# AN APPROACH TO OBJECT SYSTEM MODELING BY STATE-BASED OBJECT PETRI NETS[1]

A. Newman, S. M. Shatz, and X. Xie
Concurrent Software Systems Lab
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, Illinois 60607-7053

## Abstract

For many years, Petri nets have been used for modeling the behavior of various types of concurrent systems. While these net models are especially well suited to capture the behavior of concurrent systems, it is still the case that net models do not easily capture some important structural aspects of a system, such as modularity. In terms of software systems for distributed applications, the object-oriented paradigm has become a standard for defining modularity and reuse of software. Thus, an evolving direction in Petri net technology is the blending of net features with object-oriented capability. This paper discusses one such approach for state-based object systems.

## 1. Introduction

Computer applications have become more and more complex because of the proliferation of computer usage in different application domains. This complexity has resulted in a significant difficulty for software engineers to develop important systems that meet requirement specifications with low cost and high reliability. Thus, there is a well-recognized need for new techniques to organize the procedure of software development. Among these techniques, object-oriented design and programming have become extremely popular over the past few years, and they now form the basis for many contemporary programming standards (like CORBA) and languages (like Java).

To develop and design complex systems, in each stage, such as system analysis and system design, it is useful to combine many techniques to take advantage of properties

[1] This work was supported in part by the U. S. National Science Foundation under Grant CCR-9321743.

and strengths of the techniques. For example, object-oriented techniques have strength in focusing attention on a system's modularity. Other techniques, including those based on graph modeling, have strength in providing operational semantics (i.e., the meaning of various system-required operations) for specific properties like concurrency or distribution of function. These types of properties are of critical concern for new generation distributed and network computing applications.

Our interest in this paper is with one popular and mature graph-model technique, Petri nets. Petri nets provide a graphical and mathematical modeling tool applicable to many systems.[1] They are a promising tool for describing and studying information processing systems that have the characteristics of being concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic. As a graphical tool, Petri nets can serve as a visual-communication aid. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of systems. Thus, both practitioners and theoreticians can use Petri nets and the net models provide a powerful medium of communication between these users. However, Petri net models of practical systems tend to be large and computationally expensive to analyze. The so-called state explosion issue is a significant problem.[2]

The main idea of object-oriented techniques is breaking down a system into individual objects, each with its own specified externally observable behavior.[3] The actual details of the components are hidden from view of other designers who use the components. Message transfer is used to describe the communication between objects. Object-oriented techniques consider also how to define classes of objects and how to derive new objects from base classes. This latter feature, known as inheritence, is especially important for object-oriented programming languages so that they can support software reuse. Techniques that are centered around the concepts of objects and encapsulation, but which do not directly support inheritence features, are known as object-based techniques. There are many advantages of this technique, including support for abstraction and system maintenance. However, when we build a large, complex system, we tend to create a large number of object instances. Then, the relationships among objects also become more and more complex and object-based techniques alone do not provide a simple and direct way to describe the relationships clearly.

By combining Petri net modeling and object-oriented design we have the potential to achieve a modeling capability that can describe large, complex systems more easily and directly. Since the early 1980's, some researchers have worked in this area. In general, the proposed methods use enhanced forms of Petri nets as a base of the combination, and pursue two main approaches.[4] One is the "Petri nets inside objects" approach, in which traditional Petri net constructs are used to model the internal semantics of objects.[5,6] The other approach is the "objects inside Petri nets" approach, in which the semantics of tokens in Petri nets are expanded to include information that allows the tokens to be viewed as objects.[7] This can be considered as a step beyond the idea of colored Petri nets, where tokens have associated attributes, and thus the nets allow tokens to be distinguished from one another. In object Petri nets, the tokens have complex definitions and object-based semantics. The method discussed in this paper falls into the first category and provides an approach for object-based modeling. We describe a model called a state-based object Petri net (SBOPN). This model is most similar in spirit to one of the most advanced object-based Petri net modeling formalisms, known as LOOPN.[8] LOOPN's semantics are more rich, but in return are much less intuitive. Both methods use colored Petri nets as a foundation.[9] However, the primary encapsulator of object behavior in LOOPN is tokens, while SBOPNs use separate Petri net objects whose states are captured by special colored tokens. In this work, we focus on an object's state change and the relation between state changes and communication with other objects. It is proper to consider the SBOPN as an interpreted form of colored Petri net. It is "interpreted" since there are explicit constructs for object-based elements like objects, methods, and states, but these constructs use standard colored net elements.

The structure of the rest of this paper is as follows. Section 2 presents some basic introduction to Petri net modeling. Section 3 provides the motivation of this work by way of an example that uses a state-based net model. Section 4 presents key definitions and rules for the state-based net, and Section 5 briefly discusses some associated analysis issues and techniques. Finally, Section 6 provides a conclusion and mentions some future work.

## 2. Basic Petri Net Background

Here we provide a brief introduction to the Petri net formalism, which is a graph-based model for systems composed of concurrent events. Details on Petri nets can be found in other references.[1] Defined formally, a Petri net is a 5-tuple PN=$(P,T,F,W,M_0)$, where:

- *P* is a finite set of place nodes. A place node can contain any number of tokens, or it can be empty. When a place contains tokens, we say the place is *marked*.
- *T* is a finite set of transition nodes.
- *F* is a finite set of arcs (flows) that connect places to transitions and transitions to places (but never places to places or transitions to transitions).
- *W* is a weight function that associates a positive integer to each arc of *F*. By default, the weight of an arc is one, and a Petri net with all arc weights of one is called an *ordinary Petri net.*
- $M_0$ is the initial marking, or initial distribution of tokens to places.

Graphically, a Petri net is a directed bipartite graph. Informally, place nodes typically model conditions and transition nodes model events. With the addition of tokens, which are an attribute of place nodes, the graph takes on a dynamic capability to define the occurrence of modeled events. In a graph view, places are represented as circles, tokens as dots within the places, and transitions as bars. If a place *p* has an arc to a transition *t*, this is denoted as $p \quad {}^{\cdot}t$ or $t \quad p^{\cdot}$. Similar notation holds the case of an arc from a transition node to a place node. Associated with the transitions are *firing rules*, which determine the movement of tokens among places. For a transition *t*, if all places $p \quad {}^{\cdot}t$ are marked (each has one or more tokens) and each has a "sufficient" number of tokens, as explained below, then *t* is said to be *enabled.* An enabled transition *t* can fire, causing tokens to be removed from places $p \quad {}^{\cdot}t$ and tokens to be deposited into all places that are outputs of *t* (i.e., all $p \quad t^{\cdot}$). The weight function *W* determines both 1) the number of tokens that are required to be associated with an input place in order to enable a transition (the sufficient number of tokens mentioned previously), and 2) the number of tokens that get removed from, and deposited to, places when an enabled transition does fire. So, we can now consider the Petri net graph shown in Figure 1 and see that it is an ordinary Petri net with five transitions and two places. For the initial marking, four of the transitions are enabled and thus any one of them can fire. After the firing of any one of these transitions, only the transition *loop* is enabled. The firing of this transition returns the Petri net to its initial state.

