Petri Nets and the Representation of Standard Synchronizations

Lee W. Cooprider

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213
ARPAnet address: Cooprider@CMUA

January 1976

Abstract: This paper discusses the use of Petri Nets [Hol70] in representing synchronizations commonly found in operating systems and other multiple process programs. Several examples are presented and analysed with respect to verification and properties. The suitability of Petri Nets for this purpose is commented upon.

1. Petri Nets
1.1. Introduction 1
1.2. Other varieties 1
1.3. Representation of Processes 3
1.4. Common Subgraphs 4
2. Proof of Properties of Petri Nets 4
2.1. Introduction 6
2.2. Concurrency Restrictions 6
2.3. General Properties 6
2.3.1. Freedom from Deadlock 6
2.4. Siphons and Traps 6
2.5. Busy Waiting 7
2.6. Indefinite Delay 8
2.7. Unnecessary Constraint 9
3. Examples of Petri Net Synchronizations 9
3.1. Introduction 10
3.2. Mutual Exclusion 10
3.2.1. Problem Statement 11
3.2.2. Definitions 11
3.2.3. Notes 11
3.3. Dining Philosophers 12
3.3.1. Problem Statement 12
3.3.2. Definitions 12
3.3.3. Notes 13
3.4. Cigarette Addicts 13
3.4.1. Problem Statement[Pat71] 14
3.4.2. Definitions 14
3.4.3. Notes 15
3.5. Reader/Writer 1 16
3.5.1. Problem Statement[Cou71] 16
3.5.2. Definitions 16
3.5.3. Notes 16
3.6. Reader/Writer 2 17
3.6.1. Problem Statement [Cou71] 17
3.6.2. Definitions 17
3.6.3. Notes 18
3.7. Producer/Consumer Bounded Buffer 19
3.7.1. Problem Statement [Dij68] 20
3.7.2. Definitions 20
3.7.3. Notes 20
3.8. Producer/Consumer Multiple Streams 21
3.8.1. Problem Statement[Dij72] 21
3.8.2. Definitions 21
3.8.3. Notes 22
4. Petri Net Theorems 23
4.1. Liveness Conditions 23
4.2. Markings 23
4.3. Decidability 24
5. Discussion 24
5.1. Adequacy of Petri Nets for Synchronization 24
5.2. Suitability 24
26

5.3. Other Approaches 26

1. Petri Nets

1.1 Introduction

Petri Nets[Pet62], [Hol70] are directed graphs with two types of vertices, places (or conditions) and transitions (or events). An arc in a Petri Net can connect only dissimilar vertices, that is, a place to a transition or a transition to a place. Places are usually denoted by circles, transitions by bars or dots.

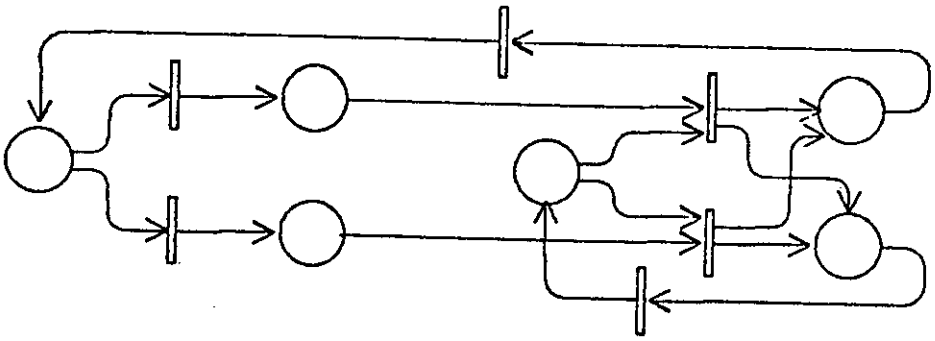

Figure 1: Petri Net Graph

In addition, the places of a Petri Net are occupied by zero or more tokens; any allocation of tokens to the places of a Petri Net is called a marking. Often the description of a Petri Net includes its initial marking.

An arc from a place to a transition designates an input place to that transition; an arc from a transition to a place designates an output place from a transition. When there is a token on every input place to a transition, it is enabled and may fire, otherwise it is disabled. If a transition fires, it takes one token from every input place and places one token on every output place.

Petri Nets are interpreted by selecting sequences of firings. Any enabled transition is selected and the marking of the Petri Net altered by the rule stated above. Another enabled transition is then selected and the net marking altered again. This process is repeated indefinitely as long as there remains an enabled transition. Any marking which can be obtained in this manner is reachable from the initial marking.

Note that the firing of one transition may disable another transition which was previously enabled. This can happen when two transitions share an input place; this configuration in a Petri Net is called conflict.

In the example in Figure 2, transitions "a" and "b" are enabled, but are in conflict so only one can fire. If "a" fires, transition "c" is the only enabled transition in the net

(see Figure 3). When it fires, tokens are placed on places 5 and 6, enabling both transitions "e" and "f" (Figure 4). If "e" fires next, and then "f" fires before "a" or "b" fires, the net returns to its initial marking.

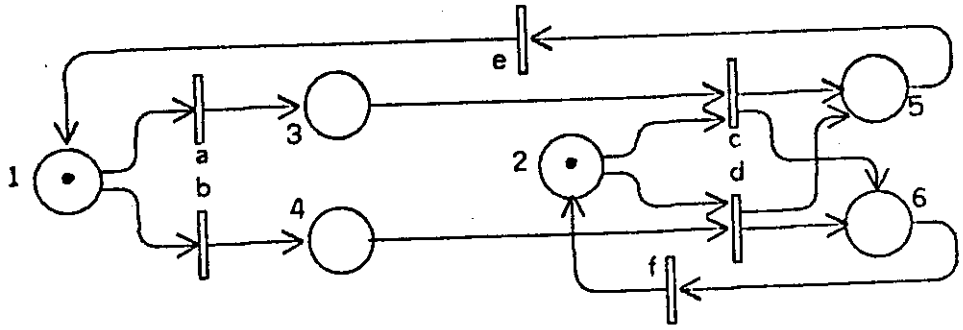

Figure 2: Initial Marking

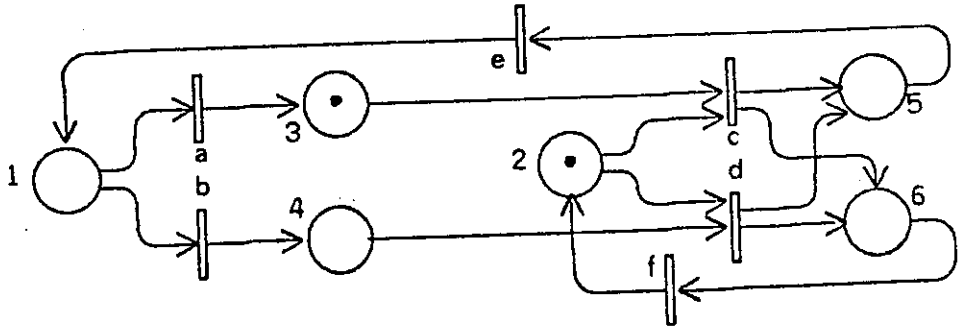

Figure 3: After firing transition "a"

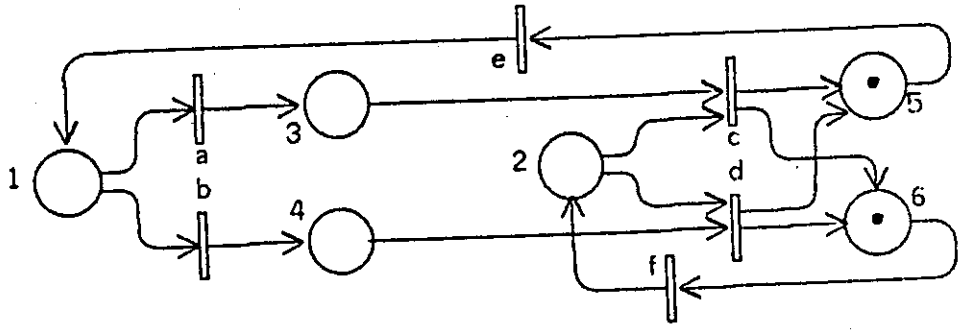

Figure 4: After firing transition "c"

1.2 Other varieties

Petri Nets have been defined with various restrictions or generalizations in attempts to provide more tractable mathematical properties or to represent systems more conveniently. These are a few of the most frequent forms of Petri Net varieties.

