24

Petri Nets

Author: Robert A. McGuigan, Department of Mathematics, Westfield State College.

Prerequisites: The prerequisites for this chapter are graphs and digraphs. See Sections 9.1, 9.2, and 10.1 of *Discrete Mathematics and Its Applications*.

Introduction

Petri nets are mathematical structures which are useful in studying systems such as computer hardware and software systems. A system is modeled by a Petri net, the analysis of which may then reveal information about the structure and behavior of the system as it changes over time. Petri nets have been used to model computer hardware, software, operating systems, communications protocols, networks, concurrency, and parallelism, for example. Petri net models have been constructed for the CDC6400 computer and the SCOPE 3.2 operating system for the purpose of evaluating system performance (see references in [2]).

More generally, industrial production systems and even general social, ecological, or environmental systems can be modeled by Petri nets. In this chapter we will introduce the basic definitions of Petri net theory, study several examples in detail, and investigate some of the deeper concepts of the theory.

Petri Nets

A **Petri net** has four components: a set P of *places*, a set T of *transitions*, an *input function* I, and an *output function* O. The input function assigns to each transition a set of places known as the *input places* of the transition. The output function assigns to each transition a set of places known as the *output places* of the transition. Conceptually, places are passive components of the system. They may store things, represent states of the system, or make things observable, for example. Transitions are the active components of the system. They may produce things, transport things, or represent changes of state of the system. The following is the formal definition.

Definition 1 A *Petri net structure* is a four-tuple (P, T, I, O) such that P is a finite set (of *places*); T is a finite set (of *transitions*), with $P \cap T = \emptyset$; I (the input function) is a function from T to the set of all finite subsets of P; O (the output function) is a function from T to the set of all finite subsets of P.

A place p is an *input place* for a transition t if and only if p belongs to $I(t)$ and p is an *output place* for t if and only if p belongs to $O(t)$. □

Example 1 Consider a simple batch processing computer system. We suppose it has four states, or conditions:

1) a job is waiting for processing,
2) the processor is idle,
3) a job is being processed,
4) a job is waiting to be output.

There are four actions of the system:

1) a job is entered in the input queue,
2) a job is started on the processor,
3) a job is completed,
4) a job is output.

If a job is entered in the input queue, then a job is waiting; if a job is waiting and the processor is idle, then a job can be started. After a job is processed it can be completed and a job will be waiting for output and the processor will be idle. If a job is waiting for output then a job can be output. Using the rule of thumb about active and passive components of a system we should have places corresponding to the four states of the system and transitions corresponding to the four actions.

This simple system can be modeled by a Petri net having four places, p_1, p_2, p_3, p_4, corresponding to the four states, and four transitions, t_1, t_2, t_3, t_4, corresponding to the four actions.

The input and output functions I and O are defined by listing their values explicitly:

$$I(t_1) = \emptyset, \quad I(t_2) = \{p_1, p_2\}, \quad I(t_3) = \{p_3\}, \quad I(t_4) = \{p_4\}$$

$$O(t_1) = \{p_1\}, \quad O(t_2) = \{p_3\}, \quad O(t_3) = \{p_2, p_4\}, \quad O(t_4) = \emptyset.$$

Note that I assigns t_1 the subset $\emptyset$ of P and O assigns $\emptyset$ to t_4. This is the formal way of saying that a transition has no input or output places. □

There is a representation of Petri net structures using directed graphs, which makes it easier to see the structure. In drawing the directed graph representation of a Petri net, we make the following convention: *places will be represented by circular vertices and transitions by rectangular vertices.* If a place p is an input place for the transition t, then there is a directed edge from the vertex for p to that for t. If p is an output place for t, then a directed edge goes from t to p. It is easy to go back and forth between the two specifications.

Example 2 Construct a directed graph representation of the Petri net structure of Example 1.

Solution: Figure 1 shows such a directed graph and is labeled to show it as a model for the simple batch processing computer system. □

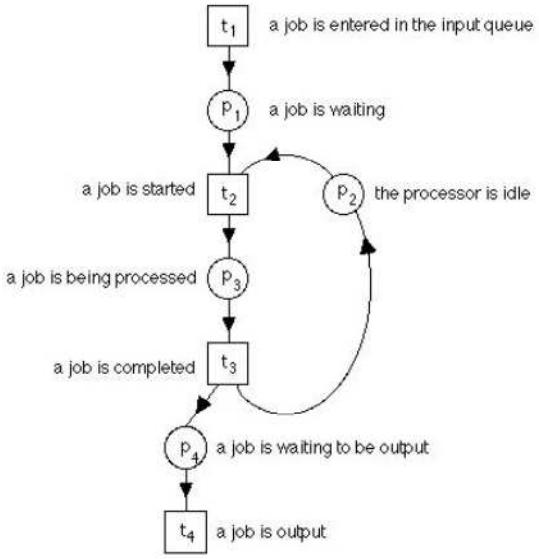

Figure 1. A Petri net graph for a computer system.

