Using Petri Nets to Introduce Operating System Concepts

John M. Jeffrey
Department of Computer Science and Information Systems
Elmhurst College
Elmhurst, IL 60126

Abstract

Graph-theoretical tools, algorithm animation, and other related visual-aids have proven very useful in computer science pedagogy. In this paper the use of the graphical aspects of Petri net theory as a tool to introduce operating system concepts is given. It is shown how a minimal amount of Petri net theory can be applied to problems often discussed in an operating systems course. Examples of models for the concept of deadlocks, for the deadlock detection algorithm, and the fork/join and parbegin/parend concurrency constructs are shown. In addition, discussion of how Petri net models are utilized to introduce Ada rendezvous solutions of classic problems, such as producer/consumer with a ring-buffer, is given.

1 Introduction

Numerous graph-oriented models have proven very useful as pedagogical tools for teaching operating system concepts. For example, precedence, process, and resource allocation graphs are utilized by several operating system textbook authors to illustrate relevant concepts and issues [3,5,7,13,16]. However, graph-oriented models are not generally utilized to visualize the dynamics associated with concepts in concurrency and related problems often presented in an operating systems course. A model that also has a dynamic component can enhance the understanding these problems. This has been demonstrated in recent years through software projects developed to show program animation of concurrent processing problems [17] or visualize the information flow in an operating system [1]. In this paper, an argument is made for incorporating the graphical aspects of Petri nets as a lecture and textbook tool for introducing operating system concepts. Although this paper does not focus on utilizing software for Petri net editing, simulation, and analysis, very sophisticated systems are available [18].

A Petri net is a graphical and mathematical tool for modeling information systems that are considered to be nondeterministic, concurrent, parallel, asynchronous, distributed, and/or stochastic [8,12]. As a graphical tool it provides students with a visualization of concepts and issues typically discussed in an operating systems course. Petri nets include tokens to represent the nondeterministic evolution of a system and make explicit the concurrent activities of a system. A formal definition of Petri nets is given in section 2.

Based upon the apparent absence of Petri nets in operating system textbooks and course syllabi, it appears that many computer science educators are unaware of Petri net theory or that a minimal amount of Petri net theory can be very valuable as a pedagogical tool for teaching operating system concepts. Very little Petri net theory seems to have been applied to the area of operating systems. A variation of a Petri net model of the SCOPE operating system of the CDC 6400 is given in [11], but the model is not strictly a Petri net. To my knowledge the only operating system book that gives any space to Petri nets is [16]. Some authors of software engineering texts devote a brief subsection to show the usefulness of Petri nets within the requirements and/or design phase(s) [4,15]. One of the motivating factors to introduce Petri nets into operating system course lectures was that many of the software engineering students exhibited a better understanding of the mutual exclusion problem, deadlock, starvation, etc. Many claimed that they would have preferred Petri nets also in their operating systems course.

Because of limited space only a sampling of the many Petri net models used in operating system lectures are discussed. Some of these Petri nets not discussed in this paper model problems such as the reader/writer problems, a simple communication protocol, the cigarette smokers problem, the dining philosophers problem and data-flow computations. For discussion of these models and further introductory Petri net theory, the interested reader is recommended to consult [8,12]; both have extensive bibliographies. The few models that are shown in this paper are intended to illustrate how a minimal amount of Petri net theory can be used to visualize and analyze problems typically presented in operating system courses.

Perhaps some concern may be raised about the amount of time needed to introduce Petri net theory and model building. It has been my experience that only one lecture of 50–65 minutes is sufficient to introduce the basics of Petri net theory. Although Petri net theory is a well-developed area

of theoretical computer science—some taking a great deal of mathematical maturity—only a sophomore-level discrete mathematics background is needed. This paper indicates precisely how much Petri net theory and related mathematics is actually needed.

2 Petri Net Background and Fundamentals

While working as a scientist at the University of Bonn, Petri nets were first conceived and developed by Carl Petri in his dissertation, which was submitted in 1962 [14] to the Mathematics and Physics faculty at the University of Darmstadt. Since then, numerous papers have been written and a rich body of theory has been developed. For more information on historic developments see for example [8].