The state of a Petri net can be represented by the markings associated with the net's places. The marking of a place refers to the number of tokens that are present in the place. The firing of a transition results in a next net marking or state. Net markings can be represented as a vector, with each entry in the vector representing the marking of a particular Petri net place. A *reachability graph* is a graph where each node represents a Petri net marking, with arcs connecting each marking with all of its next markings. The reachability graph defines a net's state space (i.e., the set of reachable states). A particularly important type of state is a *deadlock state*, which is represented by a reachability graph node that has no next marking.[1] The net in Figure 1 has no deadlock state.

In standard Petri nets, tokens have no identity. Therefore, it is arbitrary as to which particular tokens are to be removed from a transition's input place in the case that the input place contains more tokens than are needed to enable the transition. One way to change this semantic is to introduce the concept of tokens with identity. This type of Petri net is called a colored Petri net.[9]

In a colored Petri net, tokens can have attributes and the transition enabling and firing rules are based on these attributes. By convention, token attributes are also referred to as colors and hence the name, colored Petri nets.[10,11] With this extension, a marking is now the set of colors of tokens that reside in all the places in the Petri net. A transition is enabled if all the input places to the transition have a specified required set of colored tokens in them. The required tokens are indicated by an inscription on the associated arc. The arc inscription specifies a token attribute. The firing of an enabled transition causes the removal of specified colored tokens from the input places and the depositing of specified colored tokens in output places. Again, arc inscriptions -- this time on output arcs of a transition -- are used to set the attributes of deposited tokens. For example, it is possible to have a transition that is enabled if one of its input places has a token with attribute “value=3.” Upon firing, this transition can remove that token from the input place and deposit a new token with an attribute “value=4” into an output place. As indicated by this simple example, colored nets are often used when it is necessary to model data values.

Colored Petri nets are very efficient in modeling a wide range of systems as they have the virtues of standard Petri nets apart from their own advantages due to the additional feature of token attributes. In terms of modeling power (in contrast to modeling convenience), we can observe that a colored Petri net with a finite color set can be transformed into a standard Petri net.[10] The transformation involves the introduction of extra place and transition nodes so that a transition's firing is controlled by which particular places are marked rather than by which tokens are present in a given place. This process is sometimes referred to as an “unfolding” of the colored net. For instance, a colored Petri net model for the classical dining philosophers problem with *n* philosophers only needs 5 places and 4 transitions, whereas the behaviorally equivalent standard Petri net would require 5*n* places and 4*n* transitions.[12]

## 3. Motivation and Example

We begin by considering some software issues and evaluate how they are typically approached in terms of Petri net modeling and analysis. This will offer some motivations to expand the models to a state-based object Petri net.

Consider the problem of performing automated analysis of a concurrent software system -- a program composed of concurrently executing processes. One conceptually simple way to support such analysis is to convert the program code into a Petri net model where we treat all variable values, and all decisions made as a result of those values, as indeterminate. In this way, each statement of the program code is represented as a state, and movement between states is dependent upon consecutive statements, and results of branching statements. Since all variable values are indeterminate, the model provides an abstraction of the program where the execution of any leg of a branching statement is now considered to be equally likely. Thus, the Petri net model is an abstraction of the program code -- the model is conservative in that it includes all possible program behaviors, but it may also include some behaviors not possible in the program. This is the basic approach taken in almost all work in the area of static analysis, which is a type of analysis based on code evaluation, as opposed to testing, which is based on actual code execution.[13]

In the Petri net representation of a sequential program (with one process), states of the program are modeled by places in the Petri net. The presence of a token in a place represents that the program is in the corresponding state. In the case of a concurrent program with several independent processes (or threads of execution), process will be in a single state, which means that in the Petri net model a net marking will represent the global state of the concurrent system. In other words, the global state is a composite view of all the (local) states of the processes at some point in time.

Consider the classic example of a system that uses a bounded buffer to temporarily hold items, such as messages. The items are put in the buffer by producer processes and taken from the buffer by consumer processes. The three system components -- buffer, producer, and consumer -- operate asynchronously and only interact via messages initiated by the producer or consumer: *put* and *get* messages, respectively. In particular, the producer sends *put* messages to the buffer when the producer has some new item to be deposited into the buffer and the consumer sends *get* messages to the buffer when the consumer desires to remove an item from the buffer. Both the put and get actions are blocking in nature, meaning that they occur only when both the action is requested and the buffer agrees to perform the action. At any point in time, the buffer may be in one of three states: *Empty*, *Full*, or *Partially Full*. Depending on its state, the buffer may or may not be able to accept the messages *put* and *get*. Listing 1 provides an example coding of the buffer process using the Ada programming language. The Ada language provides several features that facilitate message-based concurrent programming.[14] Among these is the concept of a task, which is a component for concurrent execution. Tasks can define entries, which are services that can be called by other tasks. A task accepts a call of one of its entries by executing an accept statement for that particular entry. Also, the language provides a select statement to specify the need for a nondeterministic selection from among several alternatives.

```
-- Initialize the buffer.
BufState=Empty
-- Loop forever
loop
-- When buffer is empty, only accept put message
select
   when BufState=Empty =>
   accept put(item: in item_type) do
     -- some code for putting item in buffer
     BufState := Partial;
   end put;
-- When buffer is partial, accept put and get message
or
  when BufState=Partial =>
  accept put(item: in item_type) do
    -- some code for putting item in buffer
    -- IsFull() returns true if the buffer is full
    if (IsFull())
      BufState := Full;
    end if;
  end put;
or
  when BufState=Partial =>
  accept get(item: out item_type) do
    -- some code for getting item in buffer
    -- IsEmpty() returns true if the buffer is empty
    if (IsEmpty())
      BufState := Empty
    end if;
  end get;
-- When buffer is full, only accept get message
or
  when BufState=Full =>
  accept get(item: out item_type) do
    -- some code for getting item in buffer
    BufState := Partial;
  end get;
end select;
end loop;
```

**Listing 1 : An Ada listing for a bounded buffer example**

Figure 1 illustrates a typical Petri net model of this program as derived using translation techniques.[13] Notice that the model makes it appear as if the buffer accepts the *put* message even when in the *Full* state, and the *get* message when in the *Empty* state, though this is not possible in actual program execution. If a full reachability graph is created from the Petri net, some of the nodes in this graph will represent states of the model that do not correspond to actual program states. Naturally, such spurious states will increase the size of the reachability graph and affect the accuracy of the net model.