Example 3 As another example (adapted from [2]), consider a library system. Users have access to three desks: the request desk, the collection desk, and the return desk. All books are kept in the stacks and every book has an index card. A user goes to the request desk to ask for a book. If the book is in the stacks it is removed and the borrowed book index is updated. The user gets the book at the collection desk. Books are returned to the return desk. Books returned are replaced in the stacks and the index is updated. A Petri net model of this system is given in graphical form in Figure 2. □

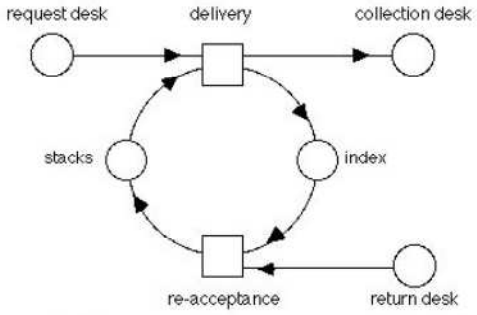

Figure 2. A library system.

For the purpose of modeling more complex systems it is necessary to use a more complicated graph structure in which more than one directed edge may go from one vertex to another.

Example 4 Figure 3 shows an example of a graphical Petri net with multiple edges between some vertices.

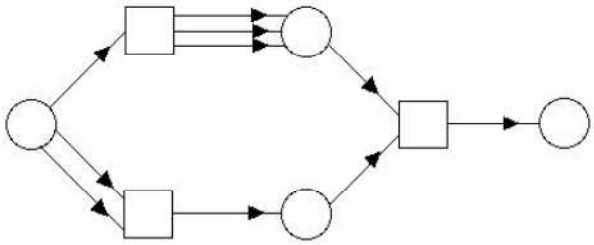

Figure 3. A graphical Petri net.

This sort of structure requires a modification of the definition of Petri net structure given previously since a place cannot appear more than once as a member of a *set* of input or output places. To overcome this difficulty we need only change the ranges of the input function I and the output function O to be the set of all finite subsets of the Cartesian product of P and N, the set of

positive integers. Thus an input or output place would be represented as an ordered pair (p, n) where the place is p and n is the number of edges between p and the transition in question. In what follows we will generally be using only the graphical representation of the net.

Modeling System Dynamics

In addition to having a static structure which is modeled by Petri net structures as discussed above, many systems change over time and it is of great interest to study this dynamic behavior. In Petri net theory this is accomplished through the use of *markings*.

Definition 2 A *marking* of a Petri net (P, T, I, O) is a function m from P to the nonnegative integers. □

We think of the marking as assigning *tokens* to the places in the net. The number assigned to a place by the marking is the number of tokens at that place. In models of systems the tokens could have many interpretations. For example in modeling computer operating systems the tokens might represent the processes which compete for resources and which must be controlled by the operating system.

As we shall see, the tokens can move from place to place in the net simulating the dynamic behavior of the system. In the graphical representation of Petri nets the tokens are represented by solid dots in the circular vertices for the places. This is practical when the number of tokens at any place is small. If there are too many tokens then we write the number of tokens assigned to a place inside the circle.

Example 5 Figure 4 shows some examples of marked Petri nets. □

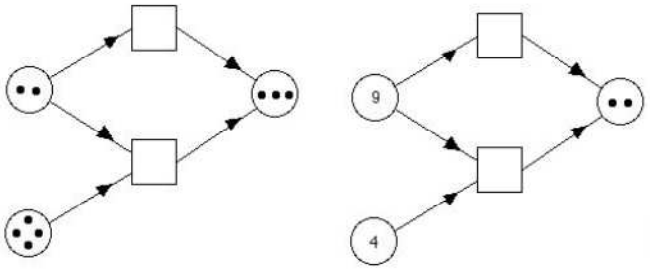

Figure 4. Marked Petri nets.

The dynamic change of the Petri net model is controlled by the marking, i.e., the number of tokens assigned to each place, and by the *firing rules* for transitions.

Definition 3 A transition is *enabled for firing* if each of its input places has at least as many tokens in it as edges from the input place to the transition. A transition may fire only if it is enabled. □

Example 6 Which of the transitions in Figure 5 are enabled for firing?

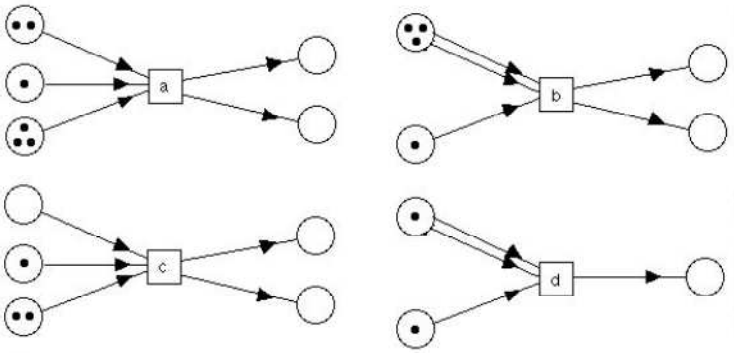

Figure 5. Transitions.

Solution: Transitions a and b are enabled while c and d are not. □

A transition is *fired* by removing one token from each input place for every edge from the input place to the transition and adding one token to each output place for every edge from the transition to that place.

