Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

FROM PETRI NETS TO COLORED PETRI NETS: A TUTORIAL INTRODUCTION TO NETS BASED FORMALISM FOR MODELING AND SIMULATION

Vijay Gehlot

Department of Computing Sciences and
Center of Excellence in Enterprise Technology (CEET)
Villanova University
800 E. Lancaster Avenue
Villanova, PA 19085, USA

ABSTRACT

Petri Net, a widely studied mathematical formalism, is a graphical notation for modeling systems. Petri Nets provide the foundation for modeling concurrency, communication, synchronization, and resource sharing constraints that are inherent to many systems. However, Petri Nets do not scale well when it comes to modeling and simulating large systems. Colored Petri Nets (CPNs) extend Petri Nets with a high level programming language, making them more suitable for modeling large systems. The CPN language allows the creation of models as a set of modules in a hierarchical manner and permits both timed and untimed models. Untimed models are used to validate the logical correctness of a system, whereas timed models are used to evaluate performance. This tutorial introduces the reader to the vocabulary and constructs of both Petri Nets and CPNs and illustrates the use of CPN Tools in creating and simulating models by means of familiar simple examples.

1 INTRODUCTION

Petri Net is a widely studied mathematical formalism (Reisig 1985). From a modeling perspective, Petri Nets provide a graphical modeling language for describing systems that are distributed and concurrent with synchronous as well as asynchronous communication mechanisms and resource sharing constraints. Historically, Petri Nets originated in the PhD dissertation work of C. A. Petri (Petri 1962) aptly titled *Kommunikation mit Automaten* (Communication with Automata). Over the decades, researchers have focused on both the theoretical aspects as well as practical applications of Petri nets. In particular, many software tools have been developed that allow the creation and analysis of Petri Nets without delving into the theoretical details.

The basic syntax and semantics of Petri Nets is extremely simple. Thus, instead of resorting to formal definitions, this tutorial paper introduces the key concepts of Petri Nets by means of illustrative examples. We then take a running example with increasing complexity to explain how Petri Nets and its extensions can be used in the modeling and simulation of systems. Readers interested in a full and formal treatment of Petri Nets may refer to (Peterson 1981) and (Reisig 1985). A comprehensive and detailed survey of early works on Petri Nets appears in (Murata 1989). A good online resource is Petri Nets World.

A Petri Net consists of *places* (depicted as circles or ovals), *transitions* (depicted as rectangles or bars), and *arcs* (depicted as arrows) that connect a place to a transition or a transition to a place (Reisig 2013). Thus, Petri Nets are also referred to as *Place/Transition Nets*. The only syntactic restriction is that two places cannot be directly connected without an intervening transition and two transitions cannot be connected directly without an intervening place. A net can have any number of places, transitions, and arcs. It is not required that the entire net be one single connected graph. Figure 1 shows a very basic

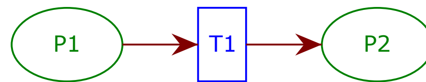

Figure 1: A basic Petri Net with two places and one transition.

Figure 2: Petri Net before and after the firing of transition T1.

Petri Net consisting of two places (P1 and P2) and one transition (T1). The interpretation of these net elements depends on the system being modeled. For example, we can interpret P1 as "Printer Available", T1 as "Print Job Arrives", and P2 as "Printer Busy". However, the same net may be given a different interpretation with P1 as "Healthy", T1 as "Bug Bites", and P2 as "Sick". Therefore, a net gives us an abstraction mechanism to see similarities among seemingly different systems. Thus, from a design and analysis perspective, it is possible to transfer interesting properties and results from one system to another. In fact, several sub-classes and structural properties of Petri Nets have been studied that provide insights into system behaviors (Reisig 1985). For example, if we restrict the net structure so that every transition has exactly one incoming arc and exactly one outgoing arc, then the resulting Petri Net is equivalent to a finite state machine.

Places may have *tokens*, which are traditionally depicted by dots residing in a place. A place with a token is termed *marked* and describes a local state of the system. Distribution of tokens across places in a given net is called a *marking* and describes the collective global state of a system. The dynamics (semantics) of a Petri Net is defined by the *firing rule* where the *firing* of a transition removes tokens from its input place and adds tokens to its output place. If more than one token is to be removed or added, the outgoing and/or incoming arcs may have an optional inscription denoting the number of tokens to be removed and/or added. A transition is termed *enabled* if all of its input places have a number of tokens that is greater than or equal to the associated arc inscriptions. Note that per the semantics described above, only an enabled transition may fire. The firing of a transition is an abstraction of the occurrence of an event and the movement of tokens describes state changes. This small basic vocabulary and simple semantics render Petri Nets very flexible in terms of application domains for modeling systems of varied nature. Note that there is no explicit notion of time in the original Petri Nets as described above. However, there do exist extensions of Petri Nets with an explicit notion of time (Ramchandani 1974; Wang 1998; Popova-Zeugmann 2013). One extension we cover later in this paper also incorporates time.

Figure 2 shows a Petri Net with 5 tokens in place P1. Instead of dots, the software tool (described later) we are using depicts tokens with a single dot and an associated integer count. When the transition T1 fires, it removes 2 tokens from place P1 and deposits 1 token in place P2, as determined by the associated outgoing and incoming arc inscriptions, respectively. Under a different interpretation, the given Petri Net captures a basic step (or requirement) of a vending machine whereby one needs to supply 2 quarters for it to dispense 1 candy bar.

The basic execution semantics of Petri Nets in terms of the firing rule above, gives rise to several interesting net configuration interpretations useful in modeling system behaviors. Figure 3 depicts some net configurations useful for expressing various communication and coordination activities of a system. Considering a healthcare application domain, the *Sequential* configuration is useful in capturing, say, the fact that a patient must see a primary care physician before consulting a specialist. The *Concurrent* configuration captures the independence of events. For example, taking blood pressure is totally independent (thus can be concurrent) of the arrival of an ambulance. The *Choice* configuration is useful in capturing, say, opting

