Development of Class-Level and Instance-Level Design Models For Distributed-Object Software [1]

X. Xie and S. M. Shatz
Concurrent Software Systems Lab
University of Illinois at Chicago
shatz@cs.uic.edu

Abstract

Leading the pack of new-generation software applications are distributed applications, which are well suited to designs based on distributed objects. To help cope with the increased complexity of such systems, developers can benefit from formal methods and associated tools that support specification and analysis. This paper presents a Petri net motivated approach for modeling distributed-object software. Both class-level and instance-level models are defined, with the goal of supporting scalable designs. Model analysis, with state space reduction due to symmetry among object instances, is also considered. A case-study is presented to illustrate the specific modeling and analysis features.

1. Introduction

Leading the pack of new-generation software applications are distributed applications such as those for internet-based commerce, cooperative design/development, and many forms of intelligent agent systems. However, due to the complexity of such systems, software engineers face increased difficulty in developing applications that meet requirement specifications with low cost and high reliability. Although there is growing interest in developing these applications using distributed object architectures, this is a new design paradigm that requires a number of advances in both research and development. To help cope with such inherent complexity, designers and developers can benefit from formal methods that provide design models that support both specification and analysis. Furthermore, it is useful to have methods that combine various design techniques and take advantage of the properties and strengths of those techniques. For example, one strength of object oriented techniques is their focus on system modularity. Other techniques, including those based on graph modeling, have strengths in providing operational semantics (i.e., the meaning of various system-required operations) for specific properties like concurrency or distribution of function.

[1] This material is based upon work supported by the U.S. Army Research Office under grant number DAAD19-01-1-0672, and the NSF under grant number CCR-9988168.

Petri nets provide a graphical and mathematical modeling tool applicable to a wide range of systems [1]. They are a promising tool for describing and studying information processing systems that have the characteristics of being concurrent, asynchronous, distributed, parallel, and non-deterministic. However, basic (low-level) Petri net models of practical systems tend to be large and difficult to develop, and, as with other state modeling techniques, the so-called state space explosion issue is a significant challenge for effective analysis [2].

Combining Petri net modeling and object oriented design presents an opportunity for achieving a modeling capability that can describe large, complex systems more easily and directly. In general, the proposed methods in this area use enhanced forms of Petri nets as a basis of the combination, and pursue two main approaches [3]. One approach is the "Petri nets inside objects" approach, in which traditional Petri net constructs are used to model the internal semantics of objects [4][5]. The other approach is the "objects inside Petri nets" approach, in which the semantics of tokens are expanded to include other information, which could include object definitions [6]. LOOPN [7] is a method based on the second approach, while G-nets [8] and CO-OPN/2 [9] are methods based on the first approach. A G-net provides well-defined mechanisms for hiding the details of an object model's internal realization and a well-defined set of interface elements to provide communications with other G-net models. But G-nets do not support other object-oriented features, such as inheritance. For both LOOPN and G-nets, there are some likely barriers to the natural use of these models as software-engineering design models. First, their main focus is on formal specification, with the result being a modeling notation that may not be able to directly leverage existing Petri net analysis concepts and methods. Second, while these models do incorporate some key object-oriented features, a software engineer, or object system designer, must have fairly specific knowledge about Petri nets to effectively develop models for the domain of distributed-object software. Third, the models do not explicitly support the derivation of general instance models from base-class models. This can result in potential difficulties in creating scalable software designs.

