TiPeNeSS: A Timed Petri Net Simulator Software with Generally Distributed Firing Delays

Ádám Horváth
University of West Hungary
Institute of Informatics and Economics
Bajcsy-Zsilinszky u. 9.
Sopron, Hungary
horvath@inf.nyme.hu

András Molnár
University of West Hungary
Institute of Informatics and Economics
Bajcsy-Zsilinszky u. 9.
Sopron, Hungary
molnaran@gain.nyme.hu

ABSTRACT

Performance analysis can be carried out in several ways, especially in case of Markovian models. In order to interpret high level of abstraction, we often use modeling tools like timed Petri nets (TPNs). Although some subclasses of TPNs (e.g. stochastic Petri nets (SPNs) [17, 19]) can be handled analytically, a general timed Petri net is hard to evaluate via numerical analysis[1]. However, the simulation of SPNs or deterministic and stochastic Petri nets (DSPNs) [16] are supported by many known tools (see, e.g. [3, 20]), it is hard to find a simulation tool for timed Petri nets with generally distributed (i.e., particular but arbitrarily chosen) firing times[2].

In this paper, we present TiPeNeSS (Timed Petri Net Simulator Software) which supports the simulation of timed Petri nets containing transitions with generally distributed firing delays. The input of the software (the Petri net and the parameters) is defined in an XML file, what allows us to generate results in batch mode. Besides, we describe a case study in which we optimize the frequency of the regular maintenance in a manufacturing process.

Categories and Subject Descriptors

I.6.7 [**Simulation and Modeling**]: Simulation Support Systems

General Terms

Modeling, Simulation, Software

[1]However, there exist some approaches for the numerical analysis of non-Markovian models, these are either restricted to special classes [8, 4, 10, 6], or use approximation methods [12].
[2]Although the specification of TimeNET claims that transitions with generally distributed firing times are enabled, the authors of this paper got only errors when tried to test that part of the tool.

Keywords

Simulation, Timed Petri Net, General Distribution

1. INTRODUCTION

Some subclasses of timed Petri nets can be analyzed numerically, such as stochastic Petri nets (SPNs) [17, 19], generalized stochastic Petri nets (GSPN) [15] or in some cases, the deterministic and stochastic Petri nets (DSPNs) [16]. However, a general TPN is hard to solve analytically, e.g. the TPN model of a manufacturing process, where the distribution of the work phases are typically deterministic, while the time between machine failures are assumed to be normally [7, 18] or gamma [13] distributed.

Although several tools exist for simulating Petri nets, we could not find one which is able to simulate Petri nets with generally distributed firing times. GreatSPN [3] has been developed at the University of Turin since the late eighties of last century, mainly for analyzing GSPNs. Later, a DSPN module was also developed to GreatSPN [9]. TimeNET [20] is a tool for analyzing and simulating DSPNs. Moreover, the user manual of TimeNET [20] states that generally distributed transitions are also enabled, and the graphical user interface ensures the use of generally distributed transitions. However, we could not find a way to run a simulation without errors when at least one generally distributed transition was placed in the model. Besides, many other tools for analyzing Petri nets are available (for a very thorough list, see [1]). However, most tools do not fit for our purposes. Many tools focus on other Petri net classes like SPNs, such as Pipe [5] or SPNP [11]; or like queuing Petri nets (QPNs), such as QPME [14]. Most software has its own graphical user interface (GUI) which supports the design of Petri nets. However, most of them cannot be driven in batch mode (only via the GUI) which does not ease the generation of multitudinous results.

In this paper, we present TiPeNeSS, a simulator software for TPNs in which we can use general distribution for firing delays (e.g. normal, uniform or gamma distribution) in addition to the distributions allowed in DSPNs (exponential, deterministic and zero-delayed). TiPeNeSS has three main functions: *i*) investigation of stability in each place; *ii*) transient and *iii*) steady state simulation. To ensure the precision of the simulation, TiPeNeSS has a statistical module which can determine the termination of the simulation process based on precision parameters (maximal relative er-

SIMUTOOLS 2015, August 24-26, Athens, Greece
Copyright © 2015 ICST
DOI 10.4108/eai.24-8-2015.2261343

ror and confidence level). The software can be run using a batch file without interacting with a graphical user interface, since the Petri net and the simulation parameters are described in an XML input file.