A Petri net is represented by an underlying bipartite directed graph and an initial state, called an *initial marking*. The two partitions are called *places* and *transitions*. Graphically, places are drawn as circles and transitions as boxes or rectangles. The arcs are weighted (labelled with positive integers) and are either from a place to a transition or vice versa. The arcs that are weighted with 1 are left unlabelled when drawing the Petri net. An arc labelled with k can be interpreted as k parallel arcs.

The state of the system that the net is modeling is represented by the assignment of non-negative integers to places. This assignment is called a *marking*. For the graphical representation, rather than labeling a place p with a non-negative integer k, k black dots, called *tokens*, are drawn inside place p. If the number of tokens is too large, then the non-negative integer is used.

Typically, the places represent some type of condition and the transitions represent an event in the system. The input (output) places to a transition represent the pre- (post-) conditions. The tokens may have many interpretations. For example, when a place is marked with a token it might represent that the corresponding condition is true. In other cases, k tokens might represent k resources, e.g. the number of tape drives available. Because Petri nets can model many types of systems, what the places, transitions, and tokens represent varies greatly. In [8] many examples of typical interpretations are given.

2.1 Definition of a Petri Net

Definition: A *Petri Net* is a 5-tuple, PN = (P,T,F,W,M_0) where:

$P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of places,
$T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions,
$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs,
$W : F \rightarrow \{1, 2, 3, \ldots\}$ is a weight function,
$M_0 : P \rightarrow \{0, 1, 2, 3, \ldots\}$ is the initial marking,
$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$. □

It is convenient to use the following notation. For a given transition t, let the *input* places and the *output* places of a transition t be denoted respectively by $\bullet t = \{p | (p,t) \in F\}$ and $t\bullet = \{p | (t,p) \in F\}$.

Formally, a marking M is defined as: $M : P \rightarrow \{0, 1, 2, \ldots\}$. It is also convenient in some cases to denote a marking M of m places as an m-vector where the i-th component denotes $M(p_i)$, i.e. $M = < M(p_1), \ldots, M(p_m) >$.

2.2 Transition Enabling and Firing

The state changes of a system are modelled via the enabling and firing rules. The rules are described as follows:

1. A transition t is *enabled* under marking M if each input place p of t is marked with at least $W(p,t)$ tokens, the weight of the input arcs to t. More formally, $\forall p \in \bullet t$ $M(p) \geq W(p,t)$ iff t is enabled.
2. A transition may or may not fire when enabled. When more than one transition is enabled, one is nondeterministically chosen depending on the model. The transitions that fire represent that the corresponding modelled events occurred.
3. A *firing* of a transition t results in $W(p,t)$ tokens being removed from each input place p of t and the addition of $W(t,p')$ tokens to each output place p'. More formally, firing an enabled transition t results in changing marking M to M' where:

$$\begin{aligned} M'(p) &= M(p) + W(t,p) \text{ if } p \in t\bullet \text{ and } p \notin \bullet t \\ M'(p) &= M(p) - W(p,t) \text{ if } p \in \bullet t \text{ and } p \notin t\bullet \\ M'(p) &= M(p) \text{ otherwise} \end{aligned}$$

If t has no input places (i.e. $\bullet t = \emptyset$), it is a *source* transition and is *vacuously* enabled. If t has no output places (i.e. $t\bullet = \emptyset$), it is called a *sink* transition. A sink transition "consumes" tokens and does not produce any tokens.

If a transition t is enabled under marking M, and M' is the resulting marking following the firing of t, we write $M \xrightarrow{t} M'$.

Note that as the transitions are fired the total number of tokens distributed throughout the net may vary, i.e. the conservation of tokens does not necessarily hold.

Example: Shown below in figure 1(a) is an example of a Petri net with both transitions enabled under an initial marking $M_0 = < 1, 2, 0 >$. Either one may be fired. We have either: $M_0 \xrightarrow{t_1} M_1 = < 0, 0, 1 >$ or $M_0 \xrightarrow{t_2} M_2 = < 1, 1, 1 >$. Figures 1(b) and (c) show markings M_1 and M_2, respectively. Under M_1 neither t_1 nor t_2 are enabled. Under M_2, t_2 is enabled and, if t_2 is fired again, we have $M_2 \xrightarrow{t_2} M_3 = < 1, 0, 2 >$. Figure 1(d) shows marking M_3. Under M_3 neither transition is enabled. For the initial marking M_0 and this rather simple net all firings are shown. □

