Chapter 9

Sequential Object Petri Nets and the Modeling of Multithreading Object-Oriented Programming Systems

Ivo Martiník

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/48470

1. Introduction

Sequential object Petri nets are the newly introduced class of Petri nets, whose definition is the main topics of this article; they feature certain original concepts and can be successfully used at a design, modeling and verification of multithreading object-oriented programming systems executing in highly-parallel or distributed environment. In this article basic characteristics of sequential object Petri nets are very briefly presented including possibilities in their definition of newly introduced tokens as non-empty finite recursive sequences over the set of non-negative integer numbers, functionalities of multiarcs and the mechanism of the firing of transitions. These properties significantly increase modeling capabilities of this class of Petri nets at the modeling of multithreading object-oriented programming systems. Sequential object Petri nets can be used also in the area of recursive algorithms modeling and they are also the initial step to explicitly represent paradigms of functional programming. The fusion of object-oriented and functional programming enables to express new kinds of programming patterns and component abstractions.

The theory of sequential object Petri nets proceeds from the theories of various types of Petri nets, starting with Place/Transition nets (Diaz, 2009) and their sub-classes, followed by High-Level nets (Jensen & Rozenberg, 1991), (Reisig, 2009) such as Predicate-Transition nets and Coloured nets (Jensen & Kristensen, 2009), enabling to model apart from the management structure of the system even data processing, and in connection with modeling of object-oriented programming systems it is Object nets (Agha et. al., 2001), (Köhler & Rölke, 2007), which are being studied lately. But practical usability of Petri nets (in their original form) in the role of the parallel programming language is mainly impeded by the static nature of their structure. They are missing standard mechanisms for description of methods alone, programming modules, classes, data types, hierarchical structures, etc.

Positive characteristics of Petri nets demonstrate only in not too much large-scale modules at high abstraction level. That is why Petri nets are often understood as the theoretical abstract module only, whose applicability for design, analysis and verification of extensive programming systems is limited. Therefore, this article briefly describes a special class of sequential object Petri nets and its possibilities of multithreading object-oriented programming systems modeling, which eliminates the stated shortcomings required for design, analysis and verification of these systems in several directions.

This chapter is arranged into the following sections: in the section 2 is described the term of the sequence over the finite set and its properties which denominates the class of the sequential object Petri nets; section 3 explains the base term of this chapter, ie. sequential object Petri net and its properties; section 4 explains in details implementation of the mechanism of firing of transitions in a sequential object Petri net; section 5 then discusses the area of object-oriented programming systems and their representation by the sequential object Petri nets; section 6 explains the example of simple class hierarchy represented by the sequential object Petri net and it is inspired by the several base classes of the Java programming language class hierarchy. Finally, the section 7 gives the conclusions of the research to conclude the chapter.

2. Sequences and their properties

Prior to the formal introduction of the term of sequential object Petri net, we present the definition of sequence over the finite set from which the denomination of this class of Petri nets has been derived. $\boldsymbol{N}$ denotes the set of all natural numbers, $\boldsymbol{N_0}$ the set of all non-negative integer numbers, $P(A)$ denotes the family of all the subsets of given set A.

Let A be a non-empty set. By the (non-empty finite) **sequence** σ over the set A we understand a mapping σ: {1, 2, …, n} $\rightarrow A$, where $n \in \boldsymbol{N}$. Mapping ε: $\emptyset \rightarrow A$ is called the **empty sequence** over the set A. We usually represent the sequence σ: {1, 2, …, n} $\rightarrow A$ by the notation $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ of the elements of the set A, where $a_i = \sigma(i)$ for $1 \leq i \leq n$. We also consider any element of the set A as the sequence over the set A, ie. mapping σ:{1} $\rightarrow A$. Empty sequence ε: $\emptyset \rightarrow A$ over the set A we usually represent by the notation $\varepsilon = \langle\rangle$. We denote the set of all **finite non-empty** sequences over the set A by the notation A^{SQ}, the set of all **finite** (and possible empty) sequences over the set A by the notation A^{ESQ}.

Note also, that the set A can be any non-empty set, which means that it can be also the non-empty set of sequences over some non-empty set B, ie. $A = B^{ESQ}$. Thus member of the sequence over the set B^{ESQ} can be then another sequence over the set B. This fact thus also similarly allows sequences over the sets $(B^{ESQ})^{ESQ}$, $((B^{ESQ})^{ESQ})^{ESQ}$, etc. The term of sequence over some non-empty set has thus **recursive character** and every such sequence can consists from **subsequences** consisting from subsequences etc. We denote the set of all **finite non-empty** sequences without the empty subsequences over the union of sets A, A^{SQ}, $(A^{SQ})^{SQ}$, $((A^{SQ})^{SQ})^{SQ}$, … by the notation A^{RQ}, the set of all **finite** (and possible empty or with empty subsequences) sequences over the union of sets A, A^{ESQ}, $(A^{ESQ})^{ESQ}$, $((A^{ESQ})^{ESQ})^{ESQ}$, … by the notation A^{ERQ}.

The **length of the sequence** $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$, where $\sigma \in A_{ERQ}$, $n \in \boldsymbol{N}$, is equal to the natural number n, the length of the empty sequence ε is equal to the number 0. The length of the sequence σ we represent by the notation $length(\sigma)$, or $@\sigma$, the set of all the elements of the sequence σ we represent by the notation $elem(\sigma)$, ie. $elem(\sigma) = \{a_i \mid a_i = \sigma(i) \text{ for } 1 \le i \le n\}$, $elem(\varepsilon) = \varnothing$. The **subsequences of the sequence** σ is the mapping $subsq: A_{ERQ} \to P(A_{ERQ})$, such that for $\forall i, 1 \le i \le n$: $((\sigma \in subsq(\sigma)) \vee (((a_i == \varepsilon) \vee (a_i \in A)) \Rightarrow a_i \in subsq(\sigma)) \vee (((a_i != \varepsilon) \wedge (a_i \notin A)) \Rightarrow ((a_i \in subsq(\sigma)) \wedge (subsq(a_i) \in subsq(\sigma)))$. The **members of the sequence** σ is the mapping $memb: A_{ERQ} \to P(A)$, so that $memb(\sigma) = \{a \mid (a \in subsq(\sigma)) \wedge (a \in A)\}$, $memb(\varepsilon) = \varnothing$.

If $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ and $\tau = \langle b_1, b_2, \ldots, b_m \rangle$ are the finite sequences, where $\sigma \in A_{ERQ}$, $\tau \in A_{ERQ}$, $n \in \boldsymbol{N}$, $m \in \boldsymbol{N}$, then by the **concatenation of the sequences** σ and τ, denoted by $\sigma\tau$, we understand the finite sequence $\sigma\tau = \langle a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m \rangle$ and its length is equal to $n + m$. We say, that the sequences σ and τ are **equal**, denoted by $\sigma == \tau$, if the following is simultaneously true: $(n = m) \wedge (\forall i, 1 \le i \le n$: $(((a_i = \varepsilon) \wedge (b_i = \varepsilon)) \vee ((a_i \in A) \wedge (b_i \in A) \wedge (a_i = b_i)) \vee ((a_i \ne \varepsilon) \wedge (b_i \ne \varepsilon) \wedge (a_i \notin A) \wedge (b_i \notin A) \wedge (a_i == b_i))))$.

If, for instance, $\tau \in A_{ERQ}$, $\tau = \langle\langle a, \langle a, b\rangle\rangle, \langle a, \langle c\rangle, b\rangle, \langle\rangle\rangle$, then $length(\tau) = 3$, $elem(\tau) = \{\langle a, \langle a, b\rangle\rangle, \langle a, \langle c\rangle, b\rangle, \langle\rangle\}$, $subsq(\tau) = \{\langle\langle a, \langle a, b\rangle\rangle, \langle a, \langle c\rangle, b\rangle, \langle\rangle\rangle, \langle a, \langle a, b\rangle\rangle, \langle a, \langle c\rangle, b\rangle, \langle a, b\rangle, \langle a\rangle, \langle b\rangle, \langle c\rangle, \langle\rangle\}$ and $memb(\tau) = \{a, b, c\}$.

When operating with sequences, notation in the form of $n^*(\sigma)$ can be utilized, where $\sigma \in A_{ERQ}$, $n \in \boldsymbol{N}$. Informally, that notation expresses sequence consisting of n concatenations of the sequence σ. If, for example $A = \boldsymbol{N_0}$, $\sigma = \langle 1, 2\rangle$, then notation $3^*(\sigma)$ represents the sequence $3^*(\langle 1, 2\rangle) == \langle 1, 2\rangle\langle 1, 2\rangle\langle 1, 2\rangle == \langle 1, 2, 1, 2, 1, 2\rangle$.

Multiset M over a non-empty set S is a function $m: S \to \boldsymbol{N_0}$. By the non-negative number $m(a) \in \boldsymbol{N_0}$, $a \in S$, we denote the number of occurrences of the element a in the multiset m. We usually represent the multiset m by the formal sum $\sum_{a \in S} m(a)`a$. By S_{MS} we denote the set of all non-empty multisets over the set S, by S_{EMS} we denote the set of all multisets over the set S.

IDENT denotes the set of all **identifiers** and it is understood to be a set of non-empty finite sequences over the set of all letters of the selected national alphabet and the set of all decadic digits that starts with a letter. Identifiers are recorded in a way usual for standard programming languages. Examples of correctly formed identifiers for example involve the *thread*, *var22*, $\alpha\beta\chi\delta$, etc. On the contrary, for example sequences *2main*, *first goal*, *_input*, are not identifiers. Moreover is it true, that if $ID_1, ID_2, \ldots, ID_n \in \boldsymbol{IDENT}$, where $n \in \boldsymbol{N}$, $n > 1$, then we call the sequence in the form $ID_1.ID_2. \ldots .ID_n$ **compound identifier** (i.e. for example the sequence *Main.Thread.Variable1* is a compound identifier). ***#IDENT*** set is understood to be the set of all non-empty finite sequences in the form #A, where $A \in \boldsymbol{IDENT}$. Then, elements of ***#IDENT*** set for example include sequences *#thread*, *#var22*, $\#\alpha\beta\chi\delta$, etc.

The **s**et $(\boldsymbol{N_0})_{RQ}$ we will denote by the symbol ***Tokens***. The set ***ArcSeq*** (*arc sequences*) is defined by the following:

i. if $x \in (\boldsymbol{IDENT} \cup \boldsymbol{\#IDENT})$, then $\langle x\rangle \in \boldsymbol{ArcSeq}$,
ii. if $x \in \boldsymbol{Tokens}$, then $x \in \boldsymbol{ArcSeq}$,

iii. if $x \in \boldsymbol{ArcSeq}$, then $<x> \in \boldsymbol{ArcSeq}$ and also $<length(x)> \in \boldsymbol{ArcSeq}$,
iv. if $x \in \boldsymbol{ArcSeq}$ and $y \in \boldsymbol{ArcSeq}$, then $xy \in \boldsymbol{ArcSeq}$,
v. if $n \in (\boldsymbol{IDENT} \cup \boldsymbol{N})$ and $x \in \boldsymbol{ArcSeq}$, then $n^*(x) \in \boldsymbol{ArcSeq}$.

The elements of ***Tokens*** set for example involve sequences <1>, <22, <0, 0>>, <<3, 2>, <4, <7, 8>>>, etc.). The set of arc sequences ***ArcSeq*** is, informally said, the set of **non-empty final recursive sequences** over the set ($\boldsymbol{IDENT} \cup \boldsymbol{\#IDENT} \cup \boldsymbol{Tokens}$) which do not contain empty subsequences and which can contain as their members even selected operations over those recursive sequences, between which is the determination of the recursive sequence length and concatenation of recursive sequences. So examples of elements of the set ***ArcSeq*** can be sequences <a, b, 1>, <#s, @(s), <1, <2, 3>>>, <a, thread, a*(<thread>), <1, 0>>, etc.

Let $AS \in \boldsymbol{ArcSeq}$, $AS = <a_1, a_2, \ldots, a_n>$, where $n \in \boldsymbol{N}$. Then the mapping *variables*: $\boldsymbol{ArcSeq} \rightarrow P(\boldsymbol{IDENT} \cup \boldsymbol{\#IDENT} \cup \boldsymbol{N_0})$ is defined so that for $\forall AS \in \boldsymbol{ArcSeq}\ \forall i, 1 \leq i \leq n$:

i. if $a_i \in (\boldsymbol{IDENT} \cup \boldsymbol{\#IDENT})$, then $a_i \in variables(AS)$,
ii. if $a_i \in \boldsymbol{Tokens}$, then $memb(a_i) \subseteq variables(AS)$,
iii. if $a_i = <x>$, where $x \in \boldsymbol{ArcSeq}$, then $variables(x) \subseteq variables(AS)$,
iv. if $a_i = length(x)$, where $x \in \boldsymbol{ArcSeq}$, then $variables(x) \subseteq variables(AS)$,
v. if $a_i = xy$, where $x \in \boldsymbol{ArcSeq}$ and $y \in \boldsymbol{ArcSeq}$, then $(variables(x) \subseteq variables(AS)) \wedge (variables(y) \subseteq variables(AS))$,
vi. if $a_i = n^*(x)$, where $n \in (\boldsymbol{IDENT} \cup \boldsymbol{N})$ and $x \in \boldsymbol{ArcSeq}$, then $(n \in variables(AS)) \wedge (variables(x) \subseteq variables(AS))$.