1) A place may not be both an input place and an output place of the same transition. This restriction established by the original definitions seems trivial but often corresponds to timing conditions in real systems.

2) A place may have only one token at a time. In some cases the firing rule is modified to prevent the firing of a transition if it would place two tokens on any place.

3) A transition has exactly one input place and output place. These concurrency-free graphs generally have only one token placed on them and are called state machines. More discussion of state machines is available in [Hol70] and [Hol68].

4) Each place has exactly one input transition and one output transition. These conflict-free graphs are called marked graphs and are discussed in [Com71].

5) Each arc from a place is either the only output from the place or the only input to some transition. These free choice nets include marked graphs and state machines and are discussed in [Hac72].

6) Every transition has at most one shared input place. These simple nets properly include free choice nets. This class is not fully understood.

7) An arc may specify that it removes or places more than one token. These generalized Petri Nets [Hac74] are abbreviations for standard Petri Nets. We will use these occasionally in this paper.

8) An arc may specify that it enables a transition only if the place at its origin contains no tokens. See section 5.1 for more on zero-testing nets.

9) Conditions can combine in or-conditions as well as the and-conditions of regular Petri Nets. This is an additional feature of the graph model of computation developed independently at UCLA and described in [Gos71] and [Cer72].

10) Coloring of tokens, assigning priority to transitions, forcing simultaneous firing of enabled transitions, associating time with transitions or places, and other bells and whistles have been explored for special application areas.

1.3 Representation of Processes

When used to represent the synchronization of concurrent computations, the features of a Petri Net usually correspond to specific aspects of the computation. Places describe states of processes, such as "Process A is in the critical region" or "a reader is reading". The behavior of other places closely resembles that of semaphores. Occasionally, places reflect general conditions of the system, such as "at least one process has entered the system".

Tokens often denote processes, so that the "flow" of a token through the net can reflect the "progress" of a particular process. (The Petri Net does not, however, actually distinguish one token from another, so the correspondence is entirely that of the user of the net). Other tokens represent counters or values in semaphores or messages.

Synchronization problems often specify the behavior of cyclic processes such as producers, readers, dining philosophers, etc. Since Petri Net places represent states of processes, the places in a net for such a problem are linked in cycles mirroring those of the processes.

1.4 Common Subgraphs

Many programming devices have direct representations in Petri Nets, as shown in Figures 5-11.

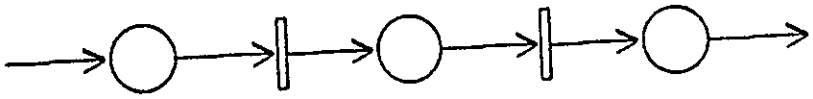

Figure 5: Sequential Flow

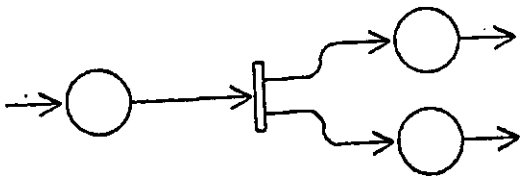

Figure 6: Process Fork

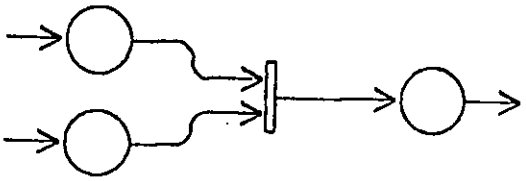

Figure 7: Process Join

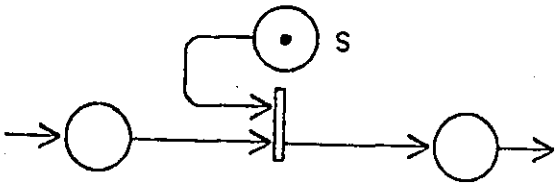

Figure 8: Dijkstra P-operation

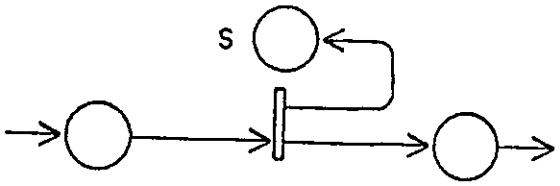

Figure 9: Dijkstra V-operation

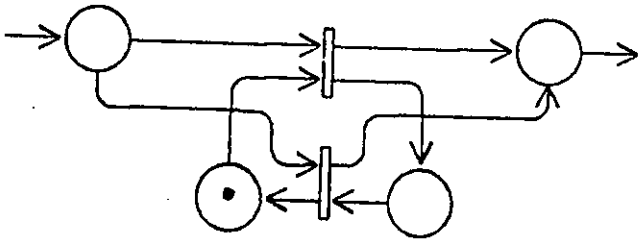

Figure 10: Toggle

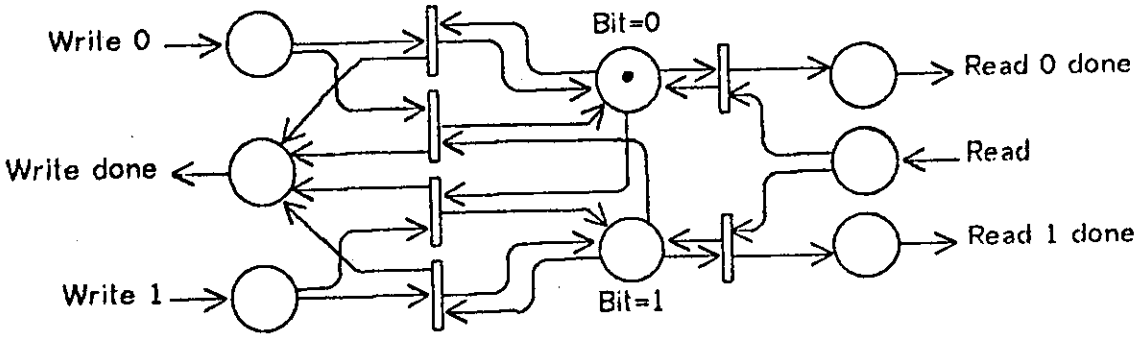

Figure 11: Bit

2. Proof of Properties of Petri Nets

2.1 Introduction

Proofs of synchronization problems generally consist of two parts. First, it must be shown that the restrictions on process concurrency are provided. For example, in the mutual exlusion problem, the solution must guarantee that only one process can be in the critical section (i.e. access the shared variables) at any one time. Second, the solution must be free of unpleasant properties such as the possibility of deadlock, indefinite delay, inordinate expense, or unnecessary constraints on concurrency.

2.2 Concurrency Restrictions

Many problem specifications can be reformulated in terms of invariants on the token loads of places in the Petri Net representation. In some cases, the invariant is directly derived by representing aspects of the problem statement. For example, the number of buffers in a buffer pool problem is generally a constant and is reflected in the Petri Net by an invariant number of tokens on a portion of the net (such that a token "represents" a buffer). In other cases, the invariant is dependent on the particular features of the net used to represent the synchronization. For example, the "semaphore" place in the mutual exclusion example in section 3.2 participates in the invariant but does not reflect any structure defined in the problem statement.

2.3 General Properties

2.3.1 Freedom from Deadlock

There are two tools for addressing deadlock questions in Petri Nets: proof of liveness and identification of subnets called siphons and traps.

A transition A is live if it is possible to enable it (eventually) by a sequence of firings from every reachable marking. This assures that the net can never reach a state such that transition A will never fire again. A Petri Net is live if every transition in it is live.

