IEEE Int. Conf. on Systems, Man, and Cybernetics, Tokyo, Japan, 12-15 October 1999, pp.841-847.

Petri Net Models of Process Synchronization Mechanisms

W.M. Zuberek
Department of Computer Science
Memorial University of Newfoundland
St.John's, Canada A1B 3X5

Abstract

Inhibitor Petri net models of several popular process synchronization mechanisms are presented and discussed. Semaphores and extended semaphores, monitors and rendezvous concepts are used in simple examples of process synchronizations. The corresponding Petri net models are used to verify basic properties such as mutual exclusion, presence or absence of deadlocks, or priorities in accessing shared resources.

1. Introduction

Petri nets are formal, mathematical models of systems with asynchronous concurrent activities [1, 18, 14]. Examples of such systems include multiprocessor computer systems, distributed databases and real-time industrial process control systems. As a modeling tool, Petri nets offer a simple and general formalism for representation of concurrent activities and synchronization of events, with a well–development formal foundation for analysis of such models.

At a higher level of abstraction, Petri nets can model many synchronization and coordination mechanisms developed for concurrent programming [3, 20]; mutual exclusion in accessing shared information and message passing in distributed systems are examples of simple applications of such mechanisms. Due to developments in processor technology, multiprocessor systems constructed from a number of similar self-contained processors, are becoming quite popular. In order to use such systems on a single task, the component processors must be able to communicate and to synchronize with each other. Many methods have been proposed for such synchronization: semaphores, critical and conditional critical regions, monitors, path expressions, rendezvous and others. Although they have been demonstrated to be adequate for their purpose, there is no widely recognized criterion for choosing among them.

The main goal of this paper is to show that many diverse synchronization mechanisms can be compared within one, uniform framework of (inhibitor) Petri net models. Moreover, simple properties of Petri nets (e.g., boundedness, absence of deadlocks) can be used for verification of typical synchronization problems. This can help to understand and clarify different notations that have been proposed in the literature to deal with parallelism.

This paper is organized in five main sections. Section 2 recalls basic concepts for inhibitor Petri nets. Semaphores with some extensions are discussed in Section 3. Monitors and their models are briefly presented in Section 4, and the concept of rendezvous – in Section 5. All synchronization mechanisms are presented rather informally, using a simple 'standard' programming notation. Several classical synchronization problems, with well–known correct and incorrect solutions, are used as illustrations.

2. Inhibitor Petri nets

An *inhibitor* (place/transition, ordinary) Petri net [18, 14] is a quadruple $\mathbf{N} = (P, T, A, B)$, where P is a finite, nonempty set of places, T is a finite, nonempty set of transitions, A is a set of directed arcs connecting places with transitions and transitions with places, $A \subseteq P \times T \cup T \times P$, and B is a finite set of inhibitor arcs connecting places with transitions, which is disjoint with A, $B \subset P \times T$, $A \cap B = \emptyset$.

A place p is an input (or an output) place of a transition t iff there exists an arc (p,t) (or (t,p), respectively) in the set A. The sets of all input and output places of a transition t are denoted by $Inp(t)$ and $Out(t)$, respectively. Similarly, $Inp(p)$ and $Out(p)$ denote the sets of input and output transitions of a place p. A place p is an inhibitor place of a transition t iff there exists an inhibitor arc (p,t) in the set B.

A *marked* inhibitor net $\mathbf{M}$ is a pair $\mathbf{M} = (\mathbf{N}, m_0)$ where $\mathbf{N}$ is an inbibitor Petri net, $\mathbf{N} = (P, T, A, B)$, and m_0 is an initial marking function which assigns a nonnegative integer number of so called tokens to each place of the net, $m_0 : P \to \{0, 1, ...\}$.

Let any function $m : P \to \{0, 1, ...\}$ be called a *marking* of a net $\mathbf{N} = (P, T, A, B)$.

A transition t is *enabled* by a marking m iff every input place of this transition contains at least one token and every inhibitor place of t contains zero tokens. The set of all transitions enabled by a marking m is denoted by $E(m)$.

Every transition enabled by a marking m can fire. When a transition fires, a token is removed from each of its input places (but not inhibitor places) and a token is added to each of its output places. This determines a new marking in a net, a new set of enabled transitions, and so on.

