An Approach for Modeling Components with Customization for Distributed Software[1]

X. Xie and S. M. Shatz
Concurrent Software Systems Lab
University of Illinois at Chicago

Abstract

Component-based software development has many potential advantages, including shorter time to market and lower prices, making it an attractive approach to both customers and producers. However, component-based development is a new technology with many open issues to be resolved. One particular issue is the specification of components as reusable entities, especially for distributed object-oriented applications. Specification of such components by formal methods can pave the way for a more systematic approach for component-based software engineering, including design analysis and simulation. This paper discusses an approach for blending Petri net concepts and object-oriented features to develop a specification approach for distributed component software systems. In particular, a scheme for modeling behavior restriction in the design of object systems is presented. A key result of this work is the definition of a "plug-in" structure that can be used to create "subclass" object models, which correspond to customized components. Algorithms that support the automatic synthesis of these models are provided, discussed, and illustrated by examples.

1. Introduction and Motivation

There is significant interest in using components in software development. Specification and implementation of a system in terms of existing and/or derived components can dramatically decrease the time required for system development, increase the usability of resulting products, and lower production costs [1]. However, component-based development is still immature, with a lack of established procedures and support from formal modeling. Techniques and tools that are based on formal methods can pave the way for advanced software engineering capabilities such as design analysis and simulation.

Reuse principles have typically placed high demands on reusable components. Such components need to sufficiently general to cover the different aspects of their use, while also being simple enough to serve a particular requirement in an efficient way. This has resulted in a situation where developing a reusable component may require three to four times more resources than developing a component for particular use [2]. Thus, component vendors desire to make full use of these components in order to achieve reasonable profit levels. Such component use

[1] This material is based upon work supported by, or in part by, the U.S. Army Research Office under grant number DAAD19-99-1-0350 and by NSF under grant number CCR-9988168.

requires the customization of general components, a process that is aided by applying different constraints to functionality to support different price policies and different user groups.

In component-based development, the final products are not closed monolithic systems, but are instead component-based products. The developers are not only designers and programmers; they are integrators and marketing investigators. Often a component is not effectively reusable because its interface or part of the implementation does not match the specified requirements of a target application. To achieve the reuse, the component needs to be customized into another component that fulfills the requirement [3]. One purpose of the customization is to apply constraints in situations where the functionality of a "base component" is more general than is actually needed, or when some base-component features exhibit characteristics not suitable for a particular application – for example some functions (or methods) may not be fault tolerant or may be resource hogs. Thus, the component's behavior must be restricted before it can be reused in a new design.

One potentially efficient and natural technique to support constraints is a particular type of inheritance known as restriction inheritance [4]. Restriction inheritance defines a subclass that constrains the behavior of a superclass. This is in contrast to augment inheritance, where a subclass augments, or extends, a superclass. Since subclassing by restriction often conflicts with the semantics and intention of inheritance, where an instance of a subclass should be an instance of the superclass and should behave like one, some researchers have suggested that restriction inheritance be avoided [1][5]. But, in our own experience, which does involve development of commercial component-based software, we have observed benefits of restriction inheritance for customizing components. First, most commercial component-based software is based on middleware technologies such as CORBA [6] and/or COM [7]. As a result, these systems mostly consist of classes. In COM, even the interface of a component is a class. So, it seems natural to use inheritance techniques (defined for class-based systems) to handle constraints. Also, restriction inheritance is efficient, simple and straightforward. Finally, since restriction inheritance is being used for the purpose of defining a wrapper for components, the original components and/or class is not intended to be used directly, which limits any potential disadvantage associated with the use of restriction inheritance.

To develop a systematic design process with the capability for automated simulation and analysis, it is valuable to define a design method's syntax and semantics in terms of some formal notation and method. This is also true for restriction inheritance. For engineering of distributed object systems, it is desirable for the formalization to provide a simple and direct way to describe component relationships and capture essential properties like nondeterminism, synchronization and concurrency. Petri nets [8] are one formal modeling notation that is in many ways well matched for general concurrent systems. In particular, the standard graphical interpretation of Petri net models is appealing as a basis for a design notation. But, standard Petri nets do not provide direct support for high-level design and object oriented features. This has motivated some recent research into methods for combining Petri net modeling and object-oriented design. In general, the proposed methods use enhanced forms of Petri nets as a

base of the combination, and pursue two main approaches [9]. One is the "objects inside Petri nets" approach, in which the semantics of tokens in Petri nets are expanded to include other information, which could include object definitions (e.g., [10]). The other approach is the "Petri nets inside objects" approach, in which traditional Petri net constructs are used to model the internal semantics of object (e.g., [11]).

In this paper we introduce a model called a State-Based Object Petri Net (SBOPN), which is developed from the basic idea introduced in [12]. An example of using SBOPN concepts in the domain of aspect orientation is described in [13]. In this paper we extend the basic SBOPN model to directly support restriction inheritance modeling for the purposes discussed earlier. SBOPN is most similar in spirit to Lakos' Language for Object Oriented Petri Nets, LOOPN [10]. LOOPN's semantics are richer, but SBOPN provides a more specific, and thus more intuitive, notation for capturing the behavior of distributed state-based objects. Like LOOPN, SBOPN is based on a generalized form of Petri net called colored Petri nets [14]. One other difference between LOOPN and SBOPN is that 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. Another language, namely CO-OPN/2 [15], is also a "Petri nets inside objects." CO-OPN/2 uses high-level Petri nets that include data structures expressed as algebraic abstract data types and a synchronization mechanism for building abstraction hierarchies to describe the concurrency aspects of a system. CO-OPN/2 is a general model that focuses on concurrency. SBOPN focuses more on the architectural modeling of state-based systems; thus it is simpler and more domain-specific.

The structure of this paper is as follows. Section 2 presents an example using restriction inheritance and informally introduces the SBOPN model**.** Section 3 provides details on SBOPN modeling and discusses the restriction subclasses and SBOPN control places. Section 4 describes our approach for synthesis of subclass models that capture instances of restriction inheritance. The approach is characterized by the use of special net structures called "plug-in structures." Section 5 uses an e-commerce case study to illustrate the application of the SBOPN model. Finally, Section 6 provides a conclusion and mentions some future work.

2. An Example and Introduction to SBOPN Modeling

Consider the classic example of a system that uses a bounded buffer to temporarily hold items, such as messages. In this version we allow an operator to enable and disable the buffer, in addition to the standard producer and consumer components. The four system components – buffer, producer, consumer and operator – operate asynchronously and only interact via messages initiated by the producer (*put* message), consumer (*get* message) or operator (*enable* and *disable* message). 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. Also, the operator can send *enable* or *disable* message to enable or disable the buffer. At any point in time, the buffer should be in one of four states: *Empty*, *Full, Partial*

(means *Partially Full*) or *Disabled*. Depending on its state, the buffer may or may not be able to accept the messages *put*, *get*, *disable* and *enable*. When the buffer is in *Empty* or *Partial* state, it can accept the *put* message and change to *Partial* or *Full* state. When it is in *Partial* or *Full* state, it can accept the *get* message and change to *Empty* or *Partial* state. When it is in any state except the *Disabled* state, it can accept the *disable* message and change to the *Disabled* state. Finally, when it is in the *Disabled* state, it can accept the *enable* message and change to its previous state (before it was disabled): *Empty*, *Partial* or *Full*. To simplify the example, we simply assume that after accepting a *disable* message, the buffer is reset to *Empty* state.

To model state-based systems, such as this buffer system, we use State-Based Object Petri Nets (SBOPN) [12]. This can be viewed as a special purpose form of (Colored) Petri net. Lack of space prevents us from giving an overview of Petri nets here; we refer the reader to a reference like [8] for such information. Figure 1 shows a simple SBOPN model of the system we have described above. Notice that there are separate models for the buffer, producer, consumer and operator. These component models are called State-Based Petri Net Objects (SBPNO) and the methods of objects are represented by *shared transitions*. For example, the *put* method is represented by a shared transition used by the buffer object and the producer object. The *system model* is called a State-Based Object Petri Net (SBOPN). To informally highlight some key features of the SBOPN model, let us consider the buffer object. There is an arc from the place p_1 to the shared transition *put*. The token labeled *D* in p_1 is called a *state token*, and *D* is the current *state-value* of this state token. This represents that the current state of the buffer is *Disabled*. The label *{Empty, Partial}* for the arc (p_1, put) shows that the *put* transition has the potential to fire only when the buffer is in the *Empty* or *Partial* state. This arc label is called a *state filter*. When all the input places of a transition satisfy the corresponding state filter, that transition is enabled. The arc from the transition *put* to the place p_1 is also labeled. This arc label $(p_1, F1)$ is called a *state-transfer tuple*, where p_1 is called a *state-transfer place* and *F1* is called a *state-transfer function*. This tuple determines the possible state(s) the buffer can be in after the *put* method is processed. The input value of a state-transfer function is the state-value of the state token consumed from the associated state-transfer place. In this simple example, the buffer can have the following changes due to the *put* method: from *Empty* to *Partial*, from *Partial* to *Partial*, or from *Partial* to *Full*. The state-transfer function *F4* indicates that a call to the *disable* method results in the buffer transitioning to the *Disabled* state, regardless of the state-value of the token consumed from place p_1.

Now, consider a need to customize this general buffer component for use in a more restricted application. First, assume the new buffer component should not allow the disable operation – the buffer cannot be "turned off." Second, to ensure tighter synchronization on producer and consumer components, the new buffer component should behave as a simple capacity-1 buffer. Thus, only when the buffer is in the *Empty* state, instead of both *Empty* and *Partial* states, should it accept a *put* message. We call this new buffer a "disable-free synchronous buffer." To model a new system that uses a disable-free synchronous buffer, we could just redesign the system model in Figure 1 to create a new model. The obvious way to do this is to remove the *disable* transition from the operator object model and from the buffer object model (along with the incident arcs), and change the state filter for the arc connecting

place p_1 to the *put* transition in the buffer model. However, there two important disadvantages inherent in performing such a redesign of the model of this new buffer component:

1. In creating the new buffer, we have changed the interface of another class, the operator. This conflicts with the basic modularity principle of object-design, i.e., the internal change of one class should not affect other classes. This is an important issue, especially when it comes to consideration of model synthesis and reuse.

2. To model the fact that the new buffer class can accept a *put* message only when it is in the *Empty* state, the redesign mentioned before would require a change in the state filter for arc (p_1, put) in the general buffer class – from *{Empty, Partial}* to *{Empty}*. But, such a change now makes it difficult to directly identify that the new object is one of many possible behaviorally restricted objects derived from a common object – borrowing from object terminology, we can think of these restricted objects components as representing subclass objects of a superclass object. We will revisit this issue in Section 3.

The first disadvantage, as we discussed, is due to the removal of a shared transition. Fortunately, SBOPNs provides an alternative way to achieve this "removal" of behavior without changing the interface of an object. For our example, this can be done by changing the state-filter for the arc $(p_1, disable)$ in the buffer object from *{Full, Partial, Empty}* to ϕ, the empty set. Since no buffer state can be an element of ϕ, the *disable* method is now unsupported (i.e., the transition representing this method is unable to ever fire in this new subclass model). It can be observed that this use of ϕ as a state filter is a special case of the technique mentioned earlier for modifying the behavior of the *put* method.