One feature of the object-oriented paradigm is that objects accept different messages depending on their states. If the state of an object could be represented by a single variable, different places in the object's corresponding Petri net model could represent each state. In order for an object, say object *A*, to accept a message from another object, say object *B*, it is necessary for *A* to be in a state in which it can receive this message, and *B* must be in a state in which it can send this message. In terms of the Petri net model, the places that represent the states of *A* and *B* respectively must contain tokens. The acceptance of a message can be modeled as the firing of a transition, which is enabled if and only if its incoming places (representing states in *A* and *B*) are marked.

The bounded buffer program from Listing 1 can also be represented by a Petri net using places to represent the *BufState* variable in the buffer, which is illustrated in Figure 2. Notice that, in Figure 2, the representation of the interaction between the producer and the buffer requires two transitions (*t2* and *t3*). The two arcs coming from the place "Ready to Produce" to two different transitions make the producer model look as if it is modeling a decision. Yet, the producer does not make a decision. It is the buffer's state that decides how the buffer accepts the *put* message from the producer. This situation occurs because the modeling of the producer and buffer are not cleanly separated in this modeling example.

The model should represent the fact that the buffer, and not the producer, changes state. This can be done with the use of "colored" tokens to represent the state of objects. Consider Figure 3. The places representing *Partial*, *Full*, and *Empty* can be represented by one place, say *P1*, while the transitions that represent the acceptance of the *put* message can be modeled by a single transition, say *T1*. Finally, the arc connecting the place to the transition can then be 'inscribed' with the set of states (*Partial* and *Empty*) in which a *put* message can be accepted. The states are represented by the color of the token occupying *P1*. In Figure 3, we show the initial state as one token in place *P1* and this token has color "Empty," denoted by the letter *E* inside place *P1*. Similar notation is used in subsequent figures. We denote as *States* the set of all possible states for the object of consideration. Finally, the arc leading out of *T1* defines a function, *F1*, which determines the next state, based on the message received and the current state. *F1* determines the next state by defining a possible change in the color of the token. The extra transition and place, *T2* and *P2*, represent the transition to the next state based on the message received. The token in *P2* defines the (new) state of the buffer, but this new state is not yet externally observable to other objects. Once the transition *T2* fires, the buffer's state is represented by a token in place *P1*, which does allow this state to be observed by other objects. Of course, in this example, we could simplify the net model by eliminating *P2* and *T2* and then directing the output arc from *T1* to *P1*. This creates a self-loop between *P1* and *T1*. In fact, this simplification will be used in a model presented later in the paper. Notice also the border around the buffer object, to emphasize its independence from the producer object. The token occupying *P1* is a colored token and the arc leading from *P1* to *T1* will "pass" the token. In other words, assuming the producer is in a state where it is ready to produce, the transition *T1* is enabled if the token in *P1* is of color *Partial* or *Empty*, but not if it is of color *Full*. This forms the basis of the state-based object net model that we are identifying.

One thing missing from this model is the definition of *F1*. This controls the color of the token based on the message received and the current state. In general, if we call *M* the message received and *S1* the current state, then a state-transition function of an object is a relation from the variables (*M*, *S1*) to the set of possible states S. For this particular example, we can define *F1* as in Table 1. Since this transition is only enabled in the event of a *put* message, without ambiguity we can say that *F1* is a function of *S1*. Notice that the output of *F1* on the input *Partial* can be either the value *Partial* or *Full*. We consider this output to be non-deterministic, as we are assuming that some detail not represented in the model actually determines the size of the buffer. Also, we are assuming that the size of the buffer is greater than one item -- otherwise, it would be possible to transition directly from the empty state to the full state. In general, when the output of a state-transition function exhibits a one-to-many behavior, as in this example, we assume that the actual output is non-deterministic.

| Starting State | Message | Ending State |
|---|---|---|
| *Empty* | *put* | *Partial* |
| *Partial* | *put* | *Partial* |
| *Partial* | *put* | *Full* |

**Table 1** The state transition function *F1* for a bounded buffer's *put* message

Since a buffer will accept messages from both a producer and a consumer, we must extend the previous model to that shown in Figure 4. A new transition has been added to accommodate the acceptance of the *get* message that would come from the consumer. Also, a new function *F2* is needed -- it is similar to function *F1* and is defined in Table 2. Finally, Figure 5 shows the producer and consumer objects. Each object is distinctly represented in this Petri net. Similar to Figure 3, the net in Figure 5 contains some internal place (P1) in which the object's state is not externally visible. Notice that the transitions that represent the acceptance of a message are drawn on the edge of their object boxes. Thus, the arcs that represent the sending or receiving of a message extend outside their object box's boundary. This allows us to draw a more simplified object interaction diagram, which emphasizes the relationship between objects. The purpose of this diagram is to emphasize which objects are sending and receiving messages, without showing the internal details of the objects -- a true modularity property. This abstraction is illustrated in Figure 6. Note that each object used in the system design is explicitly shown. Our method does not directly support replicated objects. For example, the buffer object we developed cannot interface with a set of producer objects without modifying the buffer model to explicitly contain other shared transitions for each such producer model.

| Starting State | Message | Ending State |
|---|---|---|
| *Full* | *get* | *Partial* |
| *Partial* | *get* | *Partial* |
| *Partial* | *get* | *Empty* |

**Table 2** The state transition function *F2* for a bounded buffer's *get* message

In the reachability graph of this new, higher-level Petri net, we would expect the same number of states as in the Ada net model shown in Figure 2. However, because the logic for state transitions of objects is represented in the model, it is possible to avoid the exploration of some spurious states. Therefore, we can define a new algorithm for computing the reachability graph of this object-based net. Before discussing such analysis issues, let us first formalize some of the concepts and definitions that have been raised by the previous discussion.

## 4. State-Based Object Net Definitions and Rules

In this section we define a type of Petri net called a State-Based Object Petri net (SBOPN). In effect, this is a special case of a colored Petri net model where colored tokens are used specifically for the purpose of capturing an object's state. Furthermore, object components are explicitly modeled by the use of special transitions that provide an interface to the object's methods.

Definition 1: A *state-based Petri net object* is a 7-tuple SBPNO (Type, States, Stoken, IS, ST, SF, STR), where

. *Type* is an identifier for the object’s type (or class).

. *States* is a finite set of distinct states that define the possible states of the SBPNO.

. *SToken* is a state token. This token has a value that represents the current state of the SBPNO. The value of *SToken* is an element of *States*.

. *IS* is a net model of the object’s internal structure.