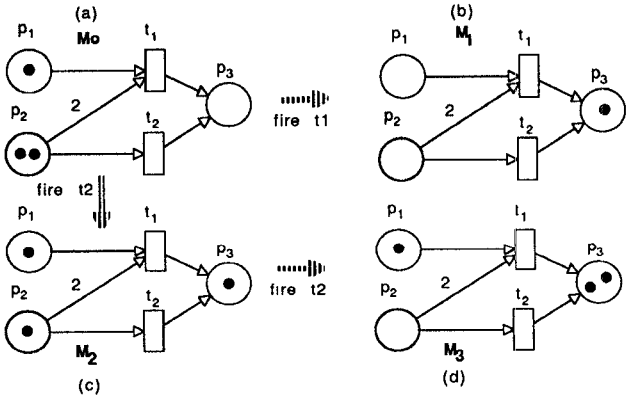

Figure 1: Example of Net Firings

It is important to note that the Petri net definition and enabling and firing rules given above permit a place to be marked with an unlimited number of tokens. These Petri nets are called *infinite capacity* nets [8]. There are also *finite capacity* nets where the marking of each place p is given an upper bound $K(p)$. Thus, an extra constraint is included in the enabling rule of a transition t: $M(p)$ cannot exceed $K(p)$ for all $p \in t\bullet$ if t were to fire. Although finite capacity nets are very useful for modeling some systems, they have not been needed in teaching operating systems thus far, so are not considered in this paper. For more discussion about finite capacity nets and their relationship with infinite capacity nets, the reader is referred to [8,12].

3 Petri Net Models of Operating System Concepts

There are many Petri net models for the classic problems typically covered in an operating systems course. In this section the usefulness of Petri nets as a pedagogical tool is illustrated through a few examples.

3.1 Modeling Deadlock Concepts

An important concept associated with resource allocation is deadlock. Within Petri net theory an important question is "does the net deadlock?" A Petri net is *deadlocked* iff no transitions are enabled. The first subsection presents two simple net models that provide a means to introduce the concept of deadlock. In the second subsection a graphical model of the bankers algorithm and deadlock detection algorithm, such as the algorithms presented in [3,5,7,13], is discussed. This model is a completely different model from the first two and exhibits a very different use of a Petri net model for the same general subject area.

3.1.1 Introducing deadlocks

The following model is used to introduce the concept of deadlock. Shown in figure 2 is a simple net model of a resource allocation scheme with only two instances of a single kind and two processes that require both resources before completing. The place R represents the single resource and each token represents an instance of the resource. The remaining places represent the thread of control of the two processes. The transitions t_1 and t_2 respectively represent the events of process a and process b requesting an instance of the resource. The transitions t_3 and t_4 represent a second request from each. Finally, transitions t_5 and t_6 model the releasing of both resource instances.

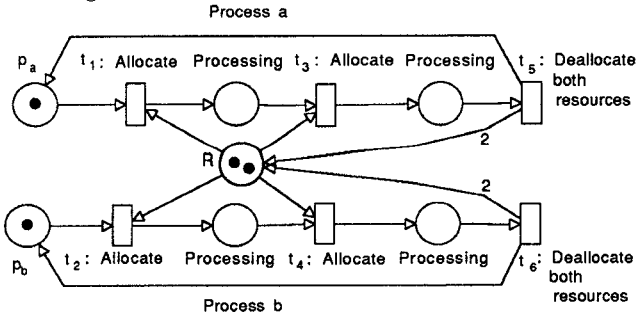

Figure 2: Resource Allocation Net Model 1

This net can deadlock if the following firing sequence occurs: t_1, t_2. At this point no transitions are enabled. Although resource allocation graphs are useful, the Petri net model adds the dynamic aspects explicitly, i.e. the possible sequences of events leading up to a deadlock. One may begin to examine what firing sequences cause deadlocks.

This particular model is simple enough to be a good starting point to illustrate the use and value of a Petri net. The students are not overwhelmed with many details. This model allows for discussion and perhaps better understanding of the four necessary conditions of deadlock: mutual exclusion, hold-and-wait, no preemption, and circular wait. This also helps develop a deeper understanding of the enabling and firing rules.

