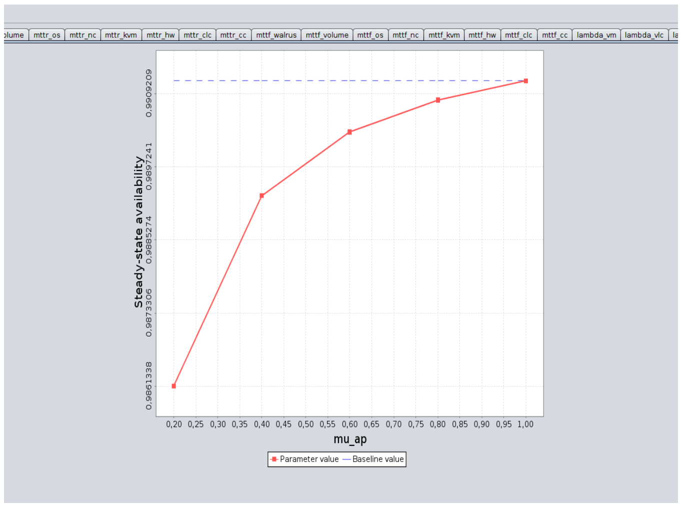


Fig. 7 Charts generated automatically by the *percentageDifference* function

The Design of Experiments (DOE) technique consists in taking a list of parameters and, for each parameter (called factor), a list of values (called levels), and performing a series of experiments with the possible combinations of factor and values. There are various possible designs for an experiment. One possible alternative is to run the experiment for all combination of levels using all factors. This is called a *full factorial design*. A drawback of this alternative is that, even for a small list of parameters, the number of experiments to be performed could be very large. One solution is to use only two levels - this is called a *two-level factorial design*. Even using only two levels, the number of experiments can be very large, if we have many parameters, and grows exponentially for each added parameter. To overcome this issue, we may perform a *fractional factorial design*, that uses only a subset of a factorial design. The Mercury tool allow us to perform two-level, full, and fractional designs, and compute the effects of each factor as described in [6].

Listing 4 shows the usage of the *designOfExperiment* function. It takes the same three mandatory parameters than the *percentageDifference* function: i) *model_*, *metric_*, and iii) *parameters*. There is no *samplingPoints* parameter, and each parameter receives a list with only two values: a min, and a max value.

```
main{
    designOfExperiment(
        model_ = "NRArchitecture",
        metric_ = "av",
        parameters =  (
            lambda_ap = [ 1/2000, 1/788.4],
            mu_ap = [ 1/5, 1 ],
            lambda_vlc = [ 1/500, 1/336 ],
            mu_vlc = [ 1/5, 1],
            lambda_vm = [ 1/4000, 1/2880 ],
            mu_vm = [ 1, 5 ],
            mu_in = [ 1/0.019166, 1/0.1 ],
            mttf_hw = [ 8760, 10000 ],
            mttr_hw = [ 100/60, 400/60 ],
            mttf_os = [ 2895, 4000 ],
            mttr_os = [ 1, 5 ],
            mttf_kvm = [ 2990 ],
            mttr_kvm = [ 1, 5 ],
            mttf_nc = [ 788.4, 2000 ],
            mttr_nc = [ 1, 5 ],
            mttf_clc = [ 788.4, 2000 ],
            mttr_clc = [ 1, 5 ],
            mttf_cc = [ 2788.4, 4000],
            mttr_cc = [ 1, 5 ],
            mttf_walrus = [ 2788.4, 4000 ],
            mttr_walrus = [ 1, 5 ],
            mttf_volume = [ 100000 ],
            mttr_volume = [ 1, 5 ]
        )
    );
}
```

Listing 4 Performing sensitivity analysis with the design of experiments technique

2.3 Recipe #3 - Reliability block diagram with variable number of blocks

The MSL language provides a construct to define a RBD model with a variable number of blocks into a series or parallel arrangement. It may be useful to answer questions like "*How much the system's availability will be improved if we add one more redundant block?*". In this recipe, we will use the model of an Eucalyptus private cloud with a variable number of worker nodes. Figure 8 shows the top level model. The cloud is composed by a frontend node running the management services of the cloud, and by a set of worker nodes.