To overcome the second disadvantages is not so easy and straightforward. We propose to model restriction inheritance by the simple addition of a "plug-in" structure to a superclass model. In other words, we want to limit the behavior of the superclass object by adding some control structure to the superclass model. Actually, this is very natural from the view of control theory since control systems limit the behavior of a system by adding some controller logic. For example, [16] describes a method for constructing a Petri net controller for a discrete event system modeled by a Petri net.

3. Subclass Component Models and Control Places

In Section 2 we informally introduced the SBOPN modeling notation via an example. Now we can formally define this notation and discuss how to derive design models for subclass components.

Definition 1 (SBPNO): A *State-Based Petri Net Object* is a 7-tuple, SBPNO = *(Type, NG, States, sp, ST, SFM, STM)*, where

- *Type* is an identifier for the object's type (or class).
- *NG = (P, T, A)* is a net graph, where
  I. *P* is a finite set of nodes, called Places.
  II. *T* is a finite set of nodes, called Transitions, disjoint from *P*, i.e., $P \cap T = \phi$.
  III. $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, known as the flow relation.
- *States* is a finite set of distinct states that define the possible states of the SBPNO. A token (as in standard, or colored Petri nets) may have associated with it a state-value, which is one of the elements of *States***.**
- $sp \in P$ is called a state place. The value associated with the token in this place indicates the current state of the SBPNO**.**
- $ST \subseteq T$ is a set of shared transitions. A shared transition in a SBPNO is a transition that is shared with other SBPNOs. Shared transitions model the acceptance of a message from other SBPNOs or the sending of a message to other SBPNOs.
- $SFM: (A \cap (P \times T)) \rightarrow 2^{States}$ is a state-filter mapping, where 2^{States} is the power set of *States*. This mapping maps each place-to-transition arc to a state filter. The basic purpose of the state filter mapping is to ensure that only those tokens that have a state-value representing one of the states in the state filter can pass (i.e., be consumed by a transition) via the corresponding arc**.**
- $STM: (A \cap (T \times P)) \rightarrow P \times STF$ is a state-transfer mapping, where *STF* is the set of state-transfer functions, $STF = \{stf \mid stf: States \rightarrow 2^{States}\}$. This mapping maps each transition-to-place arc *(t, p)* to a state-transfer tuple *(p', stf)*, where $p' \in \{p \mid (p, t) \in A\}$ is called the state-transfer place and *stf* is called the state-transfer function. The basic purpose of the state-transfer mapping is to allow the firing of transition *t* to map the state-value of the token consumed from place *p'* into a set of states, which represents the possible state-values that can be associated with the token deposited into the output place via the corresponding arc**.**

As we saw in Section 2, a SBPNO is denoted graphically as a Petri net (a subnet) inside a box and a State-Based Object Petri Net, SBOPN, is a Petri net consisting of connected SBPNOs, which are components of the system being considered. A marking of a SBOPN is the distribution of state tokens to the SBPNO components, and an SBOPN system (N, M_0) is an SBOPN, *N*, along with an initial marking M_0 (the initial states of the objects). In a SBOPN system, a transition *t* is said to be *enabled* if and only if, for each $p \in {}^\bullet t$ (where ${}^\bullet t$ is the set of input places for transition *t*), *p* contains a token whose state value is an element of the state-filter for arc *(p, t)*. When an enabled transition fires, it removes from each input place a token whose state value satisfies the corresponding state filter, and then deposits a token to each output place. The state value assigned to a deposited token is one of the elements given as an output of the corresponding state-transfer function. For example, assume an arc (*t, p*) with the state-transfer tuple *(q, f)*, where the state-transfer function $f(x) = \{x\}$. Then the firing of transition *t* will deposit a token into place *p* and the state-value of this token will be equal to the state-value of the token removed from place *q*.

To simplify SBPNO models, implicit state filters and implicit state-transfer tuples are allowed, i.e., definitions are assumed if they are not explicitly specified. For an implicit state filter, the state-filter is *States*. Note that in Figure 1, the state filters are implicit in the producer, consumer, and operator objects. An implicit state-transfer tuple can be used only when the output place associated with the arc is an input place of the transition associated with the arc – the arc is part of a self-loop. The state-transfer place is the place in the self-loop. We also require an implicit state-transfer function's output to be the state-value of the token removed from the place in the self-loop. Due to the simplicity of the producer, consumer, and operator object models, the state-transfer tuples are also implicit.

Definition 2 (firing sequence): Let $N = (Type, NG, States, V, ST, SFM, STM)$ be a SBOPN, $t_i \in ST\ (1 \le i \le n)$, and let M_i be a marking $(1 \le i \le n+1)$. If the marking M_{i+1} is reached from marking M_i by firing t_i, then we call $\sigma = t_1 t_2 \ldots t_n$ *a firing sequence* of the SBOPN system (N, M_1).

Now we can identify properties of a restriction subclass and present the definition of a restriction subclass model. First, the methods of a restriction subclass object should be a subset of the methods of the superclass object. Second, the externally observable behavior of a restriction subclass object should be observable in the behavior of the superclass object. In other words, any firing sequence of a SBPNO subclass model should be a firing sequence of the superclass model when we only consider the shared transitions. In the following definition we use the notation $\sigma|_T$, a projection of σ onto T. As an example of this projection, let $\sigma = t_1 t_2 t_1 t_3 t_2$, and $T = \{t_1, t_3\}$, then $\sigma|_T = t_1 t_1 t_3$.

Definition 3 (Restriction Subclass Model): Let $N_1 = (Type_1, NG_1, States_1, ST_1, SFM_1, STM_1)$, $N_2 = (Type_2, NG_2, States_2, ST_2, SFM_2, STM_2)$ be two SBPNOs, then N_2 is *a restriction subclass model* of N_1 if and only if:

1) $ST_2 \subseteq ST_1$
2) For any marking M_2 of N_2, there exists a marking M_1 of N_1, such that for any firing sequence σ_2 of (N_2, M_2), there exists a firing sequence σ_1 of (N_1, M_1), which satisfies $\sigma_1|_{ST1} = \sigma_2|_{ST2}$.

A particular restriction subclass model must be defined in terms of some particular superclass model and some specific method restrictions. These restrictions are captured by a restriction function, as defined next.

Definition 4 (Restriction Function): Let $N_1 = (Type_1, NG_1, States_1, IS_1, Stoken_1, ST_1, SFM_1, STM_1)$ be a SBOPN, and let function f: $SF_1 \rightarrow 2^{States1}$, where SF_1 is the domain of SFM_1, and $2^{States1}$ is the power set of $States_1$. The function f is called a *restriction function* for N_1 if and only if f satisfies: $\forall sf_1 \in SF_1, f(sf_1) \subseteq sf_1$.

Applying f to the state filters of N_1 creates a new model, which we denote as $N_1|f$. It can be shown that $N_1|f$ is a restriction subclass model of N_1, but note that $N_1|f$ features the two disadvantages discussed earlier in Section 2.

Our goal is to create a "plug-in" structure that can be added to a superclass model causing it to have the same behavior as $N_1|f$ but avoiding these disadvantages. Such a plug-in structure must be able to control the firing of some shared transition *t*. This is accomplished by using a so-called "control place" as the heart of the plug-in structure. The control place must ensure that the state-value of a token in the control place "tracks" the state-value of a token in one of the input places *p* to the transition *t*. We call such a place *p* the "controlled place."

Definition 5 (Control Place): Let $N = (Type, NG, States, ST, SFM, STM)$ be a SBPNO, and p_1 and p_2 be two places of *N*. We say that p_2 is a control place for p_1 (p_1 is a controlled place) if and only if:

1) $(ST \cap p_2^{\bullet} \neq \varnothing) \wedge (ST \cap p_2^{\bullet} \subseteq ST \cap p_1^{\bullet})$ (Note: $p_1^{\bullet}$ is the set of output transitions of the place p_1).
2) For any shared transition $t \in (ST \cap p_2^{\bullet})$, the associated state filter for the arc (p_2, t) is a subset of the state filter for the corresponding arc (p_1, t).
3) For any reachable marking *M'* from *M*, which satisfies $M(p_1) = M(p2)$, and any transition $t \in (ST \cap p_2^{\bullet})$, if *t* fires under *M'*, then the tokens consumed by *t* from p_1 and p_2 should have the same state values.

4. Synthesis of Plug-In Structures for Modeling Customized Components

4.1 Basic Plug-in Design

A straightforward way to implement a control place is to create a duplicate place. The basic idea has two steps. First, we duplicate the controlled place, such that the new place has exactly the same input and output characteristics as the controlled place. Obviously, any change in the marking of the controlled place is simultaneously reflected in the marking of the new duplicated place. Because the new SBPNO (created by the duplication process) has the same exact behavior as the original SBPNO, the new SBPNO serves as a (trivial) restriction subclass. In the second step, we modify the state filters for the arcs from the new place to all shared transitions such that they satisfy the specific requirement of the particular desired restriction subclass. This creates a model for a customized component. Recall that the specific restriction requirement (i.e., the customization feature) is determined by a restriction function, as defined in Definition 4.

Although a duplicating place can be used to create a control place and thus build a restriction subclass without changing interfaces, there is one significant disadvantage: *redundancy*. For example, in creating the "disable-free synchronous" buffer model from Section 2, we do not want to change the firing conditions of the *enable* and *get* methods. But it is necessary for the control place to connect with the associated shared transitions. Also, these additional arcs must carry the same state-filters and state-transfer functions as in the superclass model. Such extra arcs, which do not change the behavior of the methods, imply an existence of redundancy in the new model.

Since our goal is to ensure that the state-marking of a control place "tracks" that of the controlled place, we can copy the token of a controlled place into the control place, but we must be sure that this copying occurs before

allowing these places to enable any shared transition. We call this type of control place a "refreshing place" since it gets refreshed (i.e., the state-value of its current state token is updated) each time the state-value of the token in the corresponding controlled place changes. Figures 2, 3 and 4 illustrate this idea by a simple example. In Figure 2, we have a SBPNO for a component *C1*. Now suppose we want to model a restriction subclass *C2* that has the property that t_1 can be enabled only when the object is in the state *a* – instead of either state *a* or *b*, as in the component *C1*. We need t_2 to remain enabled in the *a* state.