A marking m_j is *directly reachable* from a marking m_i in a net $\mathbf{N}$ iff there exists a transition t_k enabled by the marking m_i, $t_k \in E(m_i)$, such that for all $p \in P$:

$$m_j(p) = \begin{cases} m_i(p) - 1, & \text{if } p \in Inp(t_k) - Out(t_k), \\ m_i(p) + 1, & \text{if } p \in Out(t_k) - Inp(t_k), \\ m_i(p), & \text{otherwise.} \end{cases}$$

Also, a marking m_j is (generally) *reachable* from a marking m_i in a net $\mathbf{N}$ iff there exists a sequence of directly reachable markings $(m_{i_0} m_{i_1} m_{i_2} ... m_{i_k})$ such that $m_{i_0} = m_i$ and $m_{i_k} = m_j$.

A *set of reachable markings*, $M(\mathbf{M})$, of a marked net $\mathbf{M} = (\mathbf{N}, m_0)$ is the set of all markings which are reachable from the initial marking m_0 in the net $\mathbf{N}$.

A *marking graph* $\mathbf{G}(\mathbf{M})$ of a marked Petri net $\mathbf{M}$ is a directed graph $\mathbf{G}(\mathbf{M}) = (W, D)$ where W is a set of vertices which is equal to the set of reachable markings of the net $\mathbf{M}$, $W = M(\mathbf{M})$, and D is a set of directed arcs, $D \subset W \times W$, such that (m_i, m_j) is in D iff m_j is directly reachable from m_i in $\mathbf{M}$. Quite often additional information is attached to vertices or arcs of a marking graph. In particular, the arcs connecting the nodes (i.e., markings) can be labeled by the firing transitions.

Marking graphs provide complete behavioral characterization of marked nets. One of the most important behavioral properties of nets is *boundedness*; a marked net $\mathbf{M}$ is *bounded* iff its set of reachable markings $M(\mathbf{M})$ is finite. For nets without inhibitor arcs many properties can be deduced from the structure of the net [6]; for bounded inhibitor nets, such structural properties are often insufficient, so the set (or graph) of reachable markings is used for further analyses.

Each net $\mathbf{N} = (P, T, A, B)$ can conveniently be represented by a connectivity (or incidence) matrix $\mathbf{C} : P \times T \to \{-1, 0, +1\}$ in which places correspond to rows, transitions to columns, and for all $p \in P$ and all $t \in T$, the entries are defined as:

$$\mathbf{C}[p,t] = \begin{cases} -1, & \text{if } t \in Out(p) - Inp(t), \\ +1, & \text{if } t \in Inp(p) - Out(p), \\ 0, & \text{otherwise.} \end{cases}$$

If a marking m_j is obtained from another marking m_i by firing a transition t_k then (in vector notation) $m_j = m_i + \mathbf{C}[k]$, where $\mathbf{C}[k]$ denotes the k-th column of $\mathbf{C}$, i.e., the column representing t_k.

Connectivity matrices disregard inhibitor arcs and 'selfloops', that is, pairs of arcs (p,t) and (t,p); any firing of a transition t cannot change the marking of p in such a selfloop. A pure net is defined as a net without selfloops [18].

A P–invariant (place invariant) [18, 14, 6] of a net $\mathbf{N}$ is any nonnegative, nonzero integer (column) vector I which is a solution of the matrix equation

$$\mathbf{C}^T \times I = 0,$$

where $\mathbf{C}^T$ denotes the transpose of matrix $\mathbf{C}$. It follows immediately from this definition that if I_1 and I_2 are P–invariants of $\mathbf{N}$, then also any linear (positive) combination of I_1 and I_2 is a P–invariant of $\mathbf{N}$.

A basic P–invariant of a net is defined as a P–invariant which does not contain simpler invariants. All basic P–invariants I of ordinary nets are binary vectors [18], $I : P \to \{0, 1\}$.

A net $\mathbf{N}_i = (P_i, T_i, A_i, B_i)$ is a P_i-implied subnet of a net $\mathbf{N} = (P, T, A, B)$, $P_i \subset P$, iff:

(1) $T_i = \{t \in T \mid \exists\, p \in P_i : (p,t) \in A \vee (t,p) \in A\}$,

(2) $A_i = A \cap (P_i \times T \cup T \times P_i)$,

(3) $B_i = B \cap (P_i \times T_i)$.