The models for the frontend and node were shown on previous section, in figures 2 and 3.

Lines 53 to 59 of Listing 2.3 shows an example of this construct. Using this construct, we can define a series/parallel arrangement with a variable number of similar blocks. After the block id ("*nodes*"), we pass the parameters of the grouping inside parentheses. The "*times*" parameter defines the number of blocks of the series/parallel arrangement. This value must have an initial value. If it uses a variable, the variable value must be previously inside the model (using the *set* keyword), or in the script preamble. For specifying the block that will be repeated, he have two notations. We can set the "*mttf*" and "*mttr*" parameters [2], or we can set the "*hierarchyBlock*" parameter, for hierarchical blocks. Figure 9 shows the output of the script of Listing 2.3.

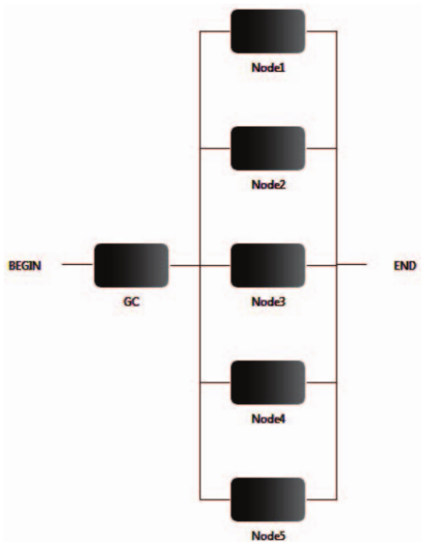
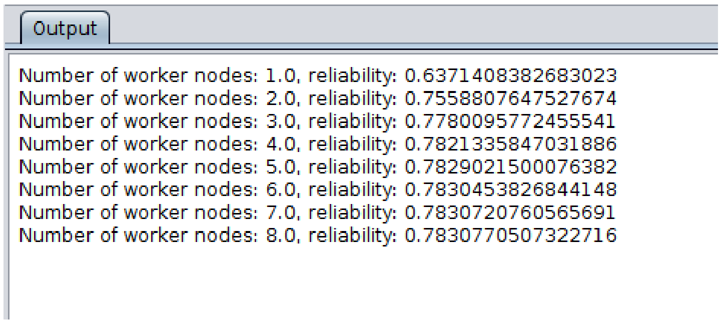


Fig. 8

```
mttf_hw = 8760;
mttr_hw = 100/60;
mttf_os = 2895;
mttr_os = 1;
mttf_kvm = 2990;
mttr_kvm = 1;
mttf_nc = 788.4;
mttr_nc = 1;
mttf_clc = 788.4;
```

