

Design Models for Components in Distributed Object Software¹

X. Xie and S. M. Shatz

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 applications. Specification of such components by formal methods can pave the way for a more systematic approach for component-based software engineering. 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.

Keywords: Distributed Software, Modeling, Object Design, Petri Nets

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 [8]. However, component-based development is still immature, with a lack of established procedures and support from formal modeling.

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 significant effort. Reuse can be aided by customization that applies 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. 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 restriction inheritance [2]. 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 [8]. But, in our own experience, which does involve

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.

development of commercial component-based software, we have observed benefits of restriction inheritance for customizing components.

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. 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 [5] 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. In this paper we introduce a model called a State-Based Object Petri Net (SBOPN), which is developed from the basic idea introduced in [6]. Here 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 [4]. 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 [3]. Another language, CO-OPN/2 [1], 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 domain-specific.

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 our example, there exists 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). 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) [6]. 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 [5] 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 objects. These components 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. 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 in Figure 1 to create a new model. But, there are disadvantages that can result from a “re-design.” First, creating the new buffer might change the interface of another class, the operator. This conflicts with the basic modularity principle of object-design. This is an important issue, especially when it comes to consideration of model synthesis and reuse. Second, the redesign might result in 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 makes it now difficult to directly identify that the new object is actually one of many possible behaviorally restricted objects derived from a common object – borrowing from object terminology, we can think of these restricted object components as representing subclass objects of a superclass object. We will revisit this issue in Section 3.

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, [9] describes a method for constructing a Petri net controller for a discrete event system modeled by a Petri net.

3. MODELING RESTRICTION SUBCLASS OBJECTS

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 objects.

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, with P as a finite set of nodes, called Places; T as a finite set of nodes, called Transitions, disjoint from P , i.e., $P \cap T = \emptyset$; and $A \subseteq (P \times T) \cup (T \times P)$ as 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.

To simplify SBPNO models, implicit (default) state filters and implicit state-transfer tuples are allowed. 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.

Now we can identify properties of a restriction subclass and present the definition of a restriction subclass model, i.e., a model for a restriction subclass object. 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. Thus, any firing sequence of a SBPNO subclass model should be a firing sequence of the superclass model when we only consider the shared transitions. 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 2 (Restriction Function): Let $N_I = (Type_I, NG_I, States_I, IS_I, Stoken_I, ST_I, SFM_I, STM_I)$ be a SBOPN, and let function $f: SF_I \rightarrow 2^{States_I}$, where SF_I is the domain of SFM_I , and 2^{States_I} is the power set of $States_I$. The function f is called a *restriction function* for N_I if and only if f satisfies: $\forall sf_I \in SF_I, f(sf_I) \subseteq sf_I$.

Applying f to the state filters of N_I creates a new model, which we denote as N_I/f . It can be shown that N_I/f is a restriction subclass model of N_I , but note that N_I/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 3 (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) $(ST \cap p_2^* \neq \emptyset) \wedge (ST \cap p_2^* \subseteq ST \cap p_1^*)$ (Note: p_1^* is the set of output transitions of the place p_1).
- 2) For any shared transition $t \in (ST \cap p_2^*)$, 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 $M(p_1) = M(p_2)$, and any transition $t \in (ST \cap p_2^*)$, if t fires under M' , then the tokens consumed by t from p_1 and p_2 should have the same state values.

3.1 Basic Plug-in Design

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 present a SBPNO model for an object $C1$. Now, suppose that we want to derive a model for a restriction subclass object $C2$ 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 superclass object $C1$. We need t_2 to remain enabled in the a -state.

To model this subclass object, we create a new place p_2 (see Figure 3) as a control place. Transition t_3 is introduced for the purpose of copying the state token from p_1 to p_2 . 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 [7]. 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 p_1 to p_2 is due to the state-transfer function F_3 . Note that t_1 is not enabled any more after t_3 fires. As we can see, p_2 now 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).

Algorithm 1: Model a restriction subclass object by use of plug-in structures

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

A restriction function (see Definition 2), $f: SF_1 \rightarrow 2^{States_1}$

Output: A restriction subclass model N_2 of N_1 (N_2 has the same externally observable behavior as the model N_1 if identified earlier).

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 $p_1 \in t^*$, 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. Create a refreshing place p_2 of p_1 . (* An algorithm for creation of a refreshing place is given in [10]. The basic idea can be understood by observing that the application of this step to the SBPNO in Figure 2 creates part of the SBPNO shown in Figure 4 – all of the model except the arcs $(p_2, t1)$ and $(t1, p_2)$, and the state-filter $\{a\}$ for the arc $(p_2, t1)$. *)

B. Add an arc r_1 from p_2 to t . Use $S2$ as the state filter for r_1 . Add an arc r_2 from t to p_2

End For

End For

The initial marking of a subclass model created by Algorithm 1 is determined by the initial marking of the source superclass model. 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 1 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.