It should be observed that in a pure net $\mathbf{N}$, each P–invariant I of $\mathbf{N}$ determines a P_I-implied (invariant) subnet of $\mathbf{N}$, where $P_I = \{p \in P \mid I(p) > 0\}$; all nonzero elements of I select rows of $\mathbf{C}$, and each selected row i corresponds to a place p_i with all its input (+1) and all output (–1) arcs associated with it.

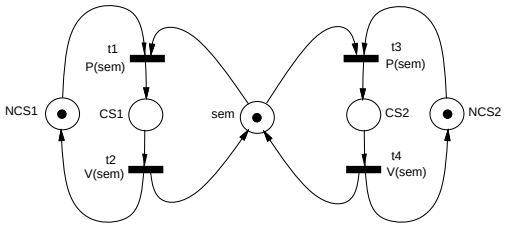

Fig.1. Net model of mutual exclusion.

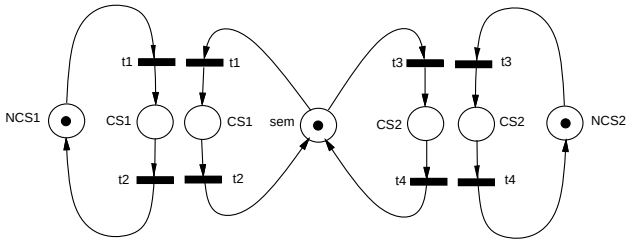

Fig.2. P–invariant–implied subnets for Fig.1.

For the Petri net shown in Fig.1, the connectivity matrix is:

$\mathbf{C}$	t_1	t_2	t_3	t_4
$NCS1$	-1	$+1$	0	0
$CS1$	$+1$	-1	0	0
sem	-1	$+1$	-1	$+1$
$CS2$	0	0	$+1$	-1
$NCS2$	0	0	-1	$+1$

and there are three basic P–invariants, $I_1 = [1,1,0,0,0]$, $I_2 = [0,1,1,1,0]$, and $I_3 = [0,0,0,1,1]$. It can be observed that the basic invariants correspond to the smallest subsets of rows of the connectivity matrix with the (component–wise) sums equal to (vector) zero.

The P_I–implied subnets are simple nets in which all transitions have single input and single output places, as shown in Fig.2. Consequently, the total number of tokens in each P–invariant subnet remains the same for all reachable marking. If a net is covered by such P–invariants, it is bounded for any initial marking m_0.

Finding basic P–invariants is a 'classical' problem of linear algebra [12, 6].

3. Semaphores

Providing mutual exclusion for a set of concurrent processes which access common (or shared) data is one of the basic synchronization problems [3, 17]. Usually such an exclusive access to shared data is localized within *critical sections*, so a synchronization mechanism is needed to guarantee that, at any time, at most one of concurrent processes is in its critical section.

An elegant solution to the mutual exclusion problem was proposed by Dijkstra in the form of *semaphores* [7]. Informally, a (counting) semaphore is an integer variable with just two indivisible operations called P (test and decrement) and V (increment). A process executing a P operation must wait until the semaphore is positive before it can decrement the semaphore's value and continue. A V operation simply increases the semaphore's value, possibly allowing some other process to execute a delayed P operation and continue. No two P or V operations on the same semaphore can be executed simultaneously.

A simple solution to mutual exclusion of two cyclic processes, *Process*1 and *Process*2, with their critical sections CS1 and CS2, respectively, uses a global semaphore *sem*, initialized to 1, as shown in Tab.1.

var sem : **semaphore** = 1;	
Process1: **process**;	Process2: **process**;
begin loop	**begin loop**
.....	.....
P(sem);	P(sem);
critical section 1;	critical section 2;
V(sem);	V(sem);
.....	.....
end loop	**end loop**
end process;	**end process**;

Tab.1. Mutual exclusion using semaphores.

A Petri net model of this solution is shown in Fig.1. The semaphore *sem* is represented by a place with the initial marking representing its initial value (i.e., 1 in this case). The processes are represented by two cyclic subnets in which P and V operations are modeled by transitions with arcs from (for P operations) or to (for V operations) *sem*; each P operations requires a positive value of the semaphore (otherwise the transition cannot be enabled), and each V operation increases the number of tokens in the semaphore by one.