[2] Notice that the parameters are in lowercase. This is due the fact that we now are using the dictionary syntax, therefore we can not use the $MTTF$ and $MTTR$ reserved words as keys

```
mttr_clc = 1;
mttf_cc = 2788.4;
mttr_cc = 1;
mttf_walrus = 2788.4;
mttr_walrus = 1;

mttf_volume = 100000;
mttr_volume = 1;

n_nodes = 1;

RBD NRFrontend{
  block hw( MTTF = mttf_hw, MTTR = mttr_hw );
  block os( MTTF = mttf_os, MTTR = mttr_os );
  block cc( MTTF = mttf_cc, MTTR = mttr_cc );
  block clc( MTTF = mttf_clc, MTTR = mttr_clc );
  block walrus( MTTF = mttf_walrus, MTTR = mttr_walrus );

  series s1(hw, os, clc, cc, walrus);

  top s1;

  metric rel = reliability( time = t );
}

RBD Node{
  block hw( MTTF = mttf_hw, MTTR = mttr_hw );
  block os( MTTF = mttf_os, MTTR = mttr_os );
  block kvm( MTTF = mttf_kvm, MTTR = mttr_kvm );
  block nc( MTTF = mttf_nc, MTTR = mttr_nc );

  series s1(hw, os, kvm, nc);

  top s1;

  metric rel = reliability( time = t );
}

RBD CloudModel{
  hierarchy frontend(
    reliability=solve(NRFrontend, rel)
  );

  parallel nodes(
    times = n_nodes,
    hierarchyBlock = (
      reliability= solve(model = Node, metric = rel)
    )
  );

  series s1( frontend, nodes );

  top s1;

  metric rel = reliability( time = t );
}

main{
  t = 100;

  for n_nodes in [1, 2, 3, 4, 5, 6, 7, 8] {
    r = solve_rm( CloudModel , rel );
    println( "Number of worker nodes: " .. n_nodes .. ",
    reliability: " .. r );
  }
}
```



```
Number of worker nodes: 1.0, reliability: 0.6371408382683023
Number of worker nodes: 2.0, reliability: 0.7558807647527674
Number of worker nodes: 3.0, reliability: 0.7780095772455541
Number of worker nodes: 4.0, reliability: 0.7821335847031886
Number of worker nodes: 5.0, reliability: 0.7829021500076382
Number of worker nodes: 6.0, reliability: 0.783045382684148
Number of worker nodes: 7.0, reliability: 0.7830720760565691
Number of worker nodes: 8.0, reliability: 0.7830770507322716
```

Fig. 9 Output of the script of Listing 2.3

2.4 Recipe #4 - Energy Flow model + availability model

The Energy Flow Model [7] formalism is used to represent the energy flow between the system components considering the respective efficiency and the maximum energy that each component can provide (considering electrical devices) or extract (assuming cooling devices). The system under evaluation can be correctly arranged, in the sense that the required components are properly connected, but they may not be able to meet system demand for electrical energy or thermal load.

Some metrics of an EFM model takes an *availability* parameter, that must inform the steady state availability of the datacenter being evaluated. This value must be computed by a separated availability model of the datacenter. Thanks to the hierarchical modeling capabilities of the scripting language, it is possible to link the EFM model and the availability model. This feature turns possible to see how the availability model parameters impact the EFM metric.

In this recipe, we will use as example the EFM model represented in Figure 10. Figure 11 shows the corresponding availability model. Listing 5 shows a script that implements these two models. In the line 48 of the script we declare a "*operationalExergy*" metric, that takes two parameters: *time* and *availability*. The *availability* parameter is obtained by the RBD model named *AvailModel*, using the *solve* function.

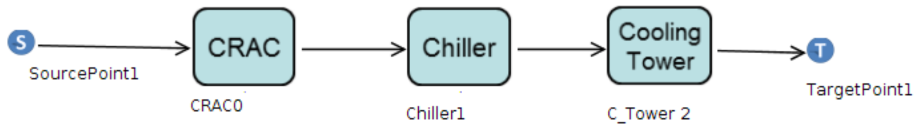


Fig. 10 EFM model

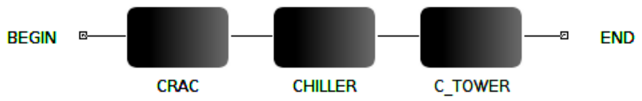


Fig. 11 Availability model of the EFM of Figure 10

```
RBD AvailModel{
    block crac( MTTF = mttf_crac , MTTR = mttr_crac );
    block chiller( MTTF = mttf_chiller , MTTR = mttr_chiller );
    block c_tower( MTTF = mttf_ctower , MTTR = mttr_ctower );

    series s1 ( crac , chiller , c_tower );

    top s1;

    metric aval = availability;
}


EFM EFM1{
    component source(
        type = "Source",
        parameters = (
            demandedEnergy = 10
        )
    );

    component target(
        type = "Target",
        parameters = (
            demandedEnergy = 10
        )
    );
    component crac(
        type = "CRAC",
        parameters = (
            efficiency = e,
            retailPrice = r
        )
    );
    component chiller(
        type = "Chiller"
    );
    component tower(
        type = "C_Tower"
    );

    arc source -> crac;
    arc crac -> chiller;
    arc chiller -> tower;
    arc tower -> target;


    metric m = operationalExergy( time = 1000, availability =
    solve( AvailModel, aval ) );
}


main {
    mttf_crac = 1000;
    mttr_crac = 1;
    mttf_chiller = 6000;
    mttr_chiller = 10;
    mttf_ctower = 10000;
    mttr_ctower = 20;

    m = solve( EFM1, m );
    println( "Operational exergy: " .. m );
}
```

