Advanced Stochastic Petri Net Modeling with the Mercury Scripting Language

Danilo Oliveira
Federal University of Pernambuco, Brazil
dmo4cin.ufpe.br

Rubens Matos
Federal Institute of Education, Science, and Technology of Sergipe, Brazil
rubens.junior@ifs.edu.br

Jamilson Dantas
Federal University of Pernambuco, Brazil
jrd@cin.ufpe.br

João Ferreira
Federal University of Pernambuco, Brazil
jfsj3@cin.ufpe.br

Bruno Silva
Federal University of Pernambuco, Brazil
bs@cin.ufpe.br

Gustavo Callou
Federal Rural University of Pernambuco, Brazil
gustavo.callou@ufrpe.br

Paulo Maciel
Federal University of Pernambuco, Brazil
prmm@cin.ufpe.br

André Brinkmann
Johannes Gutenberg University Mainz, Germany
brinkman@uni-mainz.de

ABSTRACT

Formal models are widely used in performance and dependability studies of computational systems. Graphical modeling tools allow users to compose such models with ease, but they complicate the creation of models with a dynamic/complex structure, the hierarchical arrangement of different models, and the automatic execution of models with different parameter configurations. To overcome this problem, we created a scripting language for the Mercury tool that supports the combination of different modeling approaches (e.g., Stochastic Petri Nets and Reliability Block Diagrams) in a single project. In this paper, we focus on the extensions developed to improve the capabilities of Generalized Stochastic Petri net Modeling: substitution transitions, phase-type delays for timed transitions, support for nets with a variable structure, and event-based programming for simulation.

CCS CONCEPTS

• **Computing methodologies → Modeling methodologies**; *Model verification and validation*; *Simulation languages*;

KEYWORDS

Generalized Stochastic Petri nets, hierarchical modeling, phase-type distribution, discrete-event simulation

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
VALUETOOLS 2017, December 5–7, Venice, Italy
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6346-4/17/12.
https://doi.org/10.1145/3150928.3150959

ACM Reference Format:
Danilo Oliveira, Rubens Matos, Jamilson Dantas, João Ferreira, Bruno Silva, Gustavo Callou, Paulo Maciel, and André Brinkmann. 2017. Advanced Stochastic Petri Net Modeling with the Mercury Scripting Language. In ,. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3150928.3150959

1 INTRODUCTION

Information and Communications Technology (ICT) systems are becoming increasingly ubiquitous, so that many professional and personal activities have become dependent on such systems. Users and system administrators require efficient and reliable infrastructures. However, ensuring those properties is not a simple task, given that developing computing systems is not easy. Frequently, even after a long cycle of analysis, design, and implementation, the deployed systems do not operate at satisfactory performance and dependability levels. The use of formal models from the earliest stages of system development helps to evaluate and ensure non-functional requirements such as performance, reliability, availability, and security [1]. Models also help to predict the system performance in future scenarios with bigger workloads, offering guidelines for the system administrator on how to modify the system to ensure desirable levels of performance and dependability.

Software packages for the creation and evaluation of formal dependability and performance models are essential tools for system analysts, since they provide support for dealing with the complexity of each formalism. Such software engines often provide a graphical user interface with drag-and-drop support, and evaluation mechanisms for obtaining metrics from the models. Each software has its strengths and weaknesses and can be specialized on a particular formalism, or provide support for different types of models. The Mercury tool [2] was initially developed by the MoDCS research group[1] as the evaluation engine of the Astro framework [3]. Mercury can also be used without the Astro framework, as a modeling tool for the following formalisms: Continuous Time Markov Chains

[1] http://www.modcs.org

(CTMC) [4], Generalized Stochastic Petri nets (GSPN) [5], Energy Flow Models (EFM) [3], and Reliability Block Diagrams (RBD) [6]. In addition, this tool offers useful features, such as moment matching of empirical data, support for almost 30 distribution probabilities for the simulation of Generalized Stochastic Petri nets, and sensitivity analysis of CTMC and RBD models. Mercury features can also be accessed from other tools through an application programming interface (API).