Thus mapping *variables* assigns to each arc sequence $AS \in \boldsymbol{ArcSeq}$, $AS = <a_1, a_2, \ldots, a_n>$, where $n \in \boldsymbol{N}$, the set of members from the sets ***IDENT***, ***#IDENT*** a $\boldsymbol{N_0}$ contained in it. The set of *variables*(*AS*) associated with a particular arc *AS* will be identified in the text by the term **variables of the arc sequence *AS***. So if for example *AS* = <a, thread, a*(<thread>), <1, 0>>, then *variables*(*AS*) = {a, thread, 1, 0}.

3. Sequential object Petri nets and their properties

Sequential Object Petri Net is an ordered pair $SOPN = (\Sigma, PN)$, where

i. Σ is a finite non-empty set of **pages,**
ii. *PN* is a **page number function**, $PN: \Sigma \rightarrow \boldsymbol{N}$, that is injective.

By elements of the finite non-empty set Σ of pages we routinely mark identifiers from the set ***IDENT***. Injective function *PN* of numbering of pages of the net assigns to each page of sequential object Petri net *SOPN* the unique natural number within the net.

Let $SOPN = (\Sigma, PN)$ is a sequential object Petri net. **Page** of the sequential object Petri net *SOPN* is an ordered touple $PG = (P, IP, OP, T, A, MA, IOPN, AF, MAF, TP, IPF, OPF, SP, IF)$, $PG \in \Sigma$, where:

i. *P* is a finite set of **places**,
ii. *IP* is a finite set of **input places**, $P \cap IP = \varnothing$,

iii. OP is a finite set of **output places**, $P \cap OP = \emptyset$,
iv. T is a finite set of **transitions**, $(P \cup IP \cup OP) \cap T = \emptyset$,
v. A is a finite set of **arcs**, $A \subseteq ((P \cup IP) \times T) \cup (T \times (P \cup OP))$,
vi. MA is a finite set of **multiarcs**, $MA \subseteq ((P \cup IP) \times T) \cup (T \times (P \cup OP))$, $A \cap MA = \emptyset$,
vii. $IOPN$ is a function of **input and output place numbers**, $IOPN$: $(IP \cup OP) \rightarrow \mathbf{N}$, that is injective,
viii. AF is an **arc function**, AF: $(A \cup MA) \rightarrow \boldsymbol{ArcSeq}$,
ix. MAF is a **multiarc function**, MAF: $MA \rightarrow \boldsymbol{ArcSeq}$,
x. TP is a function of **transition priorities**, TP: $T \rightarrow \mathbf{N}$,
xi. IPF is an **input place function** of multiarcs, IPF: $(T \times (P \cup OP)) \rightarrow AIP$, where $(T \times (P \cup OP)) \subseteq MA$, $AIP = \{p \mid \exists \gamma \in \Sigma : p \in IP \in \gamma\}$,
xii. OPF is an **output place function** of multiarcs, OPF: $((P \cup IP) \times T) \rightarrow AOP$, where $((P \cup IP) \times T) \subseteq MA$, $AOP = \{p \mid \exists \gamma \in \Sigma : p \in OP \in \gamma\}$,
xiii. SP is a finite set of **subpages**, $SP \subseteq \Sigma$,
xiv. IF is an **initialization function**, IF: $(P \cup IP \cup OP) \rightarrow \boldsymbol{Tokens}$EMS.

The finite set of places P is used for expressing of conditions of a modeled programming system and in the net layout we notate them with circles. IP is a finite set of input places of the net page representing its input interface. Additionally, no input place of the net page can be identical with any of its places. Input places are represented in the page layout with circles of highlighted upper semicircle. Then, OP is a finite set of output places of the net page representing its output interface. Additionally, no output place of the net page can be identical with any of its places. The definition admits even such possibility that the selected input place is identical with any of output places of the given net page. Output places are represented in the net page layout with circles of highlighted lower semicircle.

Likewise the finite set of transitions T is used for describing events in the programming system and we notate them with rectangles. That set is disjoint with the set of places P of the given net page. A is the finite set of arcs being principally oriented while connecting the place with transition or transition with place and in the layout of net we represent them by oriented arrows drawn in full line. It is worth considering that none of output arcs of any transition can be associated with any input place of the net page, and none of input arcs of any transition can be associated with any output place of the net page. MA is finite set of multiarcs, newly introduced type of arc in sequential object Petri nets. Functionalities of multiarc are used for the modeling of synchronous and asynchronous calling of methods in the given programming system and they follow the principles of the multiarcs in the bi-relational P/T Petri nets (Martiník, 2011). Multiarcs are represented in layouts of the net with oriented arrows drawn with dash line. The set of arcs of the given page is disjoint with the set of its multiarcs, hence it is not allowed the existence of the ordered pair (*place, transition*) or (*transition, place*) connected by both types of oriented arcs.

$IOPN$ function of the identification of input and output places of the net page assigns to each input and output place of the particular net page **unique** natural number which is

used at the implementation of mechanism of execution of transitions associated with multiarcs of the net page. With each arc or multiarc of the net page is associated the value of its arc function *AF*, which assigns to each arc or multiarc (one) **arc sequence**, i.e. the element of ***ArcSeq*** set. With each multiarc of the net page is additionally associated the value of its multiarc function *MAF*, which assigns to each such multiarc (one) **arc sequence**, i.e. the element of ***ArcSeq*** set. The layout of the net page shows values of *AF* and *MAF* functions associated with particular multiarc in the form *AF* | *MAF*. With all transitions of the net page are associated values of their functions of transition priorities *TP* assigning each transition with (the only) value of such transition priority, which is the value of a certain natural number. If the value of function of transition priorities is not explicitly indicated in the net layout with the particular transition, we assign it to the value of natural number 1.

The input place function of multiarcs *IPF* assigns each multiarc of the net page connecting ordered pair (*transition*, *place*) a certain **input place of the selected net page**. The definition admits even the possibility of assigning the selected multiarc of the particular net page with some of the input places of the same net page (ie. it is allowed to model recursive methods). The particular input place p of the selected net page γ ($\gamma \in \Sigma$) is in the layout of network identified by ordered pair of natural numbers ($PN(\gamma)$, $IOPN(\gamma.p)$), where the first member of the pair indicates the value of page number function *PN* and the second member of the pair identifies the selected input place p on the net page γ with the particular value of function *IOPN*. We present this ordered pair in layouts of net pages usually in the form $PN(\gamma).IOPN(\gamma.p)$. The output place function of multiarcs *OPF* assigns each multiarc of the net page connecting the ordered pair (*place*, *transition*) with a certain **output place of the selected net page.** The definition again admits the possibility of assigning to the selected multiarc of the given net page some of the output places of the same net page. The particular output place p of the selected net page γ is marked in a similar way as in case of the function *IPF*.

A part of each net page can be the finite set *SP* of its subpages, which are by themselves the net pages (i.e. elements of the set Σ). Initialization function *IF* assigns each place including input and output places of the net page with a **multiset of tokens.** That function is also identified in literature as M_0. We routinely mark identifiers from the set ***IDENT*** by elements of the set of places, input places, output places and transitions.

Figure 1 shows the sequential object Petri net $SOPN = (\Sigma, PN)$, where Σ = {**Main**, **Sub**}, *PN* = {(**Main,** 1), (**Sub,** 2)}. Net page of this sequential object Petri net **Main =** (*P, IP, OP, T, A, MA, IOPN, AF, MAF, AP, IPF, OPF, SP, IF*), where *P* = {**P1**, **P2**}, *IP* = {**in**}, *OP* = {**In**}, *T* = {**T1**, **T2**}, *A* = {(**in**, **T1**), (**P1**, **T1**), (**T2**, **In**)}, *MA* = {(**T1**, **P2**), (**P2**, **T2**)}, *IOPN* = {(**in**, 1), (**In**, 2)}, *AF* = {((**in**, **T1**), <a>), ((**P1**, **T1**), <b, 1>), ((**T1**, **P2**), <a>), ((**P2**, **T2**), <a>), ((**T2**, **In**), <b>)}, *MAF* = {((**P2**, **T2**), <b>), ((**T1**, **P2**), <b,1>)}, *TP* = {(**T1**, 1), (**T2**, 1)}, *IPF* = {((**T1**, **P2**), (2.1))}, *OPF* = {((**P2**, **T2**), (2.2))}, *SP* = ∅, *IF* = {(**in**, 1`<1>), (**P1**, ∅), (**P2**, ∅), (**In**, ∅)}. Net page **Sub =** (*P, IP, OP, T, A, MA, IOPN, AF, MAF, AP, IPF, OPF, SP, IF*), where *P* = ∅, *IP* = {**start**}, *OP* = {**Start**}, *T* = {**T1**}, *A* = {(**start**, **T1**), (**T1**, **Start**)}, *MA* = ∅, *IOPN* = {(**start**, 1), (**Start**, 2)}, *AF* = {((**start**, **T1**), <c, 1>), ((**T1**, **Start**), <c>)}, *MAF* = ∅, *TP* = {(**T1**, 1)}, *IPF* = ∅, *OPF* = ∅, *SP* = ∅, *IF* = {(**start**, ∅), (**Start**, 1`<3>)}.

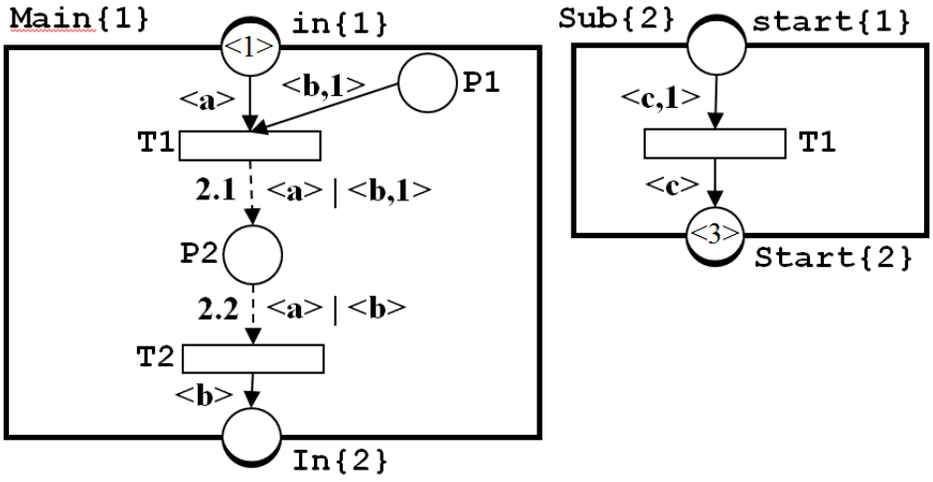

Figure 1. Sequential object Petri net

If **page_identifier** is the identifier of the selected net page and **element_identifier** is the identifier of a place, input place, output place or transition of the net page, we call the compound identifier in the form **page_identifier.element_identifier** so called **distinguished identifier of the element of net page**, which uniquely identifies it within the given sequential object Petri net. Designs of distinguished identifiers of subpages of the net and of its elements can be also executed for cases of sub subpages of the net pages, etc.

For the sake of better transparency we will not indicate in layouts of nets explicit values of page number function *PN*, and explicit values of function of input and output place numbers *IOPN* of identification of input and output places of individual net pages any more. Moreover, we will not indicate values of functions *IPF* and *OPF* of particular multiarcs in the form of **a.b** pair of natural numbers, but in the form of the pair of identifiers **page.ioplace**, where **page** $\in \Sigma$ is the net page and **ioplace** $\in IP \in$ **page**, perhaps **ioplace** $\in OP \in$ **page** is particular input, or output place of the net page while it holds that **a** = *PN*(**page**) and **b** = *IOPN*(**page.ioplace**).