Example 7 Show the result of firing each of the transitions in Figure 6.

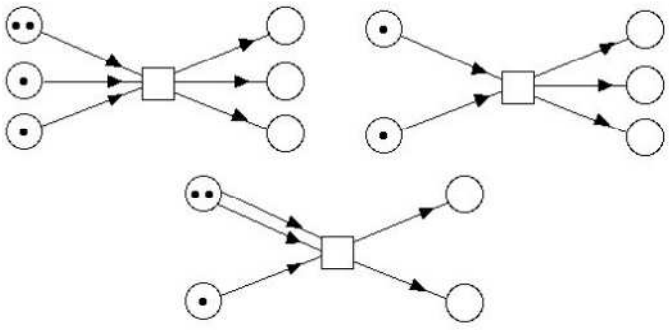

Figure 6. Transitions before firing.

Solution: Figure 7 shows these transitions after firing. □

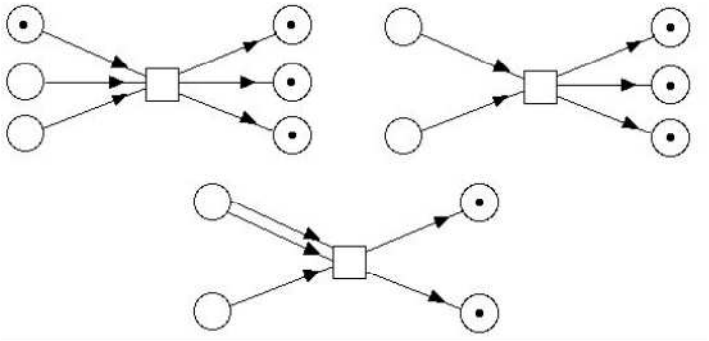

Figure 7. Transitions after firing.

Firing a transition generally changes the marking of the Petri net to a different one. Thus the dynamics of the system are acted out on the markings. Note that since only enabled transitions may be fired, the number of tokens at each place is always non-negative. Note also that firing transitions does not necessarily conserve the total number of tokens in the net.

We may continue firing transitions until there are no more enabled for firing. Since many transitions might at any time be enabled for firing, the sequence in which the transitions are fired might not be unique. The only restriction is that transitions do not fire simultaneously. Consequently, the marking which results after firing a certain number of transitions could vary depending on the particular transitions fired and their order.

Thus, there is possibly some nondeterminism in the execution of Petri net firing. Figure 8 shows an example of two transitions, each of which is enabled for firing, but once one of them is fired, the other is no longer enabled.

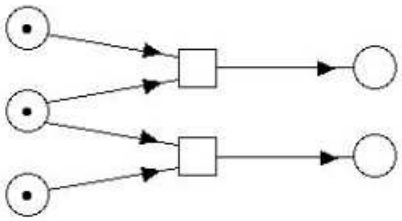

Figure 8. Transitions enabled for firing.

The goal is to have the Petri net model the system, so each possible firing sequence would correspond to a possible behavior of the system, even though they would not all necessarily result in the same final marking.

Applications

We now look at some examples of Petri net models. The models we consider in detail all concern problems related to computer hardware and operating systems.

Example 8 Construct a Petri net model for calculating the logical conjunction of two variables x and y, each of which takes the values "true" or "false". Each is assigned a value independently of the other.

Solution: Figure 9 shows the required net. To see how this net works, we walk through the firing sequence corresponding to $x =$ "true" and $y =$ "false". We start with one token in the places corresponding to $x =$"true" and $y =$ "false", the leftmost and rightmost places on the top line of Figure 9. Now only one transition is enabled for firing, the second one from the left. Firing that transition puts one token into the place labeled $x \wedge y =$ "false". No further transition firings are possible and the desired result is obtained. The reader may check that other choices for values of x and y also yield the correct results. □

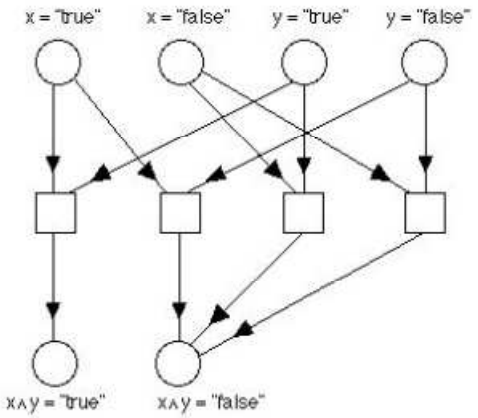

Figure 9. Petri net model for $x \wedge y$.

Example 9 Construct a Petri net that computes the negation of x, which again takes only the values "true" and "false".

Solution: Figure 10 shows a net that computes the negation of the variable x. This net works similarly to that in Figure 9. If we want to know the value of $\neg x$ when x has a certain value, we put one token in the place for that value, carry out the firing, and observe which place has a token in it. □

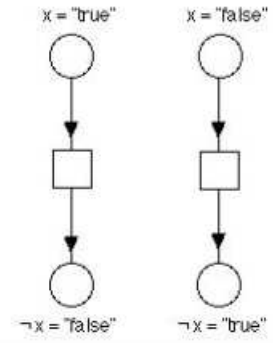

Figure 10. Petri net model for $\neg x$.