Although graphical user interfaces are simpler and easier for typical users, sometimes a scripting language and a command line tool can offer a more flexible workflow when evaluating user models. This work presents the scripting language MSL developed for the Mercury tool. MSL improves the expressiveness of Mercury while also adding more flexibility to the modeling workflow. The scripting language extends the graphical interface by providing better support for hierarchical modeling and symbolic evaluation of parameters. In this work, we are interested in showing the following features for Petri net modeling:

- Support for hierarchical (also known as substitution) transitions on Generalized Stochastic Petri net models. This transition type can be used to reduce the complexity of models or to express a recurring structure in the model for ease of reuse;
- Phase-type delay for transitions in GSPN models. This class of probability distributions can be used to approximate distributions that do not fit in an exponential distribution;
- Support for the creation of Petri nets with variable/conditional structure;
- Event-based programming for simulation.

The remainder of this paper is organized as follows. Section 2 describes the core of the scripting language and the representation of GSPNs. Then, Section 3 presents the mentioned extensions provided by the language. Section 4 presents an overview of related software packages. Finally, conclusions and future directions are presented in Section 5.

2 THE MSL SCRIPTING LANGUAGE

In this section, we present the characteristics of the MSL language, with particular emphasis on GSPN modeling. For knowing more about the other formalisms (namely: Continuous Time Markov Chains, Discrete Time Markov Chains, Reliability Block Diagrams, Dynamic Reliability Block Diagrams, and Energy Flow Models), we refer the reader to the tool's manual [2].

The main motivation for creating this scripting language has been to allow the use of the Mercury evaluation engine with greater flexibility than the GUI provides. With this purpose, the language provides two important features:

- Support for **hierarchical modeling**. The resulting metrics of a model can serve as an input parameter for top-level models;
- Support for **symbolic evaluation**. The input parameters of models can contain expressions and variables that are assessed at the time of evaluation.

[2]Available at http://www.modcs.org/?page_id=1397

Those two features are present in the SHARPE tool [7] as well, which also grants a scripting language. However, Mercury has some unique features regarding GSPN models:

- Syntax for **phase-type distributions** [8]. Phase-type distributions can be used to approximate general distributions in a Markovian model. The drawback of using it is that it complicates the structure of the model. The scripting language now allows the representation of a transition delay in terms of phase-type models and, therefore, simplifies the models.
- **Loop and conditional structures** inside the model declaration for creating nets with variable structure. The variables for controlling those structures can be treated as model parameters;
- **Event handlers** for transitions (for simulations). The events can be coded inside the script, or by using the Java programming language. With an event handler, the modeler can modify variables, model parameters, and even the net structure at runtime.

The first two items can be used in conjunction with the numerical evaluation of GSPNs (i.e., extracting the metrics by converting the GSPN to an equivalent Markov chain). To the best of our knowledge, our tool is the first to provide such features for GSPN modeling when applying analytical methods.

2.1 Generalized Stochastic Petri Net Syntax

This section illustrates the syntax for GSPNs, using a model for an M/M/1/K queue extracted from [9] and depicted in Figure 1. The transition "*generate*" creates tokens that correspond to service requests. For each generated token placed in the place "*generated*"', a choice is made. A token can be queued to be processed by the server if there is a free position in the queue. If the queue is full, however, the token will be discarded by the immediate transition "*loss*". The place "*free*" controls the firing of this transition, using the inhibitor arc. Tokens waiting for service are put in the "*queue*" place. The transition "*service*" represents the processing of the queued requests by the server. The script for the M/M/1/K queue GSPN model is shown in Listing 1.

