SOFTWARE DESIGN MODELLING WITH FUNCTIONAL PETRI NETS

F.S. Bakpo
Department of Computer Science,
University of Nigeria, Nsukka
fbakpo@yahoo.com

ABSTRACT
Petri Nets use two basic primitives: events and conditions to view or model a system. Events are the actions that take place in the system. The occurrence of events is controlled by the "state" of the system, which can be described as a set of conditions. An immediate application of such a model is in the control structures of conventional programming languages. Control structures are the backbone of every programming language. In this paper, an equivalent functional Petri Net (FPN) model is developed for each of the three constructs of structured programs and a FPN Software prototype proposed for the conventional programming construct: if-then-else statement. The motivating idea is essentially to show that FPNs could be used as an alternative approach for program design.

1. INTRODUCTION

The concept of Petri Nets has its origin in Carl Adam Petri's dissertation "Kommunikation mit Automaton", submitted in 1962 to the Faculty of Mathematics and Physics at the Technische Universitat Darmstadt, Germany [9]. Since then the use and study of Petri nets have increased considerably. For a review of the history of Petri nets and an extensive bibliography the reader is referred to [1, 6, 8, 12, 14]. A Petri net has been described in [2] and [3] as a convenient graphical and mathematical modeling tool allowing for easy representation of concurrency, synchronization and conflict among parts of the modeled system. Other areas of immediate applications include distributed data base system, communication protocols, system programs, computer hardware systems, workflow management and performance study of complex processes [1,10,13,14]. A Petri Net has been defined as a quadruple N = (P, T, F, W) where: P is the set of places (graphically represented as circles) with /P/ = n and P≠∅. T is the set of transitions (graphically represented as bars\) with /T/ = m and T≠∅,T∩P = ∅.F ⊆ (P x T) U (T x P) the flow relation of N. W: F →N\ {0} attaches a weight to each arc of the net. Figure 1 depicts an example of a Petri net.

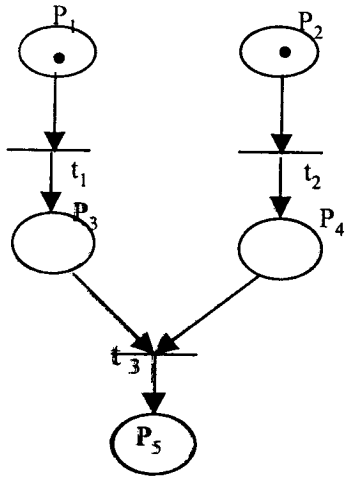

Fig. 1: A Petri net diagram

It consists of places (circle), transitions (bar) and directed arcs (flow relations) that

connect them. Input arcs connect places with transitions, while output arcs start at a transition and end at a place. A place may contain zero or more marking also called weights or tokens. The current state of a modeled system (the marking) is given by the number (and type) of tokens in each place. Transitions are active components and are used in modeling activities (events) which can occur (the transition fires), thus changing the state of the system (the marking of the Petri net). Transitions may fire only if they are enabled, which means that all the preconditions for the activity (or events) must be fulfilled. Usually, this happens if here are enough tokens available in the input places. When the transition fires, it removes tokens from its input places and adds some at its entire output place. The number of tokens removed/added depends on the cardinality of each arc. In figure 1, places PI, P2 initially hold one marker each, as follows:

Before firing: $P_1\ P_2, P_3, P_4, P_5 : = \ 1, 1, 0, 0, 0,$

During firing: $P_1 P_3 : = \ P_1 - I, P_3 + \ 1 \qquad \text{if } P_1 > 0$

$P_2, P_4 : = P_2 - 1, P_4 + \ 1 \qquad \text{if } P_2 > 0$

$P_3 P_4, P_5 : = \ P_3 \ \ - 1 \ , P_4 \ \ - 1, P_5 + l \qquad \text{if } P_3 > 0 \ \& \ P_4 > 0$

After firing: $P_1, P_2, P_3, P_4, P_5 : = \ 0, 0, 0, 0, 1.$

The Petri net in figure 1 may be used to model a program, which compares two (sorted) arrays to determine whether they have the same set of elements. As we all know, one of the capabilities that make computers so useful is their ability to make conditional decisions and to execute different instructions based on the values of data being processed. These decisions are .usually built into language control structures and constitute the program logic. Consequently, Petri nets could be used to model and verify the correct behavior of programs.