. *ST* is a set of shared transitions. A shared transition in a SBPNO is a transition that is shared with other SBPNOs. This represents the acceptance of a message from other SBPNOs or sending a message to other SBPNOs.

. *SF* is a set of state filters. For each arc from any place to any transition, there is an associated state filter. A state filter is defined as a set of states *S* *States*. Only the state tokens that represent one of the colors in *S* can pass via the corresponding arc. The default state filter of an arc (if none is specified) is to allow any token to be passed.

. *STF* is a set of state-transition functions. For each arc from any transition to any place, there is an associated state-transition function. A state-transition function is a function $F_{st}$: $States \rightarrow 2^{States}$, where $2^{States}$ is the power set of *States*. In a SBPNO, this function can change the color of a state token. If no state-transition function is defined, the default is that the color of the token remains unchanged.

A SBPNO is denoted graphically as a Petri net (a subnet) inside a box.

Remark 1: In general, we only define state filters and state-transition functions for arcs connected with shared transition. For other arcs, we use the default state filter and state-transition function.

Definition 2: A state-based object Petri net, SBOPN, is a Petri net consisting of connected SBPNOs as defined in Definition 1.

Definition 3: A marking of a SBOPN is the distribution of state tokens to SBPNOs in a SBOPN, and a SBOPN system is a SBOPN with its initial marking $M_0$.

Now that we have a formal definition for a SBOPN system, we can identify the following high-level steps for modeling a system using the SBOPN formalism.

Step 1: Determine the objects of the system. First, each physical entity of the system should be an object. Then, we introduce other abstract objects based on the need of the particular application.

Step 2: Determine the states of each object. Then for each object, create a Petri net to model the object's basic control flow. This provides the object's internal structure.

Step 3: Determine the messages that each object can accept in each state, and identify the next state that is active after the acceptance of such messages.

Step 4: Determine the communication among objects. Each communication is represented by a shared transition with a state filter and state-transition function depending on the information in Step 3.

Step 5: Determine the initial state of each object.

For instance, in our previously considered system example, we identify the natural objects of producer, consumer and buffer according to Step 1. Applying Step 2, we find that the buffer object has three states: *Empty*, *Partial*, *Full*. The producer and consumer objects each have a single state according to our previous explanation of the system. For Step 3, we can establish a few facts that were previously defined in Table 1. For example, if the buffer is in the *Partial* state and it receives a *put* message, then the buffer can change state to either the *Partial* state or the *Full* state. Of course, the actual resulting state will depend on whether or not the buffer's current size is one less than full when it receives the *put* request message. But, for the purposes here, we are not distinguishing that special case of the *Partial* states. So, now Step 4 and Step 5 together can lead us to the corresponding SBOPN system shown in Figure 7. As we mentioned in Section 2, we have simplified this model by removing the internal places and transitions shown in Figures 4 and 5. Note that the states of Producer object and Consumer object never change, so their state tokens have a constant state value.

## 5. Some Analysis Issues

Creating an SBOPN system does provide us with a model that explicitly identifies objects and their state behavior. Furthermore, the model can serve as an executable specification by performing simulations of the net. Associated with this capability is the need to define reachability generation, which is a basis for behavior analysis that can verify and/or identify properties of the corresponding system. In some cases, we are interested in trying to establish that some properties are true (like eventual reception of a message in a communication system), but in other cases we may be interested in showing that some properties are not true (like concurrent use of some shared resource). In this section, we discuss two methods to support analysis of an SBOPN system. One method is indirect, in that it does not apply directly to the SBOPN model, but relies on a transformation of the SBOPN model. The other method is direct, in that it defines how to perform reachability analysis directly on an SBOPN system.

### 5.1 Unfolding of SBOPN

It is commonly understood that to do analysis of a colored Petri net, we can first unfold the net to create an ordinary Petri net, and then apply standard net analysis techniques and algorithms to the ordinary net. This is because a colored Petri net is actually a folded model of an ordinary Petri net. Considering our SBOPN, we can say that it is in fact a folded model of a colored Petri net with some constraints. If we unfold the state filters and state-transition functions, the SBOPN will be a colored Petri net. Then we can use further unfolding or special colored Petri net techniques to do the subsequent analysis.

The unfolding process is quite straightforward. First, we must separate each state in each state filter. Second, we modify the non-deterministic state-transition functions to form deterministic ones. After we combine these two modifications, each transition, whose input and/or output arc has a non-default associated state filter and/or state-transition function, will be replaced by several new transitions. The new net will no longer have any state filters or state-transition functions. For instance, in our previous producer, consumer, and buffer SBOPN, we would unfold the two shared transitions to create six transitions as shown in Figure 8. Note that Figure 8 is a colored Petri net, so the color set is a “global color set” (i.e., it applies to all places of the net model). The arc inscriptions in Figure 8 are transition color functions as defined by colored Petri net notation.

Formally, we can express the unfolding process for an SBOPN as an algorithm as shown in Listing 2. Naturally, once an SBOPN is unfolded to create a colored Petri net, we can use available colored Petri net techniques and tools to aid the analysis.

```
Algorithm Unfold SBOPN
1 For each transition t, whose input and/or output arc has
  a non-default associated state-filter and state-
  transition function f
2   For each state S_1 in state-filter
3     For each state S_2 in f(S_1)
4       add a transition t_{S1,S2} to the SBOPN with the same
          input and output arcs as t, and replace the state-
          filter with the inscription S_1 and replace the
          function f with the inscription S_2.
5     End for (S_2)
6   End for (S_1)
7   delete t
8 End for (t)
```

**Listing 2** Pseudo-code algorithm for unfolding an SBOPN

## 5.2. Reachability Graph Construction

If we can produce the reachability graph of the SBOPN system, then we can directly identify many useful properties, since we will have an explicit state-space view of the system. So, the first thing needed is to define the firing of a transition in a SBOPN system.

Definition 4: Let *p* and *t* be a place and transition in a SBPNO system, and let *e* be an arc connecting them. Then we denote *p*>*e* as true if and only if a token in *p* has one of the state values specified in *e*'s state filter.

Definition 5: A transition *t* is said to be enabled if and only if, for each *p* ˙*t*, and for each arc *f* connecting *p* and *t*, *p>f*.