The rest of the paper is organized as follows. Section 2 provides a formal description of TPNs. In Section 3, we present TiPeNeSS, our TPN simulator software, while Section 4 describes a case study in which we demonstrate an application of TiPeNeSS through the evaluation of a manufacturing process model. Finally, we summarize the paper and point out the future plans in Section 5.

2. TPN FORMALISM

In this section, we provide a short introduction to TPNs, while a detailed description with applications can be found in [15].

TPNs are bipartite directed graphs with two types of nodes: places and transitions. The places, graphically represented as circles, correspond to the state variables of the system; while the transitions, graphically represented as boxes, correspond to the events that can induce a state change. The arcs connecting places to transitions and vice versa express the relation between states and event occurrence.

Places can contain tokens drawn as black dots within places. The state of a TPN, called marking, is defined by the number of tokens in each place. In this paper, we use the notation M to indicate a marking in general. We will denote by $M(p)$ the number of tokens in place p in marking M. Now we recall the basic definitions that are necessary for the rest of the paper.

A TPN system is a tuple

$$(P, T, I, O, H, M_0, \tau, w, e),$$

where:

- P is the finite set of *places*. A marking $M \in \mathbb{N}^{|P|}$ defines the number of tokens in each place $p \in P$.
- T is the set of *transitions*. The distribution of time between the firings of $t \in T$ has general distribution, e.g. exponential, deterministic, normal etc. Note that $P \cap T = \emptyset$.
- $I, O, H : \mathbb{N}^{|P|} \to \mathbb{N}$ are the multiplicities of the *input arc* from p to t, the *output arc* from t to p, and the *inhibitor arc* from p to t, respectively.
- $M_0 \in \mathbb{N}^{|P|}$ is the *initial marking* of the net.
- $\tau_t : \mathbb{N}^{|P|} \to \mathbb{R}$ is the mean delay for $\forall t \in T$ (note that τ_t may be marking-dependent).
- $w : \mathbb{N}^{|P|} \to \mathbb{R}^+$ is the firing weight for $\forall t \in T^Z$, where $T^Z \subset T$ denotes the set of immediate transitions.
- $e_t : \mathbb{N}^{|P|} \to \{R, E, A\}$ is the execution policy or memory policy to be used for transition t, while R, E and A denote *resampling*, *enabling memory* and *age memory* policy, respectively. Further description of memory policies can be found at the end of this section.

A transition is "enabled" if each of its input places contains the "necessary" amount of tokens where "necessary" is defined by the input function I. Formally, transition t is enabled in marking M if for all places p of the net we have $M(p) \geq I(t,p)$. An enabled transition can fire and the firing removes tokens from the input places of the transition and puts tokens into the output places of the transition. The new marking M' after the firing of transition t is formally given $M'(p) = M(p) + O(t,p) - I(t,p)$, $\forall p \in P$.

The firing of a transition occurs after a given delay. The delay associated with a transition has a specific distribution whose parameter depends on the firing intensity of the transition and on the current marking. We define the effect of the firing of transition t with an integer vector $L(t)$, place indexed, defined as: $L : T \to \mathbb{N}^k$ and the ith entry of $L(t)$ is $L(t)_i = O(t,p_i) - I(t,p_i)$ for $1 \leq i \leq k$ and $\forall t \in T$. For sake of avoiding cumbersome notation we assume that $\nexists t, t' \in T : t \neq t', L(t) = L(t')$ and $\nexists t : L(t) = 0$.

Based on the fundamental work of Marsan et al. [15], the applied memory policies can be the following.

- **Resampling (RS)**: At each and every firing, all transitions of the TPN are discarded, and new values of timers will be generated (restart mechanism).
- **Enabling memory (EM)**: At each firing, the timers of all transitions that are disabled are restarted, while the timers of transitions that are not disabled hold their present value (continue mechanism).
- **Age memory (AM)**: At each firing, the timers of all transitions hold their present values (continue mechanism).

3. TiPeNeSS, A TIMED PETRI NET SIMULATOR SOFTWARE

In this section, we describe our solution for simulating timed Petri nets with generally distributed firing delays.

3.1 The Structure of TiPeNeSS

The software requires a user defined parameter file which contains the description of the investigated TPN model and the simulation parameters. The simulation core reads the input file and constructs the basic model with the defined initial conditions. After initialization, the core simulates the behavior of the model applying the simulation method defined in the parameter file.

During the simulation, the change of the token distribution is processed by the statistical module. The core updates the statistical values of the investigated parameters based on the results produced by the statistical module. The core continuously checks the accuracy of the estimated values in order to determine the fulfilment of the termination condition for the simulation process. Upon finishing the simulation, the core creates a result file in plain text format which contains the short description of the model, the simulation parameters and the estimated values of the investigated parameters. The structure of TiPeNeSS is shown in Fig. 1.