Layouts of sequential object Petri nets are usually further adjusted in the sense of notations of declarations of headings of methods and their calling within the text of the program, similarly as shown in Figure 2. Here, identifiers of input and output places of the net pages are complemented by (informative) notation of the shape of tokens, which are accepted by those input and output places (see the notation of the input places **Main.in<a>, Sub.start<c, 1>** and of the output places **Main.In<b>, Sub.Start<c>**). We will not record values of functions *AF* and *MAF* in the form of the ordered pair separated by | line any more. The value of the arc function *AF* is indicated separately and the value of the multiarc function *MAF* is indicated behind the value of the input place function *IPF* on the net page, perhaps with a value of the output place function *OPF* of particular multiarc (see notation **Sub.start<b, 1>** and **Sub.Start<b>** of the net page **Main**) in the sense of declaration of calling of methods with the entry of values of input parameters or output values of these methods.

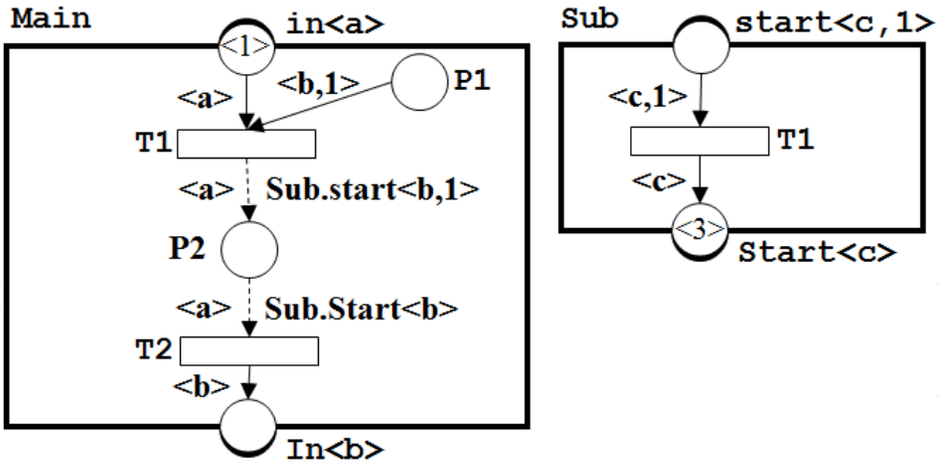

Figure 2. Sequential object Petri net

Let $SOPN = (\sum, PN)$ is a sequential object Petri net, $PG \in \sum$ is its net page. By the marking M of the net page PG we understand the mapping M: $(P \cup IP \cup OP) \rightarrow \boldsymbol{Tokens}^{\text{EMS}}$, where $P \in PG$, $IP \in PG$, $OP \in PG$. By the marking of the net $SOPN$ we understand the marking of all its net pages.

Let $SOPN = (\sum, PN)$ is a sequential object Petri net, $PG \in \sum$ is its page. Then:

i. by $InputArcs(x)$ we denote the set of all input arcs of selected place, output place or transition x, ie. $\forall\ x \in (P \cup OP \cup T) \in PG$: $InputArcs(x) = \{a \in A \mid \exists\ y \in (P \cup IP \cup T)$: $a = (y, x)\}$.
ii. by $InputMultiArcs(x)$ we denote the set of all input multiarcs of selected place, output place or transition x, ie. $\forall\ x \in (P \cup OP \cup T) \in PG$: $InputMultiArcs(x) = \{a \in MA \mid \exists\ y \in (P \cup IP \cup T)$: $a = (y, x)\}$.
iii. by $InputNodes(x)$ we denote the set of all input nodes of selected place, output place or transition x, ie. $\forall\ x \in (P \cup OP \cup T) \in PG$: $InputNodes(x) = \{y \in (P \cup IP \cup T) \mid \exists\ a \in (A \cup MA)$: $a = (y, x)\}$. We denote the set $InputNodes(x)$ by $\bullet x$.
iv. by $OutputNodes(x)$ we denote the set of all output nodes of selected place, input place or transition x, ie. $\forall\ x \in (P \cup IP \cup T) \in PG$: $OutputNodes(x) = \{y \in (P \cup OP \cup T) \mid \exists\ a \in (A \cup MA)$: $a = (x, y)\}$. We denote the set $OutputNodes(x)$ by $x\bullet$.
v. by $TransitionInputVariables(x)$ we denote the set of all variables included in the values of arc functions AF (resp. multiarc functions MAF) of all the input arcs and multiarcs of the transition x, ie. $\forall\ x \in T \in PG$: $TransitionInputVariables(x) = \{v \mid ((v \in variables(AF(y)) \vee (v \in variables(MAF(y))) \wedge ((y \in InputArcs(x)) \vee (y \in InputMultiArcs(x)))\}$.

4. Firing of transitions in sequential object Petri nets

An important term used at the implementation of the mechanism of firing of transitions in a sequential object Petri net is the term of **binding of the arc sequence** contained in the value of AF, or MAF, function of the particular arc, or multiarc, to the token found at a certain marking of the net page in the place associated with this arc (for short, we will refer to that in the text also as **binding of token**).

Let $T \in \boldsymbol{Tokens}$, $T = <t_1, t_2, \ldots, t_n>$, where $n \in \boldsymbol{N}$, $AS \in \boldsymbol{ArcSeq}$. We denote, that there exists **input binding of the arc sequence** AS to the token T, if there exists mapping $InputBind$: $AS \rightarrow T$ that satifies the following:

i. If $AS = <a_1, a_2, \ldots, a_n>$, where $a_1, a_2, \ldots, a_n \in (\boldsymbol{IDENT} \cup \boldsymbol{\#IDENT} \cup \boldsymbol{N_0})$, then:
a. if $a_i \in (\boldsymbol{IDENT} \cup \boldsymbol{\#IDENT})$ for $1 \le i \le n$, then $InputBind(a_i) = t_i$,
b. if $a_i \in \boldsymbol{N_0}$ for $1 \le i \le n$, then $(InputBind(a_i) = t_i) \wedge (a_i = t_i)$,
c. $\forall$ u $\in variables(AS) \forall\ v \in variables(AS)$: $((u = v) \Rightarrow (InputBind(u) == InputBind(v)))$.
ii. If $AS = <a_1, a_2, \ldots, a_{k-1}, a_k, a_{k+1}, \ldots, a_m>$, where $m < n$, $1 \le k \le m$, $a_k \in \boldsymbol{\#IDENT}$, $a_1, \ldots, a_{k-1}, a_{k+1}, \ldots, a_m \in (\boldsymbol{IDENT} \cup \boldsymbol{N_0})$, $m \in \boldsymbol{N}$, then:
a. $InputBind(a_k) = <t_k, t_{k+1}, \ldots, t_{k+n-m}>$,
b. if $a_i \in \boldsymbol{IDENT}$ for $1 \le i \le k$-1, then $InputBind(a_i) = t_i$,
c. if $a_i \in \boldsymbol{IDENT}$ for k+1 $\le i \le m$, then $InputBind(a_i) = t_{n-m+i}$,
d. if $a_i \in \boldsymbol{N_0}$ for $1 \le i \le k$-1, then $(InputBind(a_i) = t_i) \wedge (a_i = t_i)$,
e. if $a_i \in \boldsymbol{N_0}$ for k+1 $\le i \le m$, then $(InputBind(a_i) = t_{n-m+i}) \wedge (a_i = t_{n-m+i})$,
f. $\forall\ u \in variables(AS) \forall\ v \in variables(AS)$: $((u = v) \Rightarrow (InputBind(u) == InputBind(v)))$.
iii. In other case the mapping $InputBind$ is not defined.

Then, input binding of the arc sequence AS to the token T via mapping $InputBind$: $AS \rightarrow T$ can be successfully realized in the following two cases:

i. Arc sequence is in the form $AS = <a_1, a_2, \ldots, a_n>$, where $n \in \boldsymbol{N}$, i.e. it holds that $\boldsymbol{length(AS) = length(T) = n}$, while generally **more than one** of elements $a_1, a_2, \ldots, a_n$ of that sequence can be the element of the set $\boldsymbol{\#IDENT}$. Then, at the input binding of the arc sequence AS to the token T we execute, informally said, binding of mutually corresponding elements of sequences according to their order. If an element a_i of the arc sequence AS is a nonnegative integer, then such element must be bound to single-element t_i of token T, where $1 \le i \le n$, which is also nonnegative integer and the value of both those numbers must be identical. If u and v are two identical variables of the arc sequence AS, then the values of elements of the token T bound to them must be identical. Figure 3 shows a very simple example of input binding of the arc sequence AS = <1, a, a, #c, 3> to the token T = <1, 10, 10, 2, 3>:

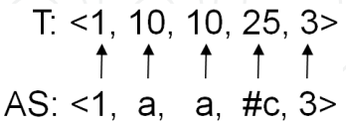

Figure 3. Binding of arc sequence to token

ii. Arc sequence is in the form $AS = <a_1, a_2, \ldots, a_{k-1}, a_k, a_{k+1}, \ldots, a_m>$, where $m \in \boldsymbol{N}$, holds that $\boldsymbol{length(AS) < length(T)}$, i.e. $m < n$, and at the same time just a single element $a_k \in \boldsymbol{\#IDENT}$, where $1 \le k \le m$. Then, that only element a_k is bound to the **sequence** $<t_k, t_{k+1}, \ldots, t_{k+n-m}>$ of elements of the token T. In binding of other elements of the sequence AS the same rules hold as it was in the case of (i). An example of that type of binding of the arc sequence AS = <x, #y, 5, z> (and thus element $a_2 \in \boldsymbol{\#IDENT}$) to the token T = <4, 8, 10, 2, 5, 19> is shown in Figure 4.

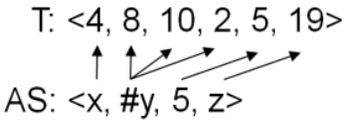

Figure 4. Binding of arc sequence to token

Next examples of the arc sequences binding to tokens in a sequential object Petri net involve:

- arc sequence <a, a, 1> can be successfully bound to token <2, 2, 1>, where *InputBind*(a) = 2, *InputBind*(1) = 1,
- arc sequence <a, a, 1> cannot be successfully bound to token <1, 2, 3> (it would hold that *InputBind*(a) = 1 and *InputBind*(a) = 2),
- arc sequence <#x> can be successfully bound to token <1, 2, 3>, where *InputBind*(#x) = <1, 2, 3>,
- arc sequence <x, #y> can be successfully bound to token <1, 2, 3>, where *InputBind*(x) = 1, *InputBind*(#y) = <2, 3>,
- arc sequence <x, y> can be successfully bound to token <<1, 2>, <3, 3>>, where *InputBind*(x) = <1, 2>, *InputBind*(y) = <3, 3>,
- arc sequence <x, #y> can be successfully bound to token <<1, 2>, <3, 3>, 4>, where *InputBind*(x) = <1, 2>, *InputBind*(#y) = <<3, 3>, 4>.

Let $SOPN = (\sum, PN)$ is a sequential object Petri net, $PG \in \sum$ is a net page, $t \in T$ is a transition of the net page PG, $p \in {\bullet}t \in (P \cup IP)$ is a place or input place of the net page PG, $q \in t{\bullet} \in (P \cup OP)$ is a place or output place of the net page PG, M is a marking of the net $SOPN$. Transition t is ***enabled*** in the marking M of the net $SOPN$, if:

i. $\forall\ (p, t) \in (InputArcs(t) \cup InputMultiArcs(t))\ \exists\ InputBind: AF(p, t) \rightarrow e$, where $e \in M(p)$,
ii. $\forall\ (p, t) \in InputMultiArcs(t)\ \exists\ InputBind: MAF(p, t) \rightarrow e$, where $e \in M(OPF(p, t))$,
iii. $\forall\ u \in TransitionInputVariables(t)\ \forall\ v \in TransitionInputVariables(t)$:

$$((u = v) \Rightarrow (InputBind(u) == InputBind(v))).$$

If transition t is enabled in the marking M of the net $SOPN$, we record that fact symbolically in the form of t ***en*** M.

Let $AS = <a_1, a_2, \ldots, a_n> \in \boldsymbol{ArcSeq}$, $n \in \boldsymbol{N}$. If transition t is ***enabled*** in the marking M of the net $SOPN$, then we say, that there exists partial mapping $OutputBind$: $\boldsymbol{ArcSeq} \rightarrow \boldsymbol{Tokens}$, if

$$OutputBind(AS) = OB(a_1)\ OB(a_2)\ \ldots\ OB(a_n)$$

where OB: $\boldsymbol{ArcSeq} \rightarrow \boldsymbol{Tokens}$ and $\forall i$, $1 \le i \le n$:

a. $OB(a_i) = <InputBind(a_i)>$, if $a_i \in \boldsymbol{IDENT}$,
b. $OB(a_i) = InputBind(a_i)$, if $a_i \in \#\boldsymbol{IDENT}$,
c. $OB(a_i) = a_i$, if $a_i \in \mathbf{Tokens}$,
d. $OB(a_i) = <OB(x)>$, if $a_i = <x>$, where $x \in \boldsymbol{ArcSeq}$,
e. $OB(a_i) = <@(OB(x))>$, if $a_i = @(x)$, where $x \in \boldsymbol{ArcSeq}$,
f. $OB(a_i) = OB(b_1)\ OB(b_2)\ \ldots\ OB(b_k)$, if $a_i = b_1 b_2 \ldots b_k$, where $b_1, b_2, \ldots, b_k \in \boldsymbol{ArcSeq}$, $k \in \boldsymbol{N}$,
g. $OB(a_i) = OB(b)^*(OB(x))$, if $a_i = b^*(x)$, where $b \in (\boldsymbol{IDENT} \cup \boldsymbol{N_0})$, $x \in \boldsymbol{ArcSeq}$.