Listing 5 Script for a EFM model

2.5 Recipe #5 - Using phase type distributions in a Stochastic Petri Net Model

When modelling a certain system of the real world using Stochastic Petri Nets, there are two options for evaluating the metrics: stationary/transient analysis and simulation. Stationary/transient analysis provides more accurate results, but the drawback is that the delay associated with the transitions must be exponentially distributed. If an user collects real world data to parametrize his/her model, and the histogram indicates that the data is not even close to an exponential distribution, the assumption of an exponentially distributed delay makes the model deviates from the real system.

Consider, for instance, that the service time from the SPN from Figure 12 is measured from the real system, and the user obtained the data depicted in the histogram of Figure 13. As we can observe, the histogram curve is different from a exponential distribution. If the user assumes an exponential service time from these data, he/she can obtain different results from the real system.

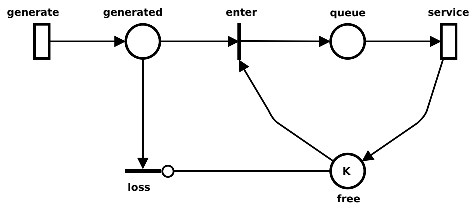


Fig. 12 Stochastic Petri Net model for a M/M/1/k queue [8]

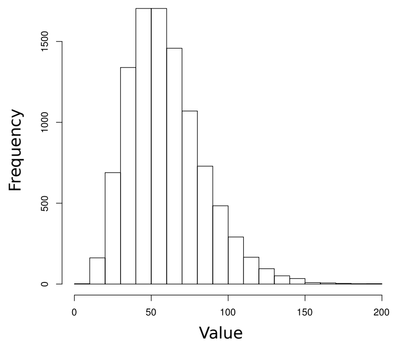
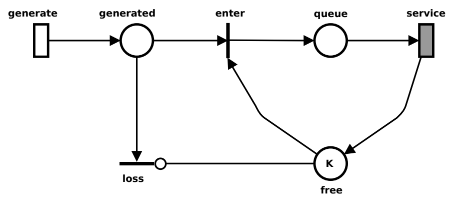


Fig. 13 Histogram plot from collected data

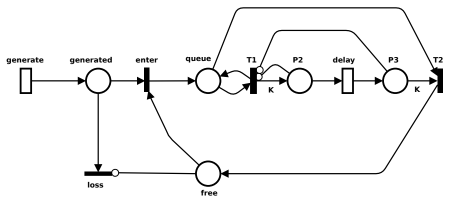
A existent solution to overcome this problem is to use *phase-type* distributions [9]. A phase-type distribution can be expressed as a composition of exponential distributions. A important characteristic of this class of probability distributions is that they can be used to approximate an empirical distribution [10].

A tradeoff found in using phase-type distributions to approximate the time of firing of transitions is that the model can become more complex and difficult to understand. To simplify the use of this class of distributions in SPN models, the MSL language defines a special syntax for expressing another distribution types than the exponential distribution. When the evaluation engine for the scripting language detects a phase-type distribution delay, it generates the structure for the phase-type transition as a subnet, and inserts this subnet on the actual Petri net by using the hierarchical transitions of the engine. As consequence, the model will be simpler than if the structure of the phase-type transition was mixed with the Petri net structure. In the graphical representation on the Mercury interface, exponential transitions are displayed with a white background, and non-exponential transitions are displayed with a shadowed background.