To model this subclass, we create a new place p_2 (see Figure 3) as a control place candidate. Transition t_3 is introduced for the purpose of copying the state token from p_1 to p_2. As in the duplicating place technique, the state filter associated with p_2's connection to t_1 is *{a}*. However, under the general firing rule that controls the behavior of a SBPNO, we cannot guarantee that the tokens in p_1 and p_2 are of the same value when t_1 is enabled. For example, in Figure 2, suppose p_1 has initial state *a*, then the firing sequence is $t_1^* t_2 t_1^*$. Now consider Figure 3, where both p_1 and p_2 have initial state *a*. Once t_2 fires, p_1 has state *b*, while p_2 still has state *a*. If t_3 does not yet fire, p_1 and p_2 have different states, but t_1 is still enabled. As a result, we could get the same firing sequence as *C1*, $t_1^* t_2 t_1^*$. However, *C2* is supposed to only allow the restricted firing sequence $t_1^* t_2$, where we ignore the internal transition t_3 in the firing sequence. So the construction in Figure 3 does not yet provide for a proper modeling of the control place.

The problem is that when t_2 fires, the token in p_2 remains unchanged and thus is not "tracking" the marking of p_1. To solve this problem, we need to force t_3 to fire immediately after t_2 fires, i.e., to refresh p_2 immediately. This is accomplished by using a special form of Petri net arc called an activator arc [17]. An activator arc can be used to connect a place to a transition. For nets with activator arcs, the transition firing rules are as follows: 1) Those enabled transitions with activator arcs have the highest priority, and 2) A transition that has activator arc input(s) cannot fire twice in succession for the same input marking, i.e., the net's marking must be modified in some manner before the transition can fire again. For example, in Figure 4, t_1, t_2 and t_3 are enabled, but t_3 has an activator arc (denoted by the arc with a solid bubble), so it fires first. After firing t_3, we get the same marking, so t_3 cannot fire again. As a result, only t_1 or t_2 can next fire. Now, if t_1 fires, because the marking remains unchanged, we have the same situation as before t_1 fires. But if t_2 fires, both t_1 and t_3 are enabled. Since the marking has changed, only t_3 can fire, which copies the token *b* from p_1 to p_2, i.e., p_2 is refreshed. This copying of the state-value from *p1* to *p2* is due to the state-transfer function *F3*. Note that t_1 is not enabled any more after t_3 fires. As we can see, now p_2 serves as a proper control place to ensure we have only one firing sequence $t_1^* t_2$ (again, ignoring the internal transition t_3 in the firing sequence).

We now present two algorithms for synthesis of restriction subclass models using plug-in structures. The first algorithm is used to create a refreshing place. Its purpose is to support the second, more important, algorithm, which synthesizes a restriction subclass model.

Algorithm 1: Create a refreshing place in a SBPNO.

Input: A SBPNO $N = (Type, NG, States, ST, SFM, STM)$, and a place p_1 that satisfies $p_1^\bullet \in ST$.

Output: A new SBPNO (a modified version of N) with a refreshing place p_2 for p_1.

Procedure:

1) Add to N_1 a place p_2 and a transition t'.
2) Add an arc r_1 from p_1 to t', and an arc r_2 from t' to p_1.
3) Add an arc r_3 from t' to p_2, and an arc r_4 from p_2 to t'. Use (p_1, F) as the state-transfer tuple for r_3, where F is defined as $F(x) = \{x\}, x \in States$.
4) Add an activator arc from p_1 to t'.

As an example, applying Algorithm 1 to the SBPNO in Figure 2 creates part of the SBPNO shown in Figure 4 – all of the model except the arcs $(p2, t1)$, $(t1, p2)$ and the state-filter $\{a\}$ for the arc $(p2, t1)$.

Algorithm 2: Model a restriction subclass by use of plug-in structures

Input: 1) A SBPNO $N_1 = (Type_1, NG_1, States_1, ST_1, SFM_1, STM_1)$.

2) A restriction function (see Definition 4), $f: SF_1 \rightarrow 2^{States1}$

Output: A restriction subclass model N_2 of N_1 (N_2 has the same externally observable behavior as the model $N_1|f$ identified in Section 3).

Procedure:

1) Make a copy N_1. Call this new model N_2 and let N_2 be the source net for the following step:
2) For each transition t in ST_1:

   For each $p_1 \in t^{\bullet}$, let $S1$ be the state filter for the arc (p_1, t). If $S2 = f(S1)$ is a proper subset of $S1$, i.e., $S2 \neq S1$, then create a control place p_2 of p_1 by applying the following steps:

   A. Use Algorithm 1 to create a refreshing place p_2 of p_1.
   B. Add an arc r_1 from p_2 to t. Use $S2$ as the state filter for r_1.
   C. Add an arc r_2 from t to p_2

   End For

   End For

The initial marking of a subclass model created by Algorithm 2 is determined by the initial marking of the superclass used to create it. All places except the created control places have the same initial marking as in the superclass model. The control places take on the same initial marking as their corresponding controlled places. As an example, applying Algorithm 2 to the SBPNO in Figure 2 creates the SBPNO shown in Figure 4. In this case, N_1 is the model shown in Figure 2 and the restriction function f is defined as $f(\{a, b\}) = \{a\}, f(\{a\}) = \{a\}$. Note that the structure within the dashed box in Figure 4 is the plug-in structure. As we can see, Figure 4 is more complex than Figure 2. And the switchable plug-in structures introduced in next subsection are even more complicated. Our proposal of modeling restriction inheritance would not be practical if we have to manually handle this complexity

introduced by plug-in structure. Fortunately, since the synthesis of restriction subclass models is based on an algorithmic process, automated tools can be used to hide the internal details of modeling and analysis.

4.2 Switchable Plug-in Structures

One advantage of Algorithm 2 is that the plug-in structures created are potentially controllable. By controllability we mean that a switch can be added to the structure to control its activity, i.e., the switch can be used to "turn on" or "turn off" the functionality of the plug-in structure. We call such a plug-in a "switchable plug-in." Switchable plug-ins offer a key advantage: They allow an model to represent a family of restriction subclass models, corresponding to a family of components. The basic idea is that a single component-model with n potential customizations (defined by n plug-in structures) can in fact model a family of 2^n customized components. The family members correspond to the various combinations of enabled customization features. This technique will be discussed shortly by a specific example.

To transform a plug-in structure into a switchable plug-in, a new place node must be added. For example, Figure 5 shows the same model as Figure 4, but with a switchable plug-in. Place p_3 serves as this new switch place. When there is a token in the switch place p_3, the "plug-in" structure is active. In this case, the plug-in behaves as before we introduced the switch place, i.e., like Figure 4. But when there is no token in p_3, the transition t_3 will never be enabled. So, in this case, the model behaves as before we introduced the plug-in, i.e., like Figure 2. Notice that we have introduced a new state value called *internal* to the state set. Although it is possible to create the switching capability for this particular example without introducing this new *internal* state, use of this special state is required for creating general-purpose switchable plug-ins. To explain this point, consider the following situation.

Suppose that we wanted to create a subclass *C3* of class *C1*, where *C3* does not support method *t1* at all. In this case, by Algorithm 2, the SBOPN for class *C3* would look like the model in Figure 4, except that the state filter for the arc (p_2, t_1) would be ϕ instead of *{a}*. Now, to make the plug-in of this model switchable, we would introduce a switch place *p3* as was done in Figure 5. But, since the state filter is the empty set, there is no way for the switch place to enable transition *t1* – it is always disabled, regardless of the state value of the token we put in *p2*. So, it is clear that in a *switchable* plug-in we cannot allow ϕ as the state filter for a restricted transition. A simple solution is to introduce a new state value that is reserved for use within the switchable plug-in structure. This is the *internal* state referred to earlier. Now, the state filter can become *{internal}*, as opposed to ϕ. To create the initial marking of this subclass *C3*, it is necessary that the initial markings of the control place *p2* and the switch place *p3* have the state-value *internal*. In general, to model a restriction subclass using switchable plug-ins, we can use Algorithm 2 with the following two simple modifications:

1. For each plug-in, create a switch place (connected to/from the transition for the refreshing place).
2. For each plug-in, modify the state filter (for the arc from the control place to the restricted transition) to include the state *internal*.

As an example, let us revisit the buffer example from Section 2. Now, the modified algorithm mentioned above can be applied to the model in Figure 1 to create a model for a "disable-free synchronous" buffer. The resulting model (with two switchable plug-ins) is shown in Figure 6. Note that the initial marking of all places belonging to plug-ins are *internal*. Note that the plug-in associated with the disable method employs a state filter of {internal}. Thus, if this plug-in is "turned-on" (by marking the switch place), the disable method will become inactive. For the plug-in associated with the put method, the state filter is set to {i,Empty}. Thus, the put method is active only when the buffer is in the empty state. Most importantly, note that this one subclass model actually models a family of buffer types. The binding of the model to a specific buffer behavior is accomplished by varying the initial markings of the switch places (*p2'* and *p3'*). The following table defines the options:

p2'	*p3'*	Model
Marked	Marked	A "disable-free synchronous" buffer
Marked	Unmarked	A "disable-free" buffer
Unmarked	Marked	A "synchronous" buffer
Unmarked	Unmarked	A general buffer

The ability to model a family of components can be very helpful for commercial component-based development. It supports flexible analysis of varying configurations of customized components in the design phase, which can reduce the overall cost of development. This has the potential to aid configuration management and support, which is becoming a major challenge that organizations face in component-based software development [18].

4.3 Some Analysis Issues

Basic SBOPN models (without plug-in structures) are derived from standard colored Petri nets. Basic SBOPN models, with state filters and state-transfer functions, can be transformed into colored Petri nets [12]. This is important since we want SBOPN models to be able to use a full set of analysis techniques already existing for mature models like colored Petri nets or ordinary Petri nets. But, the subclass models that correspond to customized components in this paper use activator arcs. Thus, we must understand the impact of these arcs in terms of analysis potential. After all, activator arcs are special arcs with unique semantics. In the generally case, there is no equivalent ordinary Petri net structure for a Petri net with activator arcs. But, in our models, activator arcs are used only in plug-in structures. Thus, it is possible to convert an SBOPN model with activator arcs to a general SBOPN model and preserve liveness, safeness and boundedness of the model. To simplify our discussion, we use Figure 5 as an example to explain some key aspects of this translation. The results apply in general.

Consider the switch place *p3* in Figure 5. In the case that *p3* is not marked, it can be observed that removal of the plug-in will not change the liveness, safeness and boundedness properties of the model. Now consider the case when *p3* is marked. In this case, *p3* can never disable *t3*. Thus, *p3* and the corresponding arcs can be removed without changing the model's behavior. From the structure of the plug-in, it is clear that the plug-in will not affect the safeness or boundedness of the model. A similar analysis of *p2*'s impact on the liveness of the model confirms that that both state-filters (on the arcs *(p1, t1)* and *(p2, t1)*) can be changed to *{a}* without changing the liveness property of *t1*. Now, since both state-filters associated with *t1* are equal, and whenever *t1* fires, the tokens in *p1* and *p2* have identical state-values, the plug-in structure can be removed without impacting the liveness of *t1*. Furthermore, because of the 1-to-1 correspondence between a plug-in and a shared transition, the translation just described does not impact the liveness of transition *t2*. Further conversion of an object model to a colored Petri net or ordinary Petri net is now assured, providing a basis for various analysis capabilities. Further discussion on specific analysis techniques using these lower-level, basic net models is beyond the scope of this paper.