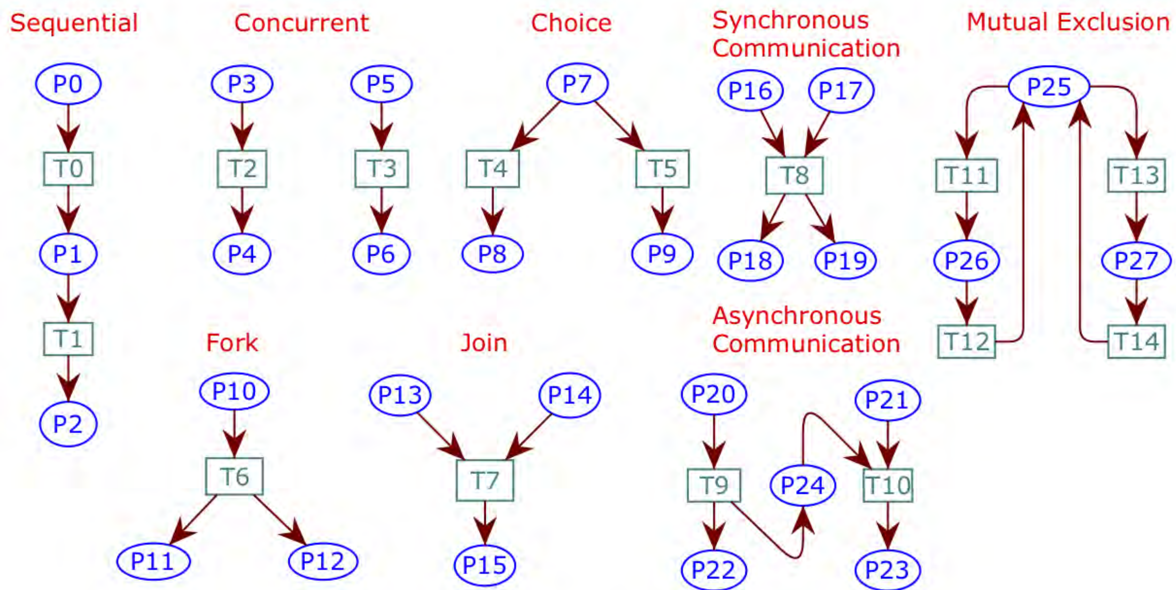

Figure 3: Useful Petri Net configurations for systems modeling.

for treatment A vs treatment B. The *Join* configuration as well as *Synchronous Communication* depicted in the figure are useful in modeling coordination where, say, the clinical team must come to a unanimous decision before sending the patient to the next stage of care. The *Asynchronous Communication* easily captures the fact of a test sample being delivered to a lab and then the lab processing it asynchronously. Finally, the depicted *Mutual Exclusion* is useful in expressing resource sharing constraints such as a single infusion pump which cannot be hooked to two different patients at the same time.

The rest of this paper is organized as follows. In Section 2 we give details of constructing a model of a simple print job system. Section 3 gives an overview of Colored Petri Nets (CPNs). Following this, in Section 4, we present a CPN version of the model which we extend to a timed version in Section 5. We present a hierarchical version of the model in Section 6. We present an overview of the CPN Tools software in Section 7. Finally, we present our conclusions in Section 8.

2 FIRST EXAMPLE

We start with a simple system consisting of print jobs and printers. User print jobs arrive at printers. If a printer is available, then a job is accepted for printing. During this period, the associated printer is unavailable. Once the printing is done, the printer becomes available for another print job. Figure 4 shows a Petri Net model of such a system. On the left is the net with initial marking showing three available printers (green circle with 3) in the place *Printers* and the place *Jobs* showing five jobs (green circle with 5) queued for printing. A simulation of the model consists of a sequence of transition firings. By simulating a model, it is possible to investigate different scenarios and identify desired and undesired behaviors of the system. Note that multiple transitions may be enabled in a given state. The underlying simulation engine for the Petri Nets may fire these non-deterministically. Extensions of Petri Nets do allow probabilistic as well as priority-based selection of transitions to fire (Haas 2002; David and Alla 2004).

The initial marking is created via the inscriptions `3`()` and `5`()` shown next to the two places. Although we have included only a fixed number of print jobs here, it is very easy to incorporate continuous arrival of jobs in the net shown, which we will do in a refinement of this example. The transition *StartJob* is highlighted in green indicating that it is enabled in the current marking and can fire. The net on the right depicts a possible marking reached after a sequence of firings. It shows two possible actions, that is,

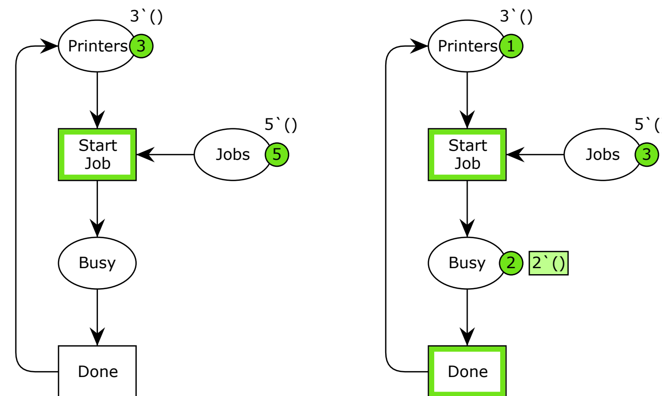

Figure 4: Petri Net model of a print job system—initial marking (left) and an intermediate marking during simulation/execution (right).

step	transition
1	Start_Job
2	Start_Job
3	Start_Job
4	Done
5	Start_Job
6	Done

data	counter	step
3	1	0
2	2	1
1	3	2
0	4	3
1	5	4
0	6	5

Figure 5: Sample simulation report (left) and token count data log for place Printers (right).

a new job can be started as well as a currently active jobs can be finished. Thus the two transitions are concurrent in the shown marking. For analysis purposes, the simulation software used here generates a simulation report, which gives details of various transition firings, as well as permits a highly customizable automatic data logging facility. For example, if we were interested in knowing whether the modeled system can reach a state where none of the printers is available, we can do so easily by logging the number of tokens in place *Printers*. A sample simulation report and log of tokens for this example net, as generated by the software, is shown in Figure 5.

Although with the small vocabulary of Petri Nets we have been able to capture the structure and behavior of our system of interest, the language is limited in terms of level of details one may want to include in a model. For example, if we wanted to distinguish an HP printer from an Epson printer or a Canon printer, we cannot do so easily. Also, we may want to keep track of jobs via username or job ids and these details are not directly representable in the language of Petri Nets we have described so far. Fortunately, extensions of Petri Nets exist that make it convenient to capture the variety of details and constraints associated with a realistic system. One such extension is Colored Petri Nets (CPNs), which we describe next.