It should be observed that the semaphore *sem* and both critical sections belong to one of the P–invariant–implied subnets shown in Fig.2. Since the initial marking assigns only one token to this subnet (the initial value of *sem*), the places CS1 and CS2 cannot be marked simultaneously, so at most one of critical sections can be 'active' at any time. Consequently, the solution guarantees the mutual exclusion of critical sections. Moreover, the model can easily be extended to any number of processes with any number of interactions between processes (controlled by identical or independent semaphores). Then, however, deadlocks can be created.

Semaphores are often used in resource allocation systems providing exclusive use of (shared) resources. Tab.2 shows two cyclic processes, *Process*1 and *Process*2, dynamically requesting (P operations) and releasing (V operations) two resources $r1$ and $r2$ controlled by semaphores $R1$ and $R2$. It is known [16] that in such *single–request systems*, immediate granting of requests may result in a deadlock.

var R1,R2 : **semaphore** = 1,1;	
Process1: **process**;	Process2: **process**;
begin loop	**begin loop**
.....	.....
P(R1);	P(R2);
P(R2);	P(R1);
use r1,r2;	use r1,r2;
V(R2);	V(R1);
V(R1);	V(R2);
.....	.....
end loop	**end loop**
end process;	**end process**;

Tab.2. Single-resource allocation using semaphores.

A Petri net model of this process synchronization is shown in Fig.3, and its reachability graph in Fig.4. The node 9 clearly indicates a deadlock which can be reached by executing P(R1) operation by *Process*1 and then P(R2) operation by *Process*2 (or first P(R2) by *Process*2 and then P(R1) by *Process*1).

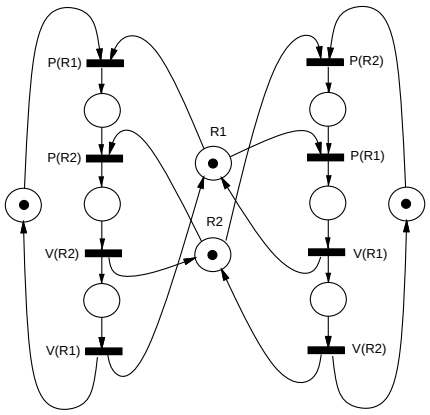

Fig.3. Resource allocation using semaphores.

It should be noted that since the net in Fig.3 does not use inhibitor arcs, the deadlock can also be identified by structural methods in this case [6].

The possibility of a deadlock in such resource allocation schemes is well known in operating system theory [19], and it can be avoided either by *ordered resource policy* or by *general requests* in which all resources are requested (and allocated) simultaneously.

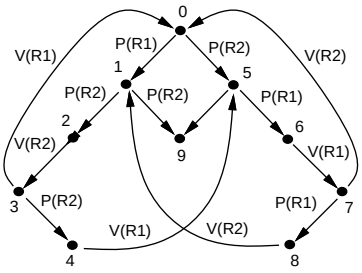

Fig.4. Marking graph for the net shown in Fig.3.

The second solution uses multiple semaphore operations, i.e., P and V operations which update simultaneously a list of semaphores [2], as shown in Tab.3, with a net model shown in Fig.5.

var R1,R2 : **semaphore** = 1,1;	
Process1: **process**;	Process2: **process**;
begin loop	**begin loop**
.....	.....
P(R1,R2);	P(R1,R2);
use r1, r2;	use r1, r2;
V(R1,R2);	V(R1,R2);
.....	.....
end loop	**end loop**
end process;	**end process**;

Tab.3. Resource allocation with multiple semaphores.

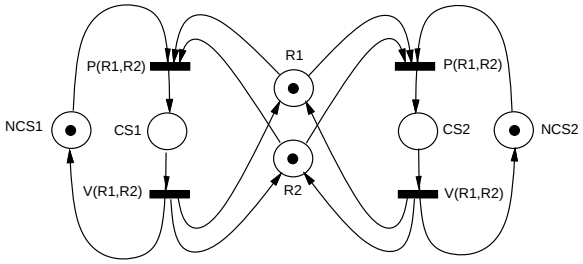

Fig.5. Resource allocation with multiple semaphores.