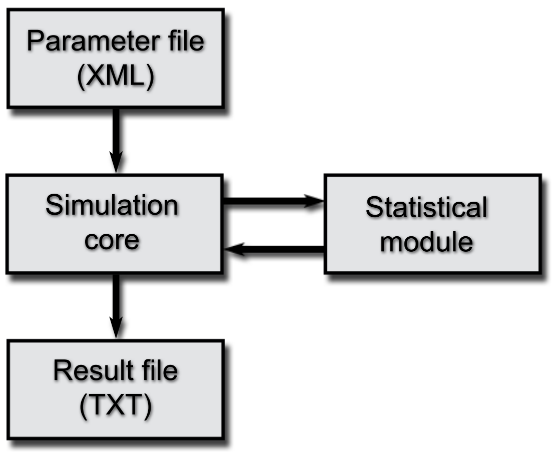

Figure 1: The structure of TiPeNeSS.

3.2 The Core Module

The core module is responsible for the whole simulation process. Its input is an XML parameter file, and its output is a text file containing the expected token distribution and some other parameters (e.g. precision parameters). To ensure the precision of the simulation, the core module interacts with a statistical module during the whole run time. The core module is based on the use of an event queue which is the most common simulation technique in event driven simulation.

Since one simulation run does not provide enough data to obtain the behavior of the model, many samples are taken usually to obtain the model properties. Several approaches have been proposed to produce these samples (for details, see [13]). In this work, we applied the replication/deletion and batch means approaches [13].

Currently, TiPeNeSS supports the following types of investigations.

- Transient simulation
- Steady state simulation
- Stability analysis

Transient simulation runs for a fixed amount of simulated time which can be set in the parameter file. This function is mostly used to examine the estimated number of tokens after a certain amount of time. In transient simulations, we must use the replication/deletion approach to ensure the precision of the simulation, since the batches cannot be considered independent if they are not "appropriately long".

On the other hand, steady state simulation is used to determine the steady state behavior of certain parameters. In this investigation, we can use both the replication/deletion and the batch means approaches, since we can use long batches in order to ensure their statistical independency [13].

Finally, stability analysis is used to determine whether a system parameter has a steady state distribution. Since stability is a prerequisite of running steady state simulation, we may want to be sure of the stability of the system.

Regarding to the transition types of TPNs, the core module supports different types of transitions based on the distribution of their firing delays. Currently, it can either be zero-delayed (immediate transitions), deterministic, exponential, normal or gamma distribution. In the future, we can implement additional types of firing delays, of course. To achieve this, we need to implement only the cumulative distribution function of the newly added distribution.

Since a generally distributed TPN requires the application of different memory policies (see Section 2), we implemented the policies in the core module. In the current version of TiPeNeSS, this property can be defined at each transition individually. Later, the interpretation of this property can be extended to transition pairs.

3.3 The Statistical Module

The software contains a standalone statistical module which provides statistical methods for the simulation. This module allows us to calculate the sample mean which is used to estimate the value of the investigated parameters.

In each iteration, we calculate the sample mean of an investigated variable as follows.

$$\bar{x}_n = \frac{\left(\sum\limits_{i=1}^{n-1} t_i\right) \cdot \bar{x}_{n-1} + t_n \cdot x_n}{\sum\limits_{i=1}^{n} t_i},$$

where $\sum_{i=1}^{n-1} t_i$ denotes the cumulative weight of the previous iteration, $\bar{x}_{n-1}$ denotes the mean of the previous iteration, t_n represents the current weight and x_n stands for the current value. Since we store the cumulative average of the previous iteration and its weight, the calculation of the weighted average requires only constant cost in each iteration.

The sample mean gives an estimation of an investigated parameter. However, it does not provide any information about its accuracy. In order to measure the precision, we need to calculate the confidence interval for the given parameter. The calculation of the confidence interval requires two additional parameters: the sample variance and the sample size. The sample variance can be calculated as follows.

$${S_n}^2 = \frac{\sum\limits_{i=1}^{n-1} t_i}{\sum\limits_{i=1}^{n} t_i} {S_{n-1}}^2 + \frac{t_n(x_n - \bar{x}_{n-1})^2}{\sum\limits_{i=1}^{n} t_i} + d^2 - 2d\bar{x}_n + 2d\bar{x}_{n-1},$$

where ${S_n}^2$ and ${S_{n-1}}^2$ represent the variance for the current and the previous iteration, respectively, while d is calculated as $d = \bar{x}_n - \bar{x}_{n-1}$. Similarly to the previous equation, the calculation cost of the sample variance is constant.