In order to determine if a transition is live, one must be able to characterize all reachable markings. In some cases, the markings are easily enumerated. Often, liveness can be deduced from the information available in the invariants on the token loads in the net.

If a Petri Net is live, every transition can be made to fire, that is, every event in the system can occur. Since events correspond to "progress" of "processes" in these nets, liveness implies that it is always possible for the system to reach a state which allows any process to continue. Hence, liveness does indeed correspond to freedom from deadlock.

For some classes of Petri Nets, there are necessary and sufficient conditions for liveness which rely only on the structure of the net (see section 4.1). Unfortunately, there are none known for the broader classes of Petri Nets.

2.4 Siphons and Traps

A siphon (or deadlock, in a special Petri Net sense) is a subset of places in a Petri Net such that every transition which places a token on a member of the subset also removes one from a member of the subset. This implies that if it should ever become blank, it will remain blank forever. In Figure 12, places A and B form a siphon.

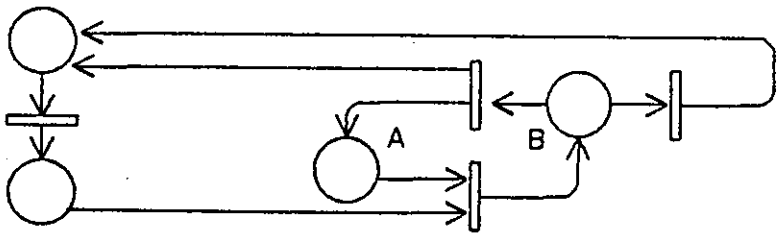

Figure 12: A Siphon

A trap is a subset of places such that every transition which takes a token from a member of the subset also places one on a member of the subset. Hence, once marked, a trap is always marked. In Figure 13, places A and B form a trap.

The definitions for siphons and traps overlap; for example, every strongly connected Petri Net is both a siphon and a trap. If a siphon contains a marked trap, it will never become blank.

If a Petri Net contains a siphon which can become blank, liveness cannot be obtained. Conversely, for some types of nets (see section 4.1), showing that

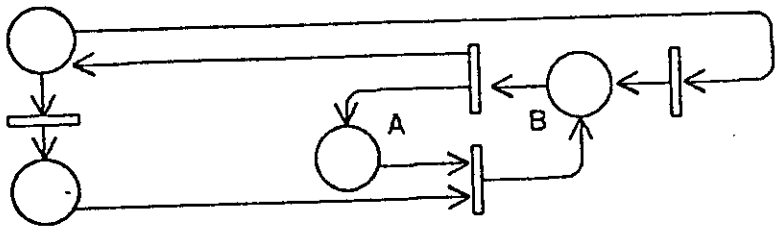

Figure 13: A Trap

each siphon contains a marked trap is sufficient for proof of liveness. An introduction to these structures is available in [Hac72] and a theoretical treatment is in [Sch75].

2.5 Busy Waiting

Busy waiting occurs when one process cannot proceed until notified by another process, but rather than blocking, it loops until notified. In some cases, the loop is a test on a memory cell which will be modified by the posting process.

Since blocking of processes is a feature of our interpretation of Petri Nets, and conditional branching is not, busy waiting does not often arise in the normal course of events. If one attempts to represent such a structure in a straightforward manner, the resulting net can proceed after the posting process places a token on the place in question, but the loop which was possible before is still possible and nothing guarantees that the process will ever leave it.

Busy waiting can, however, be represented using the "bit" net fragment of Figure 11, as shown in Figure 14.

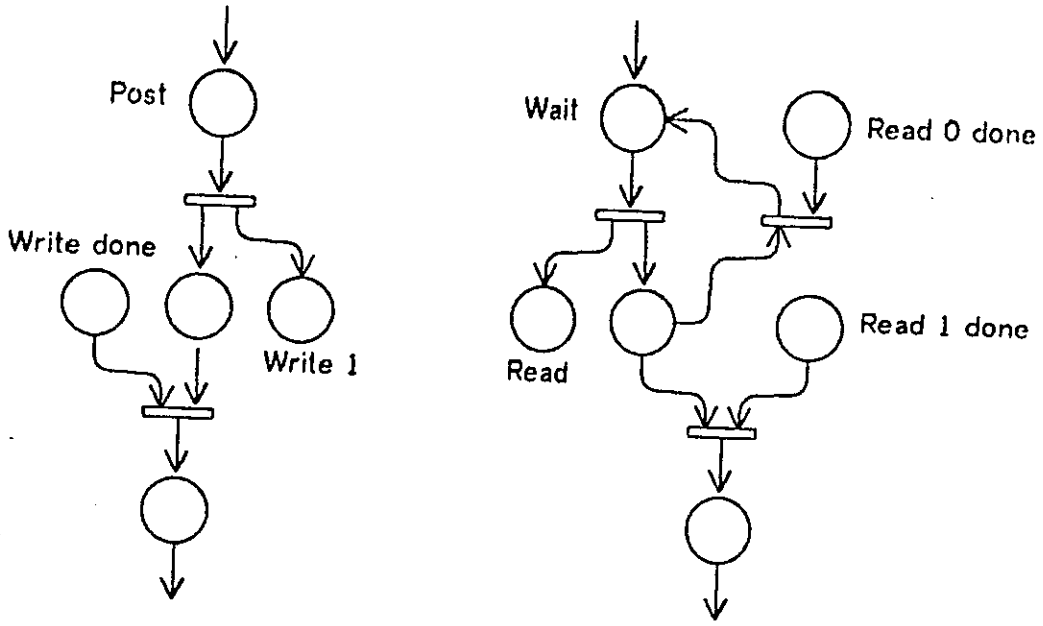

Figure 14: Busy waiting communication

2.6 Indefinite Delay

First, nothing guarantees that any transition in a net must fire. However, it is useful to assume that an enabled transition will eventually fire. This assumption translates into the requirement that every process which is not blocked will be scheduled at some time and will progress.

Second, there is no interpretation on the tokens of a Petri Net; the continuous "flow" of tokens across transitions is a figment of the user's imagination. If some of the tokens in that flow are identified with various processes in the system being represented, free token flow implies that no process is indefinitely delayed only if there is a fair scheduler in the underlying system.

2.7 Unnecessary Constraint

Argument for freedom from unnecessary constraint on processes use the same mechanisms as proofs of adequate constraint. Specifically, one shows that each transition is disabled only by conditions which reflect original problem constraints. One implicit problem constraint which must be made explicit in these proofs is that the sequence of events must follow the ordering established by the processes.

3. Examples of Petri Net Synchronizations

3.1 Introduction

The representation in Petri Nets of several standard synchronizations are presented in the next few pages. In some cases, only the portion of the system which provides the synchronization is presented. Processes in the system interact with the displayed portion by placing tokens on places designated by ingoing arrows. The displayed net indicates that the process in the system may resume by placing a token on place with an outgoing arrow, which is assumed to be a shared input place to appropriate transitions in the various processes.

For example, the net for mutual exclusion is displayed in Figure 15.

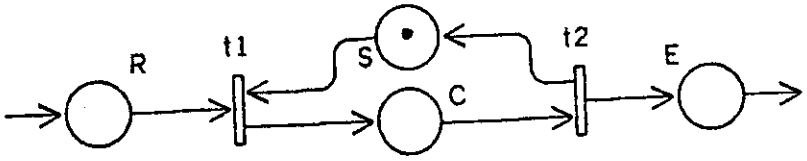

Figure 15: Mutual Exclusion

A process which is ready to execute its critical section places a token on R; When the critical section has been executed, this net places a token on E. (If desired, the critical sections could also be included in the net by the addition of several places and transitions. For our purposes, the critical section can be considered a parameter to the synchronization). A process which uses this net might be constructed as in Figure 16.