5. A Case Study

In the recent years, business via the Internet (e-commerce) has become more and more important in industry. Since the Internet is a global-scale distributed system, e-commerce systems face issues such as non-determinism, synchronization, and parallelism. The inherent complexity of such systems requires architects, designers, and developers to use techniques and tools with formal methods characterized by a sound mathematical basis. Our SBOPN is one such technique. To demonstrate the usage of SBOPN in the e-commerce domain, the case study presented in this section will focus on design of e-commerce systems. The first part of the case illustrates how to use SBOPN to model a web-shopping system. The second part of the case features one of the benefits of SBOPN: reusability. In the example, some classes are reused directly in a new system design while some are reused by inheritance, and some are totally rebuilt.

5.1 Basic Model

A web-shopping system is a classic business-to-customer example in e-commerce. In this system, customers use the Internet to do shopping in a virtual store, instead of a physical store. Typically, a customer uses a web browser, such as Netscape or Internet Explorer, to connect to a web site of a company doing e-business, such as Amazon.com. Then he registers to the shopping system, orders goods, and possibly inputs promotion information. Finally, the customer pays online, and finishes the shopping session. In our example, we assume that there are two different types of customers, as found in many shopping stores: regular-customers, and member-customers. Member-customers are automatically granted some special discounts that are not available for regular-customers. We want to model the system using the SBOPN notations.

First, we determine the classes of this system. Here we have four different classes: "user interface" to control customers' input and output logic, "register" to hold customers' registration information, "order system" to

take orders and send out invoices, and "cashier" to receive invoices and charge customers. Second, we need to determine the states of each class, the messages each class can accept, and the pre- and post-states associated with each message. For the "user interface" class, the simplified procedure is: login system, begin order, input promotion information if necessary, pay order, end order, and logout. So it has states: *Unauthorized*, *Authorized*, *Ordering*, *WaitDone*. *Unauthorized* means the customer has not logged-in yet. *Authorized* means the customer has successfully logged-in. *Ordering* means the customer is in the process of ordering items. *WaitDone* means the customer is waiting for the shopping section to finish. Note that the customer maybe input promotion information during the order process, which is represented by the *Promote* message. Although this action does not change the state of the "user interface" class, it does affect the state of the "order system" class. Table 1 shows the pre- and post-states of each message.

Starting State	Message	Ending State
Unauthorized	*Login*	*Authorized*
Authorized	*Logout*	*Unauthorized*
Authorized	*Begin Order*	*Ordering*
Ordering	*Promote*	*Ordering*
Ordering	*Charge*	*WaitDone*
WaitDone	*End Order*	*Authorized*

Table 1. The State Changes of a "User Interface" Component

The "Register" class can accept method calls for *Login*, *Logout*, and *Check Membership*. Table 2 shows the state changes. The behaviors sequence of the "Order System" is *Begin Order*, *Check Membership*, *Promote, Invoice*, *Notify*, and *End Order*. Table 3 defines the state changes. The "Cashier" class receives an invoice and then charges the customer and sends back notification. Table 4 shows state changes of "Cashier".

Starting State	Message	Ending State
Ready	*Login*	*Busy*
Busy	*Check Membership*	*Busy*
Busy	*Logout*	*Ready*

Table 2**.** The State Changes of a "Register" Component

Starting State	Message	Ending State
Ready	*Begin Order*	*StartOrder*
StartOrder	*Check Membership*	*RegularCustomer*
StartOrder	*Check Membership*	*MemberCustomer*
RegularCustomer	*Promote*	*PromotedRegular*
MemberCustomer	*Promote*	*PromotedMember*
RegularCustomer	*Invoice*	*WaitNotification*
PromotedRegular	*Invoice*	*WaitNotification*
MemberCustomer	*Invoice*	*WaitNotification*
PromotedMember	*Invoice*	*WaitNotification*
WaitNotification	*Notify*	*FinishOrder*
FinishOrder	*End Order*	*Ready*

Table 3. The State Changes of an "Order System" Component

Starting State	Message	Ending State
Ready	*Invoice*	*ReadyCharge*
ReadyCharge	*Charge*	*ReadyNotify*
ReadyNotify	*Notify*	*Ready*

Table 4. The State Changes of a "Cashier" Component

Once we have the classes and states, it is straightforward to create the SBPNOs for these classes, as shown in Figures 7, 8, 9, and 10.

Now, we can proceed to the next step and create the system-level SBOPN. The last step is to determine the initial state of each class (these we already showed in the previous figures). The result is shown in Figure 11. Note that we omit the detailed information for each class in the system-level SBOPN view.

5.2 A New Model Based on Reuse

One of the most important benefits of component technology is reusability. In our models, reusability is achieved by modularity and inheritance of classes objects. Let us consider the web-shopping system presented in Section 5.1 again. Assume that after this system is used for several months, it turns out that the performance needs to

be improved for handle multiple customers and also it is no longer deemed necessary to give promotions for member-customers since this marketing plan has ceased to improve the bottom-line of the business.

To improve the performance, let us review the SBOPN shown in Figure 11. As we can see, the system has a *register* class and a *cashier* class. Both of them have some information about customers. So there is some overhead to use these two classes. If we combine these two classes to one class, then we can improve the performance. We name the new class *registerCashier*. To reduce the effect of the new class to other classes remaining in the system, we need this new class have all the methods in the *register* class and the *cashier* class. The new class is shown in Figure 12.

To reuse our earlier component models, but prevent member-customers from getting a promotion offer, we can create a subclass model of the *order system* class model such that the subclass model will only allow regular-customers to get promotion. This can be easily achieved by using the plug-in concept to restrict the *Promote* method. We call this subclass model a *restricted order system*. As a result, the interface of the subclass is the same as for its superclass. The new subclass design is shown in Figure 13. Note that we only require a simple plug-in, not a switchable plug-in, in this example, but for generality, we included a switchable plug-in in the design.

Because we do not change the functionality of the *user interface* class, and all methods remain the same, we can reuse the *user interface* class directly. This is the benefit of modularity of our modeling notation. As an experiment, we can use Figure 7 as the model of the *user interface* class. The system-level model of the improved web-shopping system is shown in Figure 14. In this view of the system, the internal details of the plug-in structure are suppressed in order to simplify the model.

6. Conclusion and Future Work

One challenge in component-based software engineering is to find techniques and tools that are effective in aiding the specification and design of component-based systems. One way to increase the effectiveness of these design techniques is to employ formal methods that provide a well-defined design notation and support design analysis. From our research, and experience with commercial component-based software development, we noticed that restriction inheritance seems to have practical use when customizing general components to define special components.

In this paper, we have discussed our research to blend Petri net concepts and object-oriented design in order to develop a design approach for component-based software systems development. We have selected Petri nets as our underlying design model because we have experience and expertise in applying this formalism (e.g., [19][20]), and because the formalism is mature and with strong support from theory and tools. Finally, Petri nets have an

intuitively appealing graphical interpretation. A unique feature of this work is the idea of a "plug-in" control structure to allow for modeling restriction inheritance.

For future work, we plan to develop some prototype tools that can be used to automate the creation of SBOPN designs for complex systems, including support features for synthesis and management of customizing general components to particular components. In addition, we plan to widen the scope of the work on inheritance modeling to include capabilities for modeling other types of inheritance.

References

[1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998, ISBN 0-201-17888-5.

[2] M. Larsson and I. Crnkovic, "Development Experiences of a Component-based System," *Proceedings of Engineering of Computer Based Systems (ECBS 2000)*, IEEE, 2000.

[3] R. Karl, "Design Patterns for Component-Oriented Development," *Proceedings of the 25th EUROMICRO Conference*, IEEE, ISBN 0-7695-0321-7, 1999.

[4] G. Booch, *Object-Oriented Analysis and Design, with Applications* (2nd ed.), Benjamin/Cummings, San Mateo, California, 1994.

[5] B. Henderson-Sellers and J. M. Edwards, *Booktwo of Object-Oriented Knowledge* : The Working Object : Object-Oriented Software Engineering : Methods and Management, Prentice Hall, 1994.

[6] CORBA, http://www.corba.org

[7] D. Rogerson, *Inside COM*, Microsoft Press, ISBN 1-5731-349-8.

[8] T. Murata, "Petri Nets: Properties, Analysis, and Applications," *Proceedings of the IEEE*, April 1989, pp. 541-580.

[9] 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.

[10] 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.

[11] Y. Deng, S. K. Chang, J. C. A. Figueiredo and A. Perkusich, "Integrating Software Engineering Methods and Petri Nets for the Specification and Prototyping of Complex Information Systems," *Proceedings of the 14th International Conference on the Application and Theory of Petri Nets*, Chicago, IL, USA pp. 203-223, June 1993.

[12] A. Newman, S. M. Shatz, and X. Xie, "An Approach to Object System Modeling by State-Based Object Petri Nets," *Journal of Circuits, Systems, and Computers*, Vol. 8, No. 1, Feb. 1998, pp. 1-20.

[13] X. Xie and S. M. Shatz, "An Approach to Using Formal Methods in Aspect Orientation," *Proceedings of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications* (PDPTA), (Special Session on Architectural Support for Aspect-Oriented Software Systems), Vol. 1, June 26-29, 2000, Las Vegas, Nevada, pp. 263-269.

[14] 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.

[15] D. Buchs and N. Guelfi, "A Formal Specification Framework for Object-Oriented Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. 26, No. 7, July 2000, pp. 635-652.

[16] K. Yamalidou, J. Moody, M. Lemmon and P. Antsaklis, "Feedback Control of Petri Nets Based on Place Invariants," *Automatica*, Vol. 32, No. 1, pp. 15-28, 1996.

[17] S. Ramaswamy and K. P. Valavanis, "Hierarchical Time-Extended Petri Nets (H-EPNs) Based Error Identification and Recovery for Hierarchical Systems," *IEEE Transactions on Systems, Man and Cybernetics*, Feb. 1996.

[18] A. W. Brown and K. C. Wallnau, "The Current State of CBSE," *IEEE Software*, Vol. 15, No. 5, September, pp. 37-46, 1998.

[19] A. Khetarpal, S. M. Shatz, and S. Tu, "Applying an Object-Based Petri Net to the Modeling of Communication Primitives for Distributed Software," *Proceedings of the High Performance Computing Conference* (HPC98), Boston, Mass., April 1998, pp. 404-409.

[20] S. M. Shatz, S. Tu, T. Murata, and S. Duri, "An Application of Petri Net Reduction for Ada Tasking Deadlock Analysis," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 12, Dec. 1996, pp. 1307-1322.