Petri nets can be combined to form new nets. One way is simply to take the disjoint union of the two nets. This is accomplished for the graphical representation by simply drawing the graphs next to each other. Another method combines the nets in a way similar to the composition of functions. In many nets certain of the places could naturally be considered as input places, for example those labeled x = "true" and x = "false" in Figure 10. Others could be considered as output places, for example those labeled $\neg x$ = "false" and $\neg x$ = "true" in Figure 10. To compose two nets N_1 and N_2, the number of output places of N_1 must equal number of input places of N_2. The composition is obtained by identifying the output places of N_1 with the input places of N_2 in an appropriate manner.

Example 10 Construct a Petri net for computing $\neg(x \wedge y)$ by composing the nets in Figures 9 and 10.

Solution: Figure 11 shows the composition net. The input places for Figure 9 are the places labeled x = "true", x = "false", y = "true", and y = "false". The output places in Figure 9 are the places labeled $x \wedge y$ = "true", $x \wedge y$ = "false". The input places for Figure 10 are those labeled x = "true" and x = "false". To combine two nets we match up the input and output places as shown. It is also necessary to relabel some of the places to indicate their meaning in the composed net. □

Now that we have seen how to use Petri nets to model conjunction and negation of propositional formulas we should note that we can represent all propositional functions using Petri nets since conjunction and negation are functionally complete, i.e. every possible truth table can be obtained as that of a propositional function using only conjunction and negation. (See Section 1.2 of

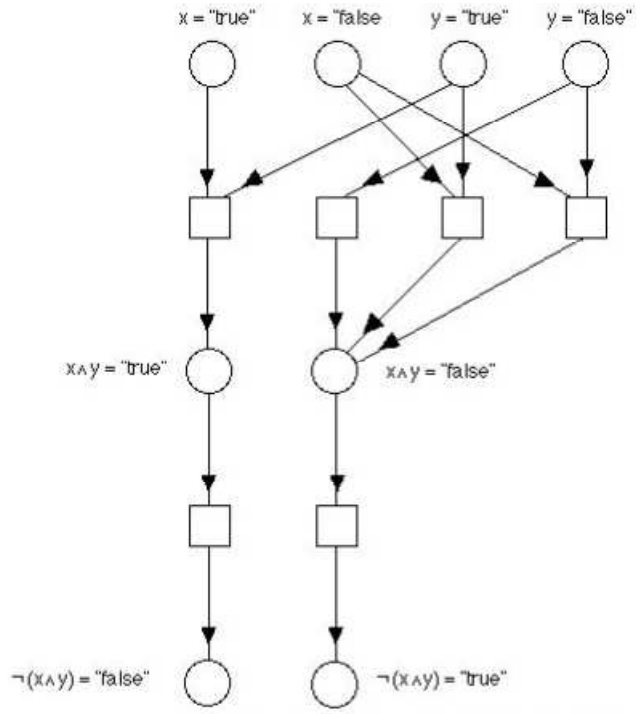

Figure 11. Composition net for nets in Figures 9 and 10.

Discrete Mathematics and Its Applications.)

Using the correspondence between propositional functions and switching circuits we can observe that computer hardware can also be modeled using Petri nets. Figure 11 can thus be seen as modeling a NAND gate.

Our next examples of Petri nets show how they can be useful in studying parallelism in computer systems or problems of control in operating systems.

First we will examine the *readers/writers problem.* There are two types of processes: reader processes and writer processes. All processes share a common file or data object. Reader processes never modify the object but writer processes do modify it. Therefore, writer processes must exclude all other processes, but multiple reader processes can access the shared object simultaneously. This sort of thing happens frequently in database applications, for example, in airline reservation systems. Depending on the actual system, there may be limits on how many processes may access the memory area simultaneously. How can this situation be controlled?

Example 11 Construct a Petri net model for the readers/writers problem

when there are six processes, two of which have write access and four have read access. Furthermore, suppose at most three reader processes can overlap in access to memory.

Solution: Figure 12 shows a Petri net model for this control problem. This example was adapted from one in reference [2]. In this example the processes are represented by the tokens, the places in the net represent the different things the processes might be doing, or states they might be in.

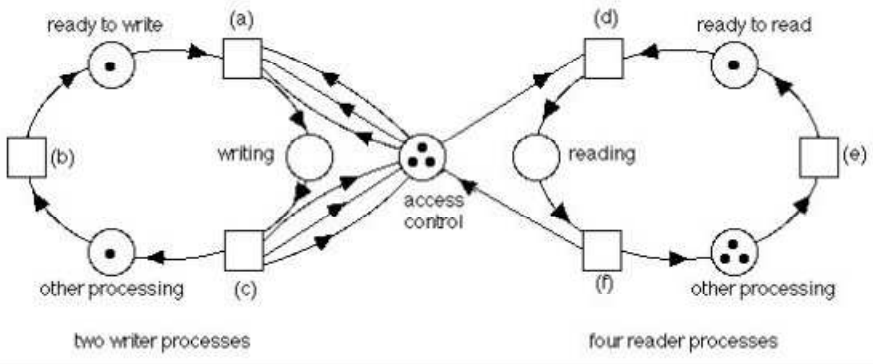

Figure 12. Petri net model for readers/writers problem.