Thus transition *t* on the net page *PG* of the net *SOPN* is ***enabled***, if the following is satisfied:

i. for all the input arcs (*p*, *t*), resp. input multiarcs (*p*, *t*), of the transition *t* there exists input binding of the value of the arc function *AF*(*p*, *t*) to some token *e* in the place *p* of the marking *M*,
ii. for all the input multiarcs (*p*, *t*) of the transition *t* there exists input binding of the value of the multiarc function *MAF*(*p*, *t*) to some token *e* in the output place of the net page that is given by the value of the output place function *OPF* of the multiarc (*p*, *t*) in the net marking *M*,
iii. if *u* and *v* are two equal variables of the set *TransitionInputVariables*(*t*), then the values of elements (resp. subsequences) bound by them in the frame of mapping *InputBind* must be equal.

Figure 5 shows the fragment of sequential object Petri net in its marking *M* and the construction of the mapping *InputBind*: *AF*(**P1**, **T1**) → <2, 0>, where <2, 0> ∈ *M*(**P1**) and *InputBind*: *AF*(**P2**, **T1**) → <1, 1, 1>, where <1, 1, 1> ∈ *M*(**P2**). It is easily to find that transition **T1** is enabled.

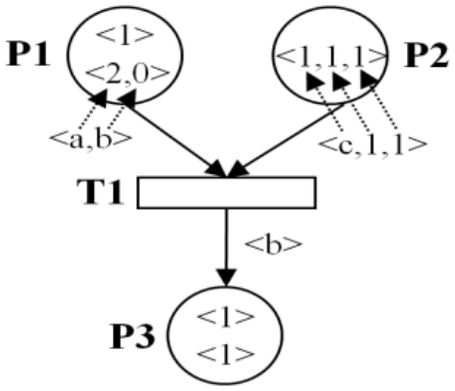

Figure 5. Mapping *InputBind* in sequential object Petri net

Partial mapping *OutputBind*: ***ArcSeq*** → ***Tokens*** is for the given transition *t* of the net page realized only in case that the transition *t* is enabled in the marking *M* of the net. Hence, partial mapping *OutputBind* assigns the selected arc sequence *AS* the **token**, being the element of the set of all ***Tokens*** (i.e. that token is not generally located in any of places of the net *SOPN* in its current marking *M*). The definition assumes that arc sequence *AS* is generally in the form $AS = \langle a_1, a_2, \ldots, a_n \rangle$, $n \in \mathbf{N}$. The value $OutputBind(\langle a_1, a_2, \ldots, a_n \rangle)$, which is generally the element of the set ***Tokens***, is given by concatenation of sequences in the form $OB(a_1)\ OB(a_2) \ldots OB(a_n)$, while individual values $OB(a_i)$ are for $1 \le i \le n$ determined according to specified rules.

Regarding recursive nature of the partial mapping *OutputBind* we will include several examples of binding of the arc sequences to elements of the set ***Tokens***. Let us assume in all cases that a certain sequential object Petri net *SOPN* is given containing the transition **T**, whose set of input variables *TransitionInputVariables*(**T**) = {a, b, c, x, #x} and in certain

marking *M* of the net *SOPN* there exists binding of those input variables given as follows: I*nputBind*(a) = 10, *InputBind*(b) = 2, *InputBind*(c) = <<1, 1>, 3>, *InputBind*(x) = <1, 2, 3>, *InputBind*(#x) = <1, 2, 3>. In the following examples we will investigate values of partial mapping *OutputBind* applied to various values of the arc sequence *AS*.

- if *AS* = <a>, then *OutputBind*(<a>) = *OB*(a) = <*InputBind*(a)> = <10>.
- if *AS* = <c, 1>, then *OutputBind*(<c, 1>) = *OB*(c) *OB*(1) = <*InputBind*(c)><1> = <<<1, 1>, 3>><1> = <<<1, 1>, 3>, 1>.
- if *AS* = <x, a, 5>, then *OutputBind*(<x, a, 5>) = *OB*(x) *OB*(a) *OB*(5) = <*InputBind*(x)><*InputBind*(a)><5> = <<1, 2, 3>><10><5> = <<1, 2, 3>, 10, 5>.
- if *AS* = <#x, a, 5>, then *OutputBind*(<#x, a, 5>) = *OB*(#x) *OB*(a) *OB*(5) = *InputBind*(#x)<*InputBind*(a)><5> = <1, 2, 3><10><5> = <1, 2, 3, 10, 5>.
- if *AS* = <b*(#x)>, then *OutputBind*(<b*(#x)>) = *OB*(b*(#x)) = *OB*(b)*(*OB*(#x)) = *InputBind*(b)*(*InputBind*(#x)) = 2*(<1, 2, 3>) = <1, 2, 3><1, 2, 3> = <1, 2, 3, 1, 2, 3>.
- if *AS* = <@(<x, a, 5>)>, then *OutputBind*(<@(<x, a, 5>)>) = <*OB*(@(<x, a, 5>))> = <@(*OB*(x) *OB*(a) *OB*(5))> = <@(<*InputBind*(x)><*InputBind*(a)>*OB*(5))> = <@(<<1, 2, 3>><10><5>)> = <@(<<1, 2, 3>, 10, 5>)> = <3>.

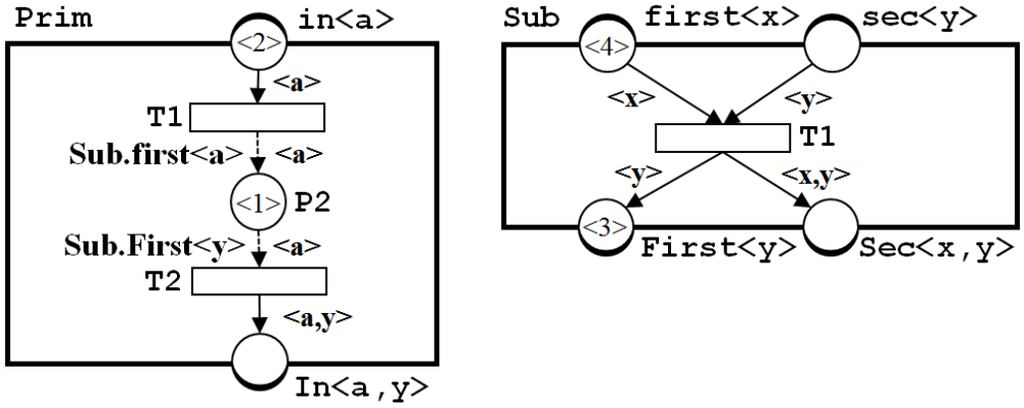

Figure 6. Marking of sequential object Petri net

Figure 6 shows the net pages **Prim** and **Sub** of a certain sequential object Petri net in their marking *M* and we are interested, if there exists input binding of transition variables associated with the transitions **Prim.T1** and **Prim.T2**. With the transition **Prim.T1** is associated one input arc (**Prim.in, Prim.T1**) whose value of the arc function *AF*(**Prim.in, Prim.T1**) = <a>, and thus the set *TransitionInputVariables*(**Prim.T1**) = {a}. We can easily determine that for the input arc (**Prim.in, Prim.T1**) there exists mapping *InputBind*: <a> → <2>, and thus holds that *InputBind*(a) = 2. With the transition **Prim.T2** is associated one input multiarc (**Prim.P2, Prim.T2**) whose value of arc function *AF*(**Prim.P2, Prim.T2**) = <a>, the value of multiarc function *MAF*(**Prim.P2, Prim.T2**) = <y> and thus the set *TransitionInputVariables*(**Prim.T2**) = {a, y}. And again, we can easily determine that for the input multiarc (**Prim.P2, Prim.T2**) there exists mapping *InputBind*: <a> → <1>, and thus holds that *InputBind*(a) = 1. With that input multiarc it is also necessary to determine the mapping *InputBind*: *MAF*(**Prim.P2**, **Prim.T2**) → e, where $e \in M(OPF(\textbf{Prim.P2}, \textbf{Prim.T2}))$. The

value of the output place function *OPF*(**Prim.P2**, **Prim.T2**) = **Sub.First**, whose marking *M*(**Sub.First**) = 1`<3>. So we investigate, if there exists mapping *InputBind*: <y> → <3>. We can easily determine that the mapping exists and it holds that *InputBind*(y) = 3. Generally, for the transition **Prim.T2** holds that *InputBind*(a) = 1 and *InputBind*(y) = 3.

So it can be stated that for both transitions **Prim.T1** a **Prim.T2** exist particular input bindings of all the transition input variables associated with their input arcs and thus, both transitions are enabled. We are further interested, if there exists the mapping *OutputBind* of the values of functions *AF* and *MAF* associated with output arcs (or multiarcs) of both transitions. With transition **Prim.T1** is associated the only output multiarc (**Prim.T1, Prim.P2**) whose value of the arc function *AF*(**Prim.T1, Prim.P2**) = <a> and the value of the multiarc function *MAF*(**Prim.T1, Prim.P2**) = <a>. So we can easily find out that *OutputBind*(<a>) = <2>. With the transition **Prim.T2** is associated the only output arc (**Prim.T2, Prim.In**) whose value of the arc function *AF*(**Prim.T2, Prim.In**) = <a, y>. So again, we can easily find out that *OutputBind*(<a, y>) = <1, 3>.

Let *SOPN* = (Σ, *PN*) is a sequential object Petri net, $PG \in \Sigma$ is a net page, $t \in T$ is a transition of the net page *PG*, *M* is a marking of the net *SOPN*.

i. If the transition *t* is enabled in the marking *M,* then we obtain by its **firing** marking *M'* of the net *SOPN*, defined as follows:

$M'(p) = M(p) \setminus InputBind(AF(p, t))$, if $(p \in \bullet t) \wedge ((p, t) \in (A \cup MA)) \wedge (\exists\, InputBind: AF(p, t) \rightarrow e, e \in M(p))$,

$M'(p) = M(p) \cup OutputBind(AF(t, p))$, if $(p \in t\bullet) \wedge ((t, p) \in (A \cup MA))$,

$M'(q) = M(q) \setminus InputBind(MAF(p, t))$, if $(p \in \bullet t) \wedge ((p, t) \in MA) \wedge (q = OPF(p, t)) \wedge (\exists\, InputBind: MAF(p, t) \rightarrow e, e \in M(OPF(p, t)))$,

$M'(q) = M(q) \cup OutputBind(MAF(t, p))$, if $(p \in t\bullet) \wedge ((t, p) \in MA) \wedge (q = IPF(t, p))$.

$M'(p) = M(p)$, otherwise.

ii. Firing of transition $t \in T$, which will change the marking *M* of the sequential object Petri net *SOPN* into the marking *M'*, is symbolically denoted as $M\ [\ t\ \rangle\ M'$.

iii. Step is understood as firing of non-empty subset from the set of enabled transitions in the given marking *M* of the sequential object Petri net *SOPN*. Step Y which will the marking *M* into the marking *M'* is symbolically denoted as $M\ [\ Y\ \rangle\ M'$.

iv. Let step *Y* be enabled at the marking *M* of the net *SOPN*. If $t_1, t_2 \in Y$ and $t_1 \neq t_2$,we say then that transitions t_1 a t_2 are **concurrently enabled** and that fact is symbolically denoted in the form of $\{t_1, t_2\}$ ***en*** *M.*

Firing of transition will result in the new marking of given sequential object Petri net, which we will obtain as follows:

- from each input place *p* of the fired transition *t* we will remove the (unique) token in the marking *M*, which is bound to the value of the arc function *AF*(*p*, *t*),
- to each output place *p* of the fired transition *t* we will add up the (unique) token which is the value of partial function *OutputBind*(*AF*(*t*, *p*)),

- from each output place of page *q*, being the value of function *OPF* of the input multiarc (*p*, *t*) of the fired transition *t*, we will remove the (unique) token in the marking *M*, bound to the value of the multiarc function *MAF*(*p*, *t*),
- to each input place of page *q*, being the value of function *IPF* of the output multiarc (*t*, *p*) of fired transition *t*, we will add up the (unique) token being the value of partial function *OutputBind*(*MAF*(*t*, *p*)),
- in all the remaining places of the net we will leave their original marking.

