

Application of Petri-Nets in Object-Oriented Environment

Dávid Bedők

*John von Neumann Faculty of Informatics
Óbuda University, Budapest, Hungary
e-mail: bedok.david@nik.uni-obuda.hu*

Abstract—The Petri-net is an effective modeling tool, especially in the area of the concurrent events, and it has a very intuitive characteristic: the non-deterministic simulation. This behavior is hard to be implemented in an imperative language, and its parameters even more difficult to be changed. This was the reason why this paper was selected the aim: embed Petri-nets into an imperative high-level language and use its beneficial properties as clean as it possible.

To reach that goal, first of all a well-formed description was needed about Petri-nets, and its extensions, and also need a new extension to model the object-oriented events inside the simulation. We have to identify the moments of the transitions' activation (transition firing), the token player movement and the change of the token distribution.

When somebody wants to implement a state machine which able to handle the transitions between the application's states, a Petri-net model creation may be needed to help the understanding of the task. In that case the model should not be only kind of static document of the product. With a new Petri-net extension it can be highlight connections between the model and the real source code events. If somebody changes the model it will take effect "immediately" in the product. In the agile world the IT industry is also changing: the speed of the changes is faster than the chance to generate a modified model of any applications.

Index Terms—Petri nets, object-oriented approach, simulation, net extension, modeling

I. INTRODUCTION

Why is it an interesting area? It is essential to answer this question. When a program can be created and used an imperative language (like C# or Java), most of the time instructions and commands are written in imperative mood to define for the computer how to solve the problem. On the other hand, at the world of the declarative thinking we try to describe what we want to solve, and don't care about the "How-To's". Therefore the imperative languages often lost in the details, and if we want to modify the whole background logic, we have to dig deep inside the code. Perhaps a developer generation has been already born, who only integrate pre-prepared API in duty time, instead of implementing a nice algorithm for the eight queen problem. For that generation a declarative modification is more manageable, and it will be a big plus if this modification immediately takes effect on the running code.

Live visualization, live or generated documentations are more and more important things nowadays. Currently a lot of people are working in the information technology ecosystem, and not all of them are engineers. The way how an IT issue

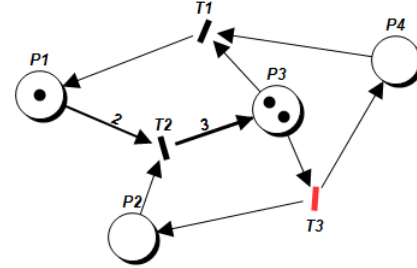


Figure 1: Sample Petri-net

be solved is changing. Not only technical skills are needed but also it would be required to deliver quality, readable and maintainable source code, nice graphical interfaces and aim professional user experiments. An engineer is working in that thing with a lot of people who has several soft skills instead of/among of technical knowledge. The modeling was always important and it helped the communication between the different areas, but in an agile world the speed of the changes are higher than the opportunity of generating or creating a modified model between the groups. One solution could be a live modeling toolset which helps for instance the managers and the front-end developers/testers to understand the behavior of the application in the current release.

II. PETRI-NETS

Today the Petri-nets are classified as a mathematical modeling language which is a good primary for describing distributed and parallel systems [1]. It offers graphical and mathematical description at the same time and it allows us to quickly review complex systems, while it is able to find the details as well. In today's modern computer science it has a key role in workflow-driven modeling/management. Perhaps its significance is even more in terms of the non-deterministic systems. In such systems the next state is cannot be known even from another well-described state.

Essentially Petri-net (or Place/Transition net) is a directed and weighted bipartite graph wherein two types of nodes occur: position and transition (Fig. 1). Weighted directed edges can be found between positions and transitions. A position is called input place if directed edge binds it to a transition, and output place if a directed edge binds a transition to that position.

Each position can contain any number of tokens. The tokens are commonly called players as well, the distribution of these determines the actual state of the Petri-net (like the serializable fields in an object-oriented environment). By definition a transition fires when its all input places contain at least as many tokens as the weight of the related edges. After firing as many token players will appear in all the output places of the transition as the weight of the related edges between the transition and the output places. These two parts of the firing process are entirely atomic operations, they cannot be distributed and no process can be wedged here. It is worth mentioning that the number of tokens is not a constant number in a Petri-net.