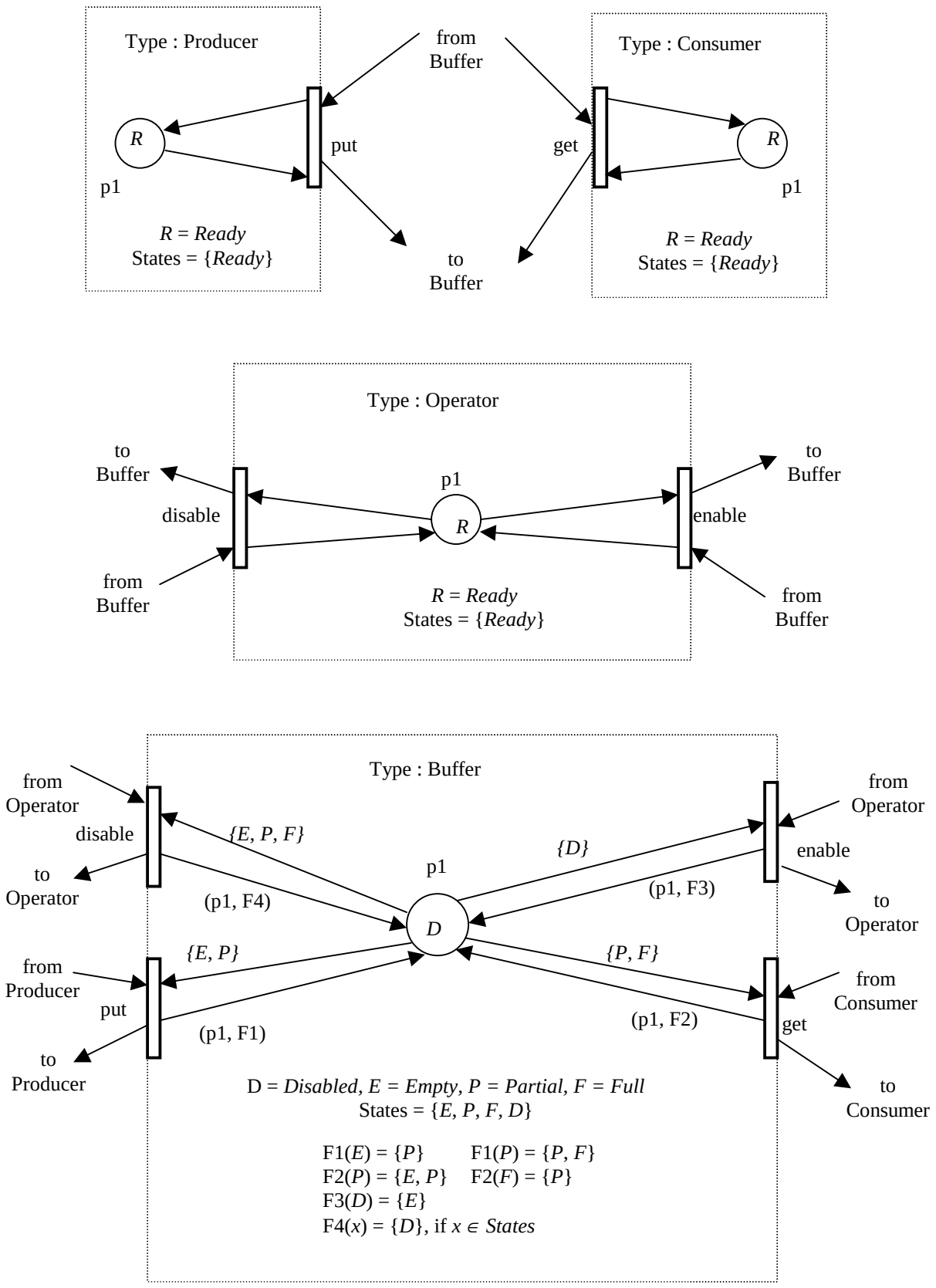


Figure 1. A SBOPN for the Buffer, Producer, Consumer, and Operator System

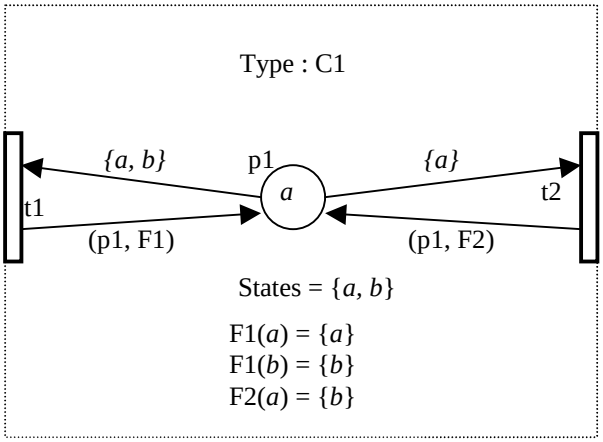


Figure 2. A SBPNO for Class *C1*

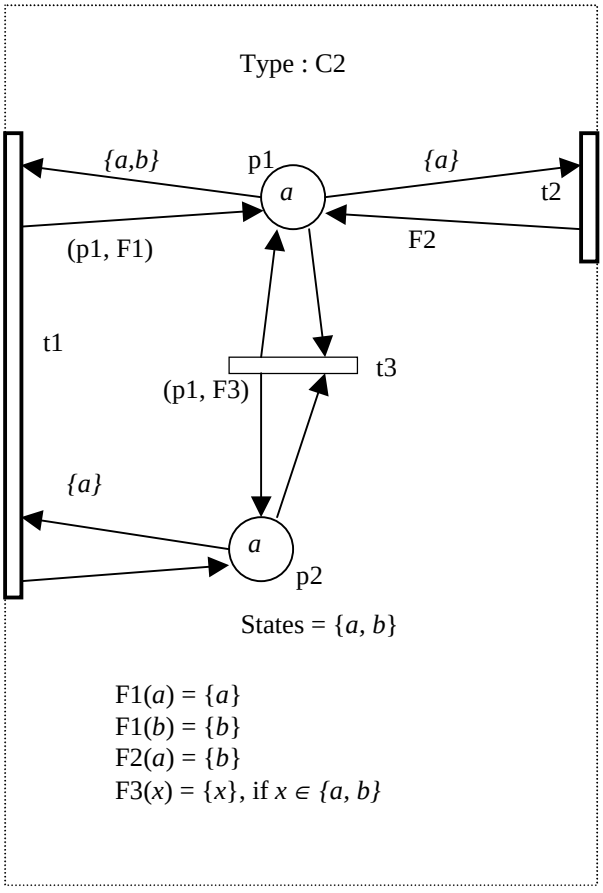


Figure 3. A SBPNO for Subclass C2 (Incomplete)