Our approach is based on a domain-specific form of Petri net model — one that is explicitly aimed at distributed systems designed in terms of collaborating objects. We call this model a State-Based Object Petri Net (SBOPN). An earlier version of the basic model was described in [10], without support for general object instantiation. Our work is perhaps most closely related to the modeling work of Hammer, Hanish, and Dillon [11] (see also [12]), due to the focus on object-oriented systems and the use of Colored Petri nets [13] as a base model. One general similarity of these works is the concern for interface modeling. In the work of Hammer, et al., object interfaces are defined in terms of Petri net place nodes (and called Points of Interactions), and combining objects involves "wiring" interfaces together with explicit external transition nodes. In our model, interfaces are defined in terms of shared transition nodes, which implies a more implicit "wiring" of interfaces. In a sense, this means that object linking is more direct, but less flexible. On the other hand, these works differ in terms of their

primary focus. In particular, unlike the model of Hammer, et al., our approach emphasizes a concern for preserving the general analysis capability of Petri nets and for providing scalable design models. The latter concern is addressed by defining the concept of instance-level models – for multiple objects representing a common object class.

Our SBOPN notation is intended to support both formal specification and analysis. Also, because the model is domain-specific, object system designers should find that model creation and understanding are simpler than would be the case with other more general-purpose formal methods (including other Petri net variants). Additionally, as we have noted, the approach presented here supports both class-level and instance-level modeling, with a capability for exploiting object symmetry during model analysis.

The following sections of the paper are organized as follows. Section 2 introduces the basic class-level SBOPN model and "single-instance" objects. Section 3 extends the basic model to support general instance-level modeling (i.e., instantiation of a class model into "multiple-instance" objects). Section 4 discusses formal analysis of instance-level SBOPN models, and introduces the concept of a "compact marking," which exploits inherent object symmetry as a way to reduce the size of the model's state space. Section 5 casts the modeling approach into a general framework for object oriented design, and illustrates the use of instance-level SBOPN models via a basic web-shopping system case study. Finally, Section 6 provides a conclusion and mentions some future work.

2. Class-Level Modeling

We begin with some a brief, and informal, introduction to colored Petri nets [13] (CPN), which is the basis for our SBOPN model. We assume that the reader is familiar with the core concepts of Petri nets [1]. In standard Petri nets, tokens have no identity. 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 [14][15]. With this extension, a transition is enabled if the input places to the transition contain a specified set of colored tokens. An inscription on the associated arc defines the specified tokens — the arc inscription specifies a token attribute. The firing of an enabled transition causes the removal of specified tokens from the input places and the depositing of specified tokens in output places. Again, arc inscriptions — this time on output arcs of a transition — are used to set the attributes of deposited tokens. A colored Petri net with a finite color set can be transformed into a standard Petri net [13]. Thus these models are theoretically equivalent in terms of modeling power.

Like standard Petri nets, the CPN model is a general model, and thus it can, in principle, be used to model distributed object software. However, because the model is general-purpose, it tends to require fairly sophisticated knowledge of the specific CPN notation. In particular, the application designer must map domain features to net elements. Also, standard CPN models do not take advantage of object-oriented techniques. Our interest is in a net-based modeling notation that is domain-specific, for the domain of distributed object software. This can provide a tool to help bridge the gap between specifications generated by software engineers, who tend to be adverse to formal methods and unsophisticated with Petri net modeling, and the mature analysis and simulation capabilities associated with Petri nets.

The SBOPN model focuses on systems composed of objects, which are independent software modules that cooperate and interact by messages. Objects are associated with internal states that control their behavior, i.e., the services they provide. Object state and behavior are bundled together, with the behavior of an object being defined in terms of methods that implement services. The method executions depend on, and modify, object states. One can view the SBOPN notation as an "interpreted" form of colored Petri net since it defines explicit constructs for key object-oriented elements, such as objects, methods, and states. Additionally, since these constructs use standard colored net elements, or define new elements that can be translated into standard elements, the SBOPN model can also take advantage of existing formal analysis techniques.