Figure 14 a) shows the SPN model with a transition with delay following an Erlang distribution. This transition is represented with a shaded background. Without this feature, our model should be depicted as in Figure 14 b), with the inclusion of additional places, arcs and transitions (displayed inside the dotted box) the shaded transition of the previous model.



(a)



(b)

Fig. 14 Stochastic Petri Net model with a Erlang transition

In the Listing 6, we show the code for the SPN model of Figure 14 a). The *service* transition is configured with a phase-type delay determined by a Erlang distribution. Instead of giving a numeric value for defining a simple timed transition, we have to pass the distribution type and its parameters. In this example, we use the *"Erlang"* string value to specify a Erlang distribution. This distribution have two parameters: *shape* - the number of phases, and *meanDelay* - the exponential delay of one phase.

```
k = 5;
arrivalTime = 1.5;
serviceTime = 0.1;
```

```
phases = 6;

SPN Model{

  place buffer( tokens = k );
  place generated;
  place queue;


  immediateTransition drop(
    inputs = [generated],
    inhibitors = [buffer]
  );

  immediateTransition enter(
    inputs = [generated, buffer],
    outputs = [queue]
  );

  timedTransition generate(
    outputs = [generated],
    delay = arrivalTime
  );

  timedTransition service(
    inputs = [queue],
    outputs = [buffer],
    delay = (
      type="Erlang",
      parameters = (
        meanDelay=serviceTime,
        shape=phases
      )
    )
  );

  // The expected number of tokens in the "queue" place
  metric equeue = stationaryAnalysis( expression = "E{#queue}" );

}

main {
  e = solve( Model, equeue );
  println( e );
}
```

Listing 6 Timed transition with phase-type delay

2.6 Recipe #6 - Sensitivity analysis on a performability model

In some performability studies, we are interested on finding how much the performance of a system is affected due the presence of failures. Thanks to the powerful *solve* function, we can link a dependability to a performance model, and measure how changing the dependability model parameters impacts the performance measures. Figure 15 shows an example of performability model. The performance model is a SPN model that represents a M/M/1/K queue. We put a place of *server_up*/*server_down* places that indicates the operational status of the server. If the server is down, the server is not able to process a request, as indicated by the inhibitor arc. The delay of the *fail*/*repair* transitions is computed by a separated RBD model. This RBD model is composed by three blocks: hardware, operating system and application. The MTTF and MTTR metrics of this model are used as delay for the *fail*/*repair* transitions on the SPN model.