2. OVERVIEW OF PROGRAMMING LANGUAGE CONSTRUCTS

In developing computer software, computer scientists and engineers study various methods and techniques of software design. One of such techniques is called structured programming. Structured programming refers to the process of designing algorithms in terms of "structured flowcharts". A flowchart is a directed network having three kinds of vertices-function vertex, predicate vertex and collecting vertex, which are illustrated in figure 2.

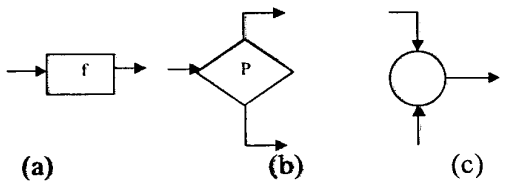

Fig. 2 Types of vertices in flowchart:
(a) function (b) predicate (c) collecting vertex

A function vertex is used to represent a function F: X ® y. A predicate vertex is used to represent a function (or predicate) P: X∈ {T, F} that is, a Boolean expression, which passes control along one of two branches. A collecting vertex represents the passage of control from either of the two incoming branches to one outgoing branch.

A structured flowchart is therefore a flowchart, which is composed of the three primitive flowcharts shown in figure 3.

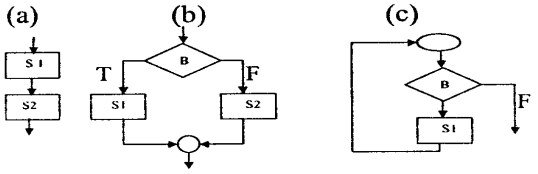

Fig 3: The three primitives of a structured flowchart:

(a) do SI; S2 od (b) if B then Sl else S2 fi (c) While B do SI od

These primitives are known respectively as Sequence, Selection, and Iteration.

Virtually all computer programs that are of some practical interest can be represented by flowcharts. It has also been shown by [4] and [10] that any flowchart can in turn be represented by a structured flowchart. An immediate corollary of this result is that the three primitive flowcharts of figure 3 are sufficient to design any algorithm. The flowcharts of figure 3(a) - (c) are referred to as program control structures. The purpose of a structured flowchart is two-fold:

(i) To enable the programmer to arrive at an acceptable structure for a program by identifying its component modules and their mutual interaction. This is aptly the main purpose.

(ii)To generate semi-automatically the program algorithm by implementing in Pseudocode the control structures indicated in the structured flowchart. This is a by- product

Software designers must limit the number of features included in a program so that it will not require more memory than the system for which it is designed.

3 THE CONCEPTS OF FUNCTIONAL PETRI NETS AND LANGUAGE CONSTRUCTS

A Complex Process (CP) is an ordered 5 - tuple = (S, K, D, Mo, I), where S - Functional Petri Net (FPN); K - set of constructions in the CP; D -set of labels associated with entry points; Mo -initial states or markings of the CP; I - system's Interpreter [2,7,11].