The assignment of tokens to places in Figure 12 is just one possibility, for illustrative purposes. Let us follow this system through some transition firings to see how it works. As the system is in Figure 12, no processes are reading or writing. Transition (a) is enabled for firing, as are (d), (e), and (b). If we fire (a) then one token is put into the writing place, the place labeled "ready to write" has one token removed from it, and the "access control" place has three tokens removed from it. After this firing, transition (d) is no longer enabled, so no process can enter the "reading" place. Even if there were another token in the "ready to write" place, we could not fire (a) again because there are no tokens in the access control place so (a) is not enabled. In fact, a little thought shows that if (a) has been fired to put a process in the write place then (a) and (d) become unable to fire, thus preventing any other process from either reading or writing. Now transition (c) is enabled. If it is fired we will have two tokens in the "other processing" place and three tokens will be put back in the access control place.

We now see how the net controls read access. Transition (e) is enabled for firing, and in fact we can fire it three times in succession so that the reader "other processing" place becomes empty and the "ready to read" place has four tokens in it. Transition (d) is now enabled for firing. Each time we fire transition (d), one token is removed from the "ready to read" place, one token is put in the reading place, and one token is taken out of the access control place. Once three firings of (d) have taken place, then the access control place

is empty, transition (d) is no longer enabled, and no additional tokens can be put in the reading place.

What happens when processes stop reading? Every time transition (f) is fired, one token is put in the access control place and one token is put in the "other processing" place. Three firings of (f) will put three tokens into access control and if there are any tokens in the "ready to write" place, transition (a) will be enabled. We can see how the three tokens in the access control place implement the requirement that no more than three reader processes may have access simultaneously, and the three edges between transitions (a) and (e) and the access control place prevent access when a process is writing. □

Next we examine the *producer/consumer problem*. Again we have a shared data object, but this time it is specified to be a buffer. A producer process creates objects which are put in the buffer. The consumer waits until there is an object in the buffer, removes it, and consumes it.

Example 12 Construct a Petri net model for the Producer/Consumer problem.

Solution: The net in Figure 13 is the required model.

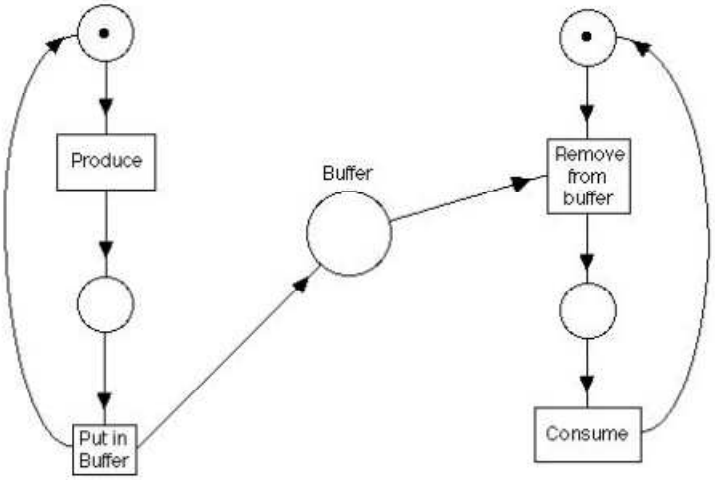

Figure 13. Petri net for producer/consumer problem.

In this model each token in the buffer represents an object which has been produced but not yet consumed. We can modify this example to work for the multiple producer/multiple consumer problem. In this case multiple producers

produce objects which are stored in a common buffer for the use of multiple consumers. If there are n producers and k consumers, we need only start the system with n tokens in the initial place of the producer process and k tokens in the initial place of the consumer process, instead of one in each as shown in Figure 13. □

The *dining philosophers problem* was suggested in 1968 by Edsger Dijkstra, a great pioneer in computer science. Five philosophers alternatively think and eat. They are seated around a large round table on which are a variety of Chinese foods. Between each pair of philosophers is one chop stick. To eat Chinese food, two chopsticks are necessary; hence each philosopher must pick up both the chopstick on the left and that on the right. Of course, if all the philosophers pick up the chopstick on their right and then wait for the chopstick on their left to become free, they will wait forever — a *deadlock* condition. Dijkstra formulated this problem to illustrate control problems that confront operating systems in which processes share resources and may compete for them with deadlock a conceivable result. To solve the problem some philosophers must eat while others are meditating. How can this be accomplished?

Example 13 Construct a Petri net to solve the problem of the dining philosophers.

Solution: The Petri net in Figure 14 solves this problem.