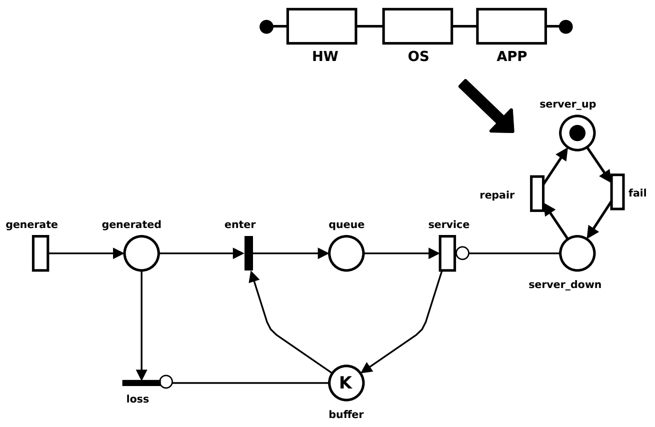


Fig. 15 Performability model

The script is exhibited on Listing 7. We create a RBD model named *DependModel*, that declares two metrics: the MTTF and the MTTR. The value of those metrics are used as input for the performance model, as shown in lines 62 and 66. On the SPN model, we declares two metrics: *m1*, that computes the stationary probability of all markings that turns the *service* transition enabled; *tp*, a composite metric that computes the throughput of the *service* transition. Finally, on the main block, we variate the *mttf_hw* parameter over a list, and show how the throughput of the server is affected by this variation.

```
arrivalTime = 1.5;
serviceTime = 1.2;
```

```
k= 5;

mttf_hw = 1000;
mttr_hw = 4;
mttf_os = 700;
mttr_os = 1;
mttf_app = 500;
mttr_app = 0.1;

RBD DependModel{
  block HW( MTTF = mttf_hw, MTTR = mttr_hw );
  block OS( MTTF = mttf_os, MTTR = mttr_os );
  block APP( MTTF = mttf_app, MTTR = mttr_app );

  series s1( HW, OS, APP );

  top s1;

  metric mttf_ = mttf;
  metric mttr_ = mttr;
}

SPN Model{

  place buffer( tokens = k );
  place generated;
  place queue;

  place server_up( tokens = 1 );
  place server_down;


  immediateTransition drop(
    inputs = [generated],
    inhibitors = [buffer]
  );

  immediateTransition enter(
    inputs = [generated, buffer],
    outputs = [queue]
  );

  timedTransition generate(
    outputs = [generated],
    delay = arrivalTime
  );

  timedTransition service(
    inhibitors = [server_down],
    inputs = [queue],
    outputs = [buffer],
    delay = serviceTime
  );

  timedTransition fail(
    inputs = [server_up],
    outputs = [server_down],
    delay = solve( DependModel, mttf_ )
  );

  timedTransition repair(
    inputs = [server_down],
    outputs = [server_up],
    delay = solve( DependModel, mttr_ )
  );

  metric m1 = stationaryAnalysis( expression = "P{(#queue>0)AND(#
    server_up=1)}" );
  metric tp( m1 / serviceTime );

}

main {
  for mttf_hw in [1000, 1200, 1300, 1400, 1500] {
    tp = solve( Model, tp );
    println( tp );
  }
}
```

Listing 7 Script for a performability model

2.7 Recipe #7 - Executing a script inside a Java program

Suppose that a programmer wants to create a specific tool that will be used to model cloud infrastructures. With this tool, an user will be able to create clusters, specify frontend and worker nodes, and compose them into a cloud infrastructure. By using the provided high level cloud model, the user will be able to extract performance and dependability metrics of a cloud. This can be achieved by converting the high level model into a SPN/RBD/CTMC model, and solving its metrics. To achieve this goal, the programmer must be able to create SPN/RBD/CTMC models, and solving it inside his/her program.

The Mercury tool exports the same API used by the scripting evaluation runtime. By using the Mercury executable archive (.jar) as a library inside an external Java program, it is possible to:

- Run scripts, i.e., parsing the script, loading the models in the runtime, and running the main block;
- Evaluate scripts, i.e., parsing the script, loading the models in the runtime, but not running the main block;

- Obtaining reference to models and metrics;
- Modifying parameters;
- Solving metrics.

The scripts can be created "on the fly" and stored into a *String* variable, or they can be stored in the disk. The Listing 8 shows a Java program that performs the above mentioned steps. To compile this class, we must to create a Java project using any IDE (Eclipse, Netbeans), and add the Mercury as a library in the dependencies. We supply a single .jar file that contains the Mercury API and also the Mercury dependencies into a single package, as shown in Figure 16.

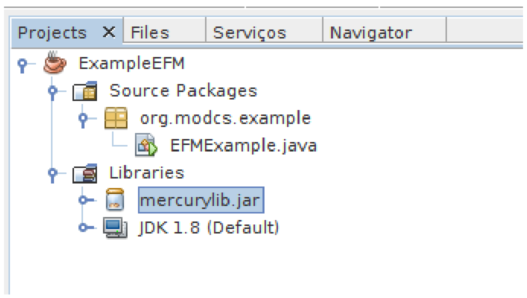


Fig. 16 Netbeans project with the Mercury API as library

In the Listing 8, we evaluate a script located in the file named *"efm_ model.mry"*, that must be in the same path of the program, or inside the root of the Netbeans/Eclipse project folder. The Script class has two constructors:

- public Script(java.io.File file);
- public Script(String script);