Besides, the statistical module can measure the accuracy of the estimation based on the value of maximal relative error which can be defined in the parameter file. This function calculates the half-length of the confidence interval and the maximal error, which is based on the sample mean of the investigated parameter and the user defined maximal relative error. If the half-length of the confidence interval is smaller than the computed maximal error, we can accept that the estimation reached the desired accuracy (we assume that the relative error does not exceed 0.15 and the sample size is at least ten [13]). This condition can be used for parameters having steady state distribution and their mean value is not equal to zero.

3.4 The Parameter File

The input of the simulation is a user defined parameter file which contains necessary information to execute simulations. The parameter file is written in XML format, what provides platform independency. Furthermore, it represents the hierarchy of the defined elements, too. The parameter file consists of two main parts: the model description and the simulation parameters. First, we will present the model description through the sample of M/D/1/2/2 queue according to Kendall's notation (see Fig. 2).

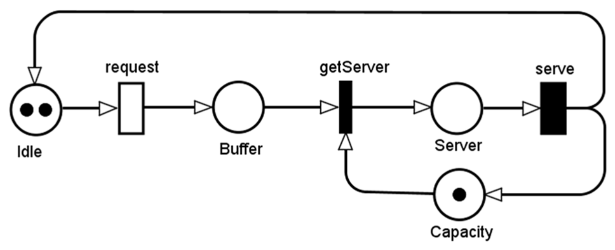

Figure 2: The Petri net model of M/D/1/2/2 system.

In the parameter file, we define our Petri net using the `<petrinet>` tag as root, while its children can be places (`<place>`) and transitions (`<`*xyz*`transition>`, where *xyz* denotes the distribution of the transition). Places contain the following sub-tags: `<name>` and `<token>`. The `<name>` tag defines the place name; this element is mandatory and it needs to be unique as it is used as a reference later on. The `<token>` tag token is optional; it is used to specify the initial number of tokens in the given place (if omitted, zero is used as default value). For example, we can define place *Idle* of Fig.2 as follows.

```
<place>
    <name>Idle</name>
    <token>2</token>
</place>
```

Since there are several different types of transitions, each with its own set of properties, we apply unique tags for the different distributions. The commonly used attributes are the transition name, the input and output places, which are independent from the distribution of transitions. The `<name>` tag is mandatory and has to be unique as it used as reference. The tags `<inplace>` and `<outplace>` are used to define the connections between places and transitions, i.e., these tags are used for defining arcs. Both input and output tags have two sub-tags: `<name>` and `<arc>`. The `<name>` tag is mandatory, it is used to reference an already defined place. The arc provides an option to assign weights to the arcs. If omitted, the arc will be declared with the default value of one. Among the arc weights, the zero value has a special meaning, it denotes an inhibitor arc.

Now, we describe the different types of transitions and their unique properties. Besides the previously described attributes, immediate transitions have two additional sub-tags: `<priority>` and `<weight>`. The `<priority>` tag is used to determine which transition has precedence over the other concurrent immediate transitions (higher number denotes higher priority). The `<weight>` tag is used to order firing weight to immediate transitions being in conflict in a marking.

In our example, *getServer* is an immediate transition having the following XML definition.

```
<immedtransition>
    <name>getServer</name>
    <inplace>
        <name>Buffer</name>
        <arc>1</arc>
    </inplace>
    <inplace>
        <name>Capacity</name>
        <arc>1</arc>
    </inplace>
    <outplace>
        <name>Server</name>
        <arc>1</arc>
    </outplace>
</immedtransition>
```

Exponentially distributed transitions can be defined using the `<exptransition>` tag. They have two sub-tags: `<delay>` and `<servertype>`. The first is mandatory and denotes the mean delay of the transition, while the second is optional and defines whether the firing rate depends on the enabling degree or not (*infinite* or *exclusive* server approach, respectively [15]). If `<servertype>` tag is omitted, the transition follows the exclusive server policy by default.