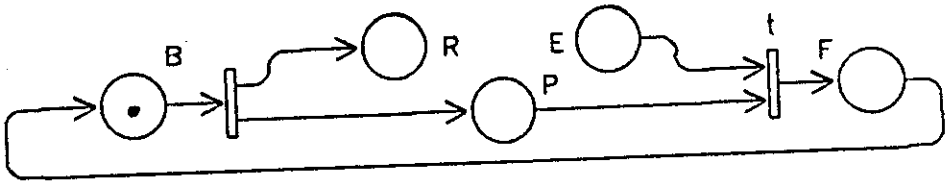

Figure 16: Use of Mutex Net

The place P preserves the identity of the process during use of the shared portion of the net by causing only transition t (of all such transitions sharing E) to be enabled when the mutual exclusion net places a token on E.

In the following examples, the net is presented with an interpretation of the places in the net with respect to the language of the problem. An alternative description of

the transitions is given by a set of formulas (after [Lie74]). In those formulas, a transition is defined by the effect it has on the token load of places. If transition t removes a token from place S and adds tokens to places Q and R, then t is described by

$$t \equiv -S + Q + R$$

Note that -R+R does not collapse to zero since the firing rule requires that tokens be present on the input places and then removed before tokens are placed on the output places. Otherwise, the implied arithmetic operations correspond to the changes in the token load on places due to the firing of a transition.

Also, each example contains a list of useful invariants of the net. The notation R' designates the token load of the place R. All values in invariants are non-negative integers.

Some of the nets use weights on arcs or places. On an arc, a weight n indicates that n tokens must be present to fire the transition or that n tokens are put on a place by the firing of the transition. On a place, a weight n indicates that n tokens are put on that place by the initial marking.

Finally, notes on each solution provide justification of the solution with proofs for some properties and discussions of others. The subtle behavior of the system is occasionally compared with that of other representations for the same synchronization.

3.2 Mutual Exclusion

3.2.1 Problem Statement

Only one process may execute a section of code called a "critical section" at a time. If no other process is in a critical section, any may execute it. (Refer back to Figure 15 for the Petri Net diagram).

3.2.2 Definitions

Token on	indicates
R	a process is ready to execute critical section
C	a process is executing a critical section
E	a process has exitted the critical section
S	semaphore ≡ no process is in a critical section

Transition descriptions

t1 ≡ -R - S + C
t2 ≡ -C + S + E

Transition invariants

$$C + S = 1$$

3.2.3 Notes

The exclusion is accomplished by providing an extra place with the inverse meaning of the place denoting a process in the critical section. This place corresponds to a semaphore [Dij68]. Note that the invariant insures that only one token exists on the pair (C,S), and therefore no two tokens can be on C at one time. Hence, no two processes execute their critical sections at one time.

If no token is on C, then the invariant insures that a token is on S, and therefore, if a token arrives on R, transition t1 is enabled and a process can enter the critical section.

3.3 Dining Philosophers

3.3.1 Problem Statement

A number of Philosophers (processes) are at dinner at a round table; each one alternately eats and meditates. A fork is placed between each adjacent pair of Philosophers (hence, there are n forks). Philosopher i needs both fork i and fork i+1 (where n+1=1) in order to eat. Provide a synchronization which allows each Philosopher equal access to food and does not deadlock.

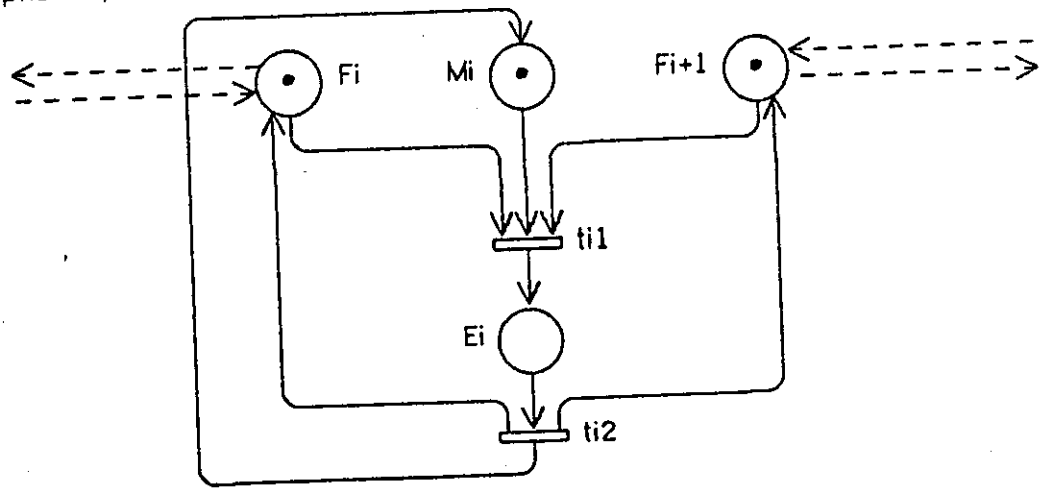

Figure 17: Dining Philosopher i

3.3.2 Definitions

Token on	indicates
Mi	philosopher i is meditating
Ei	philosopher i is eating
Fi	fork i is available

Transition descriptions

ti1 ≡ -Fi - F(i+1) - Mi + Ei
ti2 ≡ -Ei + Fi + Mi + F(i+1)

Transition invariants

∀i(Ei + Mi = 1)
∀i(Fi + E(i-1) = F(i+1) + E(i+1) = 1 - Ei

3.3.3 Notes

Equal opportunity is provided by symmetry; if all forks are available, any Philosopher may switch from meditating to eating, while if a fork is missing, its replacement immediately enables the waiting Philosopher.

The first invariant insures that there is always one token on Mi or Ei. If token is on Ei, then transition ti2 is enabled. If a token is on Mi then if tokens are present on Fi and F(i+1), transition ti1 is enabled. If a token is not present on Fi or F(i+1), invariant 2 guarantees that transitions t(i-1)2 or t(i+1)2 is enabled and by firing all such transitions tokens are returned to Fi and F(i+1) and ti1 is enabled. Since all transitions can be enabled in all reachable markings, the net is live and therefore deadlock is not possible.

It is possible that two Philosophers could collude to starve a third by a firing sequence such as ...t22,t21,t42,t41,t22,t21... In this case, Philosopher 3 is unable to get both forks simultaneously. The speeds of the processes is unknown, and no communication between processes is possible, so situations such as this must result from external conditions. Remedies for this feature are discussed with respect to other representations in [Cou74]. The net in Figure 18 implements one of these improvements.

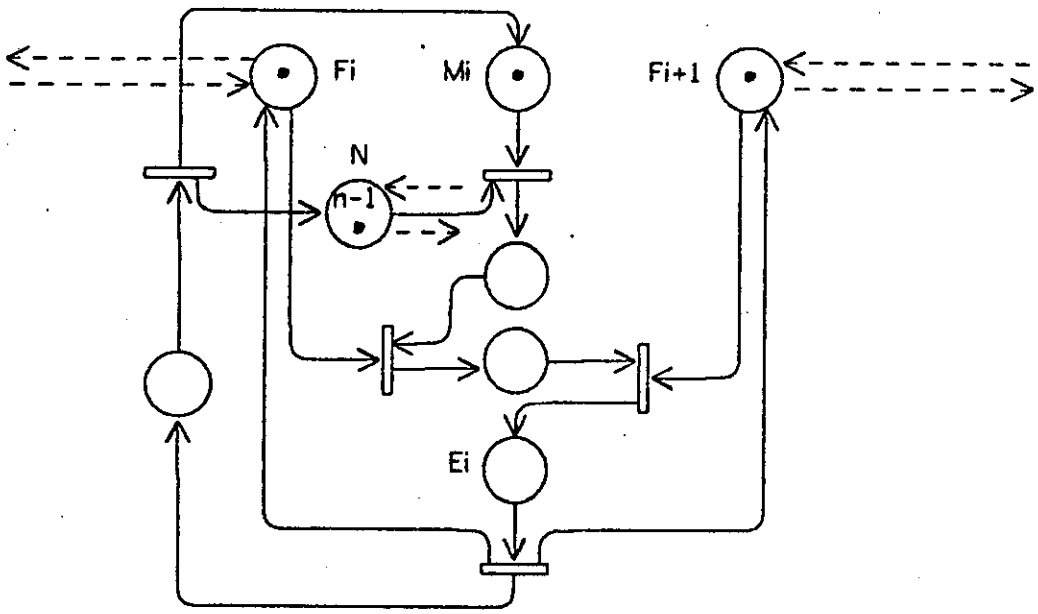

Figure 18: Dining Philosopher who cannot starve

3.4 Cigarette Addicts

3.4.1 Problem Statement[Pat71]

A system contains an agent and three cigarette addicts. The agent provides two of three types of resources on each cycle of the system. Each addict needs a different pair of the three resources to make and smoke a cigarette. The agent does not start the next cycle until the appropriate addict has completed the previous cycle. Provide a synchronization which maintains the identity of the processes and does not deadlock.