Figure 6 shows the net pages **Prim** and **Sub** of a certain sequential object Petri net in its marking *M*. From previous text we know that transitions **Prim.T1** and **Prim.T2** are **concurrently enabled.** Hence, firing of transition **Prim.T1** consists in:

- removing token <2> from the input place **Prim.in**,
- adding token <2> to the place **Prim.P2**,
- adding token <2> to the input place **Sub.first**.

Hence, firing of transition **Prim.T2** consists in:

- removing token <1> from the place **Prim.P2**,
- removing token <3> from the output place **Sub.First**,
- adding token <1, 3> to the output place **Prim.In**.

Marking M′ of the net after concurrent firing of transitions **Prim.T1** and **Prim.T2** is shown in Figure 7.

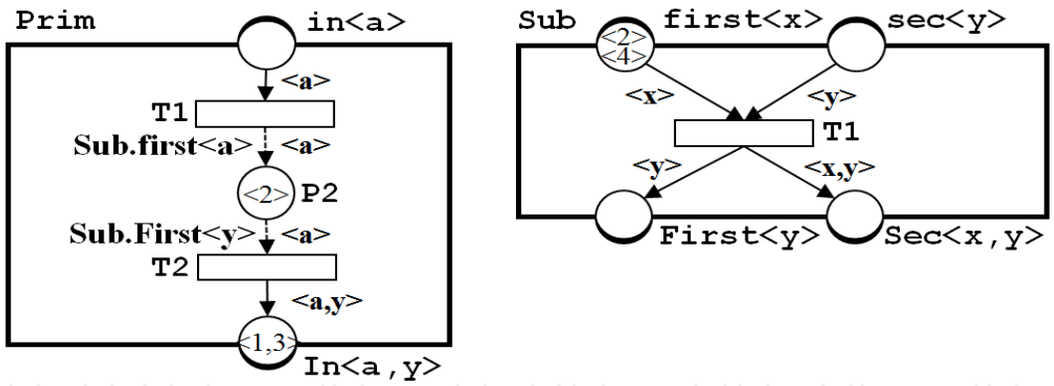

Figure 7. Firing of transitions in sequential object Petri net

Relatively complicated mechanism of firing of transitions in sequential object Petri nets can be better explained by notional substituting of all the multiarcs of the net by standard arcs, which can be realized as follows:

- if *p* is place and *t* transition of the given net page and (*p*, *t*) its multiarc whose value of arc function equals to *AF*(*p*, *t*), the value of output place function equals to *OPF*(*p*, *t*) and the value of multiarc function equals to *MAF*(*p*, *t*), we substitute this multiarc by notional pair of the following standard arcs:
  - by the arc (*p*, *t*) with the value of the arc function equal to *AF*(*p*, *t*),
  - by the arc (*OPF*(*p*, *t*), *t*) with the value of the arc function equal to *MAF*(*p*, *t*).

- if p is place and t transition of the given net page and (t, p) its multiarc whose value of arc function equals to $AF(t, p)$, the value of input place function equals to $IPF(t, p)$ and the value of multiarc function equals to $MAF(t, p)$, we substitute that multiarc with notional pair of the following standard arcs:
  - by the arc (t, p) with the value of the arc function equal to $AF(t, p)$,
  - by the arc $(t, IPF(t, p))$ with the value of the arc function equal to $MAF(t, p)$.

That notional substitution of multiarcs in the previous net is shown in Figure 8 where:

- multiarc (**Prim.T1, Prim.P2**) was substituted by the following pair of arcs:
  - the arc (**Prim.T1, Prim.P2**) with the value of the arc function AF equal to <a>,
  - the arc (**Prim.T1, Sub.first**) with the value of the arc function AF equal to <a>,
- multiarc (**Prim.P2, Prim.T2**) was substituted by the following pair of arcs:
  - the arc (**Prim.P2, Prim.T2**) with the value of the arc function AF equal to <a>,
  - the arc (**Sub.First, Prim.T2**) with the value of the arc function AF equal to <y>.

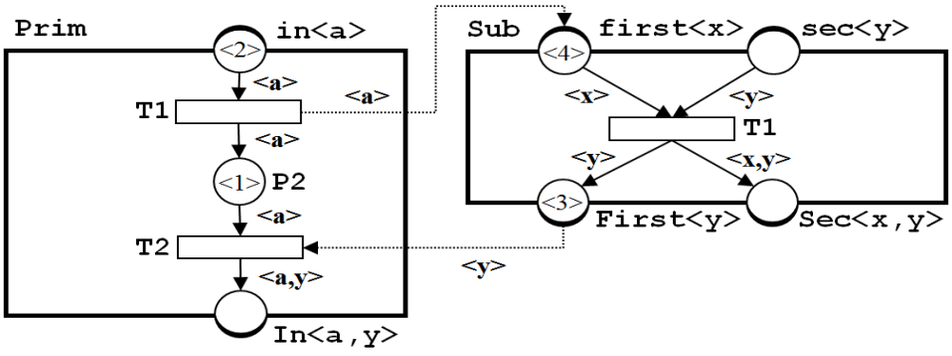

Figure 8. Substitution of multiarcs in sequential object Petri net

When enabling individual steps of the sequential object Petri net, so called *conflicts* can originate in certain markings of the net (or *conflict transitions*). At the enabling of transitions t_1 and t_2 of the given net and its marking M the conflict occurs, if both transitions t_1 and t_2 have at least one input place, each of t_1 and t_2 transitions is individually enabled in the marking M of the net, but t_1 and t_2 transitions are not in that marking M concurrently enabled, i.e. enabling of one of them will prevent enabling the other. The term of conflict transitions can be obviously easily generalized for the case of finite set $t_1, t_2, \ldots, t_n$ $(n \in \mathbf{N})$ of transitions of the given net.

A typical example of the conflict at enabling transitions in the particular marking of the net is shown in Figure 9, where transitions **T1** and **T2** of the net have a common input place **P1**, both are enabled (particular binding of tokens can be easily found), but not concurrently enabled, i.e. enabling of the transition **T1** will disable enabling of the transition **T2** and vice versa. When solving conflicts at enabling of transitions in sequential Petri nets we will therefore follow the rule which determines, informally said, that from the set of conflict transitions at the given binding of tokens the one will be enabled, whose value of transition priority function TP is the highest. If such transition from the set of conflict transitions does not exist, the given conflict would have to be solved by other means. In our studied example will be then on the basis of that rule the transition **T2** enabled (because $TP(\mathbf{T1}) = 1$ and $TP(\mathbf{T2}) = 2$).

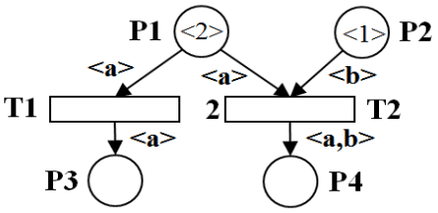

Figure 9. Conflict transitions in sequential object Petri net

5. Object-oriented programming systems and their representation by the sequential object Petri nets

This section deals with main principles applied at modeling of multithreading object-oriented programming systems with the sequential object Petri nets. All program listings are developed in the Java programming language (Goetz et. al, 2006), (Lea, 1999), (Subramaniam, 2011).

Each declared class of the object–oriented programming system is in the sequential object Petri net represented by a net page containing declared data items and methods. Their declarations are made by using elements of the net page. Individual input places of the net page then represent input points of static and non-static methods as a part of the class declaration, and output places of the net page then represent their output points. Input and output places of the net page are associated with identifiers of the particular method whose input and output point they represent while each method has one input and output place. In so doing, we abide to the convention whereby input place identifier of the particular method starts with a small letter and identifier of the output place of the same method with a capital letter. In order to differentiate graphically in layouts of the net declaration of static methods from non-static methods on a given net page, we demarcate identifiers of input and output places of the net page representing input and output points of static methods with the square brackets. Moreover, it is possible on the net page via a position of input and output places in its layout represent visibility of *public*, *protected* and *private* type of individual declared methods thus implementing the characteristic of encapsulation of the object-oriented programming.

Figure 10 illustrates the net page representing the following declaration of the class **Sys**:

```
public class Sys {
  public static void compute() { … }
  protected void run() { … }
  private void init() { … }
}
```

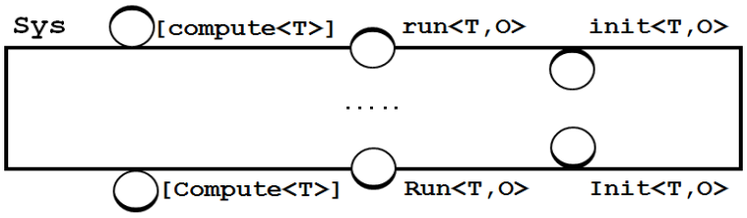

Figure 10. Net page representing declaration of class **Sys**

Static and non-static data items are on the net pages represented in the form of **tokens** in the given net marking, i.e. by elements of the set ***Tokens***. When representing values of static and non-static data items being the elements of particular tokens of the net, it is for example possible to proceed as follows:

- If the data type of the particular data item is a non-negative integer (*int*), its actual value equals to that non-negative integer.
- If the data type of the particular data item is *boolean*, we represent the truth value *false* most frequently with constant 0 and the truth value *true* with constant 1, while in layouts of the net we are also using symbolic values *false* and *true*.
- If the data type of the particular data item is *char*, whose value is the element of the set contained in the code table given by the Unicode standard, we represent its value corresponding to the value of the symbol code given in the Unicode Standard, i.e. the letter 'A' can be represented by the value 65. In order to make again layouts of nets more readable, the element of token can be entered as the particular symbol bounded by apostrophes, i.e. instead of token <65> we will indicate in layouts of nets token <'A'>.
- If the data type of the particular data item is *string*, whose value is text string, the values can be represented by sequences of codes of symbols according to the Unicode standard. Again, for better transparency of notations, it is possible to use instead of codes of symbols directly the sequence of symbols bounded by apostrophes, i.e. in layouts of nets represent data items of the data type *string* with the tokens of <'H', 'e', 'l', 'l', 'o'> shape, or to bound sequence of symbols of the string with quotation marks, i.e. to note the tokens in the form of <"Hello">.
- Data items whose data type are numbers with floating decimal point (*float*, *double*), or other numerical data type being the superset of the set of non-negative integer numbers, it is not possible to declare it directly on the basis of the stated definition of the sequential object Petri net. If a need emerges to operate with those numerical sets during simulation of multithreading object-oriented programming system, it is always easy to extend in this sense the definition of the token, arc sequence and other particular definitions, in order to ensure the support of those numerical sets (e.g. it is possible to define the token of sequential object Petri net as the finite non-empty sequence over the set of real numbers $\boldsymbol{R}$, etc.).
- In the case of data items whose data type is the pointer to instance of class, we represent their values with natural numbers. Each instance of the class has at its creation allocated unique natural number expressing its address within the programming heap. Number 0 than represents the value of pointer *null*.

Static data items are usually represented in layouts of nets with the tokens containing only their actual values. Non-static data items are usually represented by tokens containing both the value of pointer to a given instance of the class in which the particular data item is declared, and its actual value according to particular data type. So if within the declaration of the class First for example the following data items are declared:

```
public class First {
  private static boolean  indicator = true;
  private char status = 'a';
}
```

then, the static data item **indicator** can be represented by the token **<true>** and the non-static data item **status** by the token **<11, 'a'>**, where numerical value **11** represents the value of the pointer to the particular instance of class.

The dynamic creation of the instance of the class is not in the sequential object Petri net realized by the creation of a new instance of the particular net page representing declaration of the given class, but by creation of all the tokens representing non-static data items of the declared class. At the same time, each such token contains (usually as its first element) the value of the pointer to the newly created class instance. The fact, that during dynamic creation of instances of classes it is not necessary to create instances of net pages, dramatically simplifies its analysis.

All methods represented by elements of the net pages are executed by the programming threads. Each such programming thread is represented by the particular instance of class (usually by the class **Thread**). Thus, the token accepted by the input place of the net page must contain the pointer for particular programming thread realizing execution of the given method, while that pointer is within arc sequences of the net represented by default by some of the identifiers **T**, **U**, **V**, etc. By the element of the token accepted by the input place of the page representing the input point of some of the non-static methods must be then the pointer of the particular instance of class whose non-static method is (i.e. pointer **this**). That pointer is then within the arc sequences of the net standardly represented by the identifier **O** (while indeed, within representation of static methods that pointer cannot be used). When entering identifiers of static and non-static data items, parameters and local variables of methods, we use by default in layouts of the net so called Hungarian notations, where the first letter (or the first part) of identifier expresses its data type. For identification of standard data types we use the acronym **i** even for data type *int*, **b** for *boolean*, **c** for *char*, **s** for *string* and **p** for *pointer*. Hence for example the identifier **sName** within the arc sequence represents the variable of the data type *string* with the identifier **name**.