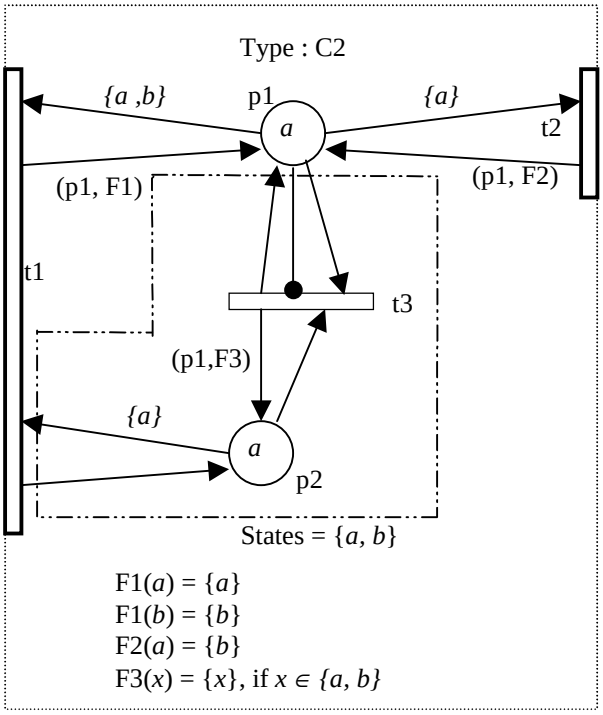


Figure 4. A SBPNO for Subclass *C2* Using a Plug-in

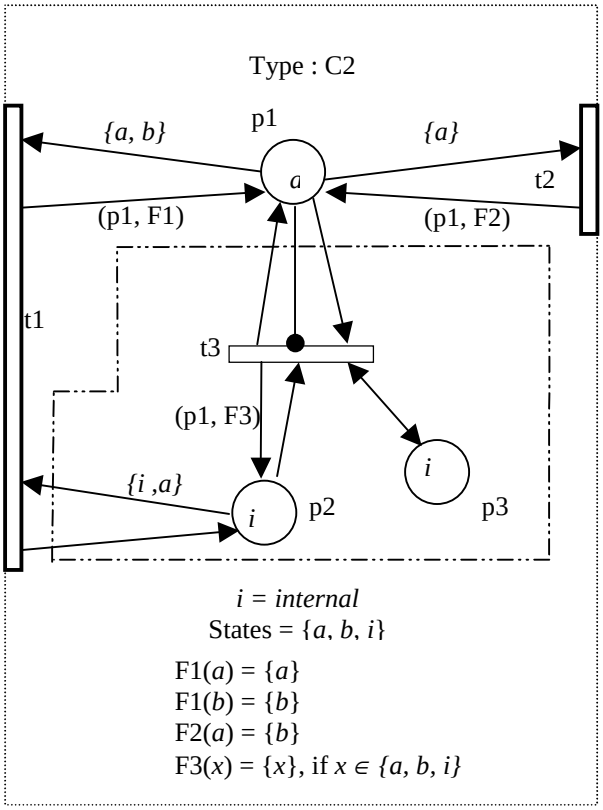


Figure 5. A SBPNO for class *C2* Using a Switchable Plug-in

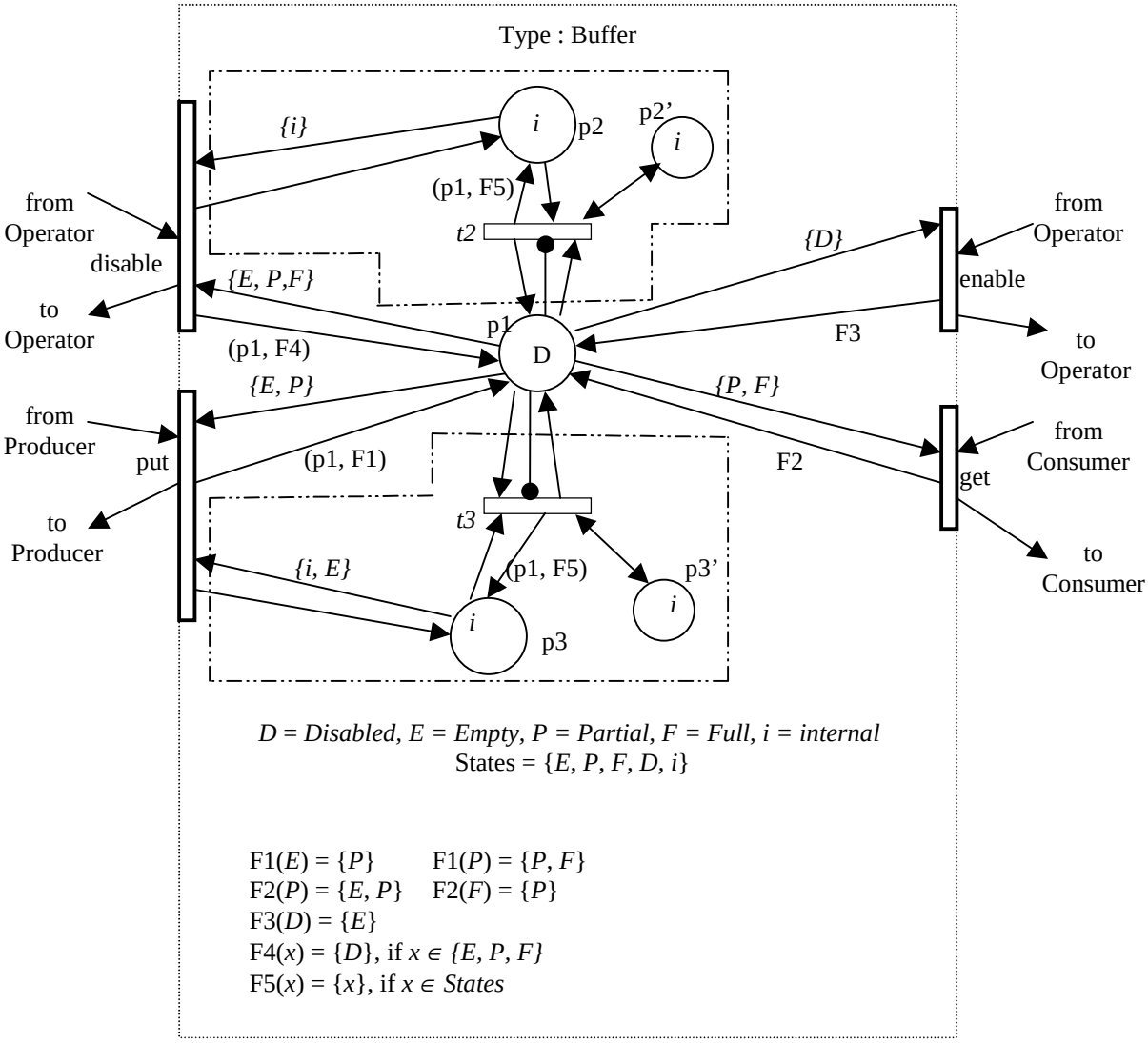


Figure 6. The SBPNO for a “Disable-Free Synchronous” Buffer
Using a Switchable Plug-in