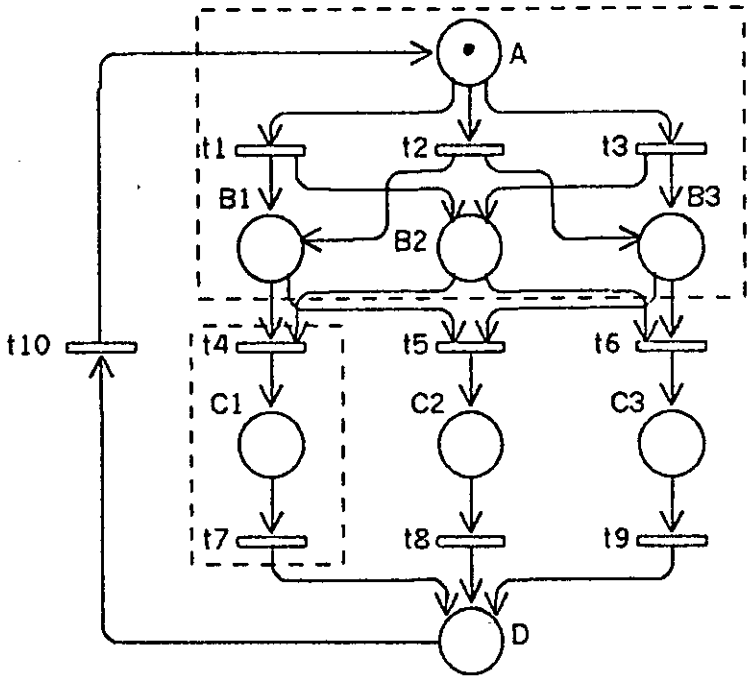

Figure 19: Three cigarette addicts

3.4.2 Definitions

Token on	indicates
A	agent ready to provide resources
Bi	resource i available
Ci	addict i processing resources
D	addict has completed processing

Transition descriptions

t1 ≡ -A + B1 + B2
t2 ≡ -A + B1 + B3
t3 ≡ -A + B2 + B3
t4 ≡ -B1 - B2 + C1
t5 ≡ -B1 - B3 + C2
t6 ≡ -B2 - B3 + C3
t(6+i) ≡ -Ci + D, i=1,2,3
t10 ≡ -D + A

Transition invariants

2(a + sum(Ci) + D) + sum(Bi) = 2

3.4.3 Notes

Places A and Bi along with transitions t1-t3 comprise the agent as required by the problem definition. The subnet consisting of Ci, t3+i and t6+i is one cigarette addict.

Note that the invariant assures that at least one transition is enabled at any time, and, in fact, that if there is no token on A, that exactly one transition is enabled. Since only one cycle exists in the net, the various cases are easily enumerated and it is evident that any transition can be enabled. Furthermore, at any time, the invariant assures that a token cannot be placed on A except by the termination of an addict. Therefore, the net is live and deadlock is not possible.

3.5 Reader/Writer 1

3.5.1 Problem Statement[Cou71]

In this bounded version of the problem, up to n reader processes may read a file simultaneously but writer processes must write the file excluding both readers and other writers. Provide a synchronization which excludes the appropriate processes but does not deadlock or otherwise cause processes to wait if they are not violating the above restriction.

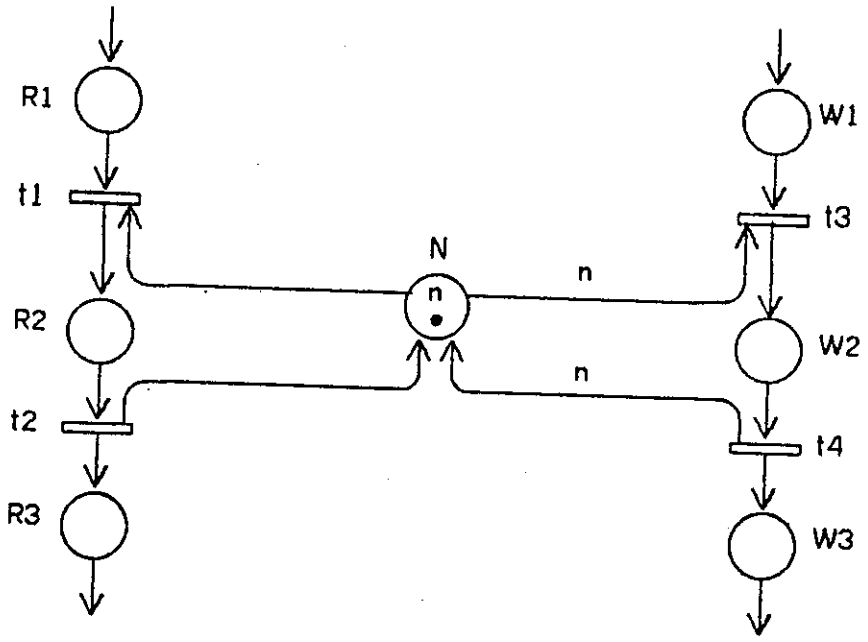

Figure 20: Bounded Reader/Writer 1

3.5.2 Definitions

Token on	indicates
R1	a reader is ready to read
R2	a reader is reading
R3	a reader has finished reading
W1	a writer is ready to write
W2	a writer has blocked readers
W3	a writer has finished writing
N	a reader may proceed

Transition descriptions

t1 ≡ -R1 - N - R2
t2 ≡ -R2 + N + R3
t3 ≡ -W1 - nN + W2
t4 ≡ -W2 + nN + W3

Transition invariants

n(1 - W2) = N + R2

3.5.3 Notes

Transition t1 can fire at most n times before t2 fires, since the invariant restricts the number of tokens on N and R2 to n. Hence, at most n readers will access the file simultaneously.

If there is a token on W2, then n tokens must have been present on N when t3 fired. Hence, the invariant insures that there are no tokens on R2, so a writer excludes readers. Furthermore, since there is a token on W2, there are no tokens on N (again from the invariant), so t3 cannot become enabled and it is impossible for a second token to be placed on W2 until the first on leaves by firing t4. Hence writers exclude one another.

When t4 fires, either t1 or t3 might become enabled and fire .

3.6 Reader/Writer 2

3.6.1 Problem Statement [Cou71]

Same as reader/writer 1 except that if writer is ready to write, no reader may start to read until all (up to m) waiting writers have finished writing.