Definition1 A functional Petri net (FPN) has been defined in [2] as a 5 – tuple S = (P,T,F, μ_0, C),where P =P_O UP_v UP_C U P_e - places designated respectively for operators, variables (operands), constants and conditions; P_o= P $_{of}$ UP_{oe} UP_{oi} - that is, an operator place may in turn be functional, logical or iteration; T = T_i, U T_o - input and output transitions; F - incidence matrix or function of the net; μ_0 -initial marking (start) of the net. C = {C_1}, i = 1, n - a set of colors, where C_1 -is the set of marking colors in the ith place (P_i) and n - is the number of places in the net. Consequently,

a FPN has been introduced as an extension of the classical Petri net obtained as a union of E-net [2,7] and colors Petri nets [5]. A FPN must satisfy the following conditions

FPN conditions	Explanations
i. $F(p) \leq 2 \rightarrow p \in \{P_v \cup P_c\}$;	Input places into a constituent module of FPN must be equal to or less than 2
ii. $F(P) \leq 1 \rightarrow \forall P \in P$;	The output place must be less or equal to.1
iii. $F(t') \leq 1 \rightarrow t \in T_i$;	The output from transition must be less or equal to 1
iv. $F(P') \leq 1 \rightarrow P \in P_o$;	The output into elementary constructs F(′P)must be equal or less than 1
v. $F(t') \geq 1 \rightarrow t \in T_1$;	The input to transition must be greater or equal to 1.
vi. $F(t') \geq \rightarrow t \in T_o$;	The output from elementary constructs F(t′) must be greater or equal to 1.

4. MAPPING SOFTWARE DESIGN CONCEPTS ONTO FUNCTIONAL PETRI NETS

Definition 2 In a CP, construction is defined as K = {Ke, Kc,}, where Ke - elementary (or primitive) constructs and Kc - constituent modules (or subgraphs) of the net. Thus, a CP consists of modules, where each module in turn consists of elementary constructs. This is similar in concepts to the Top down design methodology. There are three types of elementary constructs in FPN, depending on the type of operation. Thus, we have Ke = {$K_{ef,}$ K_{el}, K_{ei},}, where K_{ef} K_{eI}, K_{ei}, - are elementary functional, logical and iteration constructs, respectively. Figure 4 shows these three elementary constructs. A functional construct may have n+ I input variables (operands) of which one may be constant and m output variables. The net functions by processing the variables in sequence.

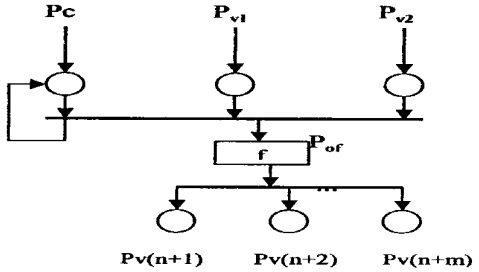

Fig. 4(a): Functional Construct

That is, P_{vi} is processed to yield $P_{v(n+l)}$ before P_{v2} is processed. The syntax is: <FUNC>. [<N>] (<INPl>, [<INP2>]) (<OUTl>,[<OUT2>]), where N is a Petri net label.

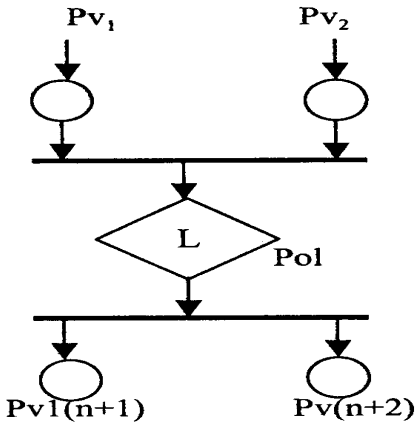

Fig 4(b) Logical construct

The syntax of a logical construct is as follows:
<N> = COND.<L> (<INPl>, <INP2>) (T = <OUT THEN>, E = <OUT ELSE>) <N> =T. <EXP> (<INP1 = OUT THEN> <INPl> ... <INPn>) (<OUTI> ... <OUTm>);<N> = E. <EXP> (<INP2 = OUT ELSE>,<INP1> ... <INPn>) (<OUT1> ... <OUTm>);
Notice in figure 4(b) that the *logical* construct is the key to determining which way the flow will go. It consists of not more than two input variables (places) at a time and two possible outcomes of the logical - true or false. The execution of a statement decreases values of variables corresponding to its input places by 1 (provided they are all positive) and increases values of variables corresponding to its output places by 1, in one multiple assignment.