Modeling of systems with priorities of some operations has resulted in the discovery of some limitations of Petri net models [11]. Inhibitor arcs has been proposed as an extension of basic Petri nets [1, 14], and it has been shown that Petri nets with inhibitor arcs are equivalent, with respect to 'modeling power', to Turing machines. Readers and writers problem [16] is a good illustration of limitations of Petri nets without inhibitor arcs.

A classical solution to this problem uses three semaphores [2], two counting semaphores *nr* and *nw*, both initialized to zero, and a binary semaphore *s* initialized to one. *Extended semaphore* operations P and V can be performed on several semaphores simultaneously, and also P operations can test semaphores for 'zero conditions' (if the second list, separated by ";", is nonempty) [2]. The solution shown in Tab.4 assumes two classes of identical *Reader* and *Writer* processes (in general case the processes may be different).

var nw,nr,s : **semaphore** = 0,0,1;	
Reader: **process**;	Writer: **process**;
begin loop	**begin loop**
.....	.....
P(s;nw);	V(nw);
V(s,nr);	P(s;nr);
read;	write;
P(nr);	V(s);
.....	P(nw);
.....	.....
end loop	**end loop**
end process;	**end process**;

Tab.4. Readers-writers synchronizations using extended semaphores.

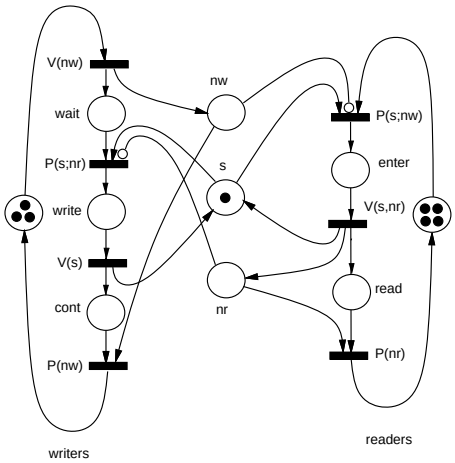

Fig.6. Readers–writers synchronization.

A Petri net model of this solution is shown in Fig.6 (inhibitor arcs have small circles instead of arrows). By analyzing P–invariants and the set of reachable markings, it can be shown that the solution provides priority of *Writer* processes over *Reader* ones (i.e., when a *Writer* process is 'ready', no new *Reder* processes are allowed to enter their "read" section), that *Reader* processes have concurrent access to "reading", and that there is mutual exclusion of *Reader* and *Writer* processes.

4. Monitors

An approach inspired by the *class* concept of Simula–67, and called *monitor* [4], is formed by encapsulating both, the shared data objects and operations that manipulate them:

```
<monitorname> : monitor;
   <decls of common variables and conditions>
   <definitions of monitor procedures>
   <definitions of other (local) procedures>
begin <initialization code> end;
```

A monitor consists of a collection of variables that can be manipulated by all monitor procedures (but

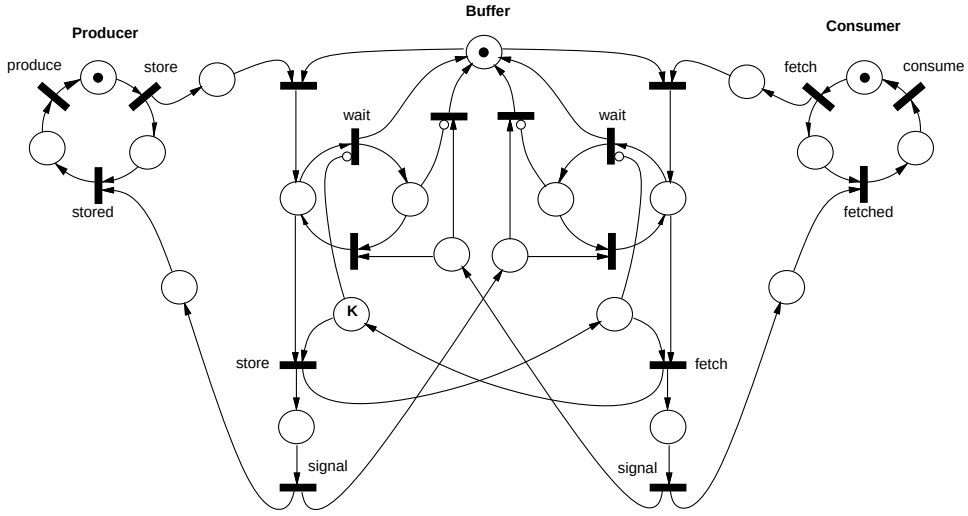

Fig.7. Model of a bounded–buffer monitor.