Once the first model is understood and the concept of deadlock is introduced, the subject of deadlock prevention and the related costs can be discussed. To illustrate the prevention protocol that calls for processes to allocate all their resources before execution begins one could use the following Petri net in figure 3.

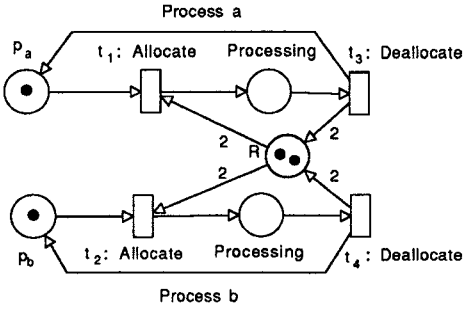

Figure 3: Resource Allocation Net Model 2

One can see that if t_1 fires, then t_2 and t_4 cannot be enabled until t_3 has been fired. This models the fact that process a has all the resources and process b cannot make any progress. Following the firing of t_3, both t_1 and t_2 are enabled. Suppose that t_1 is fired again and the same firing sequence is repeated. This illustrates the concept of *starvation*; in this case, the starvation of process b. Comparing this model with the model in figure 2, students are able to begin "seeing" the difference between starvation and deadlock. The dynamics of the model prove very useful.

Other issues such as the poor resource utilization of this particular deadlock prevention protocol can also be discussed.

3.1.2 A model for deadlock avoidance and detection algorithms

Deadlock avoidance and detection are two other major subtopics. Typically, the banker's algorithm is presented when discussing deadlock avoidance. The banker's (safety) algorithm checks for the safeness of allocating resources to a process p_i each time a request is made by process p_i. A similar algorithm is used for detecting deadlocks. The deadlock detection algorithm is run periodically. The output of the detection algorithm is either a sequence of processes, which indicates how the system of processes may complete, and thus indicate that the system is not deadlocked, or the indication that the system is deadlocked. Some sample texts that present the details of these algorithms are [3,5,7,13].

A Petri net model for the deadlock detection algorithm is presented next for a system of processes with several instances of each resource type. This model follows the algorithm as presented in [13]. Although a similar model can be derived for the banker's algorithm, only the deadlock detection algorithm is shown here.

To aid in presenting the net model we briefly review the algorithm's data structures as presented in [13].

In the following discussion we use X as an unconventional index variable to emphasize that resources are indexed by letters rather than non-negative integers.

Given n processes and m resource types, the algorithm uses an $n \times m$ allocation matrix, Allocation, whose entry, Allocation[i,X] = k, indicates that k instances of resource X are allocated to process i. Similarly, an $n \times m$ request matrix, Request, has entries, Request[i,X] = k, that indicate process i is currently requesting k instances of resource X. An m-vector, Available, whose entry, Available[X] = k, indicates k instances of resource X are currently available.

The generic net model is described as follows. There is a *process* place and transition pair for each of the processes in the system (each correspondingly subscripted). There is also a *resource* place for each resource type. Let p_i and p_X denote process and resource places, respectively. The arc set F and labeling function W consist of:

- arcs (p_i, t_i), each with weight one, for each process i;
- arcs from transitions t_i to resource places p_X with weight label k iff Allocation[i,X] = k and $k > 0$;
- and arcs from resource places p_X to transitions t_i with weight label k iff Request[i,X] = k and $k > 0$.

In the algorithm, a Boolean n-vector, Finish, is used to keep track of what processes were examined and can have their requests met. Finish[i] = true when the i^{th} process has been examined by the algorithm and its request can be met. Initially, Finish[i] is false for all i, where $1 \leq i \leq n$. The place markings correspond to Finish. A process place p_i is marked with one token iff Finish[i] = false, and marked with zero tokens otherwise. After a transition t_i fires, a token is removed from process place p_i. This corresponds to setting Finish[i] to true. The meaning of firing a transition is discussed shortly.

The resource places p_X are marked with k tokens iff Available[X] = k.