3.2 Switchable Plug-in Structures

One advantage of Algorithm 1 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.” A switchable plug-in offers a key advantage: It allows an SBOPN model to represent a family of restriction subclass models, corresponding to a family of components.

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. In general, to model a restriction subclass using switchable plug-ins, we can use Algorithm 1 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 would employ two switchable plug-ins – connected to the method transitions *disable* and *put*. The initial marking of all places belonging to the plug-ins is *internal*. For the plug-in associated with the *disable* method, the state filter is set to {*internal*}. Thus if the plug-in is “turned-on” (by marking its 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*}, so that the *put* method is only active when the buffer is in the empty state. Due to a lack of space, we cannot show this buffer model, but it is given in [10]. Of particular interest is the observation that this one subclass model actually represents 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 (which we call the *Disable-switch* and the *Put-switch*). The following table illustrates the options:

<i>Disable-switch</i>	<i>Put-switch</i>	<i>Model</i>
Marked	Marked	A “disable-free synchronous” buffer
Marked	Unmarked	A “disable-free” buffer
Unmarked	Marked	A “synchronous” buffer
Unmarked	Unmarked	A general buffer (as in Fig. 1)

The ability to model a family of components can be very helpful for 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.

As a final comment, we note that we did not explicitly discuss any issues regarding model analysis. But, we can note that the SBOPN model, including plug-in structures, can be transformed to standard Colored Petri nets, and then to ordinary Petri nets. Thus, the discussed modeling approach can leverage on existing and developing techniques for analysis, including work aimed at the state-space explosion problem. Further details are provided in [10].

4. 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. 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. A unique feature of this work is the idea of a “plug-in” control structure to allow for modeling restricted behavior on the part of object models. We discussed how this can support the modeling of customized components. One area for future work is to develop some prototype tools that can be used to automate the creation of SBOPN designs for complex systems, including support for synthesis and management of customizing general components to particular components. In addition, we plan to investigate capabilities for other forms of inheritance modeling.

REFERENCES

- [1] Biberstein, O., Buchs, D. and Guelfi, N. “CO-OPN/2: A Concurrent Object-Oriented Formalism,” *Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS), Canterbury, UK, July 21-23 1997*, Chapman and Hall, London, 1997, pp. 57-72.
- [2] Booch, G. “Object-Oriented Analysis and Design, with Applications (2nd ed.),” *Benjamin/Cummings*, San Mateo, California, 1994.
- [3] Jensen, K. “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.
- [4] Lakos, C.A. “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.
- [5] Murata, T. “Petri Nets: Properties, Analysis, and Applications,” *Proceedings of the IEEE*, April 1989, pp. 541-580.
- [6] Newman, A., Shatz, S.M. and Xie, X. “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.
- [7] Ramaswamy, S. and Valavanis, K.P. “Hierarchical Time-Extended Petri Nets (H-EPNs) Based Error Identification and Recovery for Hierarchical Systems,” *IEEE Transactions on Systems, Man and Cybernetics*, Feb. 1996.
- [8] Szyperski, W. “Component Software: Beyond Object-Oriented Programming,” Addison-Wesley, 1998.
- [9] Yamalidou, K., Moody, J., Lemmon, M. and Antsaklis, P. “Feedback Control of Petri Nets Based on Place Invariants,” *Automatica*, Vol. 32, No. 1, pp. 15-28, 1996.
- [10] Xie, X. and Shatz, S.M. “Design Support for State-Based Distributed Software,” Technical Report, Concurrent Software Systems Laboratory, Dept. of EECS, UIC, 2000.