In our example, transition *request* is described as an exponential transition that can be defined as follows.

```
<exptransition>
    <name>request</name>
    <servertype>infinite</servertype>
    <delay>0.1</delay>
    <inplace>
        <name>Idle</name>
    </inplace>
    <outplace>
        <name>Buffer</name>
    </outplace>
</exptransition>
```

Transitions with deterministic delay can be defined using the `<dettransition>` tag. Similarly to the exponential transitions, deterministic transitions also have a `<delay>` sub-tag which denotes the exact value of delay in this case.

Except the exponentially distributed transitions, we have to define the memory policy for all delayed transitions using the `<memory>` tag. The possible values are *age*, *resampling* and *enabling*, representing the age memory, the resampling and the enabling memory policy, respectively (the default memory policy is the enabling memory policy).

In case of our M/D/1/2/2 queue, transition *serve* can be described as follows.

```
<dettransition>
    <name>serve</name>
    <delay>0.06</delay>
    <inplace>
        <name>Server</name>
        <arc>1</arc>
    </inplace>
    <outplace>
        <name>Idle</name>
        <arc>1</arc>
    </outplace>
    <outplace>
        <name>Capacity</name>
        <arc>1</arc>
    </outplace>
</dettransition>
```

In case of other distributions, other distribution-specific parameters have to be defined. For example, normally distributed transitions have two specific parameters: instead of the `<delay>` tag, the mean value and the standard deviation have to be specified (tags `<mean>` and `<deviation>`)[3].

Finally, the required and optional simulation parameters can be described using the `<system>` tag which is the direct sub-tag of `<petrinet>` tag in the XML hierarchy. In Table 1, we collected the corresponding parameters.

4. SEQUENTIAL WORK PHASES IN MANUFACTURING: A CASE STUDY

In this section, we present the strengths of TiPeNeSS by modeling the operation of a production line considering the machine failures and the regular maintenance process. Evaluating the model, we determine the optimal time interval for the regular maintenance.

4.1 Production Lines in Manufacturing

A production line is a set of sequential operations in a manufacturing system or factory, where the input of the first operation is the raw material (such as metal ores, cotton, or foodstuff), while the output of the last operation is the product.

In such systems, it is crucial that the sequential operations depend on each other. For example, if the second machine in the production line breaks down, the elements pile up on the output side of the first machine, while the machines after the second machine remain unutilized. Hence, regular maintenance is a commonly applied technique to alleviate the negative effects of machine failures.

From modeling point of view, the model of a production line consists of transitions with different distributions. The work phases have typically deterministic delays, while the time between machine failures follows the normal [7, 18] or gamma [13] distribution in case of mechanical failures (in case of electronic tools, the exponential distribution must be applied). If a failure occurs despite of the application of regular maintenance, the time needed to repair the failure is exponentially distributed.

[3]Note that we use only the positive values given by the normal distribution.

4.2 A Sample Production Line

In the following sample, we model a production line with three sequential operations to demonstrate the application of TiPeNeSS. Without concretizing the functionality of the machines, we assume that machine failures can occur due to mechanical failures e.g. due to abrasion. The Petri net model of the sample production line is depicted in Fig. 3.

If the system is up (no machines have to be repaired and there is no maintenance), the raw material arrives in place *RawMaterial* which is a queue of transition *procedureA*. The service follows the FIFO policy: when a machine finishes its current job, the first element of the queue is moved to that machine, i.e., a token moves from place *RawMaterial* to place *InputA*. After an element is served, it is moved to the next queue (place *AfterFirstStep*), then to the next machine (place *InputB*), and so on. Finally, with the firing of transition *procedureC*, the element is considered completed (a token is moved to place *ManufacturedItems*).

In case of a machine failure (transitions *failureA*, *failureB* and *failureC*), the working machines can continue their work if they have elements in their queue. However, we do not allow the arriving of new elements at place *RawMaterial* until the failure is repaired. A machine failure can occur only in the case when the machine is currently working on an element. In this case, the element is booked as refuse and must be re-manufactured (i.e., if *procedureA* cannot be executed due to a machine failure, the current element must be re-manufactured, and the remaining time of *procedureA* must be re-generated, following the enabling memory policy). The time until the machine failure is repaired can be modeled with an exponentially distributed random variable. Therefore, transitions *repairA*, *repairB* and *repairC* are exponentially delayed.