The firing order is fully non-deterministic. If two different transitions can be ready to fire in one state, any transitions may be activated and this selection process is random in the general case. The system does not exclude additional rules which influence the choosing process in that special case. This aforementioned nature allows us to simulate the concurrent behavior of a distributed system in a general and in a priority case too.

There are two special transitions as well: the source and the absorbing transitions. One of them has not got input places and it can be firing at any time while the other one has not got output places and when it fires it absorbs the token players.

A. Extensions

A huge problem has to be talked about Petri-nets. If somebody wants to create a simpler but real Petri-net, a quite big graph will be built and this reduces the transparency which is something that we surely do not want when modelling a product. To compensate this several Petri-net extensions can be used. Some of them are entirely backward compatible with the original nets, but there are some High-Level Petrinets which were originally created for a specialized areas.

Reset arcs can be used in a net which is not a condition of the firing but when the transition fires it will empty the connected position (it removes the token players). It has only one orientation, it goes from position to transition, and it has not got weight. The *capacity limit* is another extension, it can be set for a position. This defines the maximum number of tokens in the related position, and it can prevent the firing process because of limit violation. Something similar is the *inhibitor arc* where the firing process will be disabled until the position contains tokens. Inhibitor arc may have weight, this marks the number of tokens where the blocking is active.

In the *Prioritised Petri-nets* a priority can be set for each transition. This changes the token game, because a transition can fire only when there is not any other higher priority ready-to-fire transition. The behavior of this type of Petri-nets remains non-deterministic if there are ready-to-fire transitions which have the same priority.

The *Coloured Petri-nets* belong to the group of High-Level Petri-nets. Here the token players are not identical, each token has a value and most of the time a type as well. It allows executing complex operations inside the model. To support that edge- and guard-expressions can be created. A transition will

not fire if the expression of the related edge is invalid, and the output tokens are also be calculated based on the expressions on the output side. From this perspective the Coloured Petri-net is some kind of high-level programming language [2].

A very interesting area is the world of *Algebraic Petri-nets* [3]. The token players have algebraic datatypes which are similar to the Coloured Petri-nets, but here we get some predefined operations/rules/calculations as well. Jacques Vautherin made the first APN in 1985, later Wolfgang Reisig improved it.

We can talk about *Hierarchical Petri-nets* which are the parents of the *Object Petri-nets*. The embedded nets were introduced here: multiple nets can communicate with each other via synchronization. A good implementation of that is the *Concurrent OO Petri Nets* [4] which are based on the Algebraic Petri-nets.

III. MATHEMATICAL MODELS

Tadao Murata created a mathematical description of the Petri-nets in 1989 [5].

Network \mathcal{G} : $\mathcal{G} = (\mathcal{P}, \mathcal{T}, \mathcal{W})$ where

Places: $p \in \mathcal{P}$

Transitions: $t \in \mathcal{T}$

Weights of edges/arcs: $w \in \mathcal{W}$ where

$\mathcal{W} : (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P}) \rightarrow \mathbb{N}^+$

Directed edges/Arcs: $a \in \mathcal{A}$ where

any $a \in \mathcal{A}$ arc has a $w(a) \in \mathbb{N}^+$ weight

Furthermore \mathcal{P} and \mathcal{T} are disjoint sets.

A $\bullet t$ notation is defined for marking the input places of a $t \in \mathcal{T}$ transition: $\bullet t = \{p \in \mathcal{P} | W(p, t) > 0\}$, and also $t \bullet$ notation is defined for marking the output places of a $t \in \mathcal{T}$ transition: $t \bullet = \{p \in \mathcal{P} | W(t, p) > 0\}$. Along the same rule $\bullet p$ and $p \bullet$ notations are defined for marking a set of transitions from where it may get ($\bullet p = \{t \in \mathcal{T} | W(t, p) > 0\}$) or to whom it may give tokens ($p \bullet = \{t \in \mathcal{T} | W(p, t) > 0\}$).

m_{p_i} means a state of a $p_i \in \mathcal{P}$ position which is the number of tokens in that place. The \mathcal{M} marks a **Marking vector** which contains all $p \in \mathcal{P}$ positions' m_p values in the whole \mathcal{G} Petrinet ($\mathcal{M} = [m_{p_1}, m_{p_2}, \dots, m_{p_k}]^T$ where \mathcal{G} net has k positions). \mathcal{M} is a state of the \mathcal{G} net. \mathcal{M}_0 marks the initial token distribution vector.