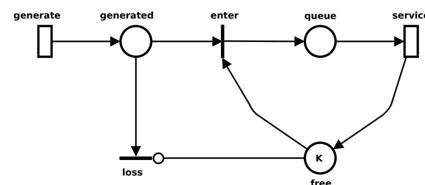

Figure 1: GSPN model for a M/M/1/k queue [9]

```
1 SPN Petri{
    place generated;
3   place queue;
    place free(tokens= k );
5   timedTransition generate(
       outputs = [generated],
7      delay = lambda
    );
9   timedTransition service(
       inputs = [queue],
```

```
      outputs = [free],
      delay = mu
   );
   immediateTransition loss(
      inputs = [generated],
      inhibitors = [free]
   );
   immediateTransition enter(
      inputs = [generated, free],
      outputs = [queue]
   );
   metric m1 =
     stationaryAnalysis(
        expression = "P{#queue>0}"
     );
}
main {
   k = 10;
   mu = 2;
   lambda = 1;
   througput =  solve( model = Petri,
      metric  = m1 ) /  mu;

   println(througput);
}
```

Listing 1: Script for GSPN model

The declaration of a GSPN model starts with the keyword *"SPN"* followed by the model identifier and the model structure between brackets. Inside the brackets, there are declarations of places, transitions, and metrics to be evaluated. In MSL, a place is created with the keyword *"place"* followed by the initial marking enclosed inside parentheses (assuming the initial marking is at least one token). To create transitions, the transition type must be specified with the appropriate keyword: "*timedTransition*" for transitions with an associated delay or "*immediateTransition*" for immediate transitions. After the keyword, we specify the transition name and then specify the arcs and parameters of the transition.

The *"metric"* statement is used for obtaining stationary/transient probabilities and the expected number of tokens of a place. In this example, the expression defines all markings in which the transition "*service*" is enabled. The main section is the place for setting the model parameters, evaluating models and printing the results. In this example, the script calculates the throughput of the transition "*service*" by dividing the metric "*m1*" by the average firing time of the transition.

3 EXTENSIONS FOR ADVANCED PETRI NET MODELING

This section presents the advanced features of GSPNs supported by MSL language. These features allow the language to overcome some limitations of the formalism and they contribute to the creation of more compact and accurate models.

3.1 Hierarchical Transitions on GSPNs

Similar to an object-oriented or modular language, Petri nets models can also benefit from mechanisms for modularization. By segmenting and modularizing Petri nets, it is possible to use "top-down" and "divide-and-conquer" techniques when conceiving complex Petri nets. Moreover, with a modular Petri net, it is possible to identify repeating structures inside a Petri net model and to define them as modules. Instead of repeating the same structure in different parts of the model, they could be instantiated in the top-level net. There are several approaches to create modular Petri nets. One of these approaches involves the concept of hierarchical transitions [10]. A hierarchical transition works like a standard transition, by taking tokens from input places and putting them in output places. However, the delay of this transition is expressed by a subnet. The following example illustrates the support of hierarchical transitions on MSL.

The Petri net model shown in Figure 2 represents a token ring network. This model was originally presented in [11], and we introduced a few modifications to enable the use of hierarchical transitions. For the sake of brevity, details about the model are suppressed. Notice that there is a recurring structure in the net, which we have highlighted with a dotted rectangle. This sub-model can be represented by a hierarchical transition and used in the top-level model. Subsequently, the final model will be clearer and will avoid the need for repeated structures.

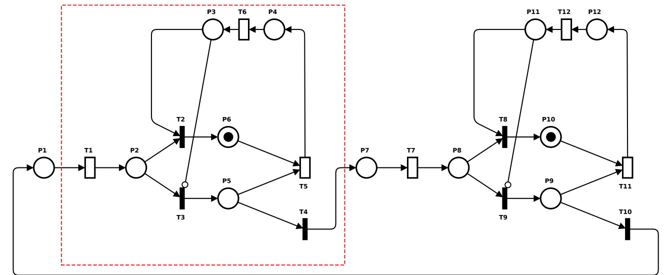

Figure 2: GSPN model for a Token Ring network

Figure 3 shows the same model with hierarchical transitions. The model is much simpler than the original one. The two transitions are depicted with different shapes to indicate the use of hierarchical modeling. In the representation of a hierarchical transition, their input and output places must be stated. They are not part of the transition, but rather serve as connectors between the sub-model and the top-level model. The places of the top-level model must be associated with the respective input and output places of the transition sub-model. For the transition "*T2*", the place "*P0*" is connected with the input place "*P_IN*" and the place "*P1*" is connected with the output place "*P_OUT*".