3 COLORED PETRI NETS

Colored Petri Nets (CPNs) extend the vocabulary of ordinary Petri Nets and add features that make them suitable for modeling large systems (Jensen and Kristensen 2009). CPNs combine the graphical components of ordinary Petri Nets with the strengths of a high-level programming language called CPN ML which is based on the functional language SML (Ullman 1998). Thus, in the CPN extension, Petri Nets provide the primitives for process interaction and the foundation for modeling concurrency, communication, and

synchronization, while the programming language provides the needed primitives for the definition of data types and the manipulations of data values.

The CPN language permits the models to be represented as a set of modules, allowing complex nets (and systems) to be represented in a hierarchical manner. CPNs allow for the creation of both timed and untimed models. Simulations of untimed models are usually used to validate the logical correctness of a system, while simulations of timed models are used to evaluate the performance of a system. Time plays an important role in the performance analysis of concurrent systems. If we were to think of Petri Nets as an assembly language, then CPNs would correspond to a high-level language. In fact, CPNs and many other extensions are referred to as *high-level* nets (Jensen and Rozenberg 1991).

According to Jensen and Kristensen (2015), "The goal of the CPN modeling language has been to develop a formally founded modeling language for concurrent systems that would make it possible to formally analyze and validate concurrent systems and that, from a modeling perspective, scale to industrial systems. A main motivation behind the research into CPNs (and many other formal modeling languages) was that the engineering of correct concurrent systems is a challenge due to their complex behavior that could result in subtle bugs if not designed with care. As concurrent systems become still more pervasive and critical to society, formal techniques for concurrent systems were (and still are) a highly relevant technology to support engineering of reliable concurrent systems."

It should be clear that the tokens in the Petri Nets described above do not carry any data value. In programming terms, such tokens can be thought of as objects of type `void` in languages like C and Java. The CPN extension of Petri Nets imparts the ability to define tokens of a variety of datatypes by adopting CPN ML as its inscription language. In fact, the qualification colored in Colored Petri Nets is synonymous with types in programming languages. In Petri Nets, tokens are conventionally depicted as black dots, and therefore in the CPN extension tokens may have many different colors (types) apart from just black. Thus in a CPN model, tokens can be coded as data values of a rich set of types (called *color sets*), and arc inscriptions can be computed expressions and not just constants.

CPN ML includes predefined data types such as integers, booleans, reals, strings, and the equivalent of `void` called `unit`. Note that there is only one value of type `unit` and is denoted by `()`, which can be thought of as a concrete representation of black tokens in Petri Nets. In fact, the example Petri Nets described above were created using the CPN software called *CPN Tools* which defaults to type `unit` for all places and tokens.

In addition to predefined data types, CPN ML provides facilities for user defined type construction including enumeration types, subrange types, product types, record types, union types, and list types.

As with Petri Nets, we introduce key concepts and constructs of CPNs by means of extensions of our running print job example. Readers interested in details, including formal definitions and theoretical foundations, may refer to (Jensen 1981), (Jensen 1994), and (Jensen and Kristensen 2009). A condensed introduction to CPNs is available in (Jensen et al. 2007).

4 SECOND EXAMPLE

Let us extend our print job example with the ability to distinguish different types of printers. Although all modeling should be at some level of abstraction, the model should include enough details of the underlying reality so as to be able to answers questions of interest. For example, if we are interested in an accurate estimate of the turnaround time, we would want to be able to distinguish various printers not just by name but also their print speed, capacity, etc. Using the rich type declaration (`colset` or color set declaration in CPN parlance) facility of CPNs, it is possible to define a printer as a token with all the attributes of interest.

As an example, let us say we want to be able to distinguish three printers: HP, Epson, and Canon. Furthermore, we want to be able to assign and track each job via a unique job identifier. Thus, a job request would consist of a job id together with a printer name. Also, an active job is a pair consisting of a printer

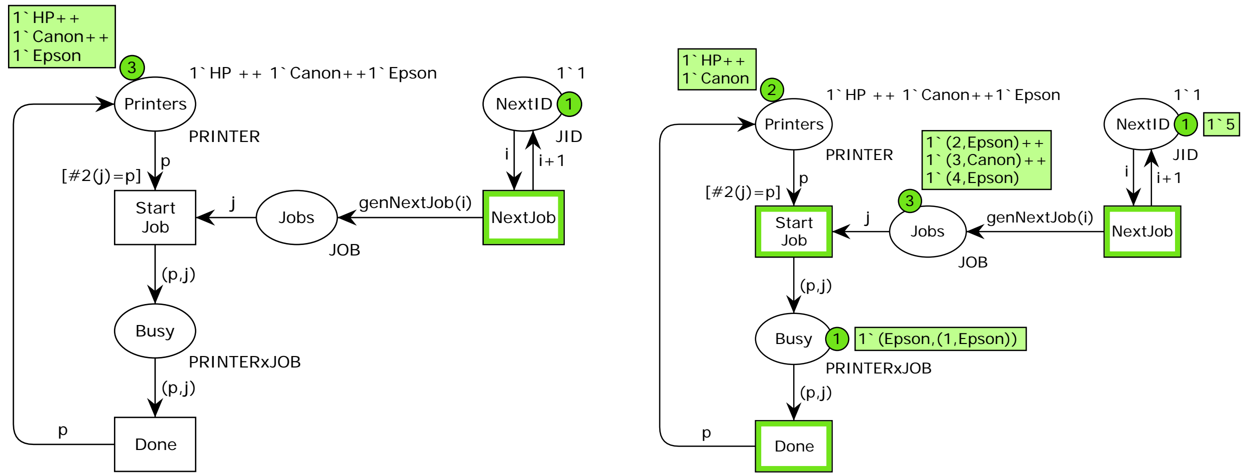

Figure 6: A CPN model of the print job system with initial marking including a generator for jobs (left). An intermediate marking of the net during simulation (right).

and a job. The color set declarations shown below achieve this refinement:

```
colset PRINTER = with HP | Canon | Epson;
colset JID = INT;
colset JOB = product JID * PRINTER;
colset PRINTERxJOB = product PRINTER * JOB;
```

With these declarations, a job with id 5 and printer Epson would be denoted by `(5, Epson)`. Of course, a further refinement could include the number of pages in addition to the name of the printer and is left as an exercise for the reader.

Since tokens reside in places, every place in a CPN is required to declare the associated color set via an inscription. A place may contain a multiset (or bag) of tokens over its associated color set. For example, the CPN ML expression `2`HP++1`Epson` defines a multiset with 2 HP printers and 1 Epson printer.