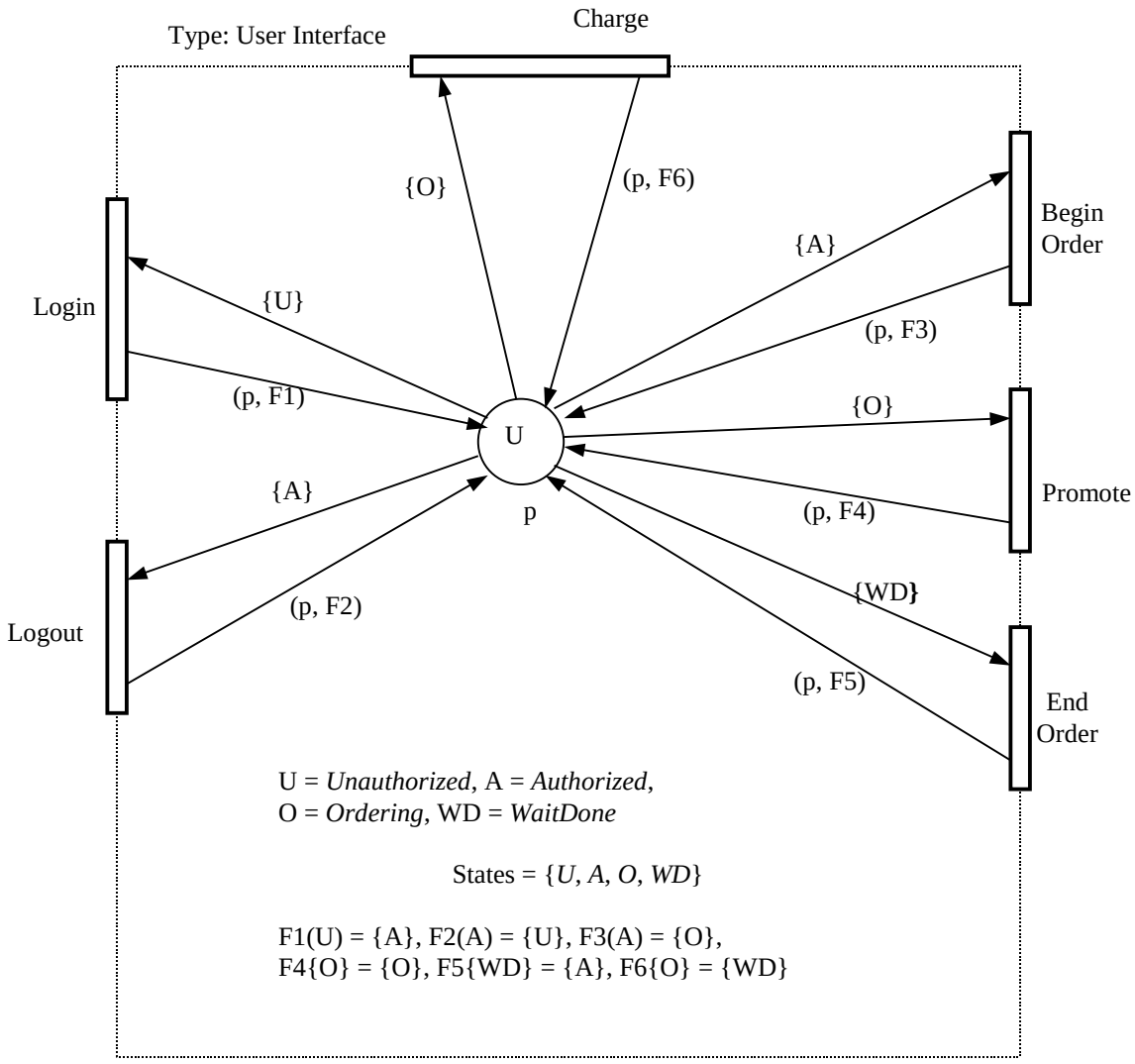


Figure 7. SBPNO of “User Interface” Component

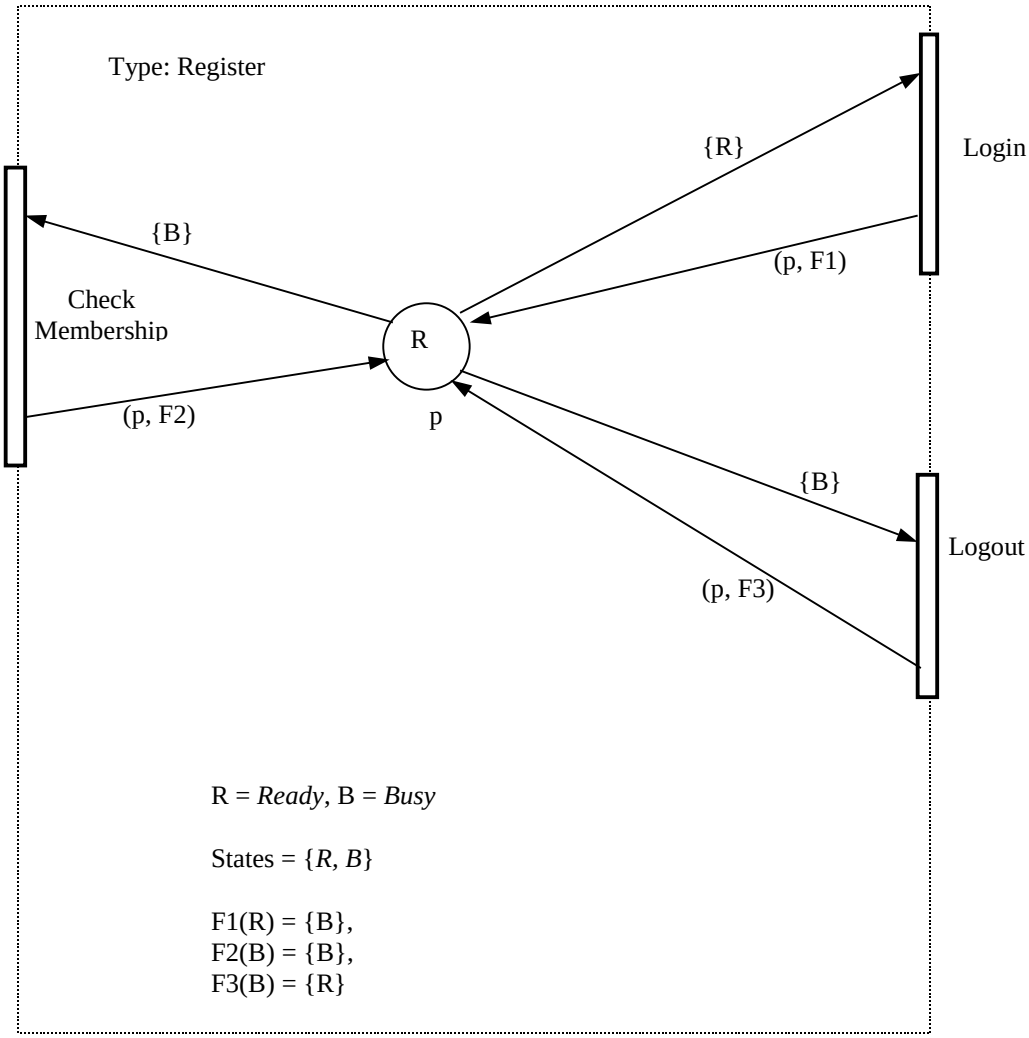


Figure 8. SBPNO of “Register” Component

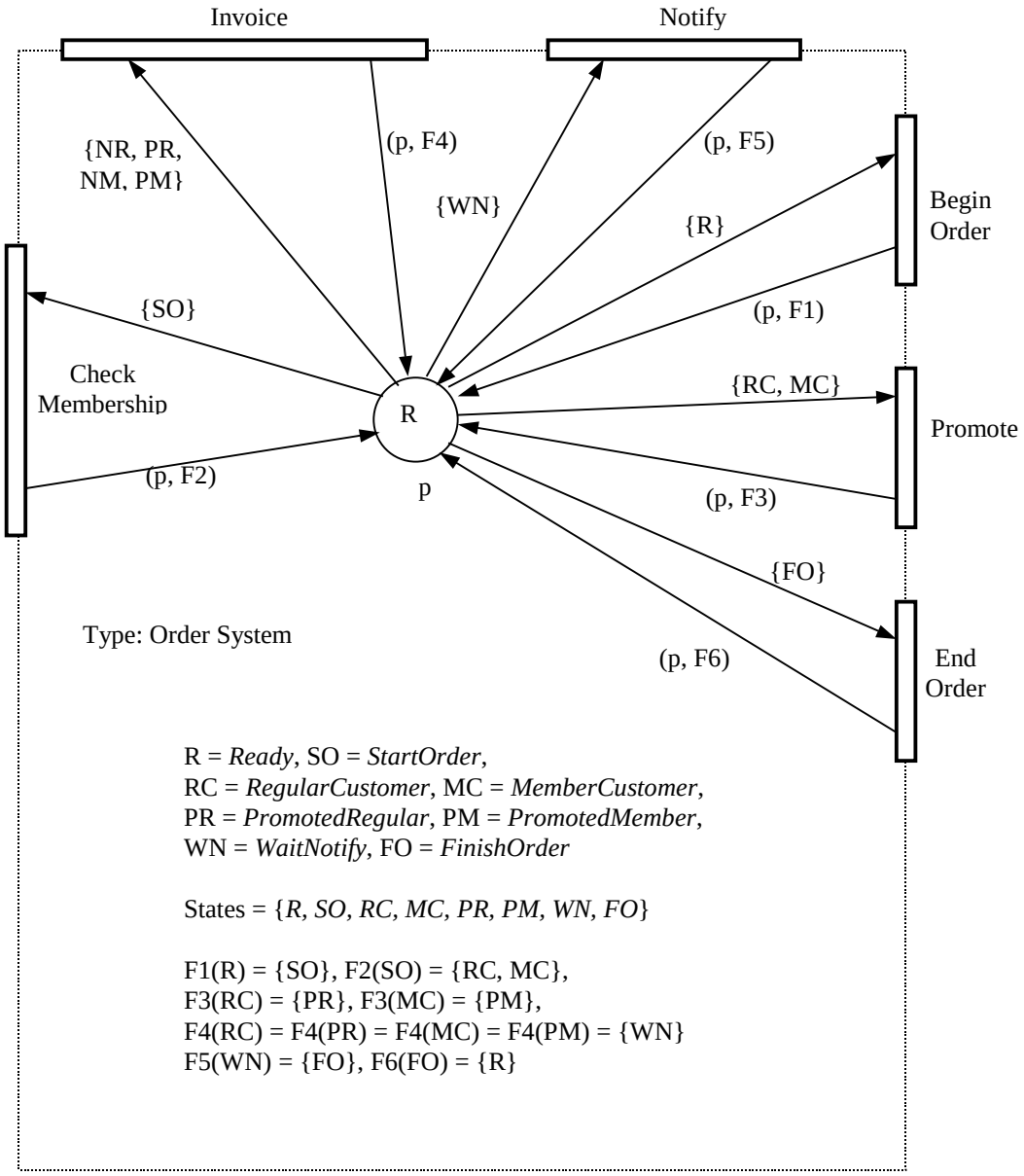


Figure 9. SBPNO of “Order System” Component

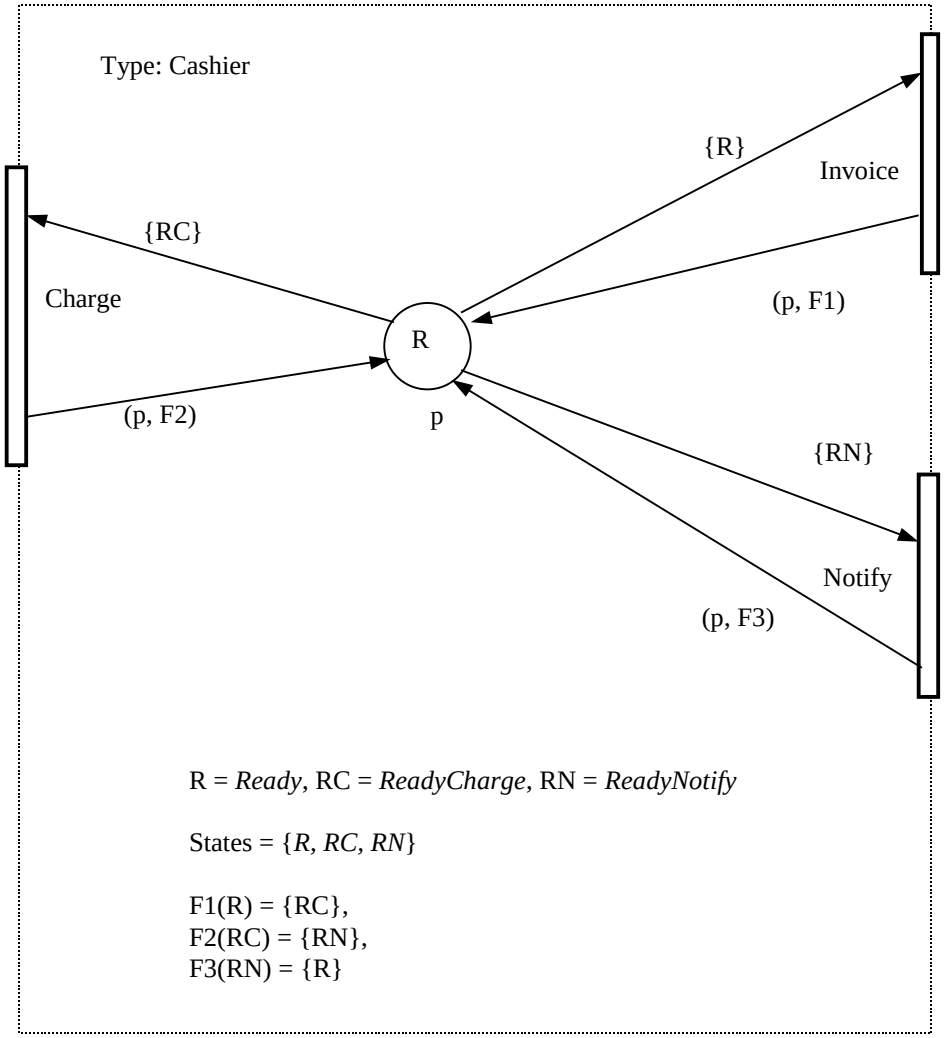


Figure 10. SBPNO of “Cashier” Component

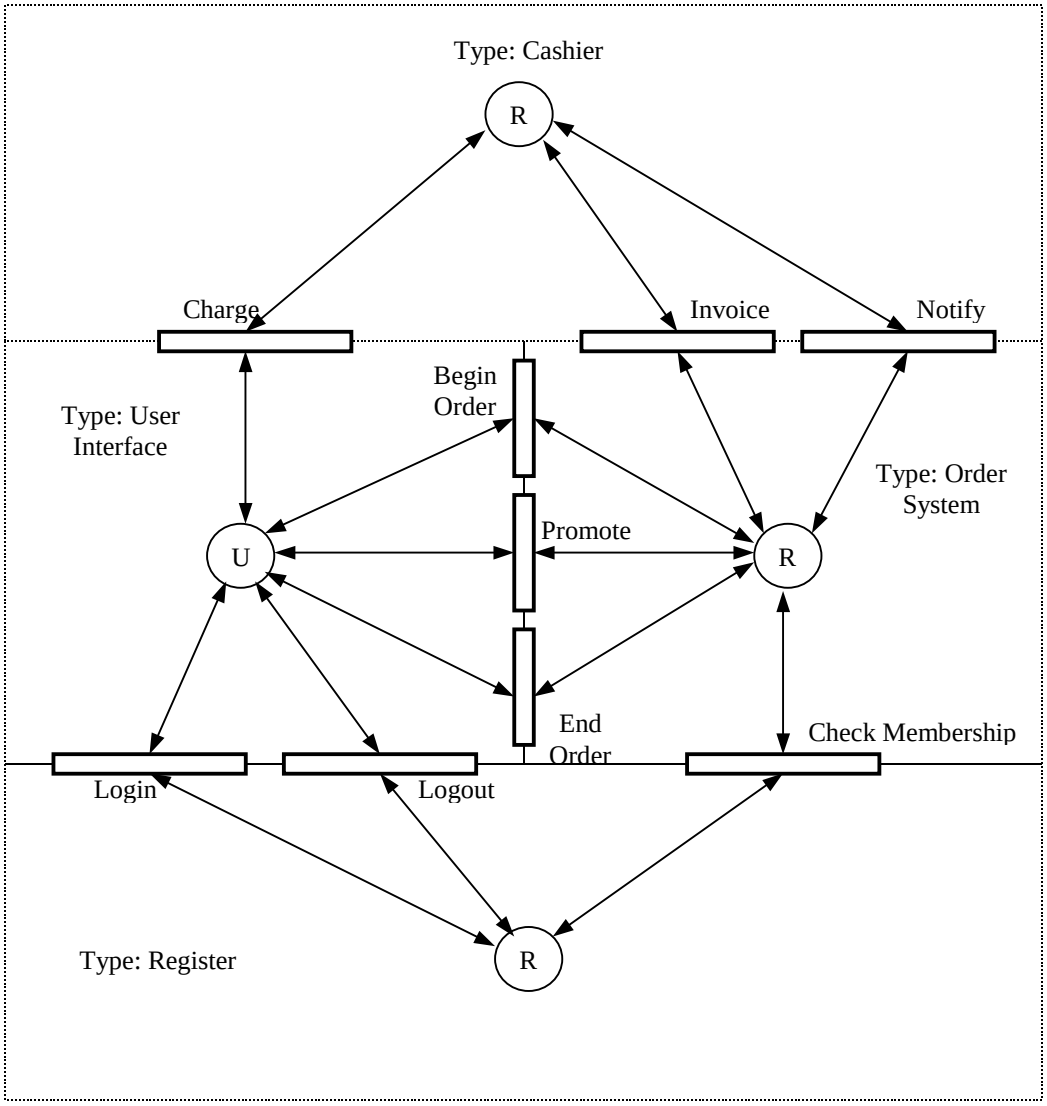


Figure 11. SBPNO of a Web-Shopping System

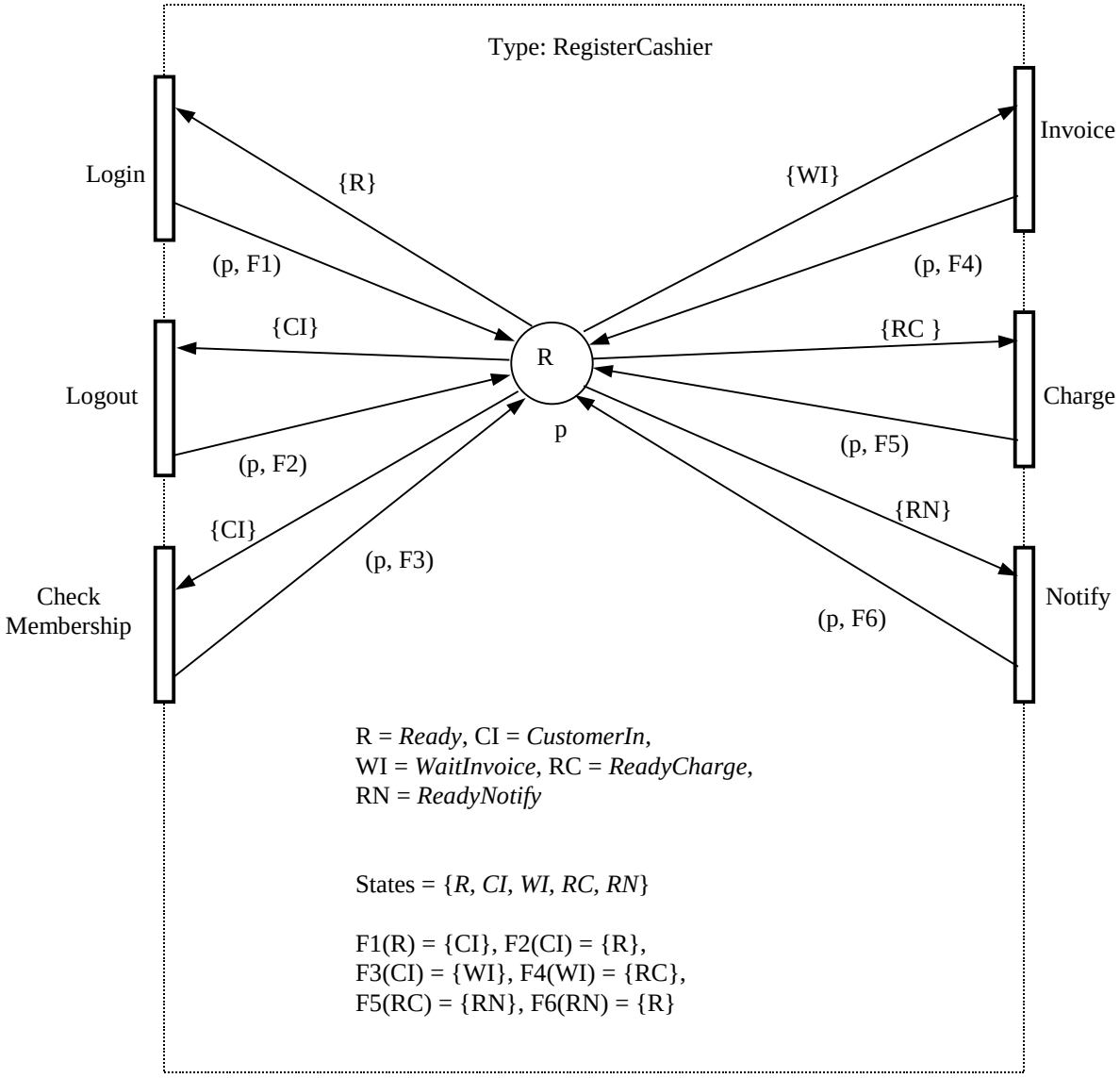
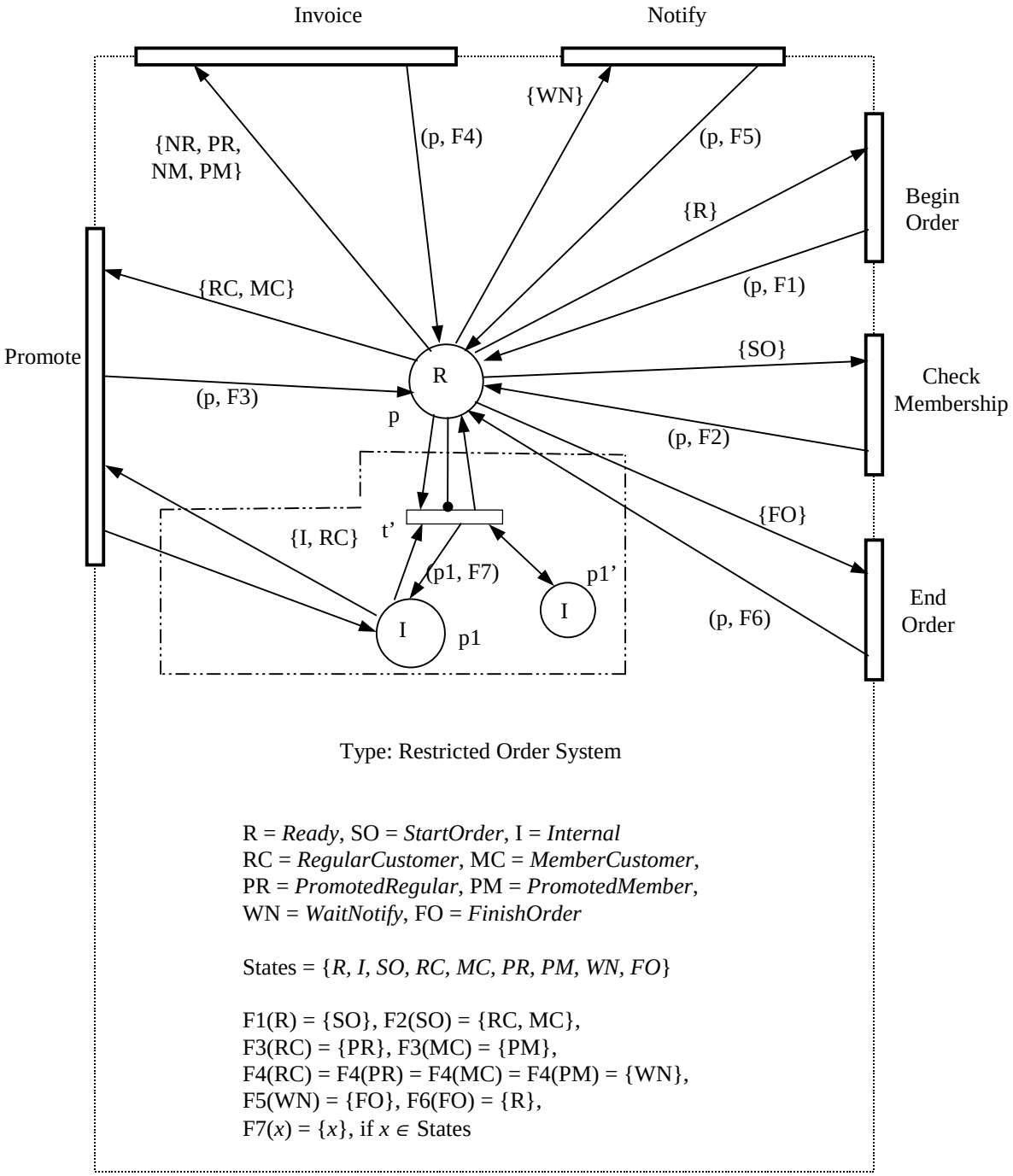


Figure 12. SBPNO of the "RegisterCashier" Component in the Improved Web-Shopping System