In addition to the data structures discussed above, the algorithm also uses an m-vector called Work, which is modified during the execution of the algorithm. Work is initialized by the Available vector. At each point in the execution of the algorithm, the value Work[X] corresponds to the number of tokens in p_X. When Request[i, X] $\leq$ Work[X] for each resource kind X and Finish[i] = false, the transition t_i is correspondingly enabled. This corresponds to the situation where the requests of the i^{th} process can be met. Note that more than one request might be met. Thus, several transitions might be enabled. One is nondeterministically chosen. If process i is chosen, then the Work vector is updated through vector addition of itself with the i^{th} Allocation row. This corresponds to the firing of transition t_i.

The reader is encouraged to compare this model with the versions of this algorithm presented in the above-mentioned textbooks.

An example taken from [13] is used to illustrate the "execution" of the Petri net model.

Suppose that five processes, $\{p_0, \ldots, p_4\}$, and three resource types, { A, B, C }, are given. Resources A, B, and C have 7, 2, and 6 instances, respectively. First, consider only the leftmost request matrix (Request); the rightmost request matrix (*Request '*) is used later in the paper. Suppose that the matrices representing the state of resource allocation at some time T are:

	Allocation			Request			Available			*Request '*		
	A	B	C	A	B	C	A	B	C	A	B	C
p_0	0	1	0	0	0	0	0	0	0	0	0	0
p_1	2	0	0	2	0	2				2	0	2
p_2	3	0	3	0	0	0				0	0	1
p_3	2	1	1	1	0	0				1	0	0
p_4	0	0	2	0	0	2				0	0	2

The following net model in figure 4 allows one to graphically depict the deadlock detection algorithm. (Ignore the dashed arc (P_C, t_2) since it corresponds to the *Request '* matrix.)

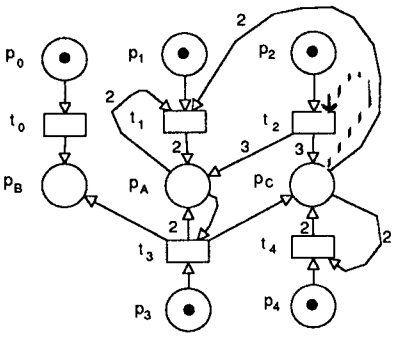

Figure 4: Deadlock Detection Model

One begins to arbitrarily (nondeterministically) fire enabled transitions until no transitions are enabled. Note that in this model, this does not mean the modelled system is deadlocked. Instead, the deadlock in the modelled system is detected if a token remains in any of the initially marked process places (as opposed to the resource places) *and* no transition is enabled. If all these process places are unmarked, then that firing sequence corresponds to a corresponding process completion sequence. For the above example t_0,t_2,t_3,t_1,t_4 is one possible firing sequence and corresponds to a solution presented in [13]. This indicates that the above system is not in a deadlocked state. Another example of the many possible firing sequences is: t_2,t_4,t_0,t_3,t_1.

Now using the rightmost request matrix, *Request '*, we have the same net, but with the dashed arc included. In this case one cannot reach a marking that has all the process places marked with zero tokens. Therefore, this system is deadlocked. The only enabled transition under the initial marking is t_0. After firing t_0 no transitions are enabled, yet there are still tokens in process places p_i for all i where $1 \leq i \leq 4$.

It should be pointed out that in [13], the authors give

resource allocation and wait-for graphs when there are single instances of each resource type. The Petri net model given above is a graphical depiction of the *general* case, i.e. several instances of each resource type. It can be thought of as a visualization of the deadlock detection algorithm (and the banker's (safety) algorithm).

Another graphical method available for deadlock detection is sequential reductions of resource allocation graphs; see [3] for example. These have also been proven useful. The Petri net model presented in this subsection is an alternative graphical technique for detecting deadlocks, and, as shown in this paper, can be used to model many other concepts in operating systems. If Petri nets are used throughout an operating systems course, then the students have a familiar graphical model.

3.2 Modeling Fork/Join and Parbegin/Parend Constructs

When introducing the concept of concurrent processing within a single process or the interprocess synchronization between processes the precedence constraints between the statements or processes are often represented graphically by a precedence graph such as in [7,13]. With respect to the concurrent processing within a single process, a *precedence graph* is a directed acyclic graph whose nodes represent statements and arcs (S_i, S_j) represent that statement S_i must complete before S_j begins. We shall use the processing within a single process for discussion.