To reduce the idle periods caused by machine failures, a regular maintenance process is done periodically. Under maintenance, no new elements are allowed to arrive in the system, and the machines are stopped. When a maintenance process is due (i.e., transition *maintenanceTime* fires), the machines can finish their current work, but do not receive new elements of the queue until the maintenance period is over. If the works in progress are finished, the maintenance process starts (i.e., a token is moved to place *UnderMaintenance* by the firing of immediate transition *doMaintenance*). The manufacturing continues after the maintenance period (by the firing of transition *endMaintenance*). If a machine failure occurs when maintenance is needed, the repair of machines has priority over the regular maintenance process. It means that while the repair of a machine holds, the maintenance will not be started. Unlike the machine failures, the time interval of the regular maintenance can be predicted. Therefore, transition *endMaintenance* is deterministically delayed.

4.3 Finding the Optimal Scheduling of the Maintenance Process

To obtain the optimal scheduling of the regular maintenance for our sample production line, we ran steady state simulations in which we applied the batch means approach. We give an overview of the precision parameters in Table 2,

Table 1: Overview of system parameters.

Parameter (tag)	Possible values	Description
`<method>`	*analysis*; *batchmean*; *repdel*.	The type of investigation can be stability analysis, batch means or replication/deletion method, respectively.
`<batch>`	Any positive double value.	The length of batches (used only in case of stability analysis and batch means method).
`<terminatingtime>`	Any positive double value.	The length of one individual run in case of replication/deletion method.
`<warmuplength>`	Any positive double value.	The length of warm-up period.
`<minsamplesize>`	Any positive integer value.	The number of batches in case of batch means method and stability analysis; the number of replications when replication/deletion method is used.
`<confidencetype>`	1; 2; 3.	The confidence level of 95%, 98% and 99%, respectively.
`<accuracy>`	A double value $x \in (0; 1)$.	The value of maximal relative error.
`<avgtoken>`	A place name defined in the model.	A significant place where we estimate the average number of tokens when steady state simulation is run.
`<token>`	A place name defined in the model.	A significant place where we estimate the number of tokens at the and of a transient simulation run.
`<outfilepath>`	The path of the result file.	The results are collected in this file.

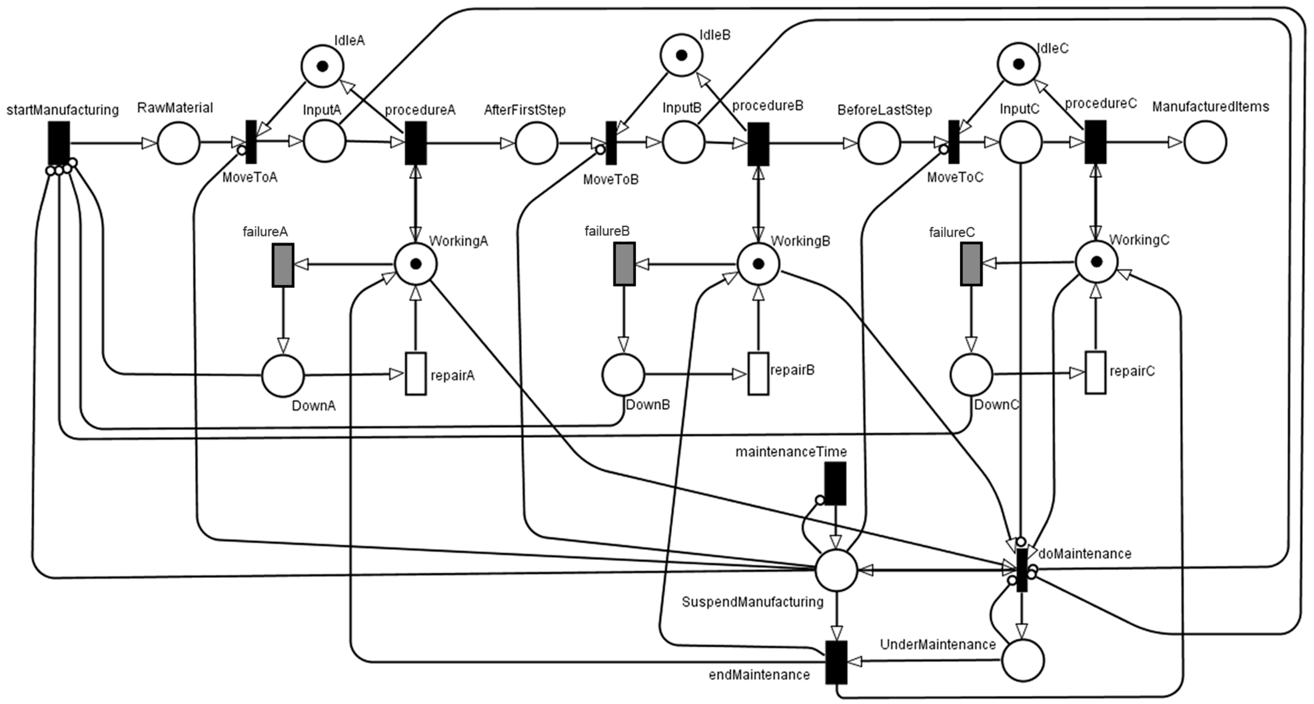

Figure 3: The Petri net model of the sample production line.