Given a marking in a standard Petri net, the set of next markings is determined exclusively from the firing of enabled transitions, and the firing of one transition produces exactly one next marking. In a SBPNO system, if a transition *t* is enabled and *t*'s input and output arcs have default associated state-filters and state-transition functions, then the result of firing *t* is similar to the result of firing any transition in a standard Petri net. Thus, the firing removes a state token from each of *t*'s input places and deposits a state token (with the same state value) into each of *t*'s output places. This firing produces exactly one next marking. On the other hand, if the input and output arcs of the enabled transition *t* have non-default associated state-filters and/or state-transition functions, then the firing is similar to that of a colored Petri net. Thus, the firing of t causes the removal of specified state tokens from *t*'s input places and the depositing of specified state tokens in *t*'s output places. The specified state tokens are identified by the state-filters and state-transition functions. However, unlike in colored Petri nets, as a consequence of the state-transition function, the firing of a transition in an SBPNO system can result in more than one marking -- in fact, such a firing can result in as many markings as the number of outputs of the state-transition function. Each marking corresponds to one of the outputs of the state-transition function. For example, consider the SBOPN in Figure 7. Given the marking *M={Ready, Ready, Partial}*, the transition *put* is enabled, and the firing of *put* will produce two markings, $M_1$*={Ready, Ready, Partial}* and $M_2$*={Ready, Ready, Full}*. These correspond to the two possible outputs of the state-transition function *F1* when the buffer is in the *Partial* state, i.e., *F1(Partial)={Partial, Full}*.

Furthermore, if there are several output arcs from a transition, the number of markings that may result is equal to the number of combinations of outputs of the state-transition functions of the outgoing arcs. For example, consider Figure 9, where object *A* is sending a message to object *B*. Let *S1* denote the current state of object *A* and let *S2* denote the current state of object *B*. Also, let *F1(S1)* be the set of possible next states in object $A$ that result from the firing of the transition while in state $S1$. Similarly, let $F2(S2)$ be the set of possible next states in object $B$ that result from firing the transition while in state $S2$. In this case, there are $F1(S1)*F2(S2)$ possible markings that can result from the firing of this transition.

When creating a reachability graph for a Petri net, each new marking is compared to previously generated markings. The new marking will be added to the reachability graph only when it is not already in the graph. This basic idea is still applicable to reachability graphs for SBOPNs.

Definition 6: Given a SBOPN $N$ and a marking $M_0$, a reachability graph of $(N, M_0)$ is a directed graph $(V, E)$, where:

- Each vertex $v$ $V$ contains a unique marking.
- Each edge $e$ $E$ connects a vertex containing a marking $M$ to some vertex containing a marking reachable from $M$ by firing some transition $t$. This edge is inscribed with the label for this transition $t$.

Now we can give an algorithm for computing a reachability graph in terms of the definitions given above. Listing 3 presents the algorithm, which takes as input an initial marking, $m$. The list of enabled transitions is created according to the rules in Definition 5. The algorithm is recursive, and is similar to the algorithm for the generation of reachability graphs for standard Petri nets. As we noted earlier, one important difference between the reachability graph generation algorithms for SBOPNs and standard Petri nets is that the algorithm for SBOPNs must consider the state transition functions from Definition 1. Attaching one of these functions to a transition means that there is more than one possible next state that can result from the firing of a transition. This is in contrast to standard Petri nets, where the firing of a transition creates exactly one next marking. This issue accounts for the inner loop, which starts on line 4. The identifier $M_{set}$ is actually a set of markings, and each marking in this set must be explored after the firing of the transition $t$.

```
Algorithm Reachability Graph (Marking m)
1   Let $T_{set}$ be the set of enabled transitions
2   For each t in $T_{set}$
3     Let $M_{set}$ be the set of markings produced by the firing of
      transition t (from marking m)
4     For each m’ in $M_{set}$
5       If (m’ is not already in reachability graph)
6         add m’ to reachability graph
7       End if
8       If the reachability graph does not yet contain an edge
        from m to m’ inscribed with t, then add such an edge
        to the reachability graph
9       If (m’ is a new vertex)
10        Reachability Graph(m’) /* Recursive call */
11      End if
12    End for
13  End for
```

**Listing 3** Pseudo-code algorithm for finding reachability graph of a SBOPN

The first *if* statement (line 5) decides if the new marking is already in the reachability graph. The second *if* statement (line 8) decides whether or not there is currently an edge inscribed with *t* from *m* to the newly generated marking *m*’. The third *if* statement (line 9) recursively calls the function if the marking is a new marking (i.e., the first *if* statement evaluated to *true*).

Let us consider the construction of a reachability graph for the net shown in Figure 7. Since the markings in the producer and consumer object will not change, we can specify each node in the reachability graph with the letters *E*, *P*, and *F*, to stand for the *Empty*, *Partial*, and *Full* states of the buffer object. We can now use the algorithm from Listing 3 to step through some markings.

The buffer is initially empty, so the marking is <*E*>. In line 1, $T_{set}$ is set to {*put*}, since the *get* message can not be accepted when the buffer is empty. In line 3, $M_{set}$ is set to

{<*P*>}, since the firing of the transition *put,* which represents acceptance of the *put* message, makes the buffer partially full. Since <*P*> is not already in the graph, an edge from <*E*> to <*P*> is added to the graph, and the algorithm is recursively called with the <*P*> marking. This time, in line 1, $T_{set}$ contains {*put*, *get*}, since either the *put* or *get* message can be accepted. If *put* fires, $M_{set}$ contains both {<*P*> and<*F*>}, from Table 1. For marking <*F*>, and edge from <*P*> to <*F*> is added to the graph, and the algorithm is called recursively again with marking <*F*>.

For this recursive call, $T_{set}$ = {*get*}, since *put* is not enabled in the *Full* state. This call will add an edge from marking <*F*> to marking <*P*>, and return, since marking <*P*> is already in the graph. After the return, the algorithm explores the firing of *get* from the marking <*P*>. Figure 10 shows the complete reachability graph. From this graph, we can see that the system is live, meaning that it does not have any deadlock states.

## 6. Conclusion and Future Work

It is commonly accepted that Petri net models are well suited for specification of concurrent systems. But standard Petri nets do not easily capture some structural aspects of such systems, including modularity. This has lead to much work on augmented forms of Petri nets, and most recently to efforts to blend some object-oriented capabilities into net models. This paper presented one such approach, which focuses on systems composed of interacting components that are characterized by some internal state and state changes that result from object interactions. The model discussed is a special purpose type of colored Petri net, where token colors are used to specifically define object states. We defined this model, called a State-Based Object Petri Net (SBOPN), and discussed the basic steps for modeling a system using the SBOPN formalism. Two approaches to support reachability analysis were also discussed: unfolding of a SBOPN into a colored Petri net, and direct reachability graph construction. Throughout this paper we used a classic example problem, the producer/consumer problem, to motivate and explain the basic definitions and analysis issues.