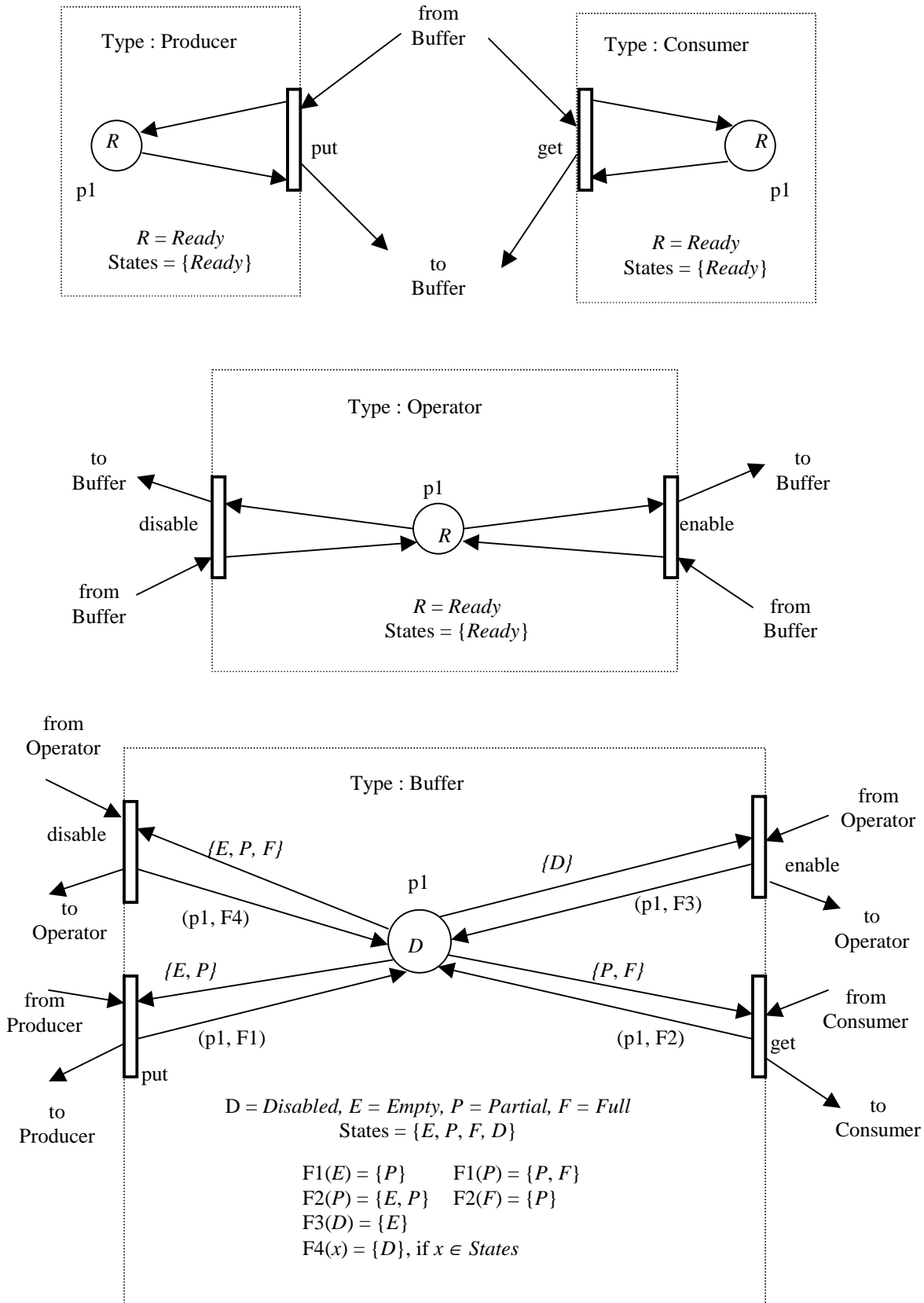


Figure 1. A SBOPN for the buffer, producer, consumer, and operator system

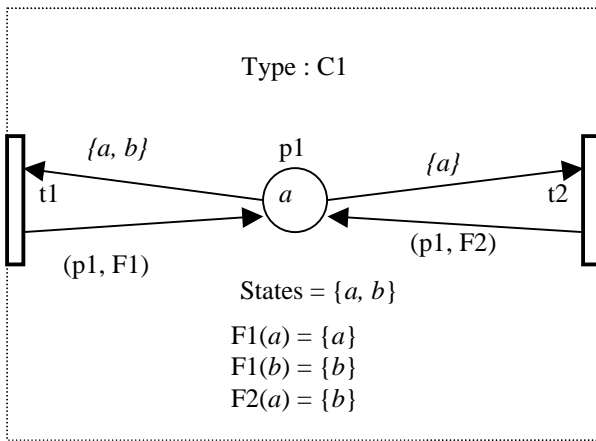


Figure 2. A Model for Object C1

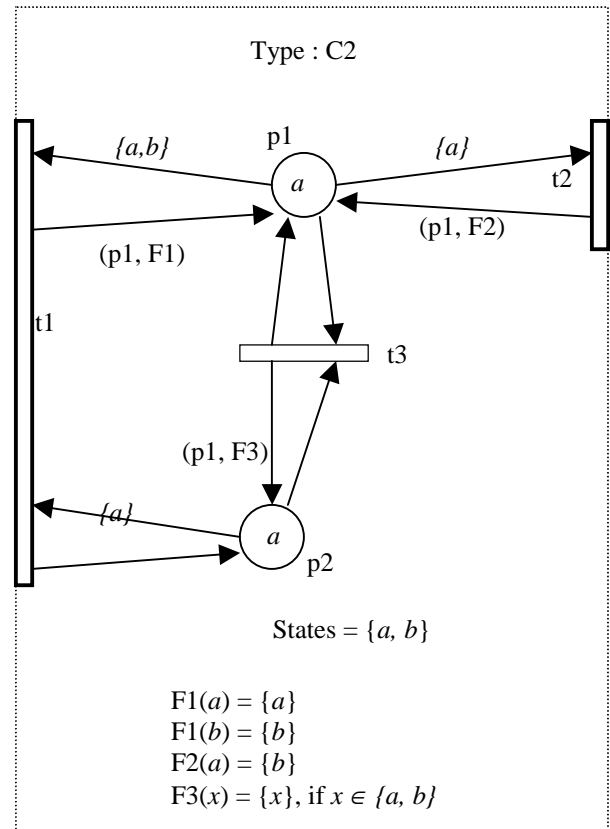