In practice $\mathcal{G} = (\mathcal{P}, \mathcal{T}, \mathcal{W}, \mathcal{M}_0)$ form is used instead of $\mathcal{G} = (\mathcal{P}, \mathcal{T}, \mathcal{W})$ to describe a Petri-net.

IV. PETRI NET MARKUP LANGUAGE (PNML)

To describe a Petri-net in a clear and accurate form is very important not only in mathematics but also in computer science. For this purpose the Petri Net Markup Language [6] was created. Originally it was the output format of the *Petri Net Kernel* application [7], but later on it became kind of standard XML based description of the Petri-nets. Three kinds of nets can be described with PNML:

- Original Petri-nets (Place/Transition nets)
- High-Level Petri-nets (e.g.: Coloured Petri nets)
- Symmetric nets

¹Notation: \mathbb{N}^+ the set of (positive) natural numbers

The PNML was designed openness, so additional properties of the elements can be added. The supplementary information is stored in a standalone Petri Net Type Definition (PNTD) file.

V. RELATED WORKS

There are lots of Petri-net simulators on the Internet, but *Renew* [8] a little bit different than the others and it has some interesting objectives which are partly similar than the goals of this paper. *Renew* looks the Petri-nets as object-oriented classes, and before the token game it instantiates the nets to play the simulation. *Renew* calls this type of net as Reference net, and expressions can be set many points of it just like in a Coloured Petri-net. The syntax of the expressions is kind of simplified Java. A very interesting part of this application that the Reference nets can be loaded in a Java application (as an instance of the *de.renew.net.NetInstance* class), and the simulation can be played via the API of this product. This possibility is very similar than the main target of this work.

According to the above-mentioned concept the instances of the simulated networks are all unique (like in the object oriented environment) and its can be initialized during the creation (some kind of constructors). In the token gameplay Java objects can be reached and methods are able to be called, etc. It is important to note that all operations, methods, initializations have to be defined in advance inside the source code of the Reference net.

In this network not only the tokens have the type but also the positions as well. This type defines what kind of tokens can be held in that place. If the position has an initialization block, its result will be the initial number of tokens, but there are global initialization blocks as well to set the initial token distribution. These lines of code are executed before the token gameplay only once. The edge expressions are also a parallel thing with the Coloured Petri-nets. At the firing process of a transition the expression defines the weight of the current edge. Of course the guard expressions are also valid and these must return a *boolean* value.

Inside the expressions method calls can be used which compatible the type of the tokens, so in case they are syntactically correct. The methods may run very often, so performance would be a key factor while planning. To check which transition will be the next firing one we all expressions have to be executed in the entire network. That is why any number of additional actions can be defined in the network as well, and these actions execute only after a transition fires.

All of these great properties would not be so convincing, because the *Renew*'s Reference net is only a type of Coloured Petri nets so far, it is some kind of high level programming language, moreover the Java language has advantages and disadvantages too in that context. But in *Renew* synchronized connections can be created between two independent Reference nets. In order to achieve this downlink and uplink properties have to be set. With these features two transitions may be run at the very same time (but in two different Reference nets).

VI. SYSTEM DESIGN

To goal of this paper to create a new Petri net extension where Petri events are able to be set inside the net. With that events connections will be created between the simulation and a real application whatever program language is used. To reach that purpose a few things have to be done. First of all the properties of the Petri-nets have to be defined, e.g. which compatible and already existing extensions will be used, and how the network will be stored/persisted in a crossplatform and language independent way.