Before introducing a formal definition of SBOPNs, let us give a sense of the graphical syntax and semantics by introducing an example. Consider the classic example of a system that uses a bounded buffer to temporarily hold items. Items can be deposited in the buffer by a producer when the buffer is empty or partial full. Likewise, items can be removed from the buffer by a consumer when the buffer is partial full or full. Figure 1 is an example SBOPN model for such a system. There are three objects: a single producer, a single consumer, and a single buffer. For now, the reader can ignore the notational details of this model. But it is important to note that the shared transitions (common to two or more object models) represent methods. For example, the buffer-object supports two methods: *put* and *get*. Also note the arc inscriptions specify pre- and post-states for an object, which are called state filters and state-transfer functions, respectively. These inscriptions specify what state an object must be in for the method to be successfully activated, and what state the object will be in after the successful execution of that method. For example, the *put* method can be called only when the buffer object is in either the *empty* or *partial* state. Likewise, if the buffer is in the *partial* state when the *put* method is called, the new state of the buffer could be either *partial* or *full*.

Now we can introduce the formal definition of an SBOPN model. Since a SBOPN is a system-level model, consisting of several object models, we start with the definition of a State-Based Object (SBO). A SBO

serves as a model for individual object classes. It defines the general behavior for a class of objects and identifies the set of states that are applicable for such objects. Of course once we associate a particular initial state with such a class model, the model can then be viewed as an object model (in contrast to the more generic class model). Thus, modeling single-instance instantiation is very straightforward.

Definition 1: A *State-Based Object*, (SBO), is a 7-tuple, SBO = *(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.
  III. $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, known as the flow relation.
  IV. $P \cap T = \phi$ and $P \cup T \neq \phi$.
- *States* is a finite set of distinct states that define the possible states of the SBO. A token (as in standard, or colored Petri nets) is always associated with 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 SBO**.**
- $ST \subseteq T$ is a set of shared transitions, where for any $st \in ST$, $(sp, st) \in A$ and $|st^{\bullet}| > 0$. A shared transition in a SBO is a transition that is shared with other SBOs. Shared transitions model the acceptance of a message from other SBOs, or the sending of a message to other SBOs.
- $SFM: (A \cap (P \times T)) \rightarrow 2^{States}$ is a state-filter mapping, where 2^{States} is the power set of *States*. This 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. This maps each transition-to-place arc (t, p) to a state-transfer tuple (p', stf), where $p' \in \{p|(p, t) \in A\}$ is called the state-transfer place and $stf: States \rightarrow 2^{States}$ 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**.**

A SBO is denoted graphically as a Petri net (a subnet) inside a box. Note that in general we define state filters and state-transition functions only for arcs connected with shared transition. For other arcs, we allow a

default state filter and state-transfer function. The default state filter is just the variable *States*, which implies that a token with any state can pass via the corresponding arc. The default state-transfer function simply assigns to the associated tokens (being deposited in an output place) a state value that is the same as the state value for the designated token consumed by the transition firing.

Definition 2: A *State-Based Object Petri Net*, SBOPN, is a Petri net consisting of connected SBOs as defined in Definition 1.

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

3. Generalized Instance-Level Modeling

In the previous section we introduced the basic SBOPN model, which serves as a class-level model. We also mentioned the straightforward process of marking individual SBO models (with state tokens) to create a system design composed of single-instances of each object class. However, for scalable object-oriented design, it is desirable to support a general form of instance-level modeling. In this section, we extend the SBOPN notation to support multiple-instance instantiation and discuss how this feature can also aid in system analysis. To support design automation, we want the process of creating instance-level models to be algorithmic. This is addressed by defining a set of model instantiation rules.

3.1 Significance of Instance-Level Models

As Figure 1 shows, the SBOPN can directly model a system at the class-level and also easily support designs based on only one single instance of each object class. We refer to such designs as “class-level.” We use the term “instance-level” to refer to models or designs that incorporate an arbitrary number of objects for any one class. To properly serve as an object-oriented model, it is important for SBOPNs to have the ability to model instantiation, i.e., to model multiple instances of a class. Such a capability is an essential feature for design modeling of scalable software systems. As we will see in our case study in Section 5, some properties in a system’s class-level model may not carry forward in a particular instance-level model. Thus, it is important to support formal analysis of a system not only at the class level, but also at the instance level. Most net-based object-oriented formalisms that we are aware of do not provide support for such instance-level analysis. But, since an SBOPN model can be mapped to a CPN, before or after applying instantiation, SBOPN models can support this analysis.