The fork/join and parbegin/parend constructs are introduced by using the precedence graphs. Using only the precedence graphs, the fork and join statements do not have explicit nodes in the graphs. What is proposed here is the introduction of Petri nets as an intermediate representation between the precedence graphs and the concurrency constructs. The Petri net model models each statement, including the fork and join statements, via transitions. In figures 5(a) and (b) the Petri net models for fork/join and parbegin/parend are shown. Note that join models the join with more than two threads of control, i.e. join with a count parameter. See [13] for details.

Given in figure 6(a) is a precedence graph taken from [13] and the corresponding fork/join (6(b)) and parbegin/parend (6(c)) Petri net models. Note that if the join were limited to merging only two threads of control, both nets would be structurally identical. One can see explicitly where the fork/join actions occur. Also, the explicit flow of tokens allow the students to visualize the threads of control. Also, the net aids in defining the count parameter's initial value for each join and the number of joins needed when writing fork/join code.

Often the discussion of the expressiveness of fork/join vs. parbegin/parend can be aided with the use of this model. By using an example of a precedence graph that cannot be translated to a Petri net begin/parend constructs, students see the power of fork/join construct. On the other hand, the students also see why the parbegin/parend is a *structured* programming construct.

3.3 Modeling the Ada Rendezvous

The Ada rendezvous has been modelled by several researchers [2,6,9]. By utilizing the work of these researchers not only the rendezvous can be visualized, but solutions to classic concurrent problems can also be visualized. For example, in [2] a solution for the producer/consumer problem with a ring-buffer is shown. Using this model gives a visualization of the rendezvous synchronization, the semantics of the *selective wait with guard(s)*, the concept of mutual exclusion, and the producer/consumer problem itself. In addition, these Petri net model instances can be compared against Petri net model instances that model similar solutions that utilize P/V semaphore operations; see for example [4]. By comparing the two solutions the students can also get a better appreciation what is "hidden" in the implementation of Ada run-time environments.