Figure 6 shows a CPN model of the print job system under discussion. The multiset inscription on the top right of the place *Printers* defines the initial marking for it. The green box and circle depict the details and count of current tokens residing in the place during simulation. The inscription `PRINTER` on the lower right side of this place declares the color set for the place. As a generalization from Petri Nets, arc inscriptions in CPNs may be expressions which are evaluated using the current markings of input places of a transition. A transition is enabled in CPNs if the variables in the arc expressions of all input places to the transition can be bound consistently from the current markings of those places, and the resulting arc expression evaluates to a multiset which is a submultiset of the current marking of the corresponding input place. Firing of an enabled transition with a given binding removes from each input place the multiset of tokens to which the corresponding input arc expression evaluates. Similarly, it adds to each output place the multiset of tokens to which the corresponding output arc expression evaluates. All variables in arc expressions must be declared using a `var` declarations. The net in Figure 6 has four variables and the corresponding declarations are given below:

```
var p: PRINTER;
var j: JOB;
var i: JID;
```

Table 1: Auto-generated report for token count in place Printers over 50 steps of simulation. The table shows count as 51 since it includes the initial marking which is at step 0 of simulation.

Name	Count	Sum	Avrg	Min	Max
AvailablePrinters	51	70	1.372549	0	3

Transitions in CPNs may have an optional *guard*, which is a list of boolean conditions. These provide additional control over the firing of the associated transitions. Let us focus on transition *StartJob*. It has two input places and one output place. The arc inscription of place Printers is simply the variable `p`. Thus, `p` can be bound to any one of `HP` or `Canon` or `Epson`. The variable `j` on the arc from place *Jobs* will be bound to a token of type `JOB` and would be a pair consisting of an id and a printer. The guard `#2(j)=p` would evaluate to true only when the second component of the pair, namely the printer, is the same as the printer associated with variable `p`. Thus, the guard guarantees that if the print job requests, say, HP printer then only the HP printer will be selected. Note that if in the current marking, no HP printer is available, then the current job with request for HP printer will not get started since the transition will not get enabled. However, if there is another job for a different printer, then that job may be started and the job for HP printer would wait until an HP printer becomes available.

Let us consider the transition *NextJob* in Figure 6. Its input place `NextID` serves as a generator for next job id in sequence starting from the initial marking where it contains integer value 1 as described by the initial marking `1‘1`. Its associated color set is `JID`, which is an integer per color set declarations above. Each time the transition fires, it removes an integer (value bound to variable `i`) and puts back the next integer in sequence (computed value of expression `i+1`) on the outgoing arc. Additionally, each firing of this transition puts the result of function call `genNextJob(i)` into place *Jobs*. The user-defined function `genNextJob` generates a random printer request for a given job id. It makes use of one of the built-in facilities of CPNs where, for any enumeration type `T`, the function `T.ran()` returns a random value of that type, and is declared as follows:

```
fun genNextJob(i) =
        (i, PRINTER.ran());
```

To complete the description of the model, once a job is started, the associated token of type `PRINTERxJOBS`, which is a pair `(p, j)` consisting of a printer and a job will get added to place *Busy*. When the transition *Done* fires, token associated with the busy printer will get returned to the pool of available printers in place *Printers*.

Figure 6 shows the marking of the net after some steps of simulation. In particular, it shows three jobs waiting in place *Jobs*. Focusing on transition *StartJob*, the variable `p` has two possible bindings given that the current marking of the place *Printers* is `1‘HP++1‘Canon`. On the other hand, the variable `j` has three possible markings based on the current marking of the place *Jobs*. However, with the binding of `j` to `(2,Epson)` the guard on the transition cannot evaluate to true and enable the transition for either of the bindings of `p`. In other words, based on our description of enabling of transitions in CPNs above, the job with id `2` cannot be started since there is no printer `Epson` available in place Printers. However, either of the other two jobs may be started at the next step of simulation. Since there are no additional constraints, the underlying simulator will pick one at random. Note that in absence of the guard, there are six possible bindings that will enable the transition *StartJob*.

Similar to what we did in the first example, we used the built-in monitoring and data logging facility of the CPN Tools software to keep track of the number of tokens in place *Printers*. The auto generated report is shown in Table 1. The weakness of the current model is our ability to do some performance analysis and answer questions like how long a job waits or what is the utilization of printers, etc. To do this, we need to incorporate a notion of time into our modeling language. Fortunately, CPNs do include a built-in notion of time. The basic idea behind the time extension is to allow each token to carry a time stamp in

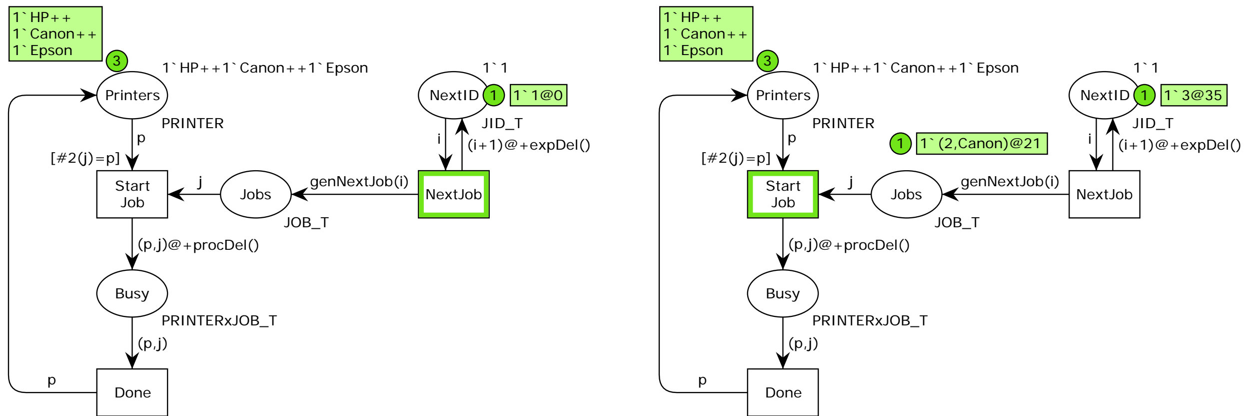

Figure 7: A timed CPN model of the print job system (left). An intermediate marking of the model during simulation (right).

addition to the data value. A global clock (counter) is used to advance time. Semantically, the time stamp specifies the global clock time at which the token becomes available to be consumed by a transition firing any time after that. Similar to many discrete event systems, the CPN Tools software does not advance time by one clock tick. Instead, it advances the time by the smallest step that makes at least one transition enabled. We discuss the details of a timed version of our current CPN model in the next section.