Figure 11 shows the net page representing the declaration of the following class **Obj** in its marking *M*:

```
public class Obj {
  private char val = 'a';
  public synchronized char getVal() { return value; }
  public synchronized void setVal(char value) { this.val = value; }
}
```

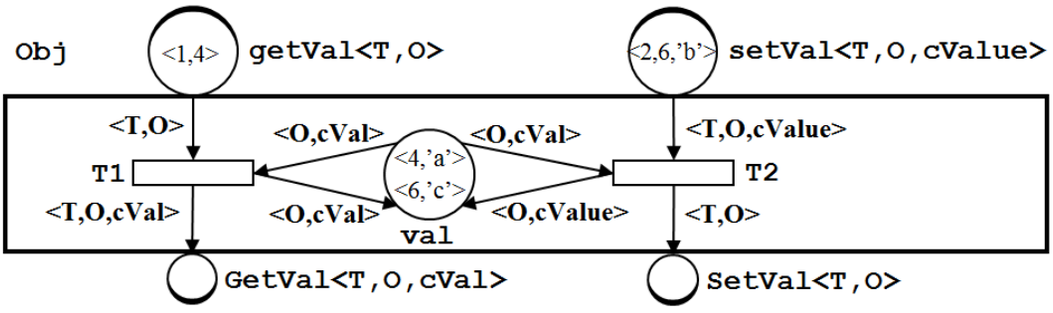

Figure 11. Net page representing declaration of class **Obj**

The current notation of that net page suggests, that within the program execution modeled by that net two instances of the class **Obj** have been already created, as in the place **Obj.val** appear tokens **<4, 'a'>** a **<6, 'c'>**, where the first element of each of those tokens is the pointer to the instance of the class **Obj** and second the actual value of the non-static data item **val** of the data type *char*. The non-static method **getVal** will be executed above the instance of class with the pointer **4** by the programming thread with the pointer equal to the value **1**, the non-static method **setVal** will be executed above the instance of the class with the pointer **6** by the programming thread with the pointer equal to the value **2**, while the value of method parameter **value** equals to character **'b'**. The shape of the net demonstrates, that over the selected instance of the class can be concurrently executed with any programming thread maximum one of **getVal** or **setVal** methods.

Additionally, in layouts of nets can be simply represented the relation of a simple inheritance between two declared classes. The identifier of the net page representing the class declaration being superclass of the given class, we indicate in the layout of the given net page of the class in its left bottom corner. If that identifier of the net page of the superclass is not explicitly indicated in the layout of the net, we consider the given class to be implicitly subclass of the top class in the hierarchy of classes created on the basis of relation of simple inheritance of classes (e.g. within the hierarchy of classes of the Java programming language it is the class **java.lang.Object**).

With the inheritance relation of classes is organically connected even the term of polymorphism of the object-oriented programming and the possibility of declaration of so called virtual methods. Figure 12 shows two net pages out of which the first represents declaration of the class **Object** and the second declaration of the class **System**, being subclass of the class **Object**. Within the class **Object** is also declared virtual method with the head **public int hashCode()**, which is in the declaration of the class **System** overwritten with the virtual method of identical method head. Input and output places of net pages **Object** and **System**, which appertain to input and output points of both virtual methods are indicated in the net layout by default while in the case of declaration of the virtual methods it is necessary that **all input places representing the input point of the given virtual method had assigned on all the net pages containing the declaration of this method the identical value of *IOPN* function and all output places representing the output point of the given virtual method had assigned on all relevant net pages the identical value of *IOPN* function** (see Figure 12).

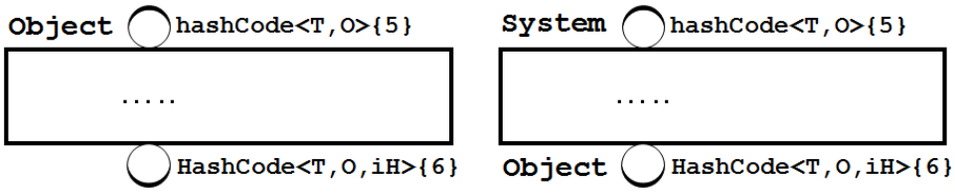

Figure 12. Virtual methods representation in sequential object Petri net

Next example demonstrates representation of classes containing declaration of virtual and abstract methods. **Virt** and **Add** classes are declared as follows:

```
public abstract class Virt {
  public abstract int compute(int a, int b);
  public int make(int a, int b)  { return compute(a, b); }
}

public class Add extends Virt {
  public int compute(int a, int b) { return (a + b); }
}
```

The **Virt** class is an abstract class with the declared abstract method **compute**. That method is called as a part of the declared method **make** of the **Virt** class. The method **compute** is then overwritten at the level of declaration of the class **Add**, being the subclass of the class **Virt**. The net page representing the declaration of the class **Virt** is shown in Figure 13. Abstract method **compute** is represented only by a pair of the input place **Virt.compute** and the output place **Virt.Compute**. Regarding the fact that it is also declaration of the virtual method, the selection of function *IOPN* values is important (i.e. in our case *IOPN*(**Virt.compute**) = 1 and *IOPN*(**Virt.Compute**) = 2). The key element of the net page is the manner of calling of the virtual method **compute** represented by the multiarc (**Virt.T1**, **Virt.P1**) and its value of input place function *IPF*, the multiarc (**Virt.P1**, **Virt.T2**) and its value of output place function *OPF*. The value of input place function *IPF* of the multiarc (**Virt.T1**, **Virt.P1**) is in the form of **O.compute** (more precisely in the **O.1** form), where identifier **O** represents the pointer of particular instance of non-abstract subclass of class **Virt**, whose non-static method **compute** is executed. The value of identifier **O** is in our case non-constant, ie. generalized at the realization of steps, and depends on binding of the specific token to the arc sequence in the form **<T, O, iA, iB>**, which shares in the given identification of the net execution of transition **Virt.T1**. According to the numerical value bound to identifier **O** the particular net page net will be determined whose value of the function *PN* of numbering net pages is identical with the numerical value bound to the identifier **O**. Then on this net page the input place with the value of *IOPN* function equal to number **1** will be selected (representing the input point of the virtual method **compute**), into which the particular token in the form **<T, O, iA, iB>** will be placed.

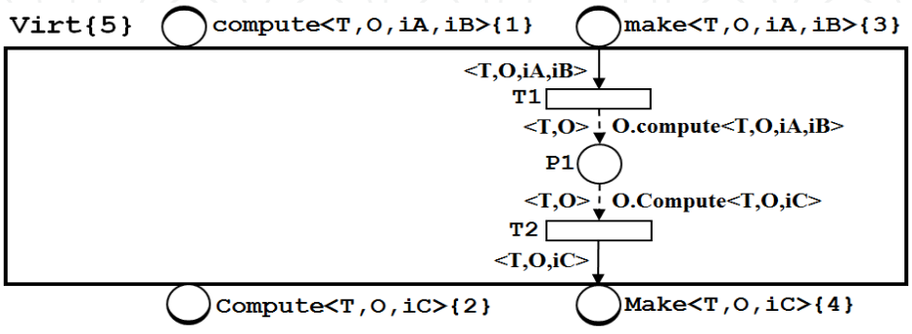

Figure 13. Polymorphism representation in sequential object Petri net

Net page representing the declaration of the class **Add**, being the subclass of the class **Virt**, is shown in Figure 14. In the class **Add** is declared the only non-static virtual method

compute, which overwrites the abstract method **compute** declared in the body of the class **Virt**. Thus, on the grounds of that fact it is necessary to correctly assign values of *IOPN* function (in our case *IOPN*(**Add.compute**) = 1 and *IOPN*(**Add.Compute**) = 2). An interesting detail of this net page is the way of determination of the addition of integral values bound to variables **iA** and **iB** of the arc sequence of the arc (**Add.compute**, **Add.T1**). The expression in the form **@(iA*<0>iB*<0>)** represents length of the sequence formed by concatenation of two sequences: first sequence containing in total **iA** of numbers **0** and the second sequence containing in total **iB** numbers **0** (in this case, it is worth mentioning, that e.g. via the expression in the form **@(iA*(iB*<0>))** product of integral values of variables **iA** a **iB** can be determined).

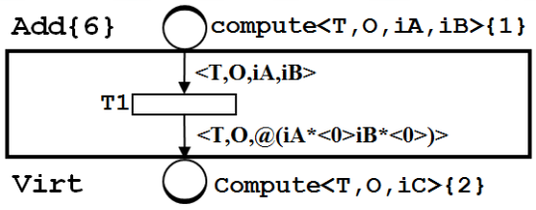

Figure 14. Polymorphism representation in sequential object Petri net

Next generalization of the form of functions *IPF* and *OPF* associated with multiarcs is possible within the sequential object Petri nets to implement the explicit support of the mechanism of **first order and higher order functions** known from functional programming (ie. given method can take another methods as parameters and return the method as the return value). Figure 15 shows the net page of the class **Func** containing declaration of method **call**. Arc sequence of the token accepted by the input place of that method contains a variable **mO**, the value of which is the value of the function *PN* for the net page with the input place **make** and output place **Make**, which are represented by particular integral values of the function *IOPN*. Values of functions *MAF* of multiarcs of the net in the form **mO.make** and **mO.Make**, whose all components are variables and whose values are not determined until particular arc sequence is bound to token of the net, provide general mechanism for the possibility of declaration of first order and higher order functions.

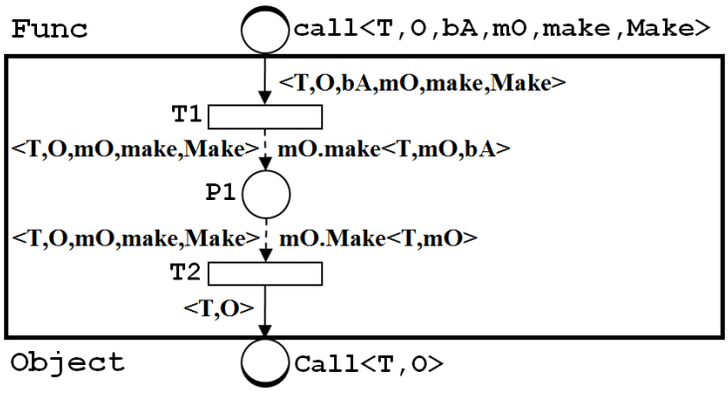

Figure 15. Higher order function representation in sequential object Petri net

Example of recursive method represented by the net page can be seen on Figure 16. This static recursive method **fact** of the class **Integer** implements calculation of the value of

factorial function whose parameter is a certain natural number **n**. Recursive algorithm for calculation of factorial of the integer value **n** in the Java programming language is represented by the following listing:

```
public class Integer {
  public static int fact(int n) {
    if (n == 1) return 1;
    else return (n * fact(n – 1));
}
```

If the value of the parameter **iN** equals to number 1, the transition **Integer.T4** is enabled and in the output place **Integer.Fact** the token **<T, 1>** is stored, whose first element **T** represents the programming thread and the second element the value of factorial of number 1. If the value of number **iN** is greater than number 1, by enabling the transition **Integer.T2** that number **iN** is substituted by the sequence consisting of **iN** numbers 0, which is then used at recursive call modeled by the multiarc (**Integer.T3**, **Integer.P2**), whose enabling will always result in elimination of one element of that sequence. The programming stack which is used at realization of recursive procedure of the algorithm is represented by the token in the place **Integer.P3** in the form of the sequence **<T, …, iN-2, iN-1, iN>**. At the moment where the sequence composed from 0 numerals contains after the series of recursive calls of algorithm one element (i.e. it is necessary to determine the value of factorial of number 1 within the stop condition of algorithm), the transition **Integer.T4** is enabled and the value of factorial of number 1 is represented in token **<T, 1>** located in the output place **Integer.Fact**. Then, by repeatedly enabling the transition **Integer.T5** and finally the transition **Integer.T6** the return from recursion is implemented with the gradual calculation of the value of factorial function, which is represented in the token of the form **<T, @(iM*(iF*0))>**. Following completion of the process of reverse return of recursion in the output place **Integer.Fact** the token in the form **<T, iF>** is stored, whose second element represents the value of factorial of the natural number **iN**.