In the MSL, a hierarchical transition can be used with the syntax shown in Listing 2. First, the script describes the structure of the transition as a normal GSPN model. The only difference is that some places will be declared with the keywords *in* and *out*, instead of the *place* keyword. Those keywords describe the input and output places of the transition, respectively. On the top level model, the script instantiates the transition by using the *substitutionTransition* keyword, followed by the transition name, and the transition parameters enclosed in parentheses. It has three parameters: *subnet*, that specifies the name of the model corresponding to the transition; *inputs*, that specifies the connections between the places of the top level model and the input places from transition; and *outputs*, that connects the outputs places.

```
SPN Transition{
```

```
    in P5in;
    out P6out;
    ...
}

SPN PetriNet{
    place P5;
    place P6;
    ...
    substitutionTransition T1(
        subnet = Transition,
        inputs = ( P5in = P5 ),
        outputs = ( P6out = P6 )
    );
    substitutionTransition T2(
        subnet = Transition,
        inputs = ( P5in = P6 ),
        outputs = ( P6out = P7 )
    );
    ...
}
```

Listing 2: Petri net with a hierarchical transition

3.2 Variable Net Structure

Sometimes Petri nets have repeated elements in their structure, e.g. presented in the Token Ring model example. The majority of existing tools cannot treat the number of repeating elements as a parameter, since the net structure is fixed, and the variable parts of the model are the transition's parameters (delay, priority, and weight) and the net's marking. To bestow greater flexibility when developing models with repeating elements, MLS provides a loop construct that can be used inside the model declaration. Listing 3 depicts the Token Ring example with a variable number of nodes to illustrate this feature. We enclose the declaration of the repeated places and transitions inside a for-loop, and we use a special operator to define a variable identifier. The hash (#) symbol is used to declare an identifier with a suffix that is given by a numeric expression, enabling us to dynamically create the places *place*_1, *place*_1, *place*_3 places, and so on. The dollar ($) operator is used to retrieve the contents of a variable, similar to the Perl, PHP or Bash languages. When we omit this operator, the value is not retrieved until the model is solved. In short: we use the $ operator in temporary variables (like the *i* variable in the script), and we do not use this operator when we want to declare a model

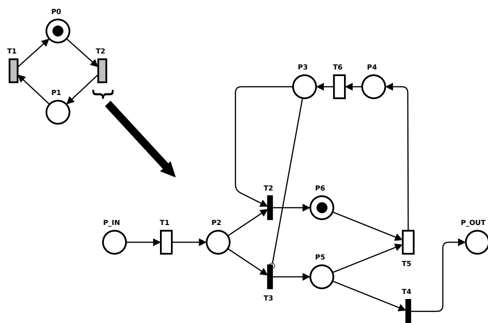

Figure 3: GSPN model for a Token Ring network (with hierarchical transitions)

parameter (the *k* and the *n* variables, for instance). The language also permits using conditional structures inside the loop, so we are able to properly create circular structures like a Token Ring net.

Figure 4 shows the generated net when assigning the value 10 to the parameter *count*. It can be observed that when combined with hierarchical transitions, those control statements can generate complex net structures with a small number of lines of code.

```
SPN TokenRingVariable{
  for i in range( 1, count ) {
    place p_#($i);
    if($i == $count){
      next = 1;
    }else{
      next = $i + 1;
    }
    substitutionTransition T#($i)(
        subnet = Transition,
        inputs = ( P5in = p_#($i) ),
        outputs = ( P6out = p_#($next) )
    );
  }
}
```