5 THIRD EXAMPLE

We now describe how to incorporate timing information to a CPN model. As mentioned before, this enrichment is crucial if we want to evaluate the efficiency of a system and its operations. Timed models are also useful in validating real-time systems, where the correctness of the system relies on the proper timing and deadlines of events and not just outputs.

As mentioned above, CPNs introduce the notion of time by associating a time stamp with a token. Thus, a token carries not only a value of its associated color set but a second value denoting its *time stamp*. This is achieved by declaring the color set of interest to be *timed*. In general, for any color set `C`, one can create a timed version by the declaration `C timed`. Assuming we have declared a timed version of the integer type, then a token belonging to this type would be written `1‘5@20`, where 5 is the value of the token and 20 is its time stamp. This token will not be available for transition enabling and firing until the global clock reads at least 20. Depending on the context, one may interpret this time stamp as the execution time of the activity associated with the affected transition. In CPNs, both timed and untimed tokens may coexist. In other words, not all token types need to be declared timed for the entire model to be timed. One implication of this property is that if both timed and untimed activities are enabled in a net, then the transitions associated with untimed activities will fire before the transitions associated with timed activities.

Figure 7 contains the timed extension of the model in Figure 6. In this model the places *NextID*, *Jobs*, and *Busy* have been declared to have timed version of their original color sets by adding the following `colset` declarations:

```
colset JID_T = JID timed;
colset JOB_T = JOB timed;
colset PRINTERxJOB_T = PRINTERxJOB timed;
```

```
1 0 NextJob @ (1:ExtendedPrintJobSystemTimed)
 - i = 1
2 0 Start_Job @ (1:ExtendedPrintJobSystemTimed)
 - j = (1,Epson)
 - p = Epson
3 12 Done @ (1:ExtendedPrintJobSystemTimed)
 - j = (1,Epson)
 - p = Epson
```

Figure 8: Sample entries from the generated simulation report for the timed CPN model of the print job system during a simulation run.

With these changes, a job in place *Jobs* will have a time stamp indicating its arrival at the printers. Similarly, an active job in place *Busy* will have a time stamp indicating the processing time. Thus, the transition *Done* will not fire until the global clock reads a time which is greater than or equal to the time stamp on an active job. In the model we have assumed that arrivals of print jobs are exponentially distributed with parameter 10.0. We achieve this as follows. When the transition *NextJob* fires, it removes a token representing the current id from the place *NextID* and puts back the timed token with next id. The timed token is generated via the inscription `(i+1)@+expDel()` on the outgoing arc. The function `expDel()` shown below returns an exponentially distributed integer value and is essentially a wrapper around the built-in `exponential` distribution function which returns a real number. Similarly, we associate a random processing delay for each job. This is achieved via the arc inscription `(p,j)@+procDel()` on the outgoing arc from transition *StartJob*. The function `procDel()` is a wrapper around the built-in uniform distribution function `discrete` as shown below.

```
val ARR_DELAY = 10.0;
fun expDel() =
     let
        val v = exponential(1.0/ARR_DELAY)
     in
        floor (v+0.5)
     end;
fun procDel() = discrete(5,25);
```

Figure 7 shows the marking of the timed net after some steps of simulation. The given marking shows that the job with id 2 and time stamp 21 is about to be started and the next job with id 3 will arrive at time 35. In a timed model, the simulation report provides the log of transition occurrences and the time at which a transition fires in addition to the step count. There is also an option to save bindings at each step. Figure 8 shows the first few lines of a simulation report generated for this model from a simulation run. For a row, the first number is the step count and the second number is the simulation clock time. Indented below each row are the bindings of the variables associated with the transition mentioned in that row.

As mentioned before, one may be interested in selected and/or refined data to be collected during simulation. This is achieved using the extensive monitoring facilities available in the CPN Tools software for data collection (Wells 2002; Lindstrøm and Wells 2002). A monitor is a mechanism in the CPN Tools that is used to observe, inspect, control, or even modify a simulation of a CPN. A variety of monitors can be defined for a given net. Monitors can inspect both the markings of places and the occurring binding elements during a simulation, and they can take appropriate actions based on the observations. Monitors can be used for each of the activities mentioned above. A monitor is associated with a relevant subnet consisting of places and/or transitions from a net. If a monitor is associated with one or more transitions,

```
Job 1 arrived at 0
Job 2 arrived at 7
Job 3 arrived at 8
Job (1,HP1) finished at 9
```

Figure 9: Sample monitoring log file generated during a simulation run.

then the monitor can check if some relevant condition is fulfilled after any one of the transitions occurs. If the relevant condition is fulfilled, the monitor can extract and record information from the subnet. If a monitor is associated with one or more places, then the monitor can examine the tokens on each of the places. The monitor can examine the markings of the places before a simulation starts, during a simulation, and when a simulation stops. If the monitor is associated with at least one transition, then the monitor can examine the tokens on the places only when one of the associated transitions occurs during a simulation. If the monitor is associated with zero transitions, then the monitor can examine the tokens on the places after every simulation step.

The code required for data extraction is automatically generated by CPN Tools when a monitor is applied to a subnet within the CPN Tools software by a user. In some cases, the user may have to override the default value for an auto-generated monitoring function. Typically, it would be the observation function that extracts information from the net. As an example, let us say we are interested in logging the arrival and completion of a print job. In this case we can associate a *write-in-file* monitor with the subnet consisting of transitions *NextJob* and *Done* of the CPN model in Figure 7. The auto-generated code with user-specified string to write out is shown below. The function `intTime()` returns the current simulation clock time as an integer.

```
fun obs (bindelem) =
let
  fun obsBindElem (ExtendedPrintJobSystemTimed'Done (1, {j,p})) =
        "Job " ^JOB.mkstr(j)^" finished at "^INT.mkstr(intTime())^"\n"
      | obsBindElem (ExtendedPrintJobSystemTimed'NextJob (1, {i})) =
        "Job " ^JID.mkstr(i)^" arrived at "^INT.mkstr(intTime())^"\n"
      | obsBindElem _ = ""
in
  obsBindElem bindelem
end
```

With this monitor in place, when we run a simulation, we will get a log file similar to one shown in Figure 9. Next we extend our print job example and introduce the hierarchical construction of CPN models.

6 FOURTH EXAMPLE

We now extend our example to have several printers connected over a network. Each printer will maintain its own queue of jobs and when a job is finished, a response is sent to the user. The creation of hierarchical nets is based on the idea that a transition can be replaced or substituted by a (sub) net that details the activities underlying the associated transition. Such transitions are called *substitution transitions* in the CPN parlance. Pictorially, a substitution transition is drawn with double rectangles. Thus at the top level, we can view this system as consisting of three components (substitution transitions), namely, *Printers*, *Network*, and *Print Jobs*, connected together as shown in Figure 10.