Let us consider scaling the example model from Figure 1 to allow several producers, several consumers, and even several buffers. To model such cases, one straightforward solution would be to craft a state-based object for each instance of each object. This would require a shared transition for every pair of object instances that can exchange any message. In Figure 2, we have 3 producers, 2 consumers and 1 buffer. Since each producer can exchange a message with the buffer, and each consumer can exchange a message with the buffer, the model requires five shared transitions. There are two main disadvantages of this solution. First, one conceptual method (e.g., putting items into a buffer) is represented by multiple transitions, because the method is related to different instances. This violates a basic tenet of object-oriented design. Second, the system becomes complicated quickly as the number of object instances increases. If we revise the model in Figure 2 for the case of l producers, m buffers, and n consumers, then we will require $(l+n)*m$ shared transitions. This is cumbersome for modeling practical systems, which may have many such object instances.

Another way to model multiple instances is for all instances of one class (e.g., producers) to share a transition for each method associated with that class (e.g., put). Now, we will not have the situation of using many transitions for one method. Figure 3 illustrates this idea. Unfortunately, as a result of this technique, we would need to change the firing rule of a shared transition. If we were to maintain the common firing rule, firing transition t1 in Figure 3 could require all producers to be ready. But, this is not the desired semantic, because we only need one producer at a time to trigger the firing and to add a new item to the buffer. So, the “and” semantic of input places of shared transition would have to be changed to an “or” semantic. Also, we would then need some way to ensure that an output token will be “directed back” to the same producer that caused the firing. Even if we could achieve such changes in the firing rules, we will face the situation of the analysis becoming more difficult and the likelihood that many analysis techniques from Petri net theory will no longer be available. We wish to avoid this situation.

3.2 Instantiation Rule of SBOPN

To solve the previously introduced problems, let us consider more closely the relationship between classes and objects (or instance). A class is an abstract concept. It presents a group of individuals, which have some common properties. An object, which has special values, is an individual of a class. For instance, “dog” is a class. It has properties, such as weight and color. A particular dog, “Sam,” is an individual (or an instance) of a dog. It has particular values for its weight and color. To distinguish any one dog from other dogs, we can give it an ID; in this example the ID is “Sam.” But in the (class-level) definition of SBOPN, we do not have the concept of an ID for each object. We can observe that when we try to model multiple producers, we cannot just create

several copies of the original producer model because in fact we want multiple *instances* of the same class (producer). This is the reason that the previous two ideas failed. What we need is instantiation, not copying.

To instantiate a SBOPN system in creating a design model, we will know the number of instances of each class (or an upper bound). So, we can give each instance a unique ID. Now the tokens in the input place(s) of a shared transition must carry both object-state and object-ID information. Naturally, we must change the inscriptions on input arcs, as used in generalized colored nets. This is done by adding ID information to the original inscription. We also need to make sure the output token denotes the same instance as that object instance that invokes the firing of a shared transition. Again, this is achieved by use of object ID information.

Because IDs are used within the class, they will not conflict with different classes. For simplicity of discussion, we can just use ID1, ID2, .., IDn to denote some arbitrary objects. To model the fact that each object instance in a desired state can invoke the shared transition with equal opportunity, we adapt the idea of variable names from colored Petri nets. This means that for each original firing requirement associated with the input arc(s), we need a variable to indicate that one instance of an object class, in some state, is required to enable a shared transition. Similarly, this shared transition requires use of the same variable to indicate a possible state change for that same object instance. Note that a shared transition connects with different classes, so we must carefully choose the variable name. If we were to use the same variable name for different classes related to the same shared transition, then the same ID from different classes would be required to invoke this transition. But, since we might not want such a constraint, it is best to avoid the conflict of variable names by using unique variable names for different classes. Fortunately, this is not difficult. Since a class name is unique through the whole system, we can use variable names like *buffer_x*, and *buffer_y* instead of *x*, and *y*. Figure 4 shows our basic idea on a design using 3 producers, 4 consumers, and 2 buffers. Here B_x is a variable name that can refer to either of the two buffer object instances.