As can be expected, a natural concern with reachability analysis of SBOPNs is the cost in terms of complexity of the underlying algorithm. In general, such reachability analysis suffers from the well-known state explosion problem. Unfortunately, SBOPNs do not magically escape this difficulty. Yet, there may be some hope in finding more efficient analysis method by exploiting the explicitly defined modularity of SBOPN models. One possible avenue for future work is to define a suitable concept of “equivalent markings” according to the object-oriented nature of the system. Then, equivalent markings can be represented by one reachable state, instead of by multiple states, as is currently the case. What we mean by equivalent markings are markings that only distinguish between states of objects of the same type, for example producers in a producer/consumer system, or readers in a reader/writer system. Another avenue for future research is to use compositional analysis, where system properties are derived or inferred from properties of individual objects in a SBOPN system.

## References

1. T. Murata, “Petri Nets: Properties, Analysis, and Applications, “*Proceedings of the IEEE*, April, 1989, pp. 541-580.
2. J. R. Burch, E. M. Clarke, K. McMillan, and D. Dill, “Symbolic Model Checking: $10^{20}$ States and Beyond,” *Informatics Computing*, Vol. 98, No. 2, June 1992, pp. 142-170.
3. G. Voss, *Object-Oriented Programming: An Introduction*, Osborne McGraw-Hill, 1991.
4. R. Bastide, “Approaches in Unifying Petri Nets and the Object-Oriented Approach,” *Proceedings of the 1st Workshop on Object-Oriented Programming and Models of Concurrency, June 1995.*
5. D. Buchs and N. Guelfi, “CO-OPN: A Concurrent Object-Oriented Petri Net Approach,” Proceedings of the *12th Int. Conf. on the Application and Theory of Petri Nets*, Denmark, in *Lecture Notes in Computer Science*, Springer-Verlag, 1991.
6. M. Baldassari and G. Bruno, “PROTOB: An Object-Oriented Methodology for Developing Discrete Event Dynamic Systems,” *Computer Languages*, Vol. 16, No. 1, Great Britain, 1991, pp. 39-63.
7. K. M. van Hee and P. A. C. Verkoulen, “Integration of a Data Model and High-Level Petri Nets,” *Proceedings of 12th Int. Conf. on the Application and Theory of Petri Nets*, Denmark, in *Lecture Notes in Computer Science*, Springer-Verlag, 1991.
8. C. A. Lakos, “Pragmatic Inheritance Issues for Object Petri Nets,” *Proceedings of TOOLS Pacific ‘95 Conference* (The 18th Technology of Object-Oriented

Languages and Systems Conference), C. Mingins, R. Duke, and B. Meyer (Eds), Prentice-Hall, 1995, pp. 309-322.

9. K. Jensen, “Coloured Petri Nets: A High Level Language for System Design and Analysis,” *Advances in Petri Nets 1990*, G. Rozenberg (Editor), in *Lecture Notes in Computer Science*, 483, Springer-Verlag, 1990.

10. J. L. Peterson, “A Note on Colored Petri Nets,” *Information Processing Letters*, Vol. 11, No. 1, Aug. 1980, pp. 40-43.

11. K. Jensen, “An Introduction to the Theoretical Aspects of Colored Petri Nets,” *Lecture Notes in Computer Science : A Decade of Concurrency*, Vol. 803, June 1993, pp. 230-272.

12. A. Valmari, “Stubborn Sets for Coloured Petri Nets,” *Proceedings of the 12th International Conference on Application and Theory of Petri Nets*, Denmark, *in Lecture Notes in Computer Science*, Springer-Verlag, 1991.

13. S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz, “Applications and Experimental Evaluation of State Space Reduction Methods for Deadlock Analysis in Ada,” *ACM Transactions on Software Engineering Methodology*, Vol. 3, No. 4, Oct. 1994, pp. 340-380.