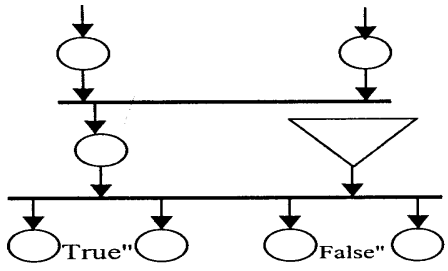

Fig 4 (c) Iteration construct

The iteration construct is adirect consequence of the logical construct. Here, if the condition stated in the logical construct is true, the subnet on the left-hand part will fire, otherwise, execution follows the right- hand part.
Similarly, a composite module Kc = {K_{cf}; K_{cl}}, where K_{cf} - is functional construct; and K_{c1}, -is a logical construct.

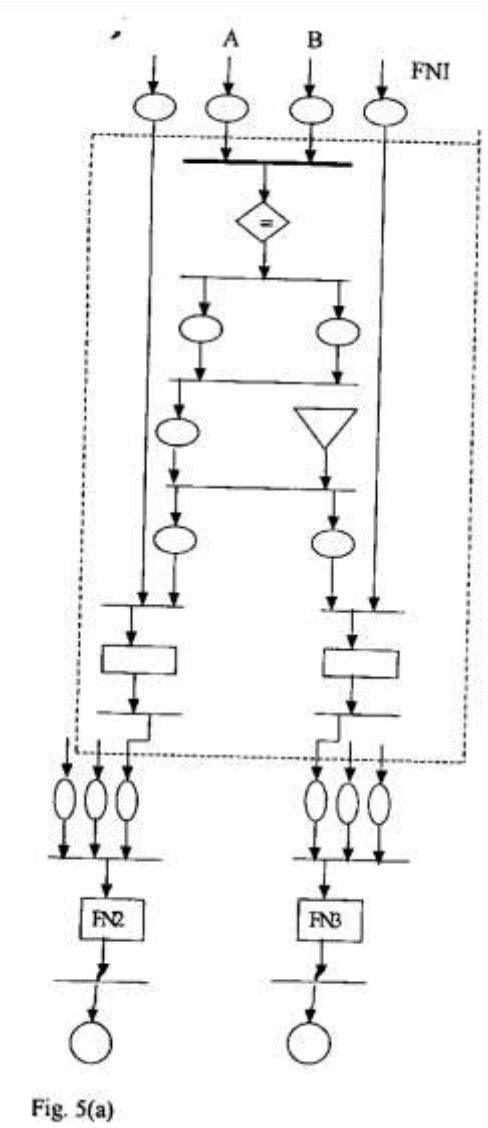

Fig. 5(a)

Definition 3 A composite module K_c, = (P_c,

T_c, F_c μ_{oc} C_c) is a sub graph of FPN in which each place $p \in P_c$, F_c – denotes incidence function, μ_{oc} - initial marking of the net for $p \in P_c$, In a FPN, places are designated as input and output whereas transitions are also designated as input and output. These definitions are as follows:

(i) An input transition of a composite module or sub graph may be defined as: Tic= {t\t Tib Λ $\exists$ p $\in$ {p/P F (t)} Λ p $\in$ P_b}; where Tib - is subsets of input transitions in the composite module and Pb - a set of places in the composite module;

(ii) An output transition of a composite module is also defined as: Toc= {t\ t $\in$ Tob, $\forall$p $\in$ Pn, $\exists$p $\in$ {p \ t $F_{(p)}$ }}.

(iii) An output (variable) place in the composite module may be defined as Pvc= {p\ P $\in$ Pb}.