Listing 3: Petri net with a repeating structure

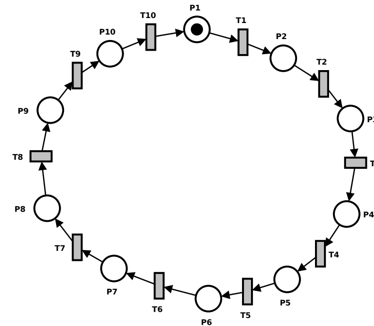

Figure 4: Hierarchical GSPN model for a Token Ring network with 10 nodes

3.3 Support for phase-type distributions

A system modeled through GSPNs can be evaluated by numerical analysis or simulation. Numerical analysis provides more accurate results, but the drawback is that the delay associated with transitions must be exponentially distributed. If a user collects real-world data in order to parameterize the model and the histogram indicates that the data is not even close to an exponential distribution, then the assumption of an exponentially distributed delay makes the model results deviate from real system results.

Consider, for instance, that the service time for the M/M/1/k model depicted in Figure 1 is derived from a real system and the collected data follows an Erlang distribution. If the user assumes an exponential service time for these data, the results may not be consistent with the real system. One solution to overcome this problem is to use *phase-type* distributions [8]. A phase-type distribution can be expressed as a composition of exponential distributions, enabling the approximation of an empirical distribution.

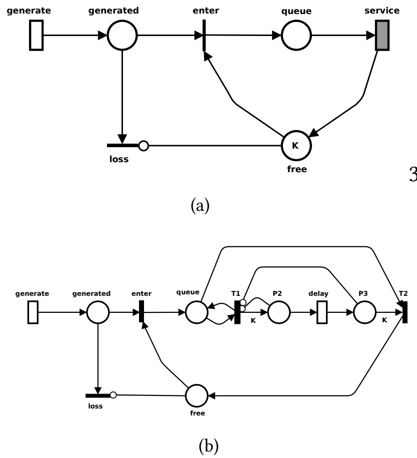

Figure 5: GSPN model with an Erlang transition

One trade-off found in using phase-type distributions to approximate a transition firing delay is that the model can become more complex and difficult to understand. The Mercury scripting language defines a special syntax for simplifying the use of phase-type distributions in GSPN models. Listing 4 shows the code for an Erlang transition with 6 stages, where the mean delay is 10 for each stage.

```
timedTransition T0(
    inputs = [A],
    outputs = [B],
    delay = (type="Erlang",
       parameters = (meanDelay=10,shape=6)
    )
);
```

Listing 4: SPN with phase-type delay

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 into the actual Petri net by using the hierarchical transitions of the engine. Therefore, the structure of the phase-type transition is not mixed with the major Petri net structure, so the model will be simpler and easier to understand. In the graphical representation of the Mercury interface, exponential transitions are displayed with a white background and non-exponential transitions are displayed with a shaded background.

Figure 5(a) depicts the GSPN model with a transition that follows an Erlang distribution. This transition is called "*service*" and is represented with a shaded background. Without this feature, our model would be depicted as in Figure 5(b), with the inclusion of additional places, arcs, and transitions (displayed inside the dotted box) to represent an Erlang-distributed delay, and hence replacing the shaded transition of the previous model. Therefore, this functionality, available in MSL, mitigates the modeling complexity of GSPNs.

3.4 Event-Based Programming for Simulation

The constructs offered by the GSPN formalism are sometimes not enough to model the aimed behavior of large or complex systems. One solution is to use simulation packages to model the system. However, using such packages is difficult to learn and complicated. Moreover, if the modeler already possesses a GSPN model and wants to extend it to implement a particular aspect, the model must be rewritten from scratch. For enhancing the simulation capabilities of GSPNs in the Mercury language, we introduced an event-based programming mechanism. This extension enables us to specify functions to be called as callbacks when a transition fires. Besides the natural behavior of transitions (move tokens through places), this mechanisms allows that transitions to implement more complex actions, such as:

- *Modifying runtime variables.* Inside a callback function we can modify runtime variables that can be used to compute a metric, or to enable/disable transitions using guard expressions;
- *Modifying model parameters.* The value of firing delays, weights, transitions, and variables used in guard expressions can be changed at simulation runtime;
- *Modify the net's marking.* It's possible to modify the whole net's marking instantaneously, in contrast with local changes provoked by transitions. As we will see in our example, it is a handy tool for implementing a checkpoint mechanism;
- *Modify the net structure.* For more complex models, it is even possible to create/delete places, transitions, arcs at runtime, creating a mutating model structure.