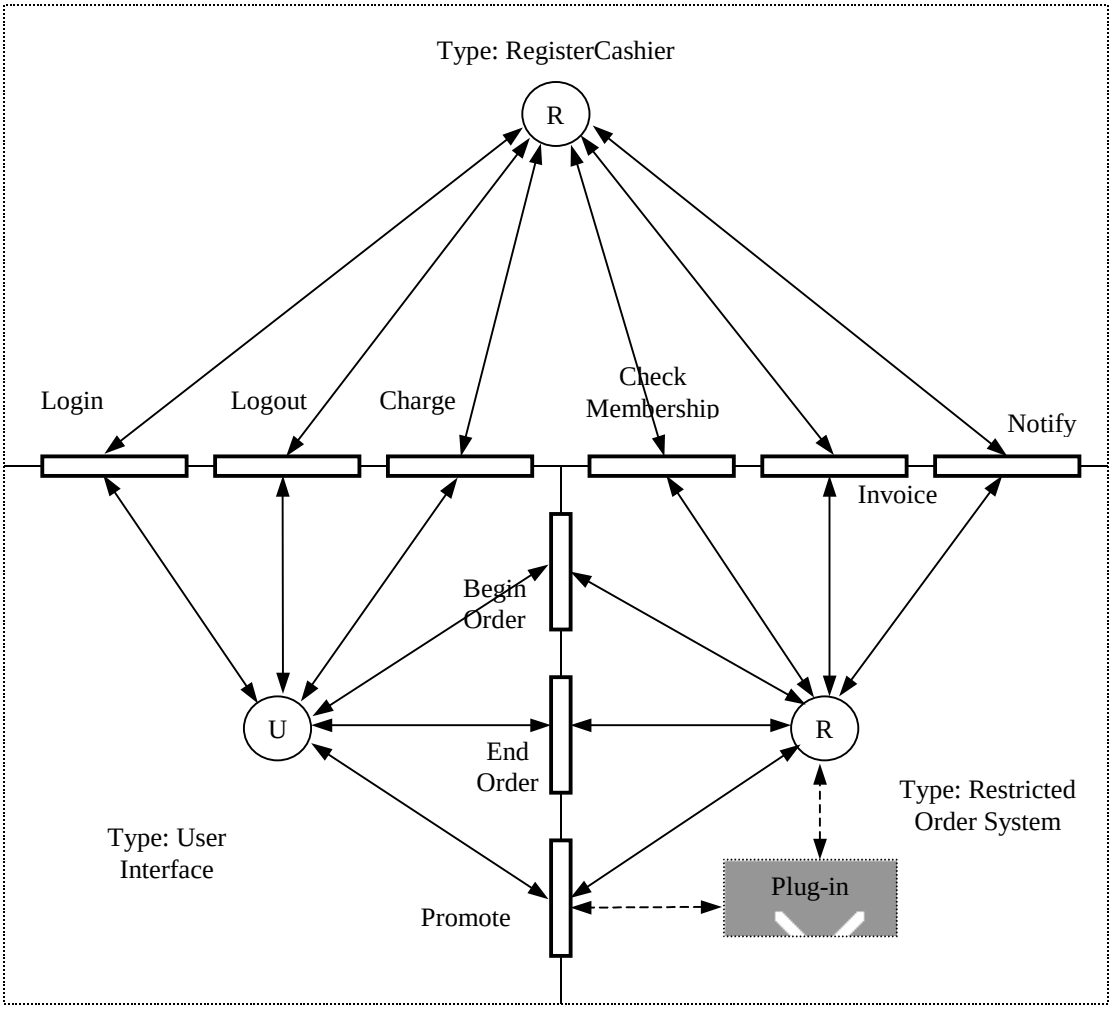
Type: Restricted Order System

R = *Ready*, SO = *StartOrder*, I = *Internal*
RC = *RegularCustomer*, MC = *MemberCustomer*,
PR = *PromotedRegular*, PM = *PromotedMember*,
WN = *WaitNotify*, FO = *FinishOrder*

States = {*R, I, SO, RC, MC, PR, PM, WN, FO*}

F1(R) = {SO}, F2(SO) = {RC, MC},
F3(RC) = {PR}, F3(MC) = {PM},
F4(RC) = F4(PR) = F4(MC) = F4(PM) = {WN},
F5(WN) = {FO}, F6(FO) = {R},
F7(x) = {x}, if $x \in$ States

Figure 13. SBPNO of the "Restricted Order System" Component in the Improved Web-Shopping System



U: Unauthorized
R: Ready

Figure 14. SBOPN of the Improved Web-Shopping System