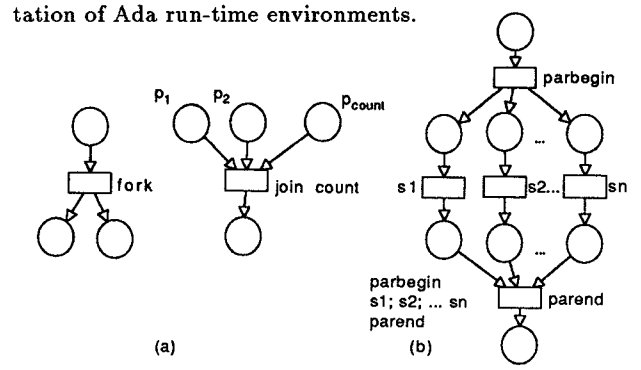

Figure 5: Fork/Join and Parbegin/Parend Models

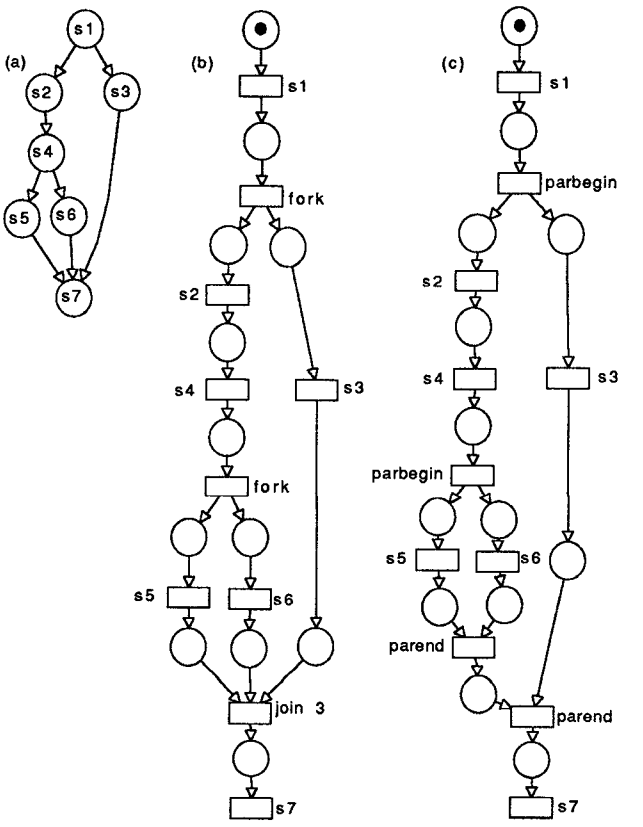

Figure 6: Transforming Precedence Graphs

4 Summary

Petri nets are a valuable pedagogical tool in an operating systems course. They act as a visual-communication aid for both the instructor and students. The explicit flow of tokens greatly enhances the understanding of the dynamic aspects of many operating system concepts, especially those like deadlocks, concurrency, mutual exclusion, etc. In addition, Petri nets can provide a kind of algorithm visualization.

From experience, the amount of time to introduce and get students started in understanding the Petri nets presented in this paper is minimal, approximately one to two lectures. As more concepts are introduced along with the Petri net models the students get a deeper understanding of Petri nets. This, in turn, seems to give a deeper understanding of the topics. The questions about the net models themselves often generate interesting and important questions about the problem being modelled. Although no software tools are currently being used, the intention is to make them available to students to edit, simulate, and analyze the net models so that the corresponding problems can be further investigated.

Petri nets have been applied to systems and subject matter found in many computer science curricula. Some systems and topics include pipelined computers, distributed systems, neural networks, and logic programming; there are many more [8,12]. Based upon the success and acceptance in the operating system course, Petri nets and variations on the Petri net model are being investigated as teaching tools in other computer science courses. A variation of the Petri net model presented in this paper, called a Predicate/Transition net, seems very promising in teaching logic programming and issues in parallelism in logic programs [8,10].

References

[1] Cañas, D., "GRAPHOS: A Graphic Operating System," *SIGCSE Bull.*, vol. 19, no. 1, pp. 201–205, 1987.

[2] Cherry, G., *Parallel Programming in ANSI Standard Ada.* Reston Publishers, 1984.

[3] Deitel, H., *Operating Systems.* Addison-Wesley, 1990.

[4] Fairley, R., *Software Engineering Concepts.* McGraw-Hill, 1985.

[5] Krakowiak, S., *Principles of Operating Systems.* The MIT Press, 1987.

[6] Mandrioli, D. and Ghezzi, C., *Theoretical Foundations of Computer Science.* John Wiley and Sons, 1987.

[7] Milenkovic, M., *Operating Systems Concepts and Design.* McGraw-Hill, 1987.

[8] Murata, T., "Petri Nets: Properties, Analysis, and Applications," *Proc. of IEEE,* vol. 77, no. 4, pp. 541–580, 1989.

[9] Murata, T., Shatz, S. M., and Shenker, B., "Detection of Ada Static Deadlocks Using Petri Net Invariants," *IEEE Trans. of Software Engineering,* vol. 15, no. 3, pp. 314–326, 1989.

[10] Murata, T. and Zhang, D., "A Predicate-Transition Net Model for Parallel Interpretation of Logic Programs," *IEEE Trans. of Software Engineering,* vol. 14, no. 4, pp. 481–497, 1988.

[11] Noe, J., "A Petri Net Model of the CDC 6400," *Proceedings of ACM SIGOPS Workshop on System Performance Evaluation,* April, pp. 362–378, 1971.

[12] Peterson, J., *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[13] Peterson, J. and Silbershatz, A., *Operating System Concepts.* Addison-Wesley, 1985.

[14] Petri, C.A., "Kommunikation mit Automaten," Bonn: Institute für Instrumentelle Mathematik, Schriften des IIM Nr. 3, 1962. English translation: "Communication with Automata," New York: Griffiss Air Force Base, Tech. Rep. RADC-TR-65-377 vol. 1, Suppl. 1, 1966.

[15] Schach, S., *Software Engineering.* Aksen Associates, 1990.

[16] Tsichritzis D. and Bernstein, P., *Operating Systems.* Academic Press, 1974.

[17] Zimmermann, M., Perrenoud, F., and Schiper, A., "Understanding Concurrent Programming Through Program Animation," *SIGCSE Bull.*, vol. 20, no. 1, pp. 27–31, 1988.

[18] Design/CPN Announcement, Feb. 1990, *Meta Software Corporation,* 150 Cambridge Park Dr., Cambridge, MA 02140.