Figure 5 illustrates a FPN model of the conventional programming language construct: **If A = B then FN2 else FN3.**

An if-then-else block permits one of two different groups of executable statements to be executed, depending on the outcome of a logical test. If the logical expression is true, then the first group of executable statements FN2 will be executed. Otherwise, the second group of executed statements FN3 will be executed. Figure 5 (a) depicts the model of such computation using the three primitive constructs

Definition 4 A set of labels D associated with entry points D = {d_{ov}; d_{bv}, d_{lv}}, where d_{ov}, d_{bv} and d_{lv}, are labels of operand, Boolean, and linked variables, respectively. Labels of operand variables are obtained as output of a functional construct while labels of Boolean variables are obtained as output of logical construct where decisions are taken based on the logical values of True or False. Labels of linked variables are obtained during compilation and refer to the various modules that will be linked together before execution.

Definition 5 In a CP, the system's interpreter is defined as:
I: μ_o X $\mu(P_v)$ = M_o X D, where M_o- denotes initial marking, and D - set of labels. The system's interpreter is discussed in more detail in [2].

5.SOFTWARE PROTOTYPE USING FUNCTIONAL PETRI NETS.

The classical Petri net allows for the modeling of states, events, conditions, synchronization, parallelism, choice, and iteration. However, Petri nets describing real processes tend to be complex and extremely large. Moreover, the classical Petri net does not allow for the modeling of data and time. To solve these problems in [4] and [14] three different extensions of the basic Petri net were proposed. These are:

(1) the extension with color to model data;
(2) the extension with time and
(3) the extension with hierarchy to structure large models.

In this section, the extension with hierarchy is employed. A hierarchy or elementary construct or subnet is an aggregate of a number of places, transitions, and subsystems.

If the dotted portion in figure 5 (a) is replaced by the logical construct (subnet) of FPN, then it is possible to modify figure 5 (a) without changing its hierarchical structure. The equivalent model is depicted in figure 5 (b).

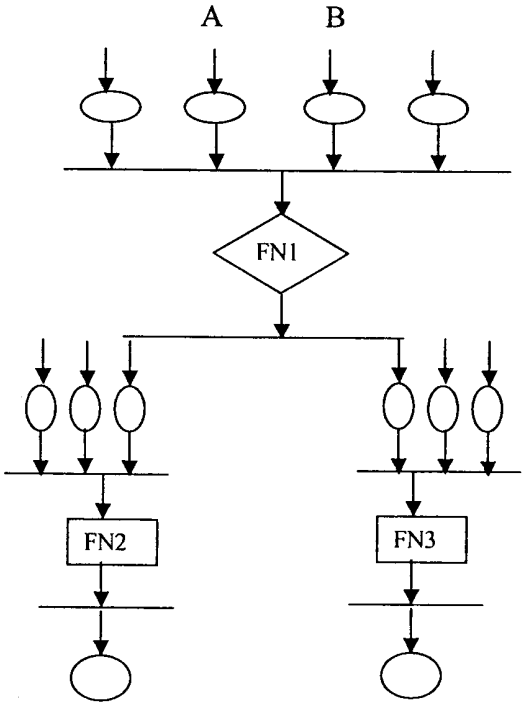

Fig 5 (b) If A = B THEN FN2 ELSE FN3

Figure 5 (b) shows the conventional IF - THEN- ELSE programming statement composed using subnets of FPN. A Petri net extended with these properties may be called a high-level Petri Net. The corresponding software prototype follows:

<d_{dv}> START/* Presence of tokens or variables */

<d_{dv}> INP (<INP1>, <INP2>)

<d_{dv}> IF <L> (<INPl>, <INP2>) (T <OUT - THEN>, E = <OUT-ELSE>);

<d_{dv}> T.<FUNC> (<INPl = OUT-T H E N >, ,<INP1>, ... ,<INPn>) (<OUT1, ... <OU Tm>);

<d_{dv}> E.<FUNC>(<INP2 =OUT - ELSE>, <INPI>,... ,<INPn>) (<OUT1>), ... ,<OUTm>);

<d_{uv}> OUT (<INPl>, <INP2>) (<OUTl>,[<OUT2>]);

<d_{dv}> END/ * Absence of token or variables */

Where d_{dv} denotes FPN labels.