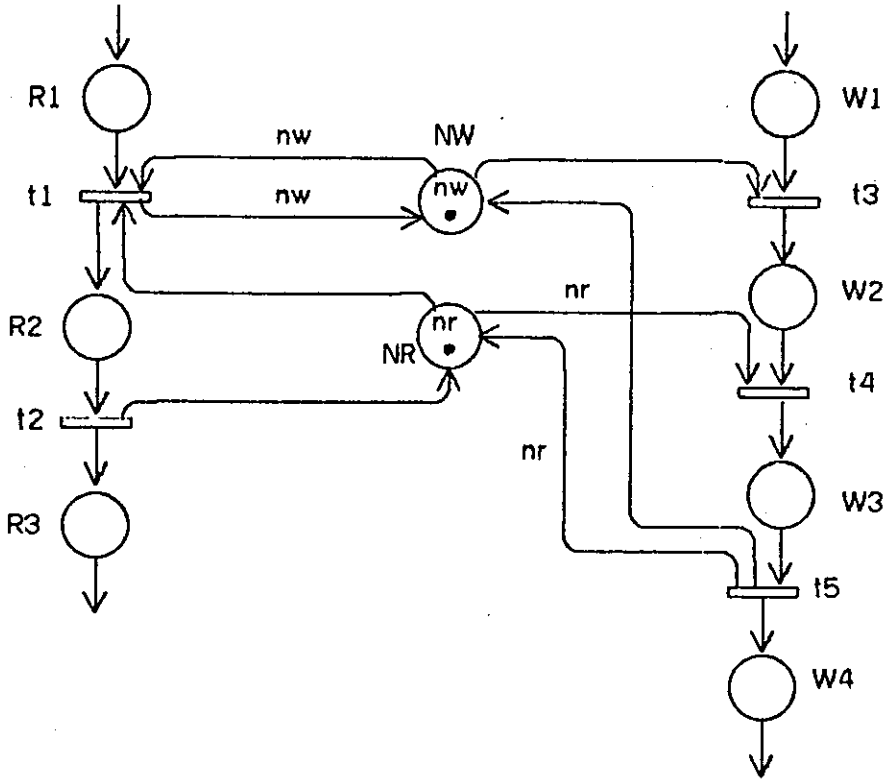

Figure 21: Bounded Reader/Writer II

3.6.2 Definitions

Token on	indicates
R1	a reader is ready to read
R2	a reader is reading
R3	a reader has finished reading
W1	a writer is ready to write
W2	a writer has blocked readers
W3	a writer is writing
W4	a writer has finished writing
NR	a reader may proceed
NW	a writer may proceed

Transition descriptions

t1 ≡ -R1 - NR - nwNW + nwNW + R2
t2 ≡ -R2 + NR + R3
t3 ≡ -W1 - NW + W2
t4 ≡ -W2 - nrNR + W3
t5 ≡ -W3 + nrNR + NW + W4

Transition invariants

W3 + W2 + NW = nw
R2 + NR = nr(1 - W3)

3.6.3 Notes

Transition t3 is always enabled when a token arrives at W1 (unless nw tokens are present on W2 and W3), since transition t1 is instantaneous. Hence, a writer can always place a token on W2. The first invariant assures that no new token can arrive on R2 since t1 is now disabled until all tokens from W2 and W3 return to NW. Therefore, writers prevent readers from starting.

A token on W3 guarantees that the number of tokens on R2 is zero via the second invariant, so writers exclude readers. Similarly, the number of tokens on NR is also zero, so t4 cannot be enabled, and therefore the number of tokens on W3 is restricted to one; i.e. writers exclude one another.

Transition t1 is enabled if a token is on R1 when no tokens are on W2 and W3 (and fewer than nr are present on R2). If tokens are present on W3, transition t5 is enabled; when it fires, either t4 or t1 is enabled. Successive firings of t5 and t4 will clear any tokens from W2 and W3 eventually enabling t1. When t1 fires, t2 is enabled. Transition t3 is disabled only if nw tokens are on W2 and W3, so clearing one by the sequence above enables t3. Again, t4 is disabled only when tokens are on R2 or W3. In that case, t2 and t5 are enabled, so firing them as needed will enable t4. T5 is always enabled if a token is present on W3. Since every transition is live, the entire net is live, assuming that tokens will be placed on R1 and W1 by other transitions in the net.

If this net were modified to accomodate the restriction that a place cannot be both the input place and output place of a transition, the arc from t1 to NW would be replaced by a place and another transition. In general, the behavior would be the same, except that the guarantee that t3 is always enabled when a token arives at W1 would not be strictly true. The result is similar to the behavior of the solution proposed by Brinch-Hansen [Bri72]: heavy activity by readers can prevent a writer from announcing its presence and therefore be delayed indefinitely [Cou72].

This solution can be extended to hierarchies of readers and writers as in [Cer72].

3.7 Producer/Consumer Bounded Buffer

3.7.1 Problem Statement [Dij68]

Producers send messages to consumers by allocating buffers from a common pool of n buffers and placing them in a common stream. Producers wait only when the pool is empty; consumers wait only when the stream is empty. Provide a synchronization which does not otherwise restrict processes and does not deadlock.

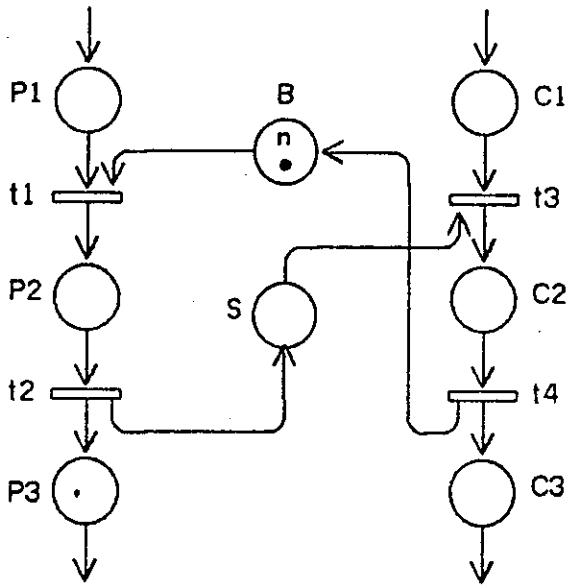

Figure 22: Producer/Consumer I

3.7.2 Definitions

Token on	indicates
P1	a producer is ready to send a message
P2	a producer has allocated a buffer
P3	a producer has placed the buffer into the stream
C1	a consumer is ready to process a message
C2	a consumer has removed a buffer from the stream
C3	a consumer has returned the buffer to the pool
B	a buffer is available from the pool
S	a buffer is in the stream

Transition descriptions

t1 = -P1 - B + P2
t2 = -P2 + S + P3
t3 = -C1 - S + C2
t4 = -C2 + B + C3

Transition invariants

B + S + P2 + C2 = n

3.7.3 Notes

If the buffer is empty, B is zero, and therefore t1 cannot become enabled. Likewise, if S is zero, t3 cannot become enabled. Hence processes wait appropriately. No other conditions affect the progress of processes.

The invariant insures that there are always n tokens distributed upon this subnet. If there are tokens on B, then P1 can be enabled. If there are tokens on C2, t4 is enabled and tokens will be placed on B. If there are tokens on S, t3 can become enabled and will place tokens on C2. Finally, if there are tokens on P2, t2 is enabled and tokens will appear on S. Hence all transitions are can eventually be enabled, the net is live and therefore deadlock-free.

3.8 Producer/Consumer Multiple Streams

3.8.1 Problem Statement[Dij72]

Same as Producer/Consumer Bounded Buffer except that several information streams share the common buffer pool. Each stream has a reservation for a number of buffers which it may use even if the pool is heavily used. Provide a synchronization which controls both the pool and the streams (i.e. consumers wait only if the corresponding stream is empty, and producers wait only if no buffers are available from the reservation or the shared group) and does not deadlock.