In an object-oriented environment the possible number of transitions are considered between object states as a finite well-defined and closed set. The simplicity and transparency are very important because the complex transitions may have become unmanageable during implementation. If the non-deterministic token gameplay is able to be wrapped in an object-oriented application, manageable but complex (non-deterministic) behavior will be got meanwhile the well-defined boundaries of the objects will not be violated.

Of course all of the extensions cannot be taken into account, on the one hand it may have contradictions, on the other hand it will not help to support the original goal. Nevertheless the following extensions will be used in the new model:

- reset arcs
- capacity limits
- inhibitor arcs
- prioritized Petri-nets

Using the *Petri Net Markup Language* would be a very good and elegant solution to persist the model, and a new *Petri Net Type Definition* would be created to store the properties of the new Petri-net extension, but it would take too much time to get enough knowledge about PNML and PNTD, so these feature remains in design phase.

There are some similarities between the new extension and the Reference-net of the *Renew* application, but in fact the purpose of the *Renew* is quite different. *Renew* creates a new High-Level Petri-net, just like the Coloured Petri-nets. It can be defined and modelled complex algorithms with these. It was not a goal to create a High-Level Petri-net and hardcode any complex algorithms, it is enough if the nets can be modelled in lower levels (so it will have lots of junctions). A simpler model will be used inside an object-oriented application for instance as part of a state machine. The token gameplay can be played and planned without the real business logic in a Petri-net simulator, but the real advantages of that extension that the same non-deterministic behavior can be used in an application which conceals the entire Petri-net token game in the background.

VII. PORTRAYAL OF THE NETWORK

One of the best choice to store a graph if its data is serialized into an XML document (Src. 1). This causes a well-defined hierarchy and a cross-platform behavior. The format of that document is clear because of the XSD scheme files and XML documents can be created with any simple text editor². The lots

²The author of this project also created a unique full-featured graphical editor for these kind of Petri-nets and its extensions, but this paper will not cover this part of the project.

```

<?xml version="1.0" encoding="utf-8"?>
<pn:PetriNetwork xmlns:pn="http://petrinetwork.hu">
  <pn:NetworkSettings>
    [...]
  </pn:NetworkSettings>
  <pn:Events>
    [...]
  </pn:Events>
  <pn:VisualSettings>
    [...]
  </pn:VisualSettings>
  <pn:VisibleSettings>
    [...]
  </pn:VisibleSettings>
  <pn:Network>
    <neit:NetworkItems xmlns:neit="http://networkitem.petrinetnetwork.hu">
      [...]
    </neit:NetworkItems>
    <neit:Edges xmlns:neit="http://networkitem.petrinetnetwork.hu">
      [...]
    </neit:Edges>
    <neit:Notes xmlns:neit="http://networkitem.petrinetnetwork.hu">
      [...]
    </neit:Notes>
  </pn:Network>
  <pn:StateHierarchy>
    <sh:States xmlns:sh="http://statehierarchy.petrinetnetwork.hu">
      [...]
    </sh:States>
    <sh:Edges xmlns:sh="http://statehierarchy.petrinetnetwork.hu">
      [...]
    </sh:Edges>
  </pn:StateHierarchy>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    [...]
  </Signature>
</pn:PetriNetwork>

```

Source 1: The frame of a *.pn.xml file

of unique XML namespaces and the separated XSD scheme files help to integrate the Petri-nets to any programming language. This is very important, because in that project the Petri-net models will be used as part of a production code (e.g.: the model is a real state machine in an application).

The root of the document is the *pn:PetriNetwork* element. Its children are the network's global settings (*pn:NetworkSettings*, *pn:VisualSettings* and *pn:VisibleSettings*), the global events (*pn:Events*), the topology (*pn:Network*) and the state hierarchy of the network (*pn:StateHierarchy*). The root element and all of its children are protected by a digital signature (*Signature* element is defined by the W3C standards's XML digital signature [9]).