which are inaccessible from outside the monitor), a set of monitor procedures that are used for manipulations of monitor variables and which are invoked by prefixed invocations:

```
<monitorname> . <proc-name> ( <list of args> )
```

and a set of local procedures used by monitor procedures only. The `<initialization code>` is executed when a monitor is created.

Execution of all monitor procedures is 'automatically' guaranteed to be mutually exclusive. This ensures that the monitor common variables are never accessed by more than one process. Moreover, special *condition* variables are used to delay processes executing monitor procedures. Two operations, *signal* and *wait*, are defined for condition variables. If x is a condition variable, then execution of $x.wait$ causes the invoking process to be blocked on x and to relinquish its mutually exclusive control of the monitor. However, there are several 'interpretations' of the operation *signal* [16]. In one [4], the process invoking the operation immediately leaves the monitor making it available for the reactivated process (the invocation of *signal* operation is required to be the last statement of the corresponding monitor procedure). In another interpretation [9], the execution of $x.signal$ depends upon the condition of the variable x; if there is no process blocked on x, the invoking process continues, otherwise the invoking process is temporarily suspended and one process blocked on x is reactivated and continued; the process suspended due to a *signal* operation continues when there is no other process executing in the monitor. Also, such processes are given priority over processes trying to begin execution of monitor procedures.

The monitor implementation of the bounded-buffer producer-consumer scheme uses a monitor *Buffer* with (buffer) operations *store* and *fetch* (the actual buffer is represented by an array B of K data elements), as shown in Tab.5. The buffer is accessed by cyclic *Producer* and *Consumer* processes shown in Tab.6.

```
Buffer : monitor;
var B : array [1..K] of data;
    first,last,count : integer;
    empty,full : condition;
procedure store (x : data);
begin if count=K then full.wait;
    B[last] := x;
    last := (last mod K)+1;
    count := count+1;
    empty.signal
end;
procedure fetch (var x : data);
begin if count=0 then empty.wait;
    x := B[first];
    first := (first mod K)+1;
    count := count-1;
    full.signal
end;
begin count := 0; first := 1; last := 1 end;
```

Tab.5. Bounded-buffer as a monitor.

```
Producer: process;
var item : data;
begin loop
    produce(item);
    Buffer.store(item)
end loop
end process;
```

```
Consumer: process;
var item : data;
begin loop
    Buffer.fetch(item);
    consume(item)
end loop
end process;
```

Tab.6. Producer and consumer processes.

Fig.7 shows a Petri net model of the producer–consumer bounded–buffer monitor; inhibitor arcs in

this model are used to indicate priorities of simultaneous events. It can be verified that the model is coverd by simple P–invariants, so it is bounded and provides mutual exclusion of its operations; reachability analysis can be used to verify correctness of priorities.

5. Rendezvous

Rendezvous is an intertask communication and synchronization mechanism introduced in Ada. The primary mechanisms is composed of *accept* statements:

```
accept <entryname> ( <parameters> ) do <body> end
```

and entry calls:

```
<taskname> . <entryname> ( <parameters> )
```

For two cooperating tasks (or processes), T_1 and T_2, let task T_1 issue a call of an entry of task T_2. There are two possible executions:

1. The entry call is issued before T_2 reaches the corresponding *accept* statement; in this case T_1 is suspended until T_2 reaches its *accept* statement and completes the execution of its body.
2. The *accept* statement is reached by T_2 before a call is received on that entry; in this case T_2 is suspended until T_1 issues its call.

As soon as T_2 reaches its *accept* statement and a call of the corresponding entry is issued by T_1, T_1 is suspended while T_2 executes the body of its *accept* statement, after which both tasks can continue their executions. This interaction is called *rendezvous*. It should be noted that the mechanism is 'asymmetric' since the same *accept* entry can be called by many tasks (there is a queue of tasks associated with each *accept* entry), while a call of an *accept* entry always uniquely identifies the entry.