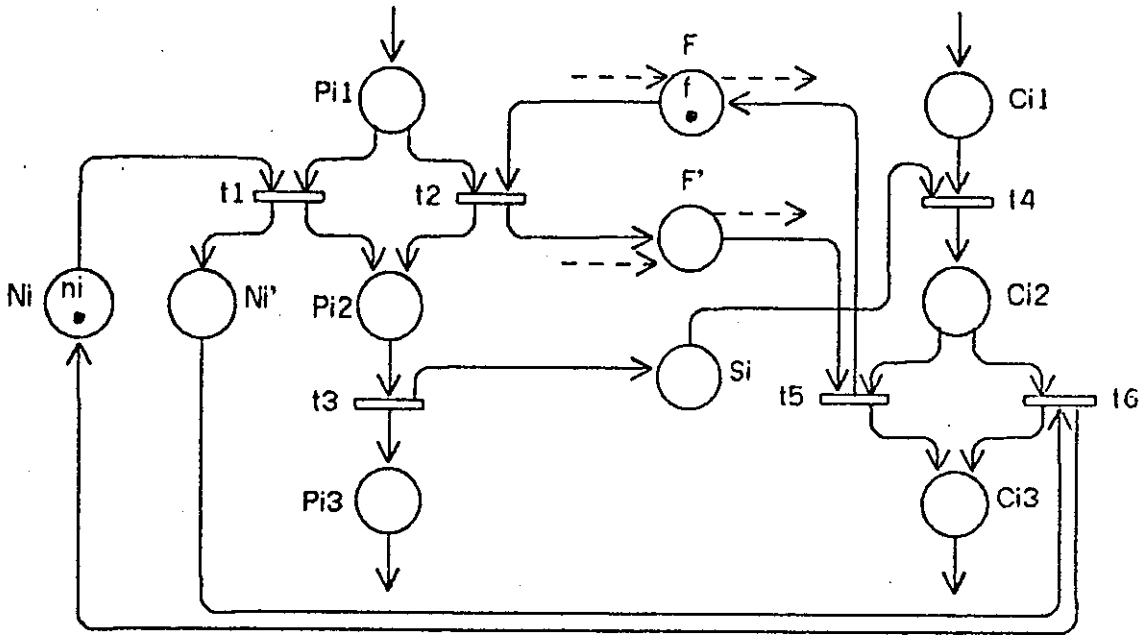

Figure 23: Producer/Consumer II -- stream i

3.8.2 Definitions

Token on	indicates
Pi1	producer i is ready to send a message
Pi2	producer i has allocated a buffer
Pi3	producer i placed a buffer into stream i
Ci1	consumer i is ready to process a message
Ci2	consumer i has removed a buffer from stream i
Ci3	consumer i has returned a buffer to the pool
F	a non-reserved buffer is available
F'	a non-reserved buffer has been allocated
Ni	a reserved buffer is available
Ni'	a reserved buffer has been allocated
Si	a buffer is in stream i

Transition descriptions

t1 ≡ -Pi1 + Pi2 + Ni' - Ni
t2 ≡ -Pi1 + Pi2 + F' - F
t3 ≡ -Pi2 + Si + Pi3
t4 ≡ -Ci1 - Si + Ci2
t5 ≡ -Ci2 - F' + F + Ci3
t6 ≡ -Ci2 - Ni' + Ni + Ci3

Transition invariants

$$F + F' = f$$
$$\forall i(N_i + N_i' = n_i)$$
$$\forall i(P_{i2} + C_{i2} + S_i \leq F' + N_i')$$
$$F + \text{sum}(N_i + S_i + P_{i2} + C_{i2}) = \text{sum}(n_i) + f$$

3.8.3 Notes

A producer i waits at Pi1 only if both t1 and t2 are disabled. This occurrs only when Ni and F both contain no tokens, that is, there are no buffers available. Consumer i waits at t4 only if Si contains no tokens, that is, the stream is empty.

The argument for freedom from deadlock is parallel to that for the simpler producer/consumer problem; the places F and F' do not enter into the proof, for the system will not deadlock even if no tokens are ever placed on F.

In other solutions to this problem, the decision to select buffers from the free or reserved sections of the pool is made in the solution [Dij72], [Coo74]. In this representation, it is not determined whether a producer will select a buffer from its reserved section before a shared buffer. The behavior of the system under load will vary due to different strategies of selection.

4. Petri Net Theorems

The following statements are a sample of the theorems known about Petri Nets which might be relevant to proofs of synchronization systems.

4.1 Liveness Conditions

1) If a state machine is strongly connected, any marking which places only one token on the net is a live, safe marking [Hol70]. No other marking is live and safe.

2) A free choice net is live if and only if every siphon contains a marked trap [Hac72].

3) A marked graph is live if and only if every elementary cycle contains a token [Com71].

4.2 Markings

1) In a live marked graph, there is a firing sequence from the initial marking which returns to the initial marking after firing each transition once [Com71].

2) In a live marked graph with marking M, M leads to marking M' which contains goal marking T if and only if M places as many tokens as T on each circuit [Hol70].

3) The maximum marking reachable from a marked graph with a given initial marking can be determined from the initial marking [Hol70].

4.3 Decidability

1) It is decidable whether a Petri Net can reach a marking which includes a given marking [Hac75a].

2) It is undecidable whether every marking reachable in one Petri Net is reachable in another Petri Net with the same number of places [Hac75a].

3) Reachability and liveness are recursively equivalent problems [Hac74]. It is not known whether they are decidable or not.

5. Discussion

5.1 Adequacy of Petri Nets for Synchronization

The Petri Net is capable of representing some behaviors of concurrent systems. It has been hypothesized that Petri Nets are adequate for representing all such systems [Pat70] and that they are not adequate [Bel73]. Subsequently, it has been shown [Age75] that Petri Nets cannot be used to represent the interaction of readers and writers [Cou71] if the number of processes is not bounded. This problem results from the inability to test for absence of tokens on a place.

This facility (which arises often in practical net construction) can be simulated in those cases in which a bound, (perhaps ridiculously high) can be established for the maximum number of tokens on each place. A new place is created which gains a token every time the place under consideration loses one, and vice versa. If the two places initially contain k tokens, testing for k tokens on the new place (with a k-weighted arc) is equivalent to testing for no tokens on the original place.

If the ability to condition a transition on the dearth of tokens on a place is added to

the Petri Net scheme, much additional power results. An arc written ---O--> indicates that the place at the origin of the arc must contain no tokens for the transition to fire.

In Figure 24, A and B represent parallel execution, as do C and D. However, C must start before D. This is insured because t1 is enabled only when a token is on S, and t2 is enabled only if that token has been removed.

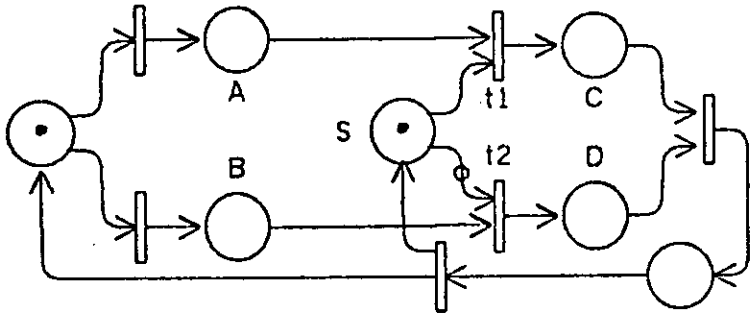

Figure 24: Zero testing net

Zero-testing can be accomplished also by establishing a partial ordering of transition priorities [Hac75b]. In such a scheme, if t1 and t2 are in conflict and priority(t1)>priority(t2), then t1 will fire. The two nets in Figure 25 are therefore equivalent.

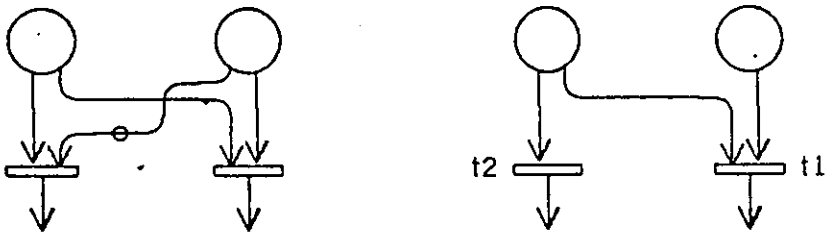

Figure 25: Equivalent nets

Petri Nets with zero-testing are adequate for representing all synchronizations of systems which can be considered to have a central clocked controller. Let S be a system of conditions and events. Between clock pulses, the conditions effectively determine which events can occur. The broadest class of such systems requires the full power of a Turing machine to determine which events are possible given the conditions obtaining. One such event is now allowed to occur and the effective procedure is invoked to determine the current class of possible events. Agerwala has shown [Age75] that there is a Petri Net (with zero-testing) for any such system.