At the next level, we assume we are going to manage the three printers independently and similarly we will separate the requests for three printers into separate print jobs categories. Figure 11 shows the expansion of the substitution transitions *Printers* and *Print Jobs*. There is no restriction on how deep one may go in terms of expanding a substitution transition. From a modeling perspective, one can fill in the

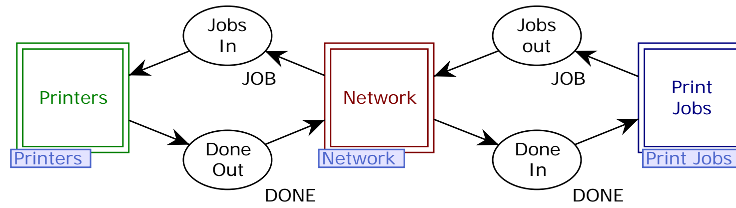

Figure 10: The top module (page) of the hierarchical CPN model of the print job system.

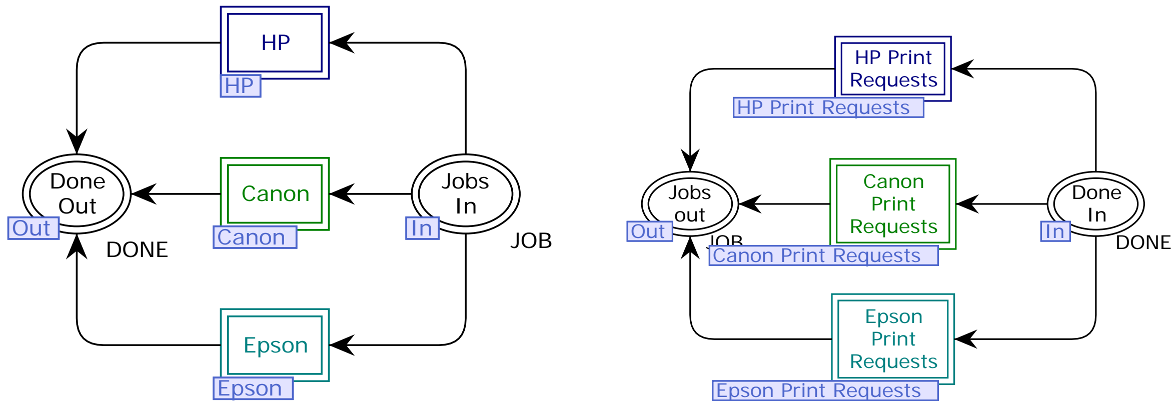

Figure 11: The subpages associated with substitution transitions Printers and Print Jobs.

details iteratively. In fact, it is not required that the details of all subpages be filled in for one to start simulating.

Each of the substitution transitions for the individual printers would expand to a net similar to the one in previous examples. However, one difference is that we will queue up jobs for each printer in an associated queue. In CPN, places are holders of a multiset and do not behave as a queue. However, it is easy to introduce a queue using the list type construction facility. We use the net shown in Figure 12 to illustrate how define and use a queue in CPNs. The declared type of place *Queue* is `INT_Q` which is defined to be `list INT`. For the enqueue operation, the transition *Enqueue* removes the current list `xs` and puts back this list appended with the new element, which is given by CPN ML expression `xs^^[x]`. Similarly, for the dequeue operation, the transition *Dequeue* removes the queue with head (front) `x` and tail `xs` and puts back the tail `xs`. The CPN ML expression `x::xs` is called a pattern and matches against a non-empty list with `x` bound to the head of the list and `xs` bound to the tail of the list.

Another change in the hierarchical model from the previous examples is that we save the current time as part of the token value itself. This allows us to directly compute the total wait time of a job upon

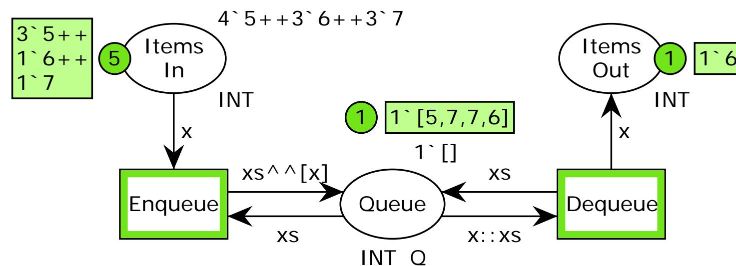

Figure 12: Illustration of queues in CPNs.

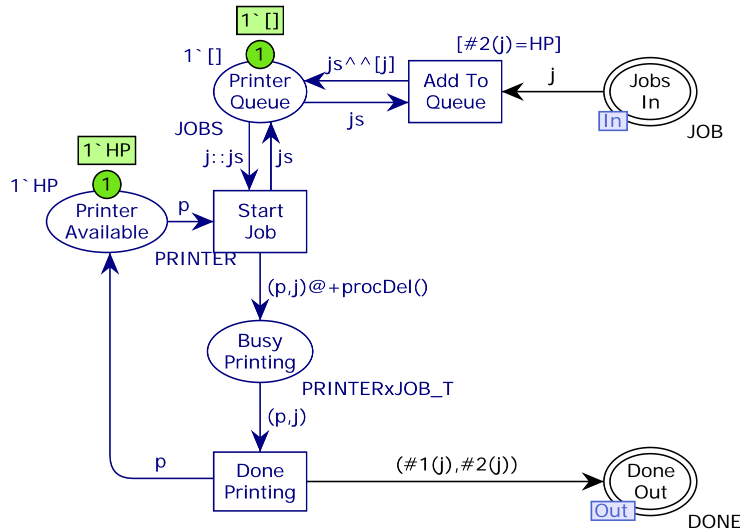

Figure 13: Subpage for HP Printer.

Table 2: Auto generated performance report when simulation replications are run.