Recall that a SBOPN consists of SBOs, and a SBO consists of 7 components (refer to Definition 1 in Section 2). Therefore, we can consider defining instantiation rules in terms of these basic components. It is useful to note that the components of *Type*, *NG*, *States*, *sp* and *ST* are properties belonging to a whole class, so these components are not impacted by the instantiation rule. So, we only need to consider *SFM* and *STM* when we perform instantiation on a class-level model with an initial marking. For formalization, we regard the instantiation rule as a function. In this way, we can clearly define the necessary modifications to an initial marking, *SFM*, and *STM* due to the application of the instantiation rule. Let *f* be the instantiation rule.

1. Consider a token with a state value *s*. According to our discussion in previous sections, if we need to instantiate a class to *n* instances, then we have

   $$f(n, s) = \{(ID1, s), (ID2, s), ..., (IDn, s)\}$$

   For a marking *M*, we apply the above rule to all tokens in *M* and denote the result as *f(n, M)*.

2. Consider a state-filter mapping *SFM*. Because *SFM* is a mapping from an arc to a state-filter, we only need to consider state filters. Let $sf = \{s1, s2, ..., sm\}$ be a state filter.

   $$f(sf) = \{(Type\_x, s1), (Type\_x, s2), ..., (Type\_x, sm)\}$$

   We can denote this mapping as $f(sf) = (Type\_x, sf)$. So, for any $sfm \in SFM$, we define $f(sfm)$ by $(f(sfm))((p, t)) = f(sfm((p, t)))$, where $p \in P$, $t \in T$.

3. Consider a state-transfer mapping *STM*. Because *STM* is a mapping from an arc to a state-transfer pair, and in the pair, only the state-transfer function depends on tokens, we only need to consider state-transfer functions. Let $F_{st} = \{s_1 \rightarrow S_1, s_2 \rightarrow S_2, \ldots, s_m \rightarrow S_m\}$ be a state-transfer function.

   $$f(F_{st}) = \{ (Type\_x, s_1) \rightarrow (Type\_x, S_1), (Type\_x, s_2) \rightarrow (Type\_x, S_2),$$
   $$\ldots, (Type\_x, s_m) \rightarrow (Type\_x, S_m)\}$$

   We can denote this mapping as $f(F_{st}) = (Type\_x, F_{st})$. So, for any $stm \in STM$, we define $f(stm)$ by $(f(stm))((t, p)) = (p', f(F_{st}))$, where $p \in P$, $t \in T$, and $stm((t, p)) = (p', F_{st})$.

Now, we can identify the impact of the instantiation rule on a complete SBO and a SBOPN.

1. Let *f* be the instantiation rule and $S = (Type, NG, States, sp, ST, SFM, STM)$ be some SBO. To instantiate the SBO system (S, M_0) into *n* instances, we have $f(n, (S, M_0)) = ((Type, NG, States, sp, ST, \{f(sfm)|sfm \in SFM\}, \{f(stm)| stm \in STM\}), f(n, M_0))$.

2. Let *f* be the instantiation rule and let *SYS* be a SBOPN consisting of the following SBOs: $S_1, S_2, ..., S_m$. To instantiate the system model *SYS* with n_1 instances of S_1, n_2 instances of S_2, ..., n_m instances of S_m, we have

   $$f(N, SYS) = (f(n_1, S_1), f(n_2, S_2), ..., f(n_m, S_m))$$

Where $N = (n_1, n_2, ..., n_m)$

To help explain the application of the above rules, consider the class-level model in Figure 1. By applying the instantiation rules for the purpose of establishing a system with 3 producers, 4 consumers, and 2 buffers, we will automatically synthesize the instance-level model shown in Figure 4.