Table 2: Overview of the precision parameters.

Parameter	Value
length of warm-up period	5000 hours
length of batches	3000 hours
confidence level	95%
maximal relative error	5%

while we collected the settings of the transitions in Table 3.

In the following, we present two investigations which help to support the decisions in the above mentioned scheduling problem.

Since the main goal of the investigations was finding the optimal scheduling of the maintenance process, the running parameter of the simulation was the delay of transition *maintenanceTime* in both cases, while we fixed the other parameters of Table 3.

In the first investigation, we ran steady state simulation in order to obtain the ratio of down time for the system components (machines). The ratio of down time consists of two

Table 3: Overview of transitions.

Transition name	Distribution	Memory policy	Mean delay [hours]	Deviation [hours]
$startManufacturing$	deterministic	AM	1/6	
$moveToA$	zero-delayed			
$procedureA$	deterministic	EM	1/12	
$failureA$	normal	EM	200	20
$repairA$	exponential		2	
$repairA$	exponential		2	
$moveToB$	zero-delayed			
$procedureB$	deterministic	EM	1/6	
$failureB$	normal	EM	250	25
$repairB$	exponential		2	
$moveToC$	zero-delayed			
$procedureC$	deterministic	EM	1/10	
$failureC$	normal	EM	300	30
$repairC$	exponential		2	
$maintenanceTime$	deterministic	EM	10..500	
$doMaintenance$	zero-delayed			
$endMaintenance$	deterministic	EM	1/2	

components: one caused by machine failures, and the other because we have to stop the system during the maintenance process. Therefore, we can obtain the ratio of down time for machine X by summing the average number of tokens in place $DownX$ and the average number of tokens in place $UnderMaintenance$. The results are shown in Fig. 4.

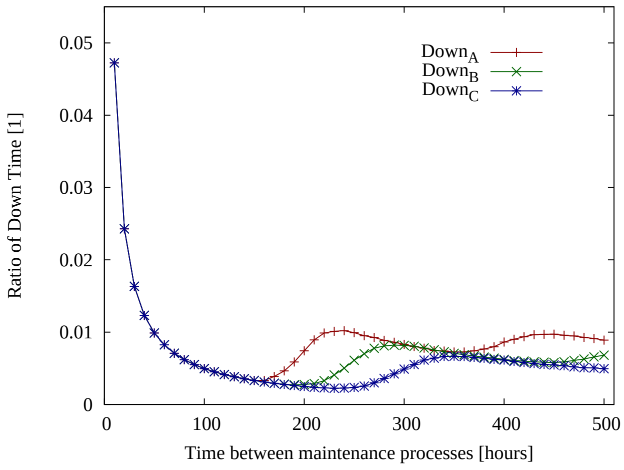

Figure 4: The ratio of down time as a function of time between maintenance processes.

In Fig. 4, we can see that each machine has its own optimal value depending on the frequency of failure occurrences. However, the maintenance requires the stopping of the whole manufacturing process in our model, so we must find a common value of 160 working hours for all machines of the system.

In the other investigation, we ran transient simulation to determine the throughput of the system for one year[4]. Fig. 5 illustrates the results.

The results of this investigation are consonant to the previous one's, we can see that the maximal throughput can be achieved if the maintenance process is scheduled in every 160 working hours (12143 manufactured items).

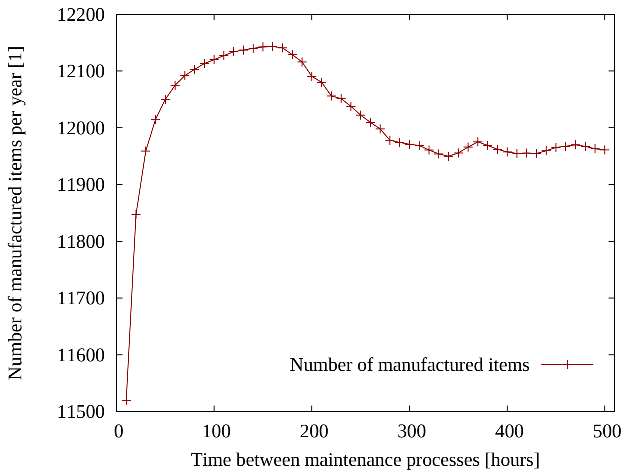

Figure 5: The number of manufactured items per year.