The following example illustrates this mechanism. In this example, we are interested in modeling an application with a checkpoint-restart mechanism. The model periodically saves the current marking as the application's state, to represent a checkpoint event. After the occurrence of a failure, the last stored marking should be restored. For binding an event handler to a transition, we use the *onFiring* keyword (Listing 5). The implementation of the callback function is written in Java, and it is shown in Listing 6. To declare an event handler in Java, it is necessary to create a class that extends the *AbstractFunction* class and overrides the *execute* method. The *@Function* annotation is used to register the class as a function in the scripting runtime.

```
place running( tokens= 1 );
place checkpointing;
timedTransition start_checkpoint(
  inputs = [running],
  outputs = [checkpointing],
  delay = ( type = "Deterministic", parameters = ( Value
     = checkpointInterval ) ),
  onFiring = onCheckpointStart()
);
```

Listing 5: Registering a callback to a transition

```
@Function( name = "onCheckpointStart")
class OnCheckpointing  extends AbstractFunction{
    @Override
    public void execute() {
        ApplicationState state = .getState();
        Marking marking = getTokenGame().getMarking();
        state.setMarking(marking);
    }
```

```
}
```

Listing 6: Callback implementation for the checkpoint activity

4 RELATED WORK

The TimeNET software package [12] is a specialized graphical tool to model Petri nets. It provides support for two classes of Petri nets: Colored Petri nets, and Extended Deterministic and Stochastic Petri nets (EDSPN). Colored Petri nets can be solved only by simulation, and EDSPN models can be solved either by simulation or numerical methods. CPN Tools [13] is a graphical modeling framework focused on Colored Petri nets that provides support for space-state based analysis methods and discrete-event simulation. It allows the use of the Standard ML programming language for creating user-defined functions that can be used by the models. The PIPE tool [11] implements more methods for structural analysis (invariants, traps, siphons, etc.) of GSPNs than the TimeNET and Mercury tools, but provides less support for obtaining metrics from the stationary and transient analysis. The GreatSPN tool [14] is another tool specialized on GSPN and CPN models that implements methods for structural analysis, Markov chain generation, and discrete event simulation. Additionally, it allows the composition of two or more interacting models.

The SHARPE tool [7] is a well established software for performance and dependability modeling that looks back on 28 years of development. It provides support for many formalisms, namely: fault trees, reliability block diagrams, acyclic series-parallel graphs, acyclic and cyclic Markov and semi-Markov models, Generalized Stochastic Petri nets, and product-form queueing networks. The SHARPE software package provides a graphical interface and a scripting language for creating and evaluating the models. The Möbius tool [15] is a modeling framework enabled for multi formalism hierarchical models. Its extensible architecture allows the incorporation of new formalisms that can be solved by Markov chain generation or by discrete event simulation. The PRISM [16] tool provides a modeling language for model verification of discrete and continuous time Markov chains.

Comparing the Mercury script language (MSL) with the SHARPE language, MSL is more verbose but has a more intuitive syntax. While both languages have similar features (e.g.: loops, symbolic evaluation, support to hierarchical models, etc.), both also have unique features. The SHARPE tool provides more formalisms and solution algorithms and it is especially useful when dealing with Markov chains, reliability graphs, and fault trees. On the other hand, the Mercury tool is particularly useful when one must deal with GSPNs, providing a number of features that are not found in other GSPN/CPN tools such as phase-type distributions, declarative loops, and event-based programming. The TimeNet and CPN tools offer support for hierarchical Petri net models, but only for Colored Petri nets. While it can be argued that CPN is a superclass of GSPN, both tools do not provide support for using hierarchical transitions on models that can be solved by Markov chain generation. The MSL language also offers a larger number of statistical distributions than CPN Tools, TimeNET, and PIPE tools.