6 ANALYSIS OF CORRECTNESS OF A COMPLEX PROCESS.

Basically, there are three types of analysis that one may examine in the design of a new system, namely:

(i) Validation, i.e., testing whether the system behaves as expected,

(ii) Verification, i.e., establishing the correctness of a system, and

(iii) Performance analysis, i.e., evaluating the ability to meet requirements with respect to throughput times, service levels, and resource utilization. As in the classical Petri nets, all formal proofs are also advocated in this paper. For a detailed review of this analysis the reader is referred to [3, 6, 8, 11]. To meet the needs of this paper, which is mapping of one component onto another, let us examine a property known as **identity of computational modules.**

Definition 6 A Complex Process that is modeled with FPN whose initial and final markings are $\mu_o^{c,}$ μ_f^c respectively, is called *identical and deadlock-free* if there exist a sequence of executed transitions τ, that $\mu_o^{c,} \rightarrow^{\tau} \mu_f^c$ and each transitions t $\in$ T is contained in τ at least once. It is believed that similar computational modules operating at different speed and sequence of execution will produce independent solutions that are identical. This property is also referred to as **identity of solutions.**

7. CONCLUSION

Software design is an application domain, which could benefit from the features of functional Petri nets. For this reason, we have presented in this paper an equivalent FPN Software prototype for the conventional language *control* structures. The concepts discussed in this paper, if properly harnessed will revitalize the existing structured programming method for the following reasons:

* FPN provides formal semantics despite its graphical nature,
* FPN models the occurrence of events by the state,
* FPN provides an abundance of analysis techniques.

REFERENCES

[1] Agerwala, T "Putting Petri Nets to work" *Computer* 12 *(1*2), 1979.

[2] Bakpo, F.S "Elaboration of an Interpreter for Functional Petri Nets' language in a Dataflow computational system". M Eng thesis (in Russian), Dept. of Electronic/Computer Engineering, Kazakh National Technical University, Almaty, 1994.

[3] Economopoulos, P "Petri Nets: A model for the analysis of the behavior and performance of concurrent systems". *Office and Database Systems Research* '87, *PP*. 99 - 132,Toronto, 1987.

[4] Hauschildt, D; Verbeek, H M W and Ven der Aalst, W M P " WOFLAN: a Petri-net- based Workflow Analyzer". *Computing Science Reports 97/12, Eindhoven University of Technology, Eindhoven, 1997.*

[5] Jensen, K "Coloured Petri Nets: Basic concepts, analysis methods and practical use", EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1996.

[6] Muler,R. E "A comparison of some theoretical models of parallel computation", *IEEE Trans. Computers, 710, 1973.*

[7] Noe, J. D and Nutt,G. J "Macro E-Nets for Representation of Parallel Systems", *IEEE Trans. Computers. Vol. C-22: PP* 718-727, 1973.

[8] Peterson, J "Petri Nets Theory and the Modeling of Systems", Prentice Hall, 1981.

[9] Petri, C. A "Kommunikation mit Automaten". Ph D thesis, Institut fur instrumentelle Mathematik, Bonn, 1962.

[10] Plax, T. P "Synthesis of parallel programs on computational models", *Programming, No* 1, *PP*. 55 - 63, 1977.

[11] Sifakis, J "Structural properties of Petri Nets", *Mathematical Foundations of Compo Sci,* 64, *PP*. 474 - 483, Springer-Verlag, Berlin, 1978.

[12] van der Aalst, W. M. P " Putting Petri nets to Work in Industry". co*mputers ill Industry*, 25: 45-54, 1994.

[13] van der Aalst, W. M. P "Petri-net-based Workflow Management Software", *Computing Science Reports 2003, Eindhoven University of Technology, Eindhoven, 2003.*

[14] van der Aa1st, W. M. P "The Application of Petri Nets to Workflow Management", *Computing Science Reports 2002, Eindhoven University of*

Technology, *Eindhoven*, 2002.