14. A. Burns and A. Wellings, *Concurrency in Ada*, Cambridge Univ. Press, 1995.

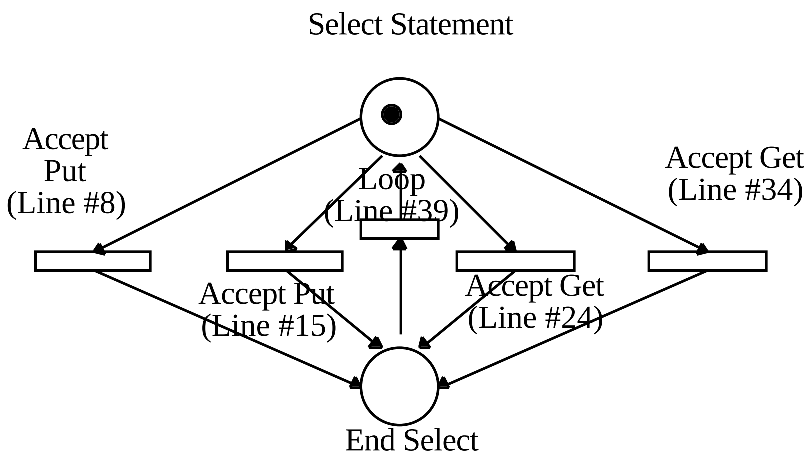


**Figure 1** A Petri net representing the code segment in Listing 1

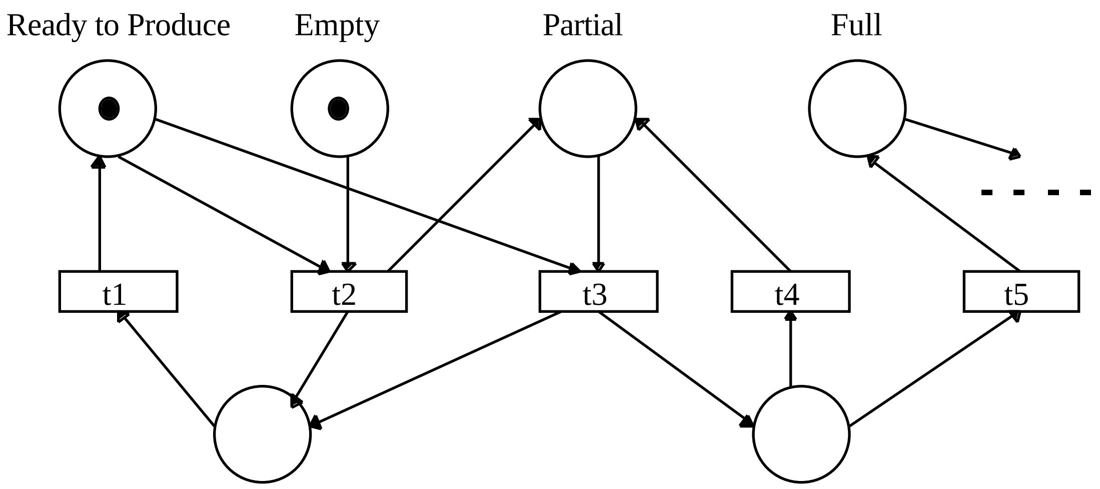


**Figure 2** Petri net segment representing producer/buffer interactions

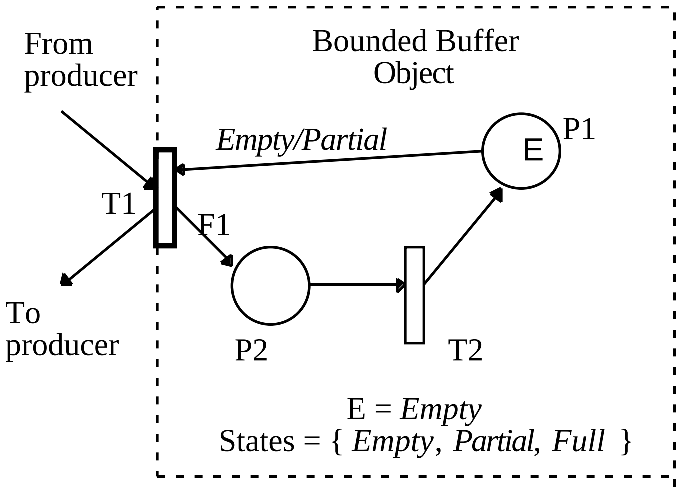


**Figure 3** A net representation of a buffer object that accepts *put* messages

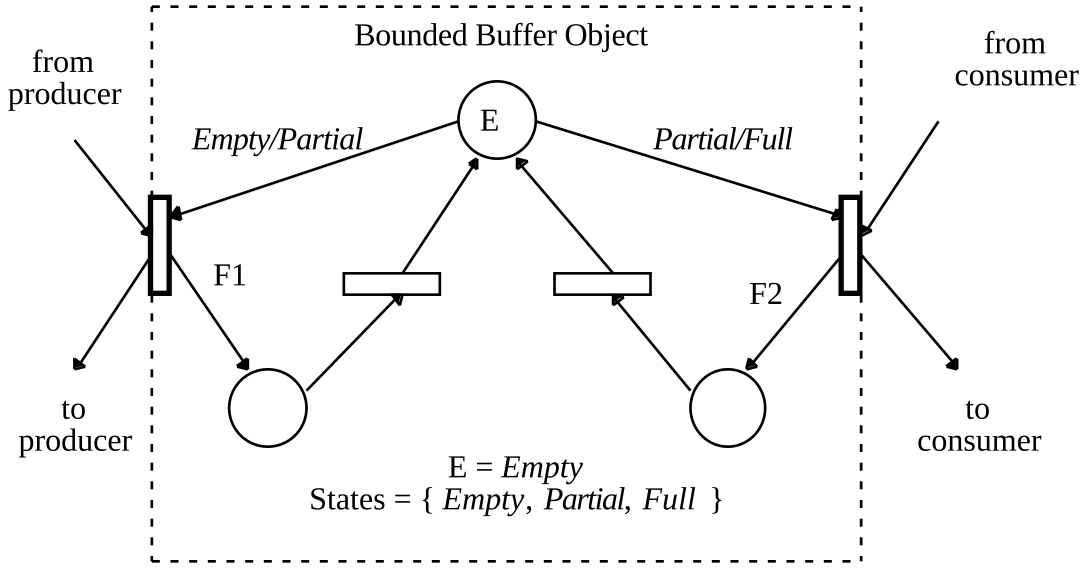


**Figure 4** A net representation of a buffer object that accepts *put* and *get* messages

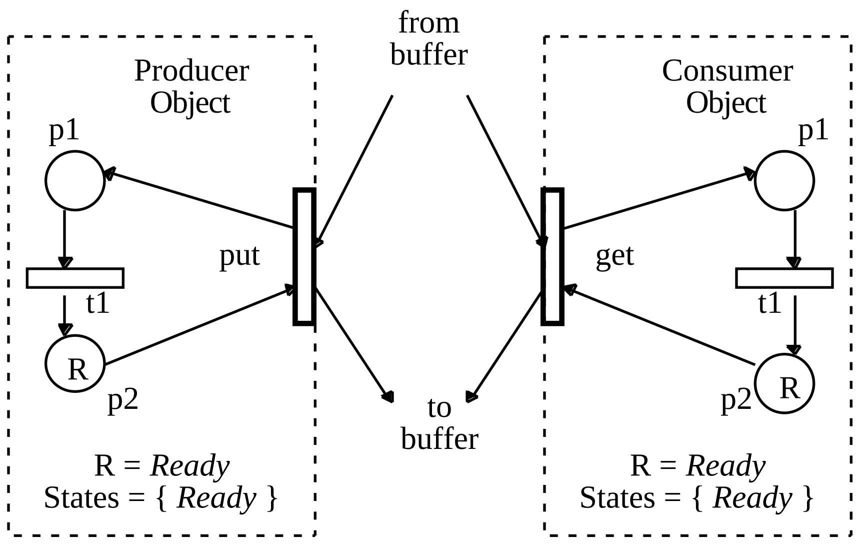


**Figure 5** A net representation of producer and consumer objects

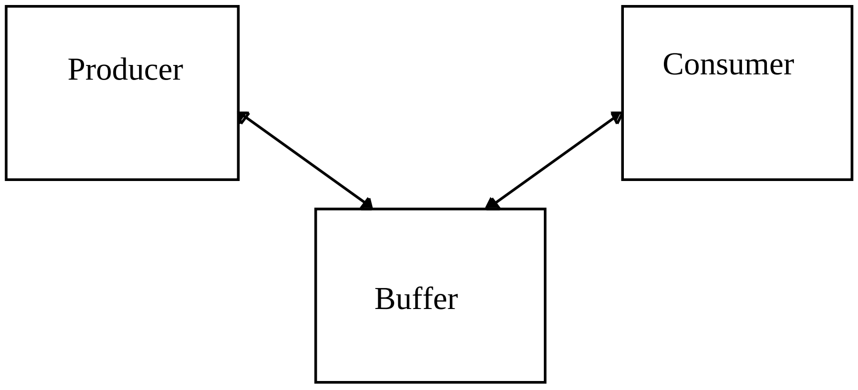