Name	Avrg	90%	95%	99%	StD	Min	Max
Canon_Queue							
count_iid	64.000000	9.700014	12.630037	20.946935	10.173495	52	79
max_iid	5.400000	1.977135	2.574356	4.269574	2.073644	3	8
min_iid	0.000000	0.000000	0.000000	0.000000	0.000000	0	0
avrg_iid	2.300330	1.565843	2.038828	3.381399	1.642276	0.500000	4.516899
Canon_Utilization							
count_iid	58.400000	7.615748	9.916189	16.446014	7.987490	47	69
max_iid	1.000000	0.000000	0.000000	0.000000	0.000000	1	1
min_iid	0.000000	0.000000	0.000000	0.000000	0.000000	0	0
avrg_iid	0.892382	0.108807	0.141673	0.234965	0.114118	0.750520	1.000000

completion via a suitable monitor rather than having to do a post processing of a log file that records the two events separately. The relevant `colset` declarations for the hierarchical net are given below, and the subpage detailing the HP printer is given in Figure 13.

```
colset AT = INT;
colset JOB = product JID * PRINTER * AT;
colset JOBS = list JOB;
colset PRINTERxJOB = product PRINTER * JOB;
colset PRINTERxJOB_T = PRINTERxJOB timed;
colset DONE = product JID * PRINTER;
```

The subpages for remaining printers and their requests are similar. CPN does include a facility to create instances of a subpage as long as all instances use the same inscriptions. If all inscriptions are not the same, then one has to use a copy and not an instance.

To collect meaningful data, we associated monitors to compute various queue sizes and utilization of printers. As suggested in (White, Jr. and Ingalls 2009), we ran simulation replications using the built-in function `CPN'Replications.nreplications`. The auto-generated performance report is given in Table 2. For brevity, only data for Canon printer is shown.

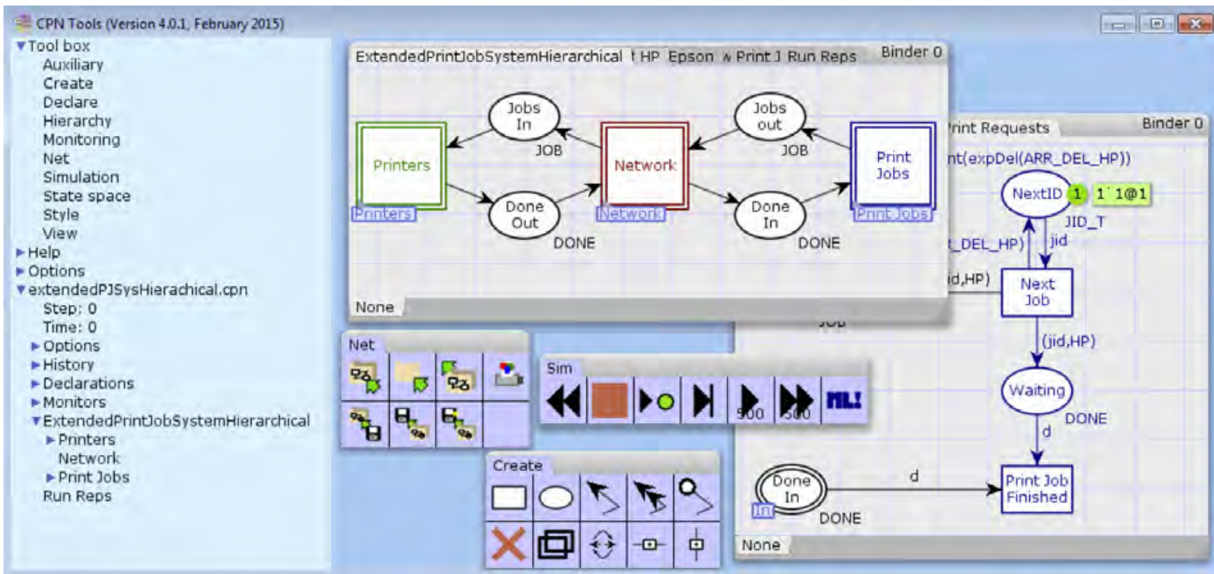

Figure 14: Screenshot of CPN Tools.

7 CPN TOOLS SOFTWARE

CPN Tools is a free software that supports editing and construction of CPN models, interactive and automatic simulation, monitoring facilities, external process communication, simulation based performance analysis, as well as state space-based model checking. It has an intuitive and flexible graphical user interface. Figure 14 shows a screenshot of CPN Tools. The software can be downloaded from the CPN Tools website.

Referring to the screenshot, the left column is called the *index* and contains a hierarchical list of objects. Expanding the *Tool box* gives a list of all the available. Any of these tools can be dragged to the section on the right, called the *workspace*. Dragging an item into the workspace creates a view of its contents. Expanding on the name of a net in the index displays all the information associated with that net, including all declarations and monitors. In the shown screenshot, the Simulation, Net, and Create tools have been dragged onto the workspace.

In CPN Tools a net is organized into binders in the workspace. Each binder may contain any number of tabbed pages. Each page contains a single (sub)net. Figure 14 shows that the overall net is currently organized in two binders. Additionally, there are three binders containing tools for creating and simulating a net. Similar to pages for nets, tools themselves may be organized as tabbed binders. Multiple nets and binders may be open at the same time. Pages may be dragged from binder to binder without changing the execution aspect of the model. Dragging a page into the workspace will create a new binder for that page.

Figure 14 also shows various tools that are useful when creating and editing a net. Technically, each is referred to as a tool *palette*. A tool in a tool palette can be picked up by clicking on the appropriate *tool cell*. The tool remains attached to the mouse pointer and can be applied multiple times by clicking the mouse. To drop a tool, simply press the `ESC` key. From the *Net* tool, the *New Net* and *Load Net* tools are used to create a new model and load an existing model, respectively. When a model is created or loaded, its model overview will be added to the index. Another way to create a net or load a net is to simply right click on a blank space in the workspace and a context sensitive menu will pop up with various options. In general, right-clicking in the CPN Tools interface anywhere will bring up a context sensitive menu applicable to where the mouse is pointing. For example, right clicking on a page will bring up options to create a transition, place, arc, etc. Or one can use the *Create* tools.

The *Sim* tools are used for running interactive or automatic simulations. It contains video tape player-like controls used to manipulate the model. *Next* allows the user to select a transition to fire. *Play* randomly fires

enabled transitions until the end of the simulation is reached or the user hits the stop button. *Fast-forward* runs the simulation for a specified number of steps set by the user, while *Rewind* resets the simulation to its initial state.

The color sets, variables, and functions that are used in inscriptions must be defined in *Declarations* for the model. The declarations that belong to a model can be seen in the index in the model overview. New declarations are added by right clicking in a declaration section and selecting the option from the menu that pops up. Declarations can be grouped in *declaration blocks*.

Inscriptions must be added to nodes and arcs. After creating a new place, transition, or arc, text-edit mode will be entered, and it will be possible to add the first inscription to the element. Arcs have only one inscription, while places and transitions have several kinds of inscriptions. The `TAB` key is used to cycle between the different inscriptions for a node.