An *accept* statement can be embedded within a *select* statement which provides a form of a (nondeterministic) multiple choice statement:

```
select  when <condition-1> => <statements-1>
    or  when <condition-2> => <statements-2>
    or  when ...
    else  <statements>
end select
```

Execution of a *select* statement is composed of three consecutive steps [21]:

1. all *when* conditions are evaluated to determine which alternatives are "open";
2. an open alternative `<statements-j>` (i.e., an alternative for which the corresponding `<condition-j>` is satisfied) is selected; an open alternative starting with an *accept* statement may be selected only if the corresponding rendezvous is possible;
3. `<statements-j>` or, if all alternatives are "closed", the *else* body `<statements>` is executed.

Bounded–buffer producer–consumer scheme is again used as an example, and the "Buffer" task is shown in Tab.7, while cyclic *Producer* and *Consumer* tasks are shown in Tab.8.

```
task body Buffer is
B : array (1..K) of data;
first,last: integer range 1..K := 1,1;
count : integer range 0..K := 0;
begin loop
    select
        when count < K =>
            accept store (item : in data)
                do B(last) := item end;
            last := (last mod K)+1;
            count := count+1
    or when count > 0 =>
            accept fetch (item : out data)
                do item := B(first) end;
            first := (first mod K)+1;
            count := count-1
    end select
end loop
end Buffer;
```

Tab.7. Bounded-buffer task.

```
task body Producer is          task body Consumer is
item : data;                   item : data;
begin loop                     begin loop
    produce item;                  Buffer.fetch(item);
    Buffer.store(item)             consume item
end loop                       end loop
end Producer;                  end Consumer;
```

Tab.8. Producer and consumer tasks.

A Petri net model of bounded-buffer task synchronization is shown in Fig.8; it looks rather simple when compared with Fig.7. The model is covered by P–invariants, so it is bounded and guarantees mutual exclusion of its operations.

A more detailed description of intertask communication and synchronization, including aborts and exceptions during rendezvous, is given in [10].

6. Concluding remarks

It has been shown that inhibitor Petri nets can represent many different process synchronization concepts. Moreover, simple properties of nets (e.g., boundedness, absence of deadlocks) are very useful in verification of concurrent programs.

Similar net models can be derived for other concurrent programming constructs, such as path expressions [8], or CSP's input and output commands [9].

Semaphores (basic as well as extended) provide a "low-level" synchronization mechanism, which is very flexible but also errorprone, so it must be used in a rigorous, consistent way in complex applications. More structured constructs restrict all accesses to shared objects to clearly identified sections of code; they also provide mutual exclusion of shared data within such sections. Monitors protect access to shared data by integrating the data and operations performed on them within one structure; in fact, the shared data are accessible only "through" corresponding operations.

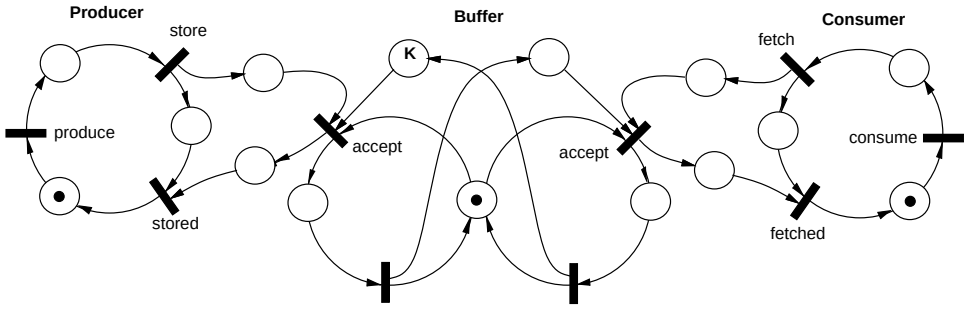

Fig.8. Model of a producer–consumer bounded–buffer synchronization.