5 CONCLUSIONS

This paper presented MSL, the scripting language of the Mercury modeling tool. Besides providing the basic functionality available on the Mercury GUI, the scripting language also increases the modeling power of the graphical tool by: (i) providing better support for hierarchical modeling; (ii) allowing parametric execution of models; (iii) using advanced techniques for Petri net modeling such as substitution transitions, phase-type delay for timed transitions, nets with variable structure, and event-based programming for simulations.

The scripting language is a very useful tool when dealing with a complex project that involves many different models, possibly arranged in a hierarchical way. Mercury scripts may be used to automate the entire workflow of a performance or dependability study. In this context, we define a workflow as the ordered set of activities that are performed in a study: conceiving the models, evaluating the models, storing the metrics, processing the metrics, performing sensitivity analysis, producing charts and reports, etc. We could also use other tools (e.g., shell scripts) to conduct the workflow, and these external tools would run the Mercury scripts.

Currently, we are working on a new powerful graphical interface that is totally integrated with the scripting environment for improving the features utilizations.

REFERENCES

[1] D. A. Menasce, V. A. Almeida, L. W. Dowdy, and L. Dowdy, *Performance by design: computer capacity planning by example.* Prentice Hall Professional, 2004.

[2] 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.

[3] B. Silva, G. Callou, E. Tavares, P. Maciel, J. Figueiredo, E. Sousa, C. Araujo, F. Magnani, and F. Neves, "Astro: An integrated environment for dependability and sustainability evaluation," *Sustainable Computing: Informatics and Systems*, vol. 3, no. 1, pp. 1–17, 2013.

[4] A. A. Markov, "Extension of the law of large numbers to dependent quantities," *Izv. Fiz.-Matem. Obsch. Kazan Univ.(2nd Ser)*, vol. 15, pp. 135–156, 1906.

[5] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with generalized stochastic Petri nets.* John Wiley & Sons, Inc., 1994.

[6] C. E. Ebeling, *An introduction to reliability and maintainability engineering.* Tata McGraw-Hill Education, 2004.

[7] K. S. Trivedi and R. Sahner, "SHARPE at the age of twenty two," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 52–57, 2009.

[8] L. Breuer and D. Baum, "Phase-type distributions," *An Introduction to Queueing Theory and Matrix-Analytic Methods*, pp. 169–184, 2005.

[9] 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.

[10] P. Huber, K. Jensen, and R. M. Shapiro, "Hierarchies in coloured Petri nets," in *Advances in Petri Nets 1990.* Springer, 1991, pp. 313–341.

[11] P. Bonet, C. M. Lladó, R. Puijaner, and W. J. Knottenbelt, "PIPE v2. 5: A Petri net tool for performance modelling," in *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*, 2007.

[12] R. German, C. Kelling, A. Zimmermann, and G. Hommel, "TimeNET: a toolkit for evaluating non-markovian stochastic petri nets," *Performance Evaluation*, vol. 24, no. 1, pp. 69–87, 1995.

[13] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen, "CPN tools for editing, simulating, and analysing coloured petri nets," in *Applications and Theory of Petri Nets 2003.* Springer, 2003, pp. 450–462.

[14] S. Baarir, M. Beccuti, D. Cerotti, M. De Pierro, S. Donatelli, and G. Franceschinis, "The GreatSPN tool: recent enhancements," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 4–9, 2009.

[15] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster, "The Mobius modeling tool," in *Petri Nets and Performance Models, 2001. Proceedings. 9th International Workshop on.* IEEE, 2001, pp. 241–250.

[16] M. Kwiatkowska, G. Norman, and D. Parker, "Prism: Probabilistic symbolic model checker," *Computer Performance Evaluation/TOOLS*, vol. 2324, pp. 200–204, 2002.