The first constructor is used to evaluate a script located inside a file, and the second is used to evaluate a script that is stored into a java.lang.String object. For evaluating this script, we call the *"evaluateScript"* method of the runtime object. This method sets the variables defined in the preamble (if this section exists), loads all models in the runtime, but does not execute the main block. If we want to run the main block after loading the models, we call the *"runScript"* method.

```
package org.modcs.example;

import java.io.File;
import org.modcs.tools.parser.model.ExecutionRuntime;
import org.modcs.tools.parser.model.Model;
import org.modcs.tools.parser.model.Script;
import org.modcs.tools.parser.model.metrics.Metric;

```

```
public class EFMExample {
    public static void main(String[] args) {

        //creating the scripting evaluatin runtime
        ExecutionRuntime runtime = new ExecutionRuntime();

        //creating an evaluating the script, without
        //running the main block
        Script scrpt = new Script( new File("emf_model.mry"));
        runtime.evaluateScript(scrpt);


        //modifying some variables and changing the model
        //parameters
        runtime.getVariableTable().setValue("mttf", 1000);
        runtime.getVariableTable().setValue("mttr", 1);
        runtime.getVariableTable().setValue("e", 0.8);
        runtime.getVariableTable().setValue("r", 5000);

        //obtaining reference to a model by passing
        //its identifier
        Model model = runtime.getModel("EFM1");

        //obtaining reference to a metric
        Metric m = model.getMetric("m2");

        //solving and printing the metric
        double val = m.solve();
        System.out.println("Metric value: " + val);


        //performing a experiment
        double[] mttfs = { 500, 1000, 1500, 2000, 2500 };

        for(double mttf: mttfs){
            runtime.getVariableTable().setValue("mttf", mttf);

            System.out.println(m.solve());
        }
    }
}
```

Listing 8 Executing a script inside a Java program

References

1. B. Silva, R. Matos, G. Callou, J. Figueiredo, D. Oliveira, J. Ferreira, J. Dantas, A. L. Junior, V. Alves, and P. Maciel, "Mercury: An integrated environment for performance and dependability evaluation of general systems," *Proceedings of Industrial Track at 45th Dependable Systems and Networks Conference (DSN)*, 2015.

2. J. Dantas, R. Matos, J. Araujo, and P. Maciel, “Eucalyptus-based private clouds: availability modeling and comparison to the cost of a public cloud,” *Computing*, pp. 1–20, 2015.
3. R. Matos, J. Araujo, D. Oliveira, P. Maciel, and K. Trivedi, “Sensitivity analysis of a hierarchical model of mobile cloud computing,” *Simulation Modelling Practice and Theory*, vol. 50, no. 0, pp. 151 – 164, 2015, special Issue on Resource Management in Mobile Clouds.
4. R. M. De Melo, M. C. Bezerra, J. Dantas, R. Matos, I. J. De Melo Filho, and P. Maciel, “Redundant vod streaming service in a private cloud: Availability modeling and sensitivity analysis,” *Mathematical Problems in Engineering*, vol. 2014, 2014.
5. F. Hoffman and R. Gardner, “Evaluation of uncertainties in environmental radiological assessment models,” in *Radiological Assessments: a Textbook on Environmental Dose Assessment*, J. Till and H. Meyer, Eds. Washington, DC: U.S. Nuclear Regulatory Commission, 1983, Report No. NUREG/CR-3332.
6. R. Jain, *The art of computer systems performance analysis.* John Wiley & Sons, 2008.
7. G. Callou, P. Maciel, D. Tutsch, J. Ferreira, J. Araújo, and R. Souza, “Estimating sustainability impact of high dependable data centers: A comparative study between brazilian and us energy mixes,” *Computing*, vol. 95, no. 12, pp. 1137–1170, 2013.
8. R. German, “A concept for the modular description of stochastic petri nets (extended abstract,” in *Proc. 3rd Int. Workshop on Performability Modeling of Computer and Communication Systems*, 1996, pp. 20–24.
9. L. Breuer and D. Baum, “Phase-type distributions,” *An Introduction to Queueing Theory and Matrix-Analytic Methods*, pp. 169–184, 2005.
10. H.-B. Mor, “Performance modeling and design of computer systems,” 2013.