This may appear too inflexible in practical applications since nested monitors as well as mutual monitor operations (i.e., monitor procedures invoked from other monitors) must be used very cautiously because of possibility of deadlocks. Rendezvous concept which provides a synchronous [15] one-way naming mechanism, is quite simple as a basic idea, however, it is considerably obscured by many "special cases", exceptions, etc. [13, 10]. Therefore successors to existing concurrent programming concepts should probably return to attractive simplicity of semaphores, but also provide a finely grained selective access control and dynamic control of access rights that could be passed from one process to another, independently of the static structure of their definitions. The development of a comprehensive set of concurrent programming primitives must be based on a better understanding of the nature of concurrency, and this remains an important topic for further research. This further research, however, cannot simply ignore several decades of work in this area, work that contributed many important, if not generally appreciated, results [5].

Acknowledgements

The Natural Sciences and Engineering Research Council of Canada partially supported this research through Research Grant OGP-8222.

References

[1] Agerwala, T., "Putting Petri nets to work"; *IEEE Computer*, vol.12, no.12, pp.85-94, 1979.

[2] Agerwala, T., "Some extended semaphore primitives"; *Acta Informatica*, vol.8, no.3, pp.201-220, 1977.

[3] Andrews, G.R., Schneider, F.B., "Concepts and notations for concurrent programming"; *ACM Computing Surveys*, vol.15, no.2, pp.3-43, 1983.

[4] Brinch Hansen, P., "The programming language Concurrent Pascal"; *IEEE Trans. on Software Engineering*, vol.1, no.3, pp.199-207, 1975.

[5] Brinch Hansen, P., "Java's insecure parallelism"; *SIGPLAN Notices*, vol.34, no.4, pp.38-45, 1999.

[6] Desel, J., "Basic linear algebraic techniques for place/transition nets"; in: *Lectures on Petri Nets I: Basic Models* (Lecture Notes in Computer Science 1491), pp.257-308, Springer–Verlag 1998.

[7] Dijkstra, E., "Cooperating sequential processes"; in: *Programming Languages*, Genuys, F. (ed.), pp.43-112, Academic Press 1968.

[8] Campbell, R.H., Habermann, A.N., "The specification of process synchronization by path expressions"; in: *Operating Systems* (Lecture Notes in Computer Science 16), pp.89-102, Springer–Verlag 1974.

[9] Hoare, C.A.R., "Communicating sequential processes"; *Communications of the ACM*, vol.21, no.8, pp.666-677, 1978.

[10] Gedela, R.K., Shatz, S.M., "Modeling of advanced tasking in Ada-95: a Petri net perspective"; Proc. 2-nd Int. Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97), pp.4-14, 1997.

[11] Kosaraju, S.R., "Limitations of Dijkstra's semaphore primitives and Petri nets"; *Operating Systems Review*, vol.7, no.4, pp.122-126, 1973.

[12] Krueckeberg, F., Jaxy, M., "Mathematical methods for calculating invariants in Petri nets"; in: *Advances in Petri Nets 1987* (Lecture Notes in Computer Science 266), G. Rozenberg (ed.), pp.104-131, Springer–Verlag 1987.

[13] Mandrioli, D., Zicari, R., Ghazzi, C., Tisato, F., "Modeling the Ada task system by Petri nets"; *Computer Languages*, vol.10, no.1, pp.43-62, 1985.

[14] Murata, T., "Petri nets: properties, analysis and applications"; *Proceedings of IEEE*, vol.77, no.4, pp.541–580, 1989.

[15] Murata, T., Shenker, B., Shatz, S.M., "Detection of Ada static deadlocks using Petri net invariants"; *IEEE Trans. on Software Engineering*, vol.15, no.3, pp.314-326, 1989.

[16] Peterson, J.L., Silberschatz, A., *Operating Systems Concepts* (5-th ed.); Addison–Wesley 1998.

[17] Raynal, M., *Algorithms for Mutual Exclusion*; MIT Press 1986.

[18] Reisig, W., *Petri nets - an introduction* (EATCS Monographs on Theoretical Computer Science 4); Springer–Verlag 1985.

[19] Shaw, A.C., *The Logical Design of Operating Systems*; Prentice Hall 1974.

[20] Wegner, P., Smolka, S.A., "Processes, tasks, monitors: a comparative study of concurrent programming primitives"; *IEEE Trans. Software Engineering*, vol.9, no.4, pp.446-462, 1983.

[21] Welsh, J., Lister, A., "A comparative study of task communication in Ada"; *Software Practice and Experience*, vol.11, no.3, pp.257-290, 1981.