4. Analysis and State Marking Compression

Although we obtain a new model after applying the instantiation rule, the net firing rule requires almost no change. The general rule is that a shared transition is enabled if and only if for each input place, there is an instance whose current state is an element of the corresponding state filter. The instance's ID is not used to determine if the shared transition is enabled, but is used to make sure that after the shared transition fires, a token corresponding to the same instance will be put into the corresponding output place. In this context, "the same instance" means instances with the same ID, but maybe in different states. For example, in Figure 4, suppose the put transition is fired and the ID of the buffer instance that invoked this firing is ID1. So, before the firing, the state token is (ID1, *Empty*), while after the firing, the state token changes to (ID1, *Partial*). Note that the ID does not change, but the state changes from *Empty* to *Partial*.

Since the firing rule is almost the same, the two basic analysis techniques defined for single-instance SBOPN models [10] are still suitable for generalized, multiple-instance models. The first technique is an indirect method, based on unfolding a SBOPN model to create an equivalent CPN model. Then, CPN techniques and tools can be applied to aid the analysis. The second technique is a direct method, based on generation of a reachability graph directly from the SBOPN model. Let's first briefly discuss the issue of "unfolding" a SBOPN model to create a more "traditional" colored Petri net.

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 actually a colored Petri net is a folded model of an ordinary Petri net. Considering our SBOPN model, we can view it as a folded model of a colored Petri net with some constraints. By unfolding the state filters and state-transfer functions, the SBOPN becomes a colored Petri net. Then we can use further unfolding or special colored Petri net techniques to allow subsequent model analysis.

The unfolding process is quite straightforward. First, note that a state filter is a set of states. Yet, only a color value or variable is allowed in a traditional colored Petri net. So, we must first separate out each state in each state filter of the SBOPN model. Second, we must modify the non-deterministic state-transfer functions to form deterministic ones – traditional colored Petri nets do not directly support non-deterministic binding of

output token values. After we combine these two modifications, each transition in the SBOPN model (whose input and/or output arc has a non-default associated state filter and/or state-transfer function) will have been replaced by several new transitions that define a colored Petri net; the new net will no longer have state filters or state-transfer functions. For example, if we apply the unfolding process to the instance-level Producer, Consumer and Buffer SBOPN given in Figure 4, we can obtain the colored Petri net model shown in Figure 5. Notice the Buffer object. The state filter {(B_x, E), (B_x, P)}, associated with the arc (*p1*, *put*), has been separated into two transitions (B_x, E) and (B_x, P). And the state-transfer function F1 maps the two inputs (B_x, E) and (B_x, P) into three different outputs. As a result, the shared transition *put* has been unfolded into three transitions, *put1*, *put2* and *put3*, in Figure 5.

As we mentioned, it is also possible to derive a reachability graph directly form a source SBOPN model. Once the reachability graph of a SBOPN model is created, we have the opportunity to directly check for system properties, since we then have an explicit state-space view of the system. In other words, this technique opens up the possibility to apply various forms of model checking [16]. However, there is the well-recognized state-explosion problem associated with reachability graph generation. It exists for both the class-level and instance-level models. Since instance-level models purposefully increase the number of objects in the system, the state-exploration problem tends to be more serious in this case. In terms of state-space analysis, the primary contribution of our model is that it does not sacrifice inherent model reduction or state-space reduction techniques that exist, and are under study, for more "conventional" Petri nets or other state-transition models. But, in addition, because of the focus on object-based design, there is a potential for some simple analysis optimization that exploits natural symmetry among commonly instantiated object models, i.e., models derived from a common class model. Instances of an object are considered as symmetric since they exhibit identical individual behavior. Thus, two system states (composite states) are symmetric if they become identical under the condition that process (in our case, object) identities are ignored. Since the basic idea of this optimization is simple, we explain it in an informal way using the case of multiple buffer objects as an example.