1) *Global events*: According to the new extension of the Petri-net global events can be defined for the network (Src. 2). All of the events belong to the <http://event.petrinetnetwork.hu> namespace, but the names of the tags are different due to the clarity of the XSD. The type of the global events is one of the following:

- DEADLOCK
- CYCLE
- TICK

Only the name identifies an event (*pe:name*), and this is not unique intentionally. If the same name is set for more than one event, you the association can be simplified between the events and the event-handler(s). For example if all of the Petri events are handled in one single method, this feature might be a good

```

<pn:Events>
  <pe:Event pe:name="sampleevent" pe:type="DEADLOCK" xmlns:pe="http://event.petrinetnetwork.hu">
    [...]
  </pe:Event>
</pn:Events>

```

Source 2: Global Petri events

```

<pn:Network>
  <neit:NetworkItems xmlns:neit="http://networkitem.petrinetnetwork.hu">
    <i:Position ... xmlns:i="http://item.petrinetnetwork.hu">
      [...]
    </i:Position>
    <i:Transition ... xmlns:i="http://item.petrinetnetwork.hu">
      [...]
    </i:Transition>
  </neit:NetworkItems>
  <neit:Edges xmlns:neit="http://networkitem.petrinetnetwork.hu">
    <edg:Edge ... xmlns:edg="http://edge.petrinetnetwork.hu">
      [...]
    </edg:Edge>
  </neit:Edges>
  <neit:Notes xmlns:neit="http://networkitem.petrinetnetwork.hu">
    <i>Note ... xmlns:i="http://item.petrinetnetwork.hu">
      [...]
    </i>Note>
  </neit:Notes>
</pn:Network>

```

Source 3: XML model of network topology

option.

2) *Topology*: The children of the *pn:Network* element define the fundamental parts of the Petri network (Src. 3) and the related annotations (*i:Note*).

Each position can have some Petri events (*pe:ItemEvent*), which type could be the followings:

- Before activation (PREACTIVATE)
- After activation (POSTACTIVATE)

3) *State hierarchy*: In the Petri-net's XML file the entire state hierarchy guided graph can be stored. This is a separate network where each junction is a state of the original Petri-net, and two junctions are connected if the target state can be reachable from the source state, while exactly one of the transitions of the Petri-network fires. Petri events can be defined to any states (*pe:StateEvent* element), which type could be the followings:

- Before activation (PREACTIVATE)
- After activation (POSTACTIVATE)

```

<pn:StateHierarchy>
  <sh:States xmlns:sh="http://statehierarchy.petrinetnetwork.hu">
    <sv:StateVector sv:name="m0" sv:unid="31" sv:radius="20" xmlns:sv="http://statevector.petrinetnetwork.hu">
      <pf:StateOrigo pf:x="26" pf:y="27" xmlns:pf="http://pointf.petrinetnetwork.hu">
        <sv:TokenDistributions>
          <sv:Position sv:unid="0">
            <sv:Token sv:unid="26" />
          </sv:Position>
        </sv:TokenDistributions>
      </sv:StateVector>
    </sh:States>
    <sh:Edges xmlns:sh="http://statehierarchy.petrinetnetwork.hu">
      <se:StateEdge xmlns:se="http://stateedge.petrinetnetwork.hu">
        <se:StartState>31</se:StartState>
        <se:EndState>28</se:EndState>
      </se:StateEdge>
    </sh:Edges>
  </pn:StateHierarchy>

```

Source 4: State hierarchy

```
public delegate void PetriHandler (AbstractEventDrivenItem
    item, EventType eventType);
```

Source 5: PetriHandler

VIII. APPLICATION OF MODEL IN OO ENVIRONMENT

Without the Petri events the created Petri-net can be used for simulation or educational usage³ With the Petri events the model can be used in production environment, for example as the engine of a state machine. In that case the Petri-net with the new Petri events works in the background according to the rules of the Petri-networks, but it stays hidden for the user of the application.

In order to use the Petri-network in any object-oriented environment the following steps are necessary:

- The data of the network's topology have to be loaded into the memory (open PN.XML file)
- The visualization properties of the network can be omitted
- The firing process of the opened network has to be played in the application

All of these features are part of an API which is a key part of the new Petri-net extension.