Each philosopher is represented by two places, meditating (M_i) and eating (E_i). There are also two transitions for each philosopher for going from meditating to eating and vice-versa. Each chopstick is represented by a place (C_i). A token in an eating or meditating place indicates which condition that philosopher is in. A token in a chopstick place indicates that that chopstick is free. A philosopher can change from meditating to eating only if the chopsticks on the left and right are both free. □

Numerous other applications of Petri net models have been developed, not all in computer science. Petri nets have been used to study the PERT technique of project planning and in studying legal processes. More information on these topics can be found in reference [1]. The bibliography of reference [2] lists 221 items including 47 papers and books on applications of Petri nets.

Analysis of Petri Nets

We have seen how useful Petri nets can be in constructing models, but perhaps

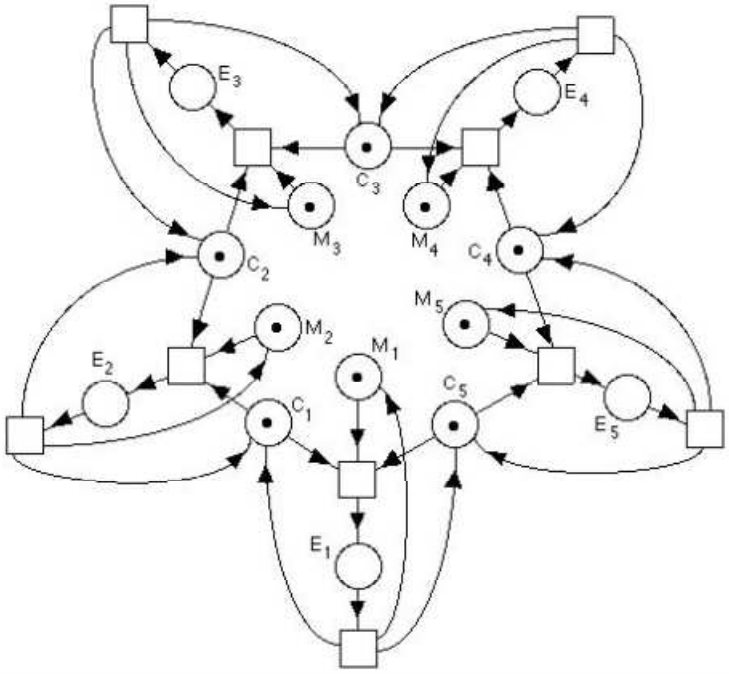

Figure 14. Petri net for dining philosophers problem.

the most important purpose of modeling is to be able to analyze the model to answer questions about the system being modeled. In this section we consider some problems related to the analysis of Petri nets. The most important analysis problem is that of *reachability*.

Definition 4 Given a Petri net and a marking m, a marking m' is *immediately reachable* from m if there is some transition which, when fired, yields the marking m'.

A marking m' is *reachable* from the marking m if there is a sequence $m_1, \ldots, m_k$ of markings such that $m_1 = m$ and $m_k = m'$ and for all i, m_{i+1} is immediately reachable from m_i. If N is a Petri net and m is a marking of N, then the set of all markings reachable from m is called the *reachability set* of m and is denoted $R(N, m)$. □

Example 14 Show that the marking of the net in Figure 15b is reachable from the marking in Figure 15a.

Solution: Firing t_1 twice and then t_2 yields the marking in Figure 15b. □

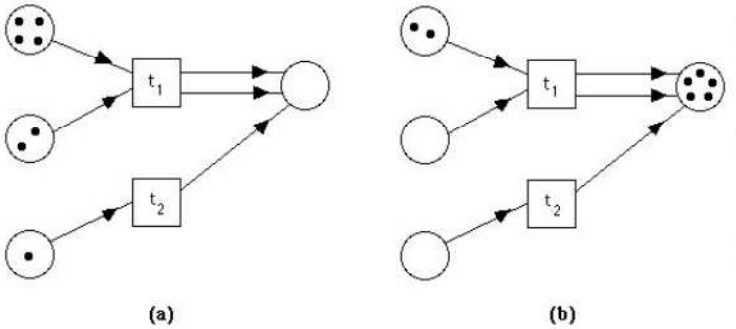

Figure 15. A reachable net.

Many interesting analysis problems for Petri nets can be phrased as questions about what kinds of markings are reachable from others. For example we introduced the concept of *deadlock* in the discussion of the dining philosophers problem. A net is in **deadlock** if no transitions can fire. In any net we might be interested in knowing whether a deadlock marking is reachable from some initial marking.

If a Petri net is to be a model of a real hardware device, one of the important properties it should have is *safeness*.

Definition 5 A place p in a Petri net N with marking m is *safe* if and only if for all $m' \in R(N, m)$, $m'(p) \leq 1$. That is, p never has more than one token. A net is *safe* if every place in it is safe. □

The reason safeness is important in modeling hardware devices is that if a place has either no tokens or one token in it, then that place can be implemented by a flip-flop. The nets in Figures 9, 10, and 11 are safe if the initial marking puts at most one token in each of the initial places. The producer/consumer net of Figure 13 is also safe with the marking shown.

Safeness is a special case of the property called *boundedness*. A place is **k-bounded** if the number of tokens in that place can never exceed k. A net is **k-bounded** if all of its places are k-bounded. Safeness is thus the same as 1-boundedness. A Petri net is **bounded** if it is k-bounded for some k. A Petri net which is bounded could be realized in hardware using counters for places while one with an unbounded place could not.