**Figure 6** An object interaction diagram for an object-based Petri net

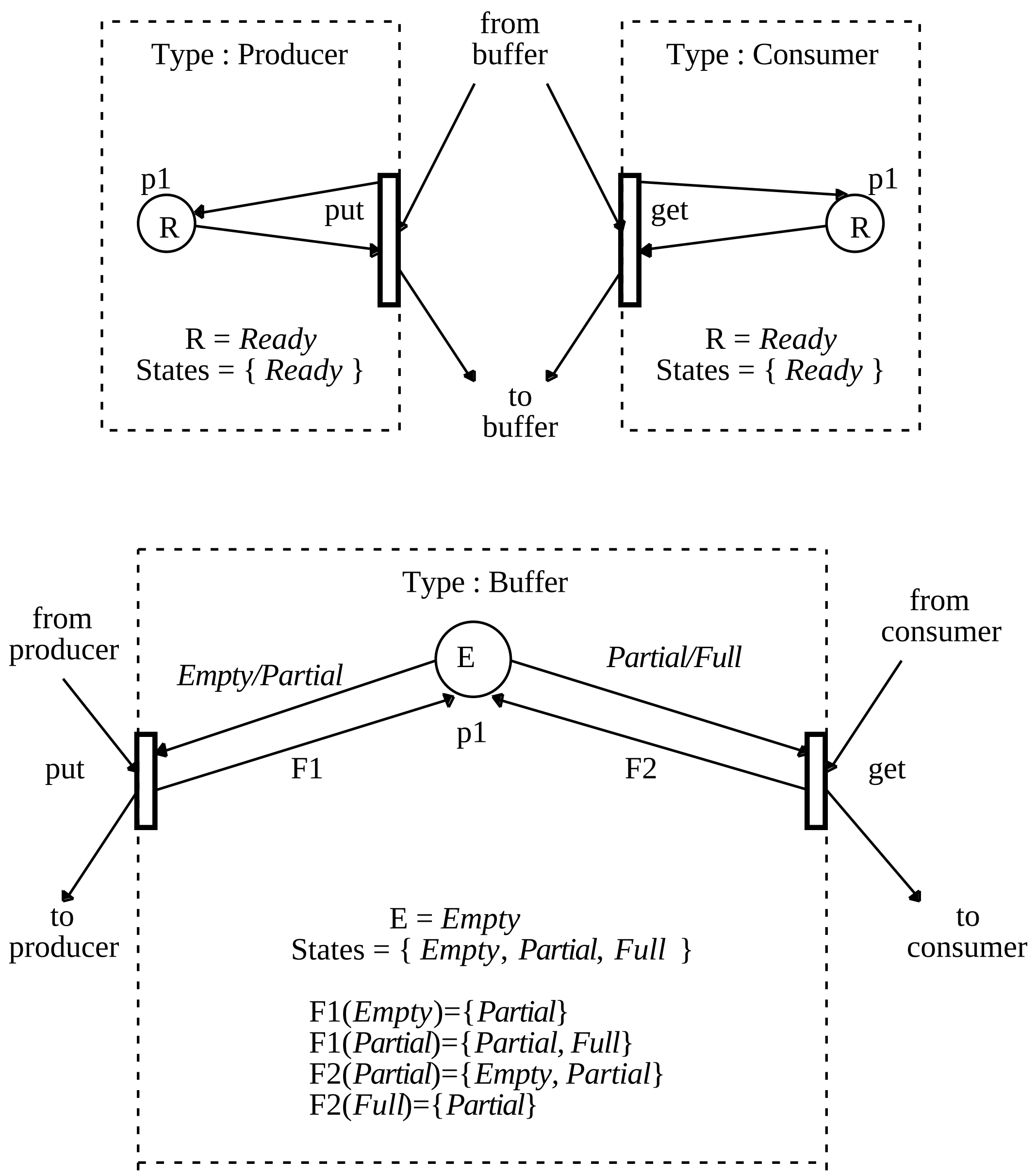
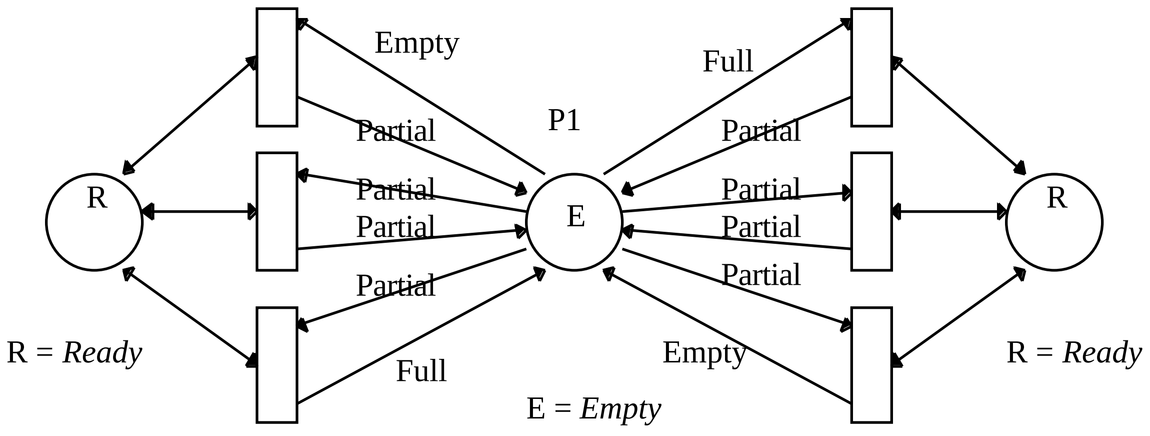


**Figure 7** A SBOPN for the producer, consumer, and buffer system



Color Set = { *Empty*, *Partial*, *Full*, *Ready* }

**Figure 8** An unfolding of the net model from Fig. 7

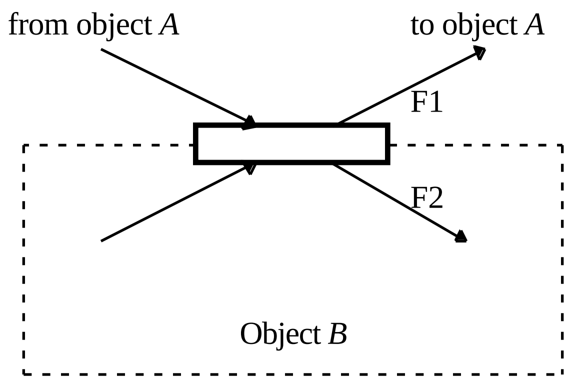


**Figure 9** A shared transition whose firing represents the acceptance by Object B of a message from Object *A* (F1 and F2 represent state-transition functions)

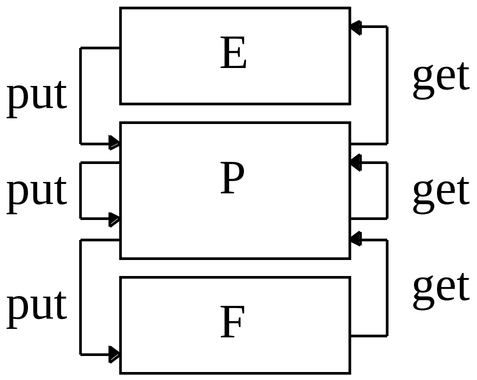


**Figure 10** A reachability graph for the SBOPN in Fig. 7