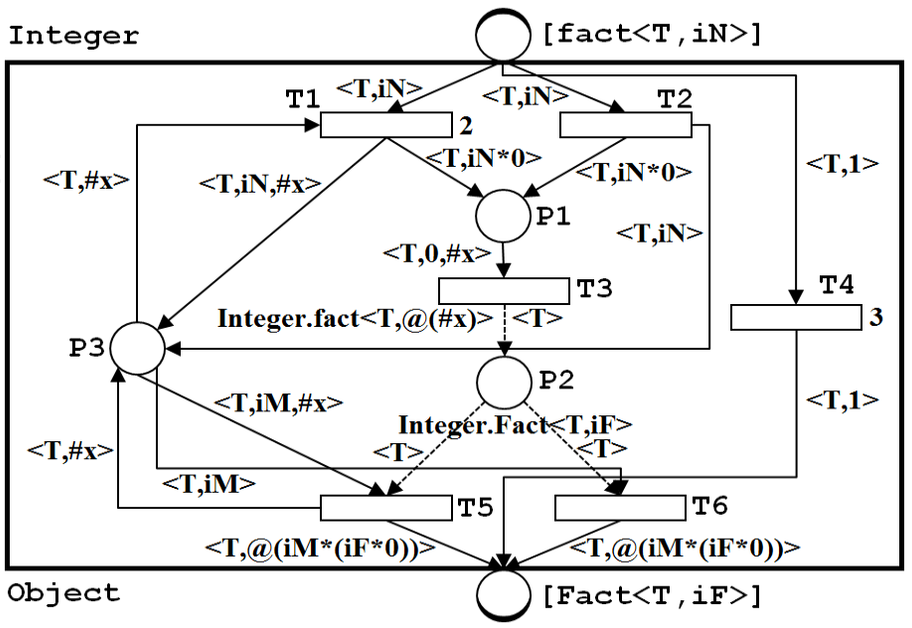

Figure 16. Recursive function representation in sequential object Petri net

Moreover, in the sequential object Petri nets can be simply represented declarations of inner classes of the selected class via subpages of the net page. So if for example the class **Obj** which within its declaration contains declaration of its own inner class **InnerObj**, that declaration can be represented by the subpage **InnerObj** of the page **Obj** (see Figure 17)

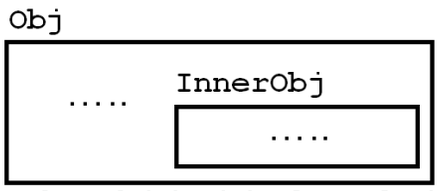

Figure 17. Inner class declaration representation in sequential object Petri net

The representation of declared interfaces, which can contain only declarations of headings of publicly accessible abstract methods, is also easy. Figure 18 shows the representation of interface ***Runnable*** (identifiers of interface we indicate in layouts of the net with spaced letters) containing declaration of the method with the head **public abstract void run()**.

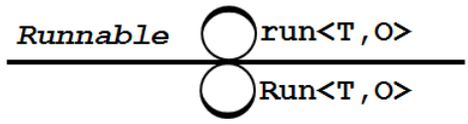

Figure 18. Interface declaration in sequential object Petri net

6. Example of simple class hierarchy represented by sequential object Petri net

In this section we will demonstrate how simple hierarchy of the classes via sequential object Petri nets is built. The model of those examples involve selected classes contained in standard library of classes of the Java programming language. (e.g. classes **java.lang.Object**, **java.lang.Thread**, etc.). For better visibility of layouts we usually present individual declared methods in separate figures representing individual parts of the particular net page (while naturally places and transitions of the net which appear in individual parts of the net page and have identical identifiers, always represent the same place or transition of the total page of the net). Thus, top of our hierarchy of classes will be the class **Object**, whose declaration made in the Java programming language is the following:

```
public class Object {
  private Object monitor;
  private Thread thread;
  private Vector<Thread> = new Vector<Thread>;
  private synchronized static int getPointer() { … }
  public Object() { … }
  protected void finalize() { … }
  public void lock() { … }
  public void unlock() { … }
```

```
  public void wait() { … }
  public void notify() { … }
}
```

The static method **getPointer** (see Figure 19) can be executed by the only one programming thread only and it is determined for assigning unique integral values of pointers to individual instances of the classes. At the first execution of the method the integral value 1 is returned in variable **P** (because *InputBind*(#p) = 0 a @(#p) = @(<0>) = 1), in the place **Object.P1** is then stored the token **<0, 0>**, and at next execution of the method is in the variable **P** returned the value 2, etc.

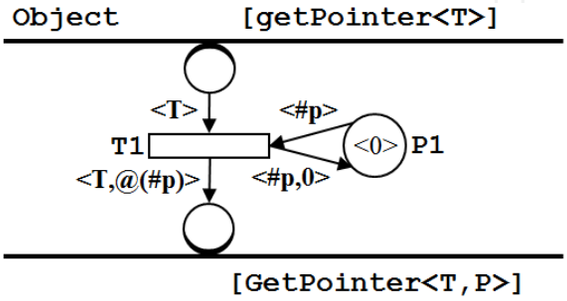

Figure 19. Declaration of class **Object** in sequential object Petri net

Within the declaration of the constructor of the object of the class **Object** (see Figure 20) is by the programming thread firstly obtained the value of the pointer for a newly created object by calling the method **getPointer**, that value is then stored to the variable **O** and the value of non-static data item **monitor** representing the monitor of particular instance of the class is then stored in the form of the token **<O>** in the place **Object.monitor**. The destructor of the object represented by the method **finalize** will then mainly ensure cancellation of the monitor of the object represented by the token **<O>** in the place **Object.monitor**.

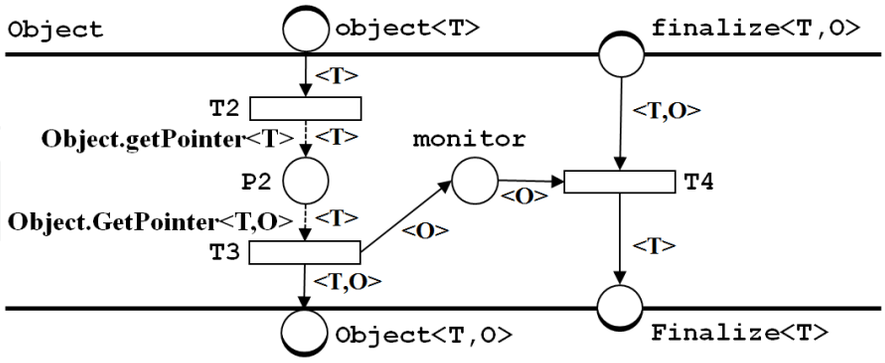

Figure 20. Declaration of class **Object** in sequential object Petri net

Entry into the critical section of the object (i.e. execution of the non-static method with the modifier **synchronized**) is conditioned by getting the object monitor with the particular programming thread. As a part of the execution of the method **lock** the programming thread **T** can respectively allocate the object monitor with the value of the pointer **O**, (i.e. the token **<O>** in the place **Object.monitor**) and then enter into the critical section of the object

while the programming thread is permitted to enter the critical section of the same object several times in sequence. First entry of the programming thread **T** into the critical section of the object **O** is realized by execution of the transition **Object.T5**. Pointer to the programming thread, which successfully obtained the object monitor, is then stored in the variable **thread** (i.e. the token **<T, O, 1>** is located in the place **Object.thread**). At repeated entry of the programming thread into the critical sections of the object **O** the transition **Object.T6** is executed and particular token in the place **Object.thread** is added with the next element which is the number **1** (i.e. at second entry of the thread **T** into the critical section of the object **O** is in the place **Object.thread** stored the token **<T, O, 1, 1>**, at the third entry the token **<T, O, 1, 1, 1>**, etc). Deal location of the object monitor and initialization of its critical section is then realized by execution of the non-static method **unlock**, whose functionalities are inverse to functionalities of the method **lock** (see Figure 21).

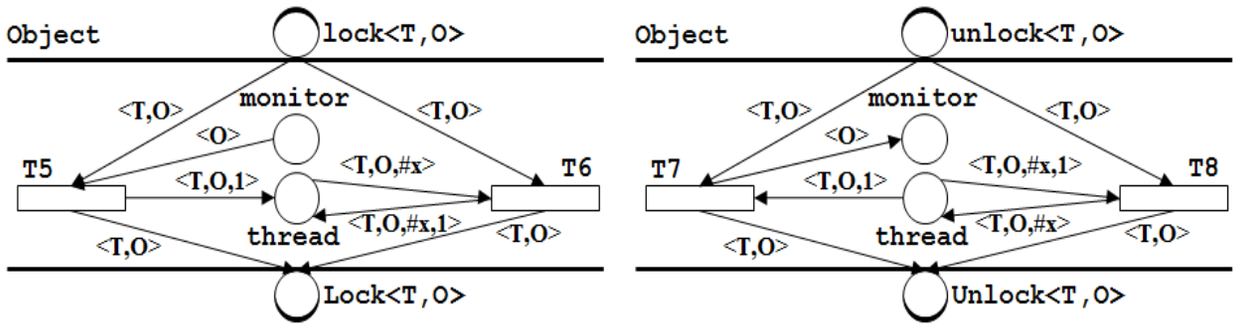

Figure 21. Declaration of class **Object** in sequential object Petri net

Method **wait** causes the current thread **T** to wait until another thread invokes the **notify** method for the object **O**. The current thread **T** must own the monitor of the object **O**. The thread releases ownership of this monitor and waits in the place **Object.pool** until another thread notifies threads waiting on the object's monitor to wake up through a call to the notify method. The thread **T** then waits in the place **Object.P3** until it can re-obtain ownership of the object monitor and resumes execution (see Figure 22).

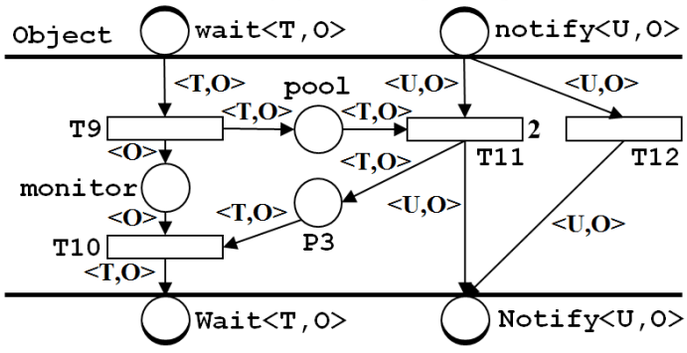

Figure 22. Declaration of class **Object** in sequential object Petri net

Next declared class in our hierarchy will be the class **Thread** representing the programming thread, which is the subclass of the class **Object** and it also implements the interface ***Runnable*** (see Figure 23). Its declaration in the Java programming language is as follows:

```
public class Thread extends Object implements Runnable {
  private Runnable runnable;
  public Thread(Runnable run) { … }
  protected void finalize() { … }
  public void start() { … }
  public void run() { … }
}
```

The method **Thread** is the constructor of the object which during its execution firstly invokes the construction of the instance of the class **Object**, that is the superclass of the class **Thread**, which in the variable value **U** returns the pointer to the newly created programming thread. In the place **Thread.runnable** (i.e. in the value of the variable **runnable**) is then stored the token **<U, pRun>**, where the value of the variable **pRun** contains the pointer to the class instance implementing the interface ***Runnable*** (see Figure 18) whose method **run** will be then executed by the newly created programming thread. That will be implemented through execution of the method **start** with programming thread **T**. The token **<U, pRun, 0>** is then located into the place **Thread.runnable** representing the fact that the execution of the method **pRun.run** was initiated. Its invocation itself is realized by asynchronous method call by the firing of the transition **Thread.T3** and by the mechanism of multiarc (**Thread.T3**, **Thread.Start**). The class **Thread** itself has the method **run** implemented the easiest possible way.

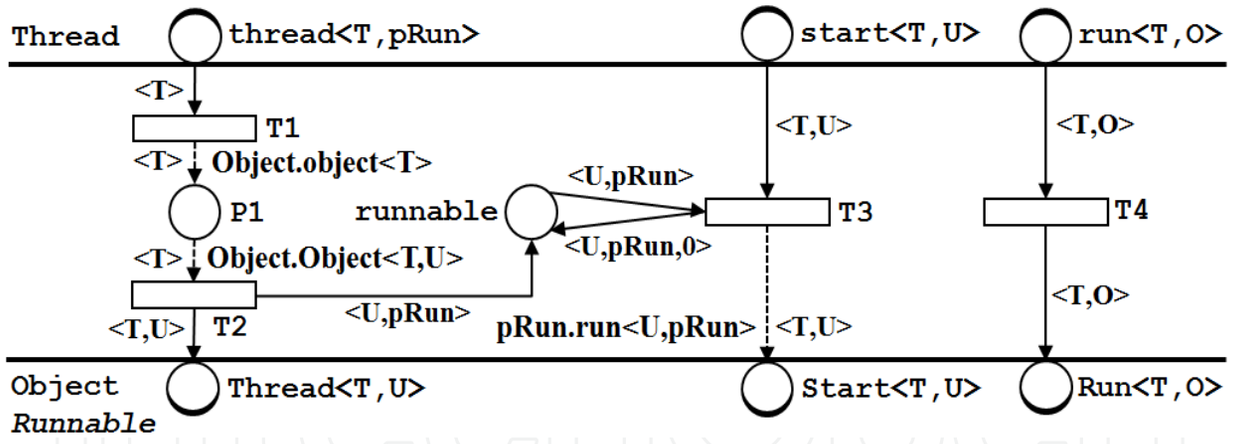

Figure 23. Declaration of class **Thread** in sequential object Petri net

As a part of execution of the destructor of the instance of the class **Thread**, i.e. the method **finalize** (see Figure 24), the particular token is removed from the place **Thread.runnable** and the destructor of the superclass **Object** is then executed. If in the place **Thread.runnable** is the token of the form **<U, pRun>**, the programming thread is not active to obeyed destruction and the transition **Thread.T7** can be fired. If, on the contrary, in the place **Thread.runnable** is the token in the form **<U, pRun, 0>**, the programming thread is active and the execution of the destructor must be delayed until the completion of the activities of this thread (i.e. returning of the thread **U** after executing the method **pRun.run** ensured by the mechanism of the multiarc (**Thread.finalize**, **Thread.T8**)).