IX. API OF PETRI-EVENTS

All the entities which can be held Petri-events (*Position*, *Transition* and *StateVector*) have a common base class, called *AbstractEventDrivenItem*, because there are some common responsibilities.

With the API the developer can be read the following properties of the network:

- the color of a token player
- the weight, type and junction points of an edge
- the text of any annotation (comment) and its owner
- the capacity and the list of tokens of a position
- the priority and the type of a transition
- the entire token distribution of a state vector
- the *EventTrunk* of any *AbstractEventDrivenItem* entity
- the non-visual properties of the network (*CertificateSubject*, *Description*, *FileName*, *FireRule*, *LastModificationData* and *Name*)
- the *EventTrunk* of the network
- list of all state's names in the network (*StatesName*)
- all unique (!) event's names in the network

With the *EventTrunk* instance the developer can register/bind new events. With this option the same model can be used in different applications if the model is general enough. The network has a *fire()* method. If this is performed exactly one transition of the network will be fired. It returns an instance of *FireReturn* which contains a *FireEvent* enum value. With this value the developer can find out the necessity of the next firing (possible values are the followings: INITFIRE, NORMALFIRE, RESETFIRE or DEADLOCK). In the case

```
using PetriNetworkLibrary.Model.NetworkItem;
[.]
Random rand = new Random();
PetriNetwork network = PetriNetwork.openFromXml(rand, @"
    networks\Demo.pn.xml");
```

Source 6: Open a Petri-network

of DEADLOCK the application can be stopped calling the *fire()* method, but this might be the developer's decision if RESETFIRE is gotten. In that case the network reached a state which has already happened (so this implies a kind of cycle, but because of the non-deterministic behavior this is not inevitable).

In order to build connections between the API and the application the developer has to bind the event handlers via the *PetriHandler* delegate (Src. 6). The class of the network keeps a dictionary of *Dictionary<String, PetriHandler>* where the unique Petri-event names are the keys. With the instance of Petri network the user can register any event handlers via the *bindPetriEvent()* method (or remove the connection via the *unbindPetriEvent()* method). The signature of the *PetriHandler* delegate contains an *AbstractEventDrivenItem* instance (event source) and the type of the event, so the developer might use that information in the event handler's code.

The Petri-network has to be loaded into the memory before usage. This step as simple as it is (Src. 6). This static method will throw validation exception if the XML cannot be validated by the official XSD, but it will not throw anything if the integrity of the network is inadequate⁴.

With the instance of the network and the name of the Petri event the event handlers can be registered (Src. 7). If the names of the events are unknown, the *EventsName* property of the network may be used to get a list of all available event names. In that case a general event handler has to be written or some conditional statements have to be used to separate the different use cases.

After the network is opened and the event handlers are bounded the token game has been ready to play (Src. 8).

X. RESULTS AND TESTING

At the end let us check out a lifelike but fictional example. Create a horserace game, where the accidental events which may happen at the court are driven by the Petri-net engine. If a simple random simulation is used for the game, the connection would be lost between the reality. Using a Petri-net (Fig. 2) can simulate lifelike events which are connected to each other. In the simplified race game the following rules are valid:

⁴This is an intentional behavior, the usage of the application is not wanted to restrict because of the validity of the network's certificate. But of course the network cannot be modified with the official simulator application if the validation unguaranteed.

```
network.bindPetriEvent("dummy", new PetriHandler(
    eventHandler));
```

Source 7: Register named event

³There is a unique Petri-net editor which can open the *.pn.xml files, but this is not the scope of this paper.

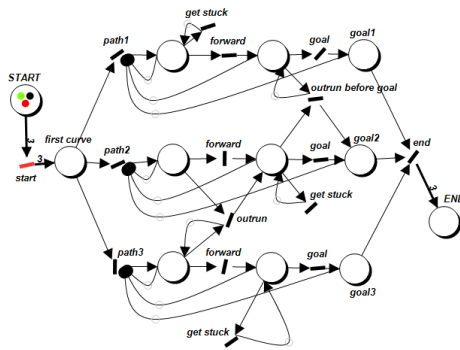


Figure 2: Horserace's Petri-net

- after the first curve a random order is going to be formed
- any horse can get stuck in anytime at the court
- there are outruns/overtakings where the tracks cross each other

The *end* transition has three input positions (*goal1*, *goal2* and *goal3*). All of these have a *POSTACTIVATE* Petri event, and the name of these events are *goal* in all cases. The final sequence will be formed by the ordered occurrence of the *goal* Petri events.