Petri Nets can potentially represent systems in which no such central controller is possible and are therefore not limited to such clocked systems. No definition of completeness or adequacy for distributed control, asynchronous systems has yet been established.

5.2 Suitability

Petri Nets are convenient for representation of interactions is system structures with fixed numbers of processes and fixed relationships between processes. The cigarette addicts problem is prototypical of such rigid relationships. A variety of difficulties arise when applying Petri Nets to more dynamic systems.

1) When adding a process to a system (for example, reader/writer or producer/consumer), the Petri Net must be modified to reflect the new process. The addition of the subnet must be done so as to not violate the invariants of the net. No study has been done on modification of Petri Nets, although Gostelow has started work on how to represent modifiable Petri Nets, using a Petri Net Interpreting Petri Net [Gos75].

2) There is no structure in a Petri Net which corresponds to local environment. Hence, when tokens from two different processes arrive at the same place, it is not possible to determine which leaves first. This results in a proliferation of places for keeping track of process identifications.

3) The number of tokens which is placed on a place cannot be parameterized by the process. Hence, an event which is conditioned by a variable number of occurences of another event cannot be represented. For example, a process wishes to resume after k ticks of an external clock [Hoa74]. The event "tick" can be associated with the process computation of k only if an arc can be given a parameter which denotes the number of tokens to be placed on a place.

4) Many synchronizations are easily constructed in terms of conditional expressions on variables evaluated when events occur. Some of these are easily converted to conditions in Petri Nets. In others, however, where the expression is a complex combination of variables, the complexity of the corresponding Petri Net far exceeds the complexity of the problem. Some improvement can be realized by providing subnets for computational structures (such as that in Figure 11), but such subnets are not generally available.

5.3 Other Approaches

Three other approaches to the use of Petri Nets in the representation of synchronization should be mentioned. Petri Net theory, in these cases, applies to classes of synchronization systems rather than specific coordination problems.

Lauer and Campbell [Lau74] show that some path expressions [Hab75] can be simulated by Petri Nets and that there exist algorithms for constructing the simulating Petri Nets. By proving properties of such nets, they are able to derive properties of the path expressions which generated the nets.

Hack [Hac75b] and others have characterized the classes of languages which can be generated by various interpretations of Petri Nets. (One such interpretation parallels the language interpretation of a finite state machine). The combination of this theory with that of language generating expressions such as path expressions should contribute to understanding the relationship between these methods.

Karp and Miller [Kar67] and Keller[Kel72] have developed vector systems for modeling parallel asynchronous processes. These formal systems have been shown to be equivalent to Petri Nets and some theoretical exchange has resulted [Hac74].

References

Age75 T. Agerwala, "A Complete Model for Representing the Coordination of Asynchronous Processes". Proceedings of the Project Mac Conference on Petri Nets and Related Methods (1975).

Bel73 G. Belpaire and J.P. Wilmotte, "A Semantic Approach to the Theory of Parallel Processes". Proceedings of the International Computing Symposium 73 (1973).

Bri72 Per Brinch-Hansen, "Structured Multiprogramming". Communications of the ACM 15,7 (September 1972).

Cer72 V.G. Cerf, "Multiprocessors, Semaphores, and a Graph Model of Computation". UCLA-ENG-7223 (1972).

Com71 F. Commoner, A.W. Holt, S. Even and A. Pnueli, "Marked Directed Graphs". Journal of Computer and System Science 5 (October 1971).

Coo74 Lee W. Cooprider, P.J. Courtois, F. Heymans and David L. Parnas, "Information Streams Sharing a Finite Buffer: Other Solutions". Information Processing Letters 3 (1974).

Cou71 P.J. Courtois, F. Heymans and David L. Parnas, "Concurrent Control with Readers and Writers". Communications of the ACM 14,10 (October 1971).

Cou74 M. Braun, P.J. Courtois, and J. Georges, "On Starvation". Report R249, MBLE Brussels (1974).

Cou72 P.J. Courtois, F. Heymans and David L. Parnas, "Comments on "A Comparison of Two Synchronizing Concepts by P.B. Hansen"". Acta Informatica 1 (1972).

Dij72 Edsger W. Dijkstra, "Information Streams Sharing a Finite Buffer". Information Processing Letters 1 (1972).

Dij68 Edsger W. Dijkstra, "The Structure of the "THE"-Multiprogramming System". Communications of the ACM 11,5 (May 1968), 341-346.

Gos71 K.P. Gostelow, Flow of Control, Resource Allocation, and the Proper Termination of Programs, UCLA-ENG-7179, 1971.

Gos75 K.P. Gostelow, "A Model of Processes Based on Petri Nets". Proceedings of the Project Mac Conference on Petri Nets and Related Methods (1975).

Hab75 A. Nico Habermann, Path Expressions, Computer Science Department, Carnegie-Mellon University, 1975.

Hac72 Michael H. Hack, Analysis of Production Schemata by Petri Nets, TR-94, Project MAC, Massachusetts Institute of Technology, 1972.

Hac74 Michael H. Hack, "The Recursive Equivalence of the Reachability Problem and the Liveness Problem for Petri Nets and Vector Addition Systems". Proceedings of the 15th Annual IEEE Symposium on Switching and Automata Theory (October 1974).

Hac75a Michael H. Hack, Decision Problems for Petri Nets and Vector Addition Systems, TM-59, Project MAC, Massachusetts Institute of Technology, 1975.

Hac75b Michael H. Hack, Petri Net Languages, CSGM-124, Massachusetts Institute of Technology, 1975.

Hoa74 C.A.R. Hoare, "Monitors: An Operating System Structuring Concept". Communications of the ACM 17,10 (October 1974), 549-557.

Hol70 A.W. Holt and F. Commoner, "Events and Conditions". (in three parts), Applied Data Research, New York. [Chapters I, II, IV and VI appear in Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York] (1970).

Hol68 A.W. Holt, "Final Report of the Information Systems Theory Project". Technical Report RADC-TR-68-305, Rome Air Development Center, Griffiss Air Force Base, New York (1968).

Kar67 R.M. Karp and R.E. Miller, "Parallel Program Schemata: A Mathematical Model for Parallel Computation". Proceedings of the 8th Annual IEEE Symposium on Switching and Automata Theory (October 1967).

Kel72 R.M. Keller, Vector Replacement Systems: A Formalism for Modelling Asynchronous Systems, TR 117, Computer Science Laboratory, Princeton University, 1972.

Lau74 Peter E. Lauer and Roy H. Campbell, A Description of Path Expressions by Petri Nets, University of Newcastle upon Tyne, 1974.

Lie74 Y.E. Lien, Termination and Finiteness Properties of Transition Systems, Report TR-74-4, Dept. of Computer Science, University of Kansas , 1974.

Pat71 S.S. Patil, Limitations and Capabilities of Dijkstras Semaphore Primitives for Coordination among Processes, CSGM 57, Project MAC, Massachusetts Institute of Technology, 1971.

Pat70 S.S. Patil, Coordination of Asynchronous Events, TR-72, Project MAC, Massachusetts Institute of Technology, 1970.

Pet62 Carl Adam Petri, "Communication with Automata". Supplement to Technical Report RADC-TR-65-377, Vol. 1, Rome Air Development Center, Griffiss Air Force Base, New York 1966. [Originally published in German: Kommunikation mit Automaten, University of Bonn] (1962).

Sch75 Hans Schmid, "Towards a Constructive Liveness Proof". Proceedings of the Project Mac Conference on Petri Nets and Related Methods (1975).