Petri nets can be used to model resource allocation systems. In this context the tokens might represent the resources to be allocated among the places. For these nets *conservation* is an important property.

Definition 6 A Petri net N with marking m is *conservative* if for all $m' \in$

$R(N, m)$,

$$\sum_{p \in P} m'(p) = \sum_{p \in P} m(p). \qquad \square$$

Conservation thus means that the total number of tokens is the same for all markings reachable from m. This is an extremely strong requirement, as the following theorem shows.

Theorem 1 If N is a conservative net then for each transition the number of input places must equal the number of output places.

Proof: If t is a transition with differing numbers of input and output places then firing t will change the number of tokens in the net. (See, for example, the second transition in Figure 6.) ■

The concepts presented so far all involve reachability in some sense. How can they be analyzed? Numerous techniques have been developed and are discussed in detail in references [1] and [2]. We will limit ourselves here to a discussion of the *reachability tree*. The reachability tree of a Petri net is a graphical method of listing the reachable markings. Our treatment follows that of reference [1], Chapter 4, Section 2. To begin, we fix an ordered listing of the places so that a marking can be written as an ordered n-tuple of nonnegative integers. An entry in an n-tuple gives the number of tokens assigned to the place in that position in the list. The initial marking of the net corresponds to the root vertex of the tree. For each marking immediately reachable from the initial one, we create a new vertex and draw a directed edge to it from the initial marking. The edge is labeled with the transition fired to yield the new marking.

Now we repeat this process for all the new markings. If this process is repeated over and over, potentially endlessly, all markings reachable from the initial one will eventually be produced. Of course, the resulting tree may well be infinite. Indeed, if the reachability set of a net is infinite, then the tree must also be infinite.

Example 15 Carry out the first three steps in construction of the reachability tree for the net shown in Figure 16, with the initial marking given there.

Solution: Figure 17 shows the result. The tree is obtained as follows: the triple (x_1, x_2, x_3) gives the number of tokens assigned to the places p_1, p_2, p_3, in that order. From marking $(1, 0, 0)$, firing t_1 yields marking $(1, 1, 0)$. From marking $(1, 0, 0)$ firing t_2 gives marking $(0, 1, 1)$. From marking $(1, 1, 0)$, firing

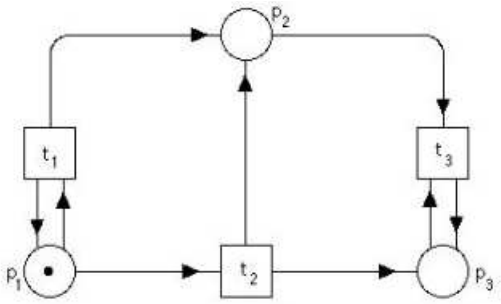

Figure 16. A Petri net.

t_1 yields $(1, 2, 0)$ while firing t_2 yields $(0, 2, 1)$. From $(0, 1, 1)$, firing t_3 yields $(0, 0, 1)$. This completes the second level down from the top. The third level is constructed similarly. □

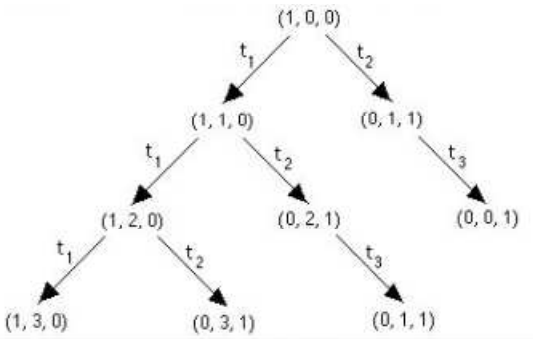

Figure 17. First three steps of reachability tree for Petri net of Figure 16.

If a net has an infinite reachability set, then of course its reachability tree will be infinite. However, it could happen that a net has a finite reachability set and still has an infinite reachability tree. This would be the case if a sequence of markings is repeated infinitely as in the net in Figure 18. For this net the tree is an infinite chain since the net alternates between the markings $(1, 0)$ and $(0, 1)$. As we have constructed it, the reachability tree contains the results of every possible sequence of transition firings, each sequence corresponding to some path from the root. Any reachability problem can, then, be solved by examining the reachability tree, though the search may be impractical.

While the reachability tree contains every reachable marking, its potential infiniteness poses a problem for analysis. Consequently, it is useful to have a finite representation of the reachability tree. We now describe a method for obtaining a finite representation of a reachability tree. For this purpose we introduce a classification of vertices in the tree. The new markings produced

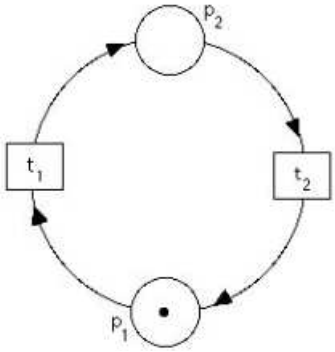

Figure 18. Petri net with infinite reachability tree.

at each step will be called *frontier* vertices. Markings in which no transition is enabled will be called *terminal* vertices. Markings which have previously appeared in the tree will be called *duplicate* vertices.