8 CONCLUSIONS

Petri Net is a widely studied mathematical formalism and provides a graphical notation suitable for modeling distributed and concurrent systems. Petri Nets provide the foundation for modeling concurrency, communication, synchronization, and resource sharing constraints that are inherent to many systems. However, Petri nets do not scale well when it comes to modeling and simulation of large systems. Colored Petri Nets (CPNs) extend Petri Nets with a high level programming language, making them more suitable for modeling large systems.

The focus of this paper has been to introduce the audience to vocabulary and concepts of Petri Nets and Colored Petri Nets (CPNs) and illustrate their use in modeling and simulation of systems. Rather than focusing on formal definitions of these frameworks, we took an example-based approach to incrementally introduce the reader to the details of Petri Nets and Colored Petri Nets. We discussed important features including hierarchy, color sets, and both timed and untimed nets. Thus, there are enough details in this paper so that even a reader who is not at all familiar with either of the two formalisms should be able to get started with the CPN Tools software to create and simulate models. As a starting point, interested readers may download all the models discussed in this paper. To practice and become more familiar with the CPN Tools software, readers may want to extend these models by incorporating following additional details and features:

- Include number of pages for each job and compute processing time based on number of pages.
- Include printers with multiple tray sizes and capacities.
- Include possibility of printers running out or paper and paper getting jammed and printers running out of ink.
- Include a monitor in the hierarchical model to compute waiting time for jobs.

These models can also serve as base models for some other applications. For example, jobs arriving at printers is not different from patients showing up at a hospital or doctor's office. Instead of jobs with preference for printers, we'd have patients with preference for doctors or even include a list of ailments to be treated.

In addition to simulation-based analysis, CPN Tools also supports state-space based analysis. Simulation can only be used to consider a finite number of executions of the model being analyzed. However, simulations cannot cover all possible executions of a modeled system in general. State-space based analysis complements simulation-based analysis and deals with all reachable states of a system. This paper does not focus on state-space based analysis. A high-level introduction to state-space analysis and its use is discussed in (Jensen et al. 2007). Finally, A CPN model of the example discussed in (White, Jr. and Ingalls 2009) is described in (Gehlot and Nigro 2010) and can serve as an additional resource.

REFERENCES

David, R., and H. Alla. 2004. *Discrete, Continuous and Hybrid Petri Nets*. Berlin: Springer-Verlag.

Gehlot, V., and C. Nigro. 2010. "An Introduction to Systems Modeling and Simulation with Colored Petri Nets". In *Proceedings of the 2009 Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, 104–118. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Haas, P. 2002. *Stochastic Petri Nets: Modelling, Stability, Simulation*. New York: Springer-Verlag.

Jensen, K. 1981. "Coloured Petri Nets and the Invariant Method". *Theoretical Computer Science* 14(3):317–336.

Jensen, K. 1994. "An Introduction to the Theoretical Aspects of Coloured Petri Nets". In *A Decade of Concurrency*, edited by J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Volume 803 of *Lecture Notes in Computer Science*, 230–272. Berlin-Heidelberg: Springer-Verlag.

Jensen, K., and L. M. Kristensen. 2009. *Coloured Petri Nets. Modelling and Validation of Concurrent Systems*. Berlin-Heidelberg: Springer-Verlag.

Jensen, K., and L. M. Kristensen. 2015. "Colored Petri Nets: A Graphical Language for Formal Modeling and Validation of Concurrent Systems". *Communications of the ACM* 58(6):61–70.

Jensen, K., L. M. Kristensen, and L. Wells. 2007. "Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems". *International Journal on Software Tools for Technology Transfer* 9(3):213–254.

Jensen, K., and G. Rozenberg. (Eds.) 1991. *High-level Petri Nets—Theory and Application*. Berlin-Heidelberg: Springer-Verlag.

Lindstrøm, B., and L. Wells. 2002. "Towards a Monitoring Framework for Discrete-Event System Simulations". In *Proceedings of the Sixth International Workshop on Discrete Event Systems (WODES'02)*, 127–134. IEEE Computer Society: Institute of Electrical and Electronics Engineers, Inc.

Murata, T. 1989. "Petri Nets: Properties, Analysis and Applications". *Proceedings of the IEEE* 77(4):541–580.

Peterson, J. L. 1981. *Petri Net Theory and the Modeling of Systems*. New Jersey: Prentice-Hall.

Petri, C. A. 1962. *Kommunikation mit Automaten*. Ph. D. thesis, Institut für Instrumentelle Mathematik, Bonn. English Translation, 1966: Communication with Automata, Technical Report RADC-TR-65-377, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York.

Popova-Zeugmann, L. 2013. *Time and Petri Nets*. Berlin: Springer.

Ramchandani, C. 1974. "Analysis of Asynchronous Concurrent Systems by Timed Petri Nets". Technical Report Project MAC, TR-120, MIT.

Reisig, W. 1985. *Petri Nets—An Introduction*. Berlin: Springer.

Reisig, W. 2013. *Understanding Petri Nets*. Berlin-Heidelberg: Springer-Verlag.

Ullman, J. D. 1998. *Elements of ML Programming*. New Jersey: Prentice-Hall.

Wang, J. 1998. *Timed Petri Nets—Theory and Application*. Boston: Kluwer Academic Publishers.

Wells, L. 2002. "Performance Analysis using Coloured Petri Nets". In *Proceedings. 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, 217–222. IEEE Computer Society: Institute of Electrical and Electronics Engineers, Inc.

White, Jr., K. P., and R. G. Ingalls. 2009. "Introduction to Simulation". In *Proceedings of the 2009 Winter Simulation Conference*, edited by M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, 12–23. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

AUTHOR BIOGRAPHIES

VIJAY GEHLOT is Professor and Graduate Program Director in the Computing Sciences Department at Villanova University. He is also a research faculty member with Villanova's Center of Excellence in Enterprise Technology (CEET). He received his PhD in Computer and Information Science from the University of Pennsylvania. His current research focus is applications of Colored Petri Nets (CPNs) in modeling and analysis of systems, including system of systems and system dynamics. Specific focus of his research works include healthcare applications pertaining to workflow and patient safety issues; enterprise service-oriented architectures and scalability issues; emergent behaviors of cyber-physical systems; broadband networks architecture, modeling, measurement, and analysis; systems biology and signaling pathways modeling. He is a member of the ACM, ACM SIGSim, and Sigma Xi. His email address is vijay.gehlot@villanova.edu.