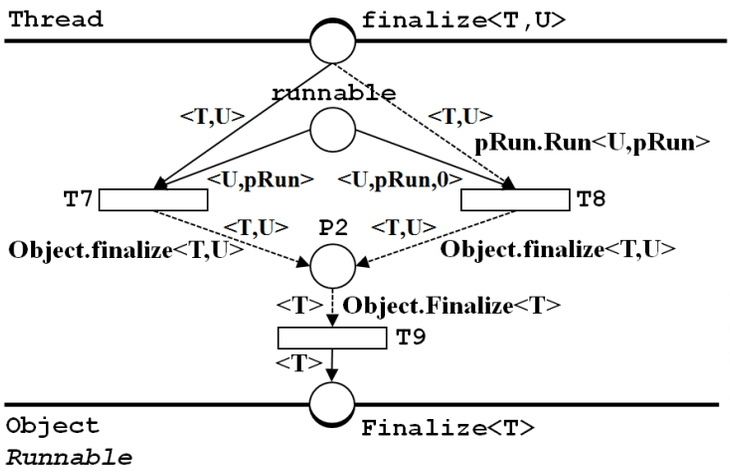

Figure 24. Declaration of class **Thread** in sequential object Petri net

Next declared class in our hierarchy will be the class **Semaphore** representing a counting semaphore. Conceptually, a semaphore maintains a set of permits. Method **down** blocks if necessary until a permit is available, and then takes it. Method **up** adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used and the **Semaphore** just keeps a count of the number available and acts accordingly. Declaration of the class in the Java programming language is the following:

```
public class Semaphore extends Object {
  private int value = 1;
  public Semaphore() { … }
  protected void finalize() { … }
  public synchronized void down() { … }
  public synchronized void up() { … }
}
```

The constructor of the object, i.e. the method **Semaphore**, following execution of the constructor of the instance of the superclass **Object** will store in the place **Semaphore.value** the token **<O, 1>** representing the initial value of the data item value of the class instance **O**. As a part of the execution of the destructor of the instance of the class **Semaphore**, i.e. of the method **finalize**, that token is removed from the place **Semaphore.value** and then the destructor of the superclass **Object** is executed (see Figure 25).

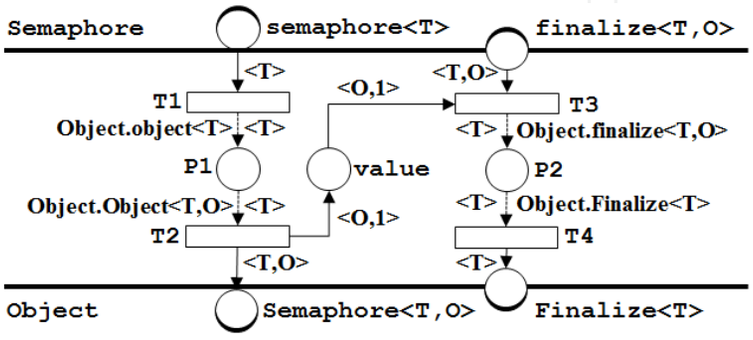

Figure 25. Declaration of class **Semaphore** in sequential object Petri net

Method **down** acquires a permit from this semaphore, blocking until one is available. Following entry into the critical section of the object instance the programming thread acquires a permit from this semaphore (ie. token in the place **Semaphore.value** can be bound to the arc sequence **<O, 1, #x>** after at least one execution of the method **up**) and will leave the critical section of the object. If no permit is available then the current thread becomes disabled for thread scheduling purposes and waits (i.e. the transition **Semaphore.T7** is executed and the method **Object.wait** is invoked, the programming thread will release the monitor of the object in order to enable execution of the method **up** by other programming thread) until some other thread invokes the **up** method for this semaphore and the current thread is next to be assigned a permit (see Figure 26).

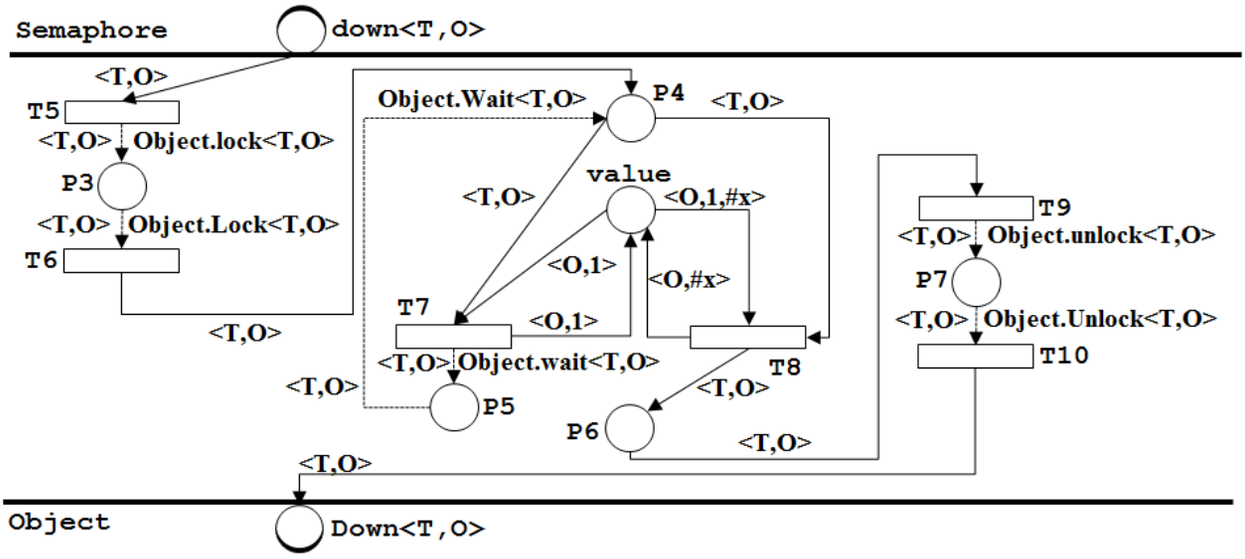

Figure 26. Declaration of class **Semaphore** in sequential object Petri net

Method **up** releases a permit, increasing the number of available permits by one. If any threads are trying to acquire a permit, then one is selected and given the permit that was just released. That thread is (re)enabled for thread scheduling purposes (see Figure 27).

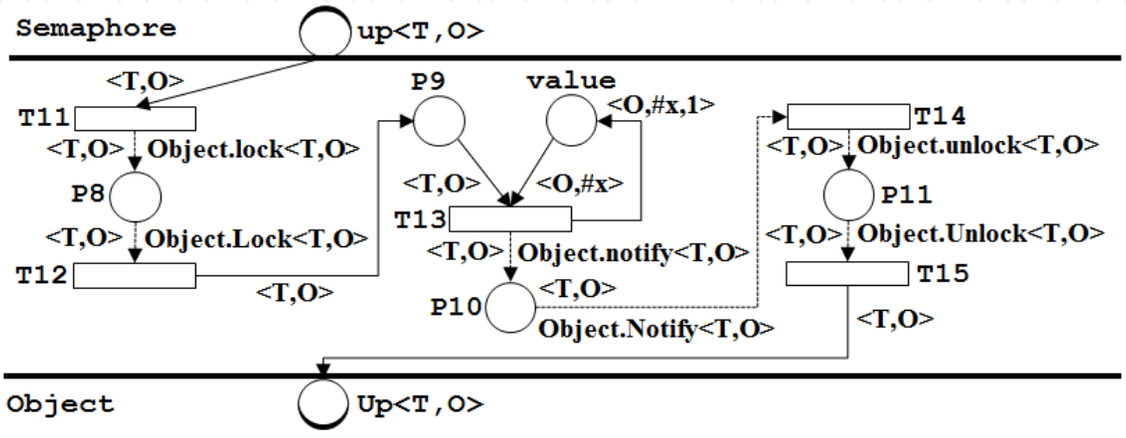

Figure 27. Declaration of class **Semaphore** in sequential object Petri net

7. Conclusion

Sequential object Petri nets represent an interesting class in the area of object Petri net classes, which can be applied at design, modeling, analysis and verification of generally distributed multithreading object-oriented programming systems. A newly introduced term of token as finite non-empty recursive sequence over the set of non-negative integer numbers, functionalities of multiarcs and the mechanism of the firing of transitions do not increase demands on performance of analysis of characteristics, as seen in other classes of high-level or colored Petri Nets.

Functional programming is one of the most important paradigms of programming that looks back on a long history. The recent interest in functional programming started as a response to the growing pervasiveness of concurrency as a way of scaling horizontally. Multithreaded programming is difficult to do well in practice and functional programming offers (in many ways) better strategies than object-oriented programming for writing robust, concurrent software. Functional programming is generally regarded a paradigm of programming that can be applied in many languages - even those that were not originally intended to be used with that paradigm. Like the name implies, it focuses on the application of functions. Functional programmers use functions as building blocks to create new functions and the function is the main construct that architecture is built from. Several programming languages (like Scala, C#, Java, Delphi, etc) are a blend of object-oriented and functional programming concepts in a statically typed language in the present time. The fusion of object-oriented and functional programming makes it possible to express new kinds of programming patterns and component abstractions. It also leads to a legible and concise programming style. Sequential object Petri nets then also fully support design, modeling, analysis and verification of programming systems based on this fusion of object-oriented and functional programming paradigms.

Author details

Ivo Martiník
VŠB-Technical University of Ostrava, Czech Republic

Acknowledgement

This paper has been elaborated in the framework of the IT4Innovations Centre of Excellence project, reg. no. CZ.1.05/1.1.00/02.0070 supported by Operational Programme 'Research and Development for Innovations' funded by Structural Funds of the European Union and state budget of the Czech Republic.

8. References

Agha, G. A.; Cinindio, F. & Rozenberg, G. (2001). *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets*. Springer, ISBN 978-3-540-41942-6, Berlin, Germany

Diaz, M. (2009). *Petri Nets: Fundamental Models, Verification and Applications*, John Willey & Sons, ISTE Ltd., ISBN: 978-0-470-39430-4, London, United Kingdom

Goetz, B.; Peierls, T., Bloch, J.; Bowbeer, J.; Holmes, D. & Lea, D. (2006). *Java Concurrency in Practice*, Addison-Wesley, ISBN 978-0321349606, Reading, United Kingdom

Jensen, K.; Kristensen, L. M. (2009). *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*, Springer, ISBN 978-3-642-00283-0, Berlin, Germany

Jensen, K.; Rozenberg, G. (1991). *High-Level Petri Nets: Theory and Application*, Springer, ISBN 3-540-54125-x, London, United Kingdom

Köhler, M.; Rölke, H. (2007). Web Services Orchestration with Super-Dual Object Nets, *ICATPN 2007, Lecture Notes in Computer Science 4546*, Springer-Verlag, pp. 263–280, ISBN 978-3-540-73093-4

Lea, D. (1999). *Concurrent Programming in Java, Second Edition*, Addison-Wesley, ISBN 0-201-31009-0, Reading, United Kingdom

Martiník, I. Bi-relational P/T Petri Nets and the Modeling of Multithreading Object-oriented Programming Systems, *Communications in Computer and Information Science.* 188 CCIS (Part 1), (July 2011), pp. 222-236. ISSN 1865-0929

Reisig, W. (2009). *Elements of Distributed Algorithms*, Springer, ISBN 3-540-62752-9, Berlin, Germany

Subramaniam, V. (2011). *Programming Concurrency on the JVM: Mastering Synchronization, STM and Actors*, Pragmatic Bookshelf, ISBN 978-1934356760, Dallas, USA