Terminal vertices are important because they yield no additional frontier vertices. Duplicate vertices yield no *new* frontier vertices, so no successors of duplicate vertices need be considered. We can thus stop generating frontier vertices from duplicate ones, thereby reducing the size of the tree. For the example in Figure 18 this will make the tree finite. In the example of Figures 16 and 17, the vertex $(0, 1, 1)$ in the third level from the root is a duplicate vertex and hence will produce no new markings.

One more case needs to be considered. If we examine the example of Figures 16 and 17 we can see that transition t_1 can be fired over and over endlessly, each time increasing the number of tokens at place 2 by one. This will create an infinite number of different markings. In this situation and in others like it there is a pattern to the infinite string of markings obtained. To aid in describing this pattern, it is useful to regard the n-tuples for markings as vectors and perform operations on them component-wise.

Suppose we have an initial marking m and some sequence of transition firings s yields a marking m' with $m' \geq m$ componentwise. We can think of m' as having been obtained from m by adding some extra tokens to some places so

$$m' = m + (m' - m)$$

componentwise, and $m' - m \geq 0$. Thus, firing the sequence s had the result of adding $m' - m$ to m. If we start from m' and fire s to get m'', then the result will be to add $m' - m$ to m', so

$$m'' = m + 2(m' - m).$$

In general we could fire the sequence n times to get a marking $m + n(m' - m)$.

We use a special symbol, ω, to represent the infinite number of markings which result from this type of loop. It stands for a number of tokens which can be made arbitrarily large. For the sake of simplifying our description it is useful to define some arithmetic "operations" using ω. For any constant a we define

$$\omega + a = \omega \qquad \omega - a = \omega \qquad a < \omega \qquad \omega \leq \omega.$$

These are all we will need in dealing with the reachability tree. We need one more notational device to simplify the description of the method for constructing the reduced reachability tree.

Definition 7 If m is a marking of a Petri net and t is a transition enabled for firing, then $\delta(m, t)$ is the marking produced from m by firing t. □

In the reduced reachability tree each vertex x is an *extended marking*; that is, the number of tokens at a place is allowed to be either a nonnegative integer or the symbol ω. Each vertex is classified as either a frontier vertex, a duplicate vertex, a terminal vertex, or an interior vertex. Frontier vertices are vertices which have not been processed by the reduction algorithm; they are converted to terminal, duplicate, or interior vertices.

The algorithm begins by defining the initial marking to be the root of the tree and initially a frontier vertex. The algorithm continues until no more frontier vertices remain to be processed. If x is a marking written as an n-tuple, let x_i be the ith component of x.

Let x be a frontier vertex to be processed.

1. If there exists another vertex in the tree which is not a frontier vertex and is the same marking as x, then vertex x is a duplicate vertex.

2. If no transitions are enabled for the marking x, then x is a terminal vertex.

3. For all transitions t_j which are enabled for firing for the marking x, create a new vertex z in the tree. The components of z are defined as follows:

   (a) If $x_i = \omega$ then $z_i = \omega$.

   (b) If there exists a vertex y on the path from the root to x with $y \leq \delta(x, t_j)$ and $y_i < \delta(x, t_j)_i$ then $z_i = \omega$.

   (c) Otherwise, $z_i = \delta(x, t_j)_i$. An edge labeled t_j is directed from vertex x to vertex z. Vertex x is classified as an interior vertex and z becomes a frontier vertex.

When there are no more frontier vertices, the algorithm stops.

Example 16 Construct the reduced reachability tree for the Petri net shown in Figure 16.

Solution: Figure 19 shows the required tree. □

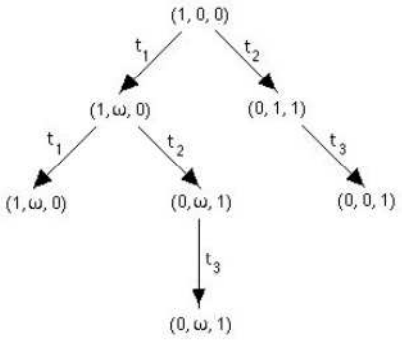

Figure 19. Reduced reachability tree.

It has been proved that the algorithm for constructing the reduced reachability tree always terminates in a finite tree. The interested reader is referred to Chapter 4 of [1] for details.

Many questions about reachability can be answered using the reduced reachability tree. However, because the use of the symbol ω condenses infinitely many vertices into one, some information may be lost by this process. Often the ω indicates a pattern which can be recognized, thus reducing the loss of information. The special reachability questions we presented can be answered using the tree. A net will be safe if the vertices in the tree all have only 0s and 1s as components. A net will be bounded provided the symbol ω never appears. A net will be conservative if the sums of the components of all the vertices are equal.

Petri nets are perhaps the most widely used modeling tool in computer science. The references each contain large bibliographies; [1] has a bibliography extending over 38 pages.

Suggested Readings

1. J. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Upper Saddle River, N.J., 1981.

2. W. Reisig, *Petri Nets, An Introduction*, Springer-Verlag, New York, 1985.