In both cases, we can see that too frequent scheduling of the regular maintenance is not worth while, since the system has to be stopped during the maintenance. On the other hand, if we schedule the maintenance infrequently, the ratio of down time will increase depending on the expected value of the number of failures till the next maintenance process.

5. SUMMARY

In this paper, we presented TiPeNeSS, a simulator software for TPNs having generally distributed delays. Besides the description of the software components, we presented the usefulness of TiPeNeSS through the optimization problem of a manufacturing system. In this example, the strength of the software are touchable, since the model of a manufacturing system contains different types of transitions.

[4]We calculated with one shift and 2032 working hours.

Our simulator software is under development, the current version is available through an open source version control system [2].

In the future, we plan to implement a graphical TPN editor to TiPeNeSS in order to decrease the probability of errors caused by the difficulties of editing the XML input file.

References

[1] List of Petri net tools on the web page of Department of Informatics, University of Hamburg, Germany. `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/complete_db.html`. Online; accessed 3rd June 2015.

[2] The source code of TiPeNeSS. `https://github.com/molnaran/tipeness`. Online; accessed 22nd July 2015.

[3] S. Baarir, M. Beccuti, D. Cerotti, M. De Pierro, S. Donatelli, and G. Franceschinis. The GreatSPN tool: recent enhancements. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):4–9, 2009.

[4] A. Bobbio and N. Telek. Markov regenerative spn with non-overlapping activity cycles. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 124–133. IEEE, 1995.

[5] 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.

[6] G. Ciardo and C. Lindemann. Analysis of deterministic and stochastic petri nets. In *Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on*, pages 160–169. IEEE, 1993.

[7] C. for Chemical Process Safety. *Appendix G: Statistical Distributions Available for Use as Failure Rate Models*, pages 695–703. John Wiley & Sons, Inc., 2010.

[8] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi. Analysis of software rejuvenation using markov regenerative stochastic petri net. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 180–187. IEEE, 1995.

[9] E. Gilberto and S. Donatelli. Dspn-tool: A new dspn and gspn solver for greatspn. In *Quantitative Evaluation of Systems (QEST), 2010 Seventh International Conference on the*, pages 79–80, Sept 2010.

[10] M. Gribaudo, M. Sereno, and A. Bobbio. Fluid stochastic petri nets: An extended formalism to include non-markovian models. In *Petri Nets and Performance Models, 1999. Proceedings. The 8th International Workshop on*, pages 74–81. IEEE, 1999.

[11] C. Hirel, B. Tuffin, and K. S. Trivedi. SPNP: Stochastic petri nets. version 6.0. In *Computer Performance Evaluation. Modelling Techniques and Tools*, pages 354–357. Springer, 2000.

[12] G. Horton. A new paradigm for the numerical simulation of stochastic petri nets with general firing times. In *Proceedings of the European Simulation Symposium*, pages 129–136, 2002.

[13] W. D. Kelton and A. M. Law. *Simulation modeling and analysis*. McGraw Hill Boston, MA, 2000.

[14] S. Kounev, S. Spinner, and P. Meier. Qpme 2.0-a tool for stochastic modeling and analysis using queueing petri nets. In *From active data management to event-based systems and more*, pages 293–311. Springer, 2010.

[15] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley and Sons, 1995.

[16] M. A. Marsan and G. Chiola. On Petri nets with deterministic and exponentially distributed firing times. In *Advances in Petri Nets 1987*, pages 132–145. Springer Berlin Heidelberg, 1987.

[17] M. K. Molloy. *On the Integration of Delay and Throughput Measures in Distributed Processing Models*. PhD thesis, UCLA, Los Angeles, CA, 1981.

[18] H. Pham. *System software reliability*. Springer, 2007.

[19] F. J. W. Symons. *Modeling and Analysis of Communication Protocols Using Numerical Petri Nets*. PhD thesis, University of Essex, 1978.

[20] A. Zimmermann and M. Knoke. *TimeNET 4.0: A software tool for the performability evaluation with stochastic and colored Petri nets; user manual*. TU, Professoren der Fak. IV, 2007.