After the model was built in a simulator and it was tested with several running branches, a game has been ready to be created and the API of that Petri-net and its Petri event extension will be used (Src. 9). What was the original objective of this horserace example? Create something which is clean and declarative enough to simulate a horserace, and of course a random horse ordering is got as well.

XI. CONCLUSION

The following steps describe a typical scenario of the usage of the new Petri-net extension:

- create a low level Petri-net with the above mentioned standard extensions
- define Petri events in several parts of the network
- persist the Petri-net in a cross-platform and well-defined XML document
- load the persisted Petri-net in an object-oriented application via the API of the new extension
- associate the tokens and the object instances if needed
- create event handlers for the Petri-events which can be obtained from the model via the API of the new extension
- execute the token gameplay (typically in a background thread)

```
using PetriNetworkLibrary.Model.NetworkItem;
[.]
FireEvent fireEvent = FireEvent.INITFIRE;
FireReturn fireReturn = null;
while (!FireEvent.DEADLOCK.Equals(fireEvent))
{
    fireReturn = network.fire();
    System.Console.WriteLine(fireReturn);
    fireEvent = fireReturn.FireEvent;
}
```

Source 8: Token game

```
using PetriNetworkLibrary.Model.NetworkItem;
[.]
private PetriNetwork network;
private List<Token> result;
[.]
this.network = PetriNetwork.openFromXml(this.rand, @"
network\Horserace.pn.xml");
this.network.bindPetriEvent("goal", new PetriHandler(
    eventHandler));
[.]
private void eventHandler(AbstractEventDrivenItem item,
    EventType eventType) {
    if (item is Position) {
        Position position = (Position)item;
        List<Token> tokens = position.Tokens;
        if (tokens != null) && (tokens.Count == 1) {
            this.result.Add(tokens[0]);
        }
    }
}
[.]
private void start()
{
    this.result.Clear();
    this.network.setStartState("start");
    FireEvent fireEvent = FireEvent.INITFIRE;
    FireReturn fireReturn = null;
    while (!FireEvent.DEADLOCK.Equals(fireEvent))
    {
        fireReturn = this.network.fire();
        fireEvent = fireReturn.FireEvent;
    }
}
```

Source 9: Horserace code example

After while the steps are performed (or a loop is created in a background thread) the event handlers will be activated and our application's state will be changed according to the Petri-net rules.

REFERENCES

- [1] Wolfgang Reisig: Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies (Springer) July 2013
- [2] Kurt Jensen, Lars Michael Kristensen: Coloured Petri Nets (Springer-Verlag Berlin Heidelberg) July 2009
- [3] Eike Best, Raymond Devillers, Maciej Koutny: Petri Net Algebra (Springer-Verlag Berlin Heidelberg) 2001
- [4] Gul A. Agha, Fiorella De Cindio, Grzegorz Rozenberg: Concurrent Object-Oriented Programming and Petri Nets (Springer-Verlag Berlin Heidelberg) 2001
- [5] Tadao Murata: Petri Nets Properties, Analysis and Applications (Proceeding of the IEEE, vol. 77, no. 4) <http://www.di.univaq.it/adimarco/teaching/bioinfo15/paper.pdf> April 1989
- [6] Petri Net Markup Language <http://www.pnml.org> (last visit September 2015)
- [7] Erik Fischer: Petri Net Kernel <http://www2.informatik.hu-berlin.de/top/pnk> (last update May 2002)
- [8] Renew (University of Hamburg) <http://www.renew.de> (last update June 2016)
- [9] W3C XML Signature Syntax and Processing (Second Edition). <http://www.w3.org/TR/xmlsig-core/> (2008)