Figure 3. A Model for Subclass Object C2 (Incomplete)

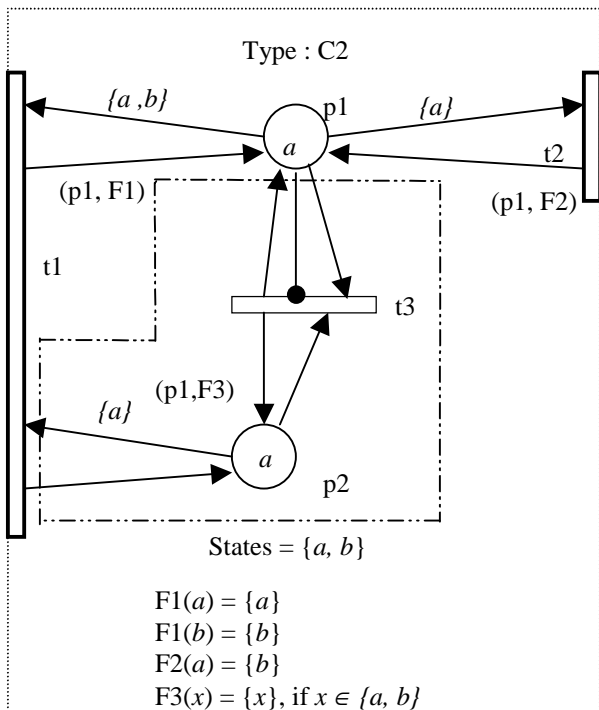


Figure 4. A Model for Subclass Object C2 Using a Plug-in

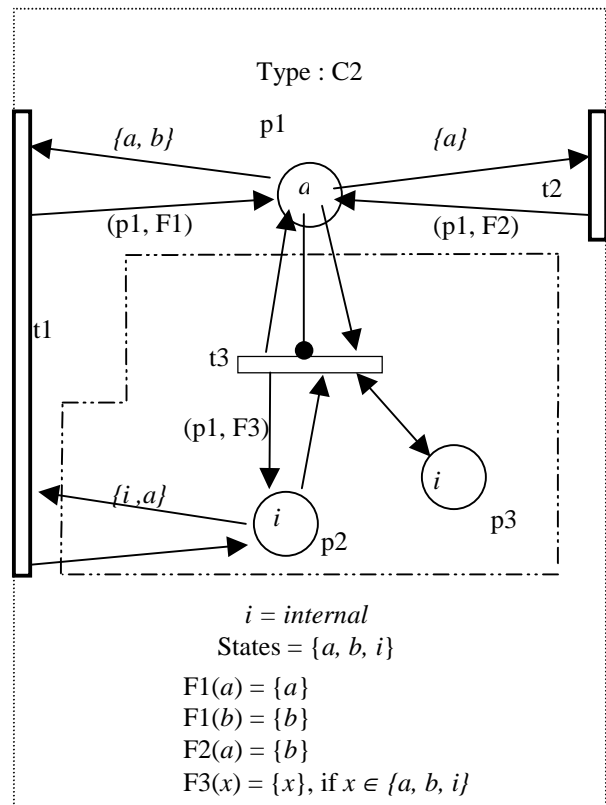


Figure 5. A Model for Object C2 Using a Switchable Plug-in