After we instantiate a SBOPN to an instance-level model, the marking of this new model is no longer just the distribution of state tokens of the SBOPN. Instead, a marking can be viewed as the distribution of tokens belonging to object instances. We can observe a few facts about such markings. First, the marking associated with an object is the tokens from each instance model as defined for the object class. In Figure 4, the marking associated with the buffer object is ((ID1, E), (ID2, E)). Because the ID number can be indirectly represented by the order of states, we can omit the IDs. So, for this buffer, we have marking (E, E). Second, the marking of the whole system is the combination of markings of each class. The marking of the system in Figure 4 is ((R, R, R)$_{\text{Producer}}$, (R, R, R, R)$_{\text{Consumer}}$, (E, E)$_{\text{Buffer}}$).

As is expected, the set of reachable markings increases exponentially as the number of instances increases. In Figure 4, if we have *n* buffers, then we have 3^n different markings. Clearly, such an exponential growth will make reachability-based analysis very difficult. Fortunately, for some special properties, it is possible to reduce the state space, and also retain the ability of analysis. For instance, for deadlock analysis one can use various analysis methods like net reduction, atomic expression evaluation, stubborn set method, sleep set method, and net symmetry [17]. In particular, the net symmetry method achieves reduction in the size of reachability graphs by detecting and exploiting symmetries that might exist in a Petri net [18]. In [17], semantic information associated with place nodes in models (of Ada tasking programs) is used to reduce substantially the time required by the symmetry algorithm.

An important observation is that some symmetry information becomes explicit once we instantiate a SBOPN because of the object-oriented nature of the system. So we can use this natural symmetry information to compress the representation of markings without a need for a separate symmetry detection phase. Actually, it is intuitive that all instances of one class can be considered as symmetric. To make this clear, we use Figure 4 as an example. As a first observation, note that since the states associated with any instances of a producer and a consumer never change (both are always in the *Ready* state), for simplicity we can use the markings of buffer instances to create the system state space. So, we can interpret the initial marking of Figure 4 as (E, E), meaning that both instances are in the *Empty* state. Now, let us consider the two markings: (E, P) and (P, E). The marking (E, P) means that instance 1 is in the *Empty* state and instance 2 is in the *Partial* state, while (P, E) means that instance 1 is in the *Partial* state and instance 2 is in the *Empty* state. Considering the symmetry of the buffer instances, both of these two cases reflect the same essential fact: there is one buffer in the *Empty* state, one buffer in the *Partial* state, and no buffer in the *Full* state. The key observation is that the information on the number of instances in a specific state is sufficient for our analysis; we do not need to know the state of each specific instance. So, we can compress the two markings (E, P) and (P, E) into one marking, (1, 1, 0), which directly conveys that there is 1 object instance in the *Empty* state, 1 instance in the *Partial* state, and 0 instances in the *Full* state.

The order in which states are represented in the compressed marking is arbitrary, but must be consistent among all markings. Here we have selected the state-ordering as (Empty, Partial, Full). Using the same technique, we can compress the following 9 markings: (E, E), (E, P), (E, F), (P,E), (P,P), (P,F), (F,E), (F,P), and (F,F). These are all markings of this system when there are two buffer instances. After compressing, we only have 6 (= 4!/ (2! * 2!) markings: (0, 0, 2), (0, 1, 1), (0, 2, 0), (1, 0, 1), (1, 1, 0), and (2, 0, 0). For a system design using 3 buffers, the state space size will reduce from 81 states to 10 states. The general case is described by the following property:

Compression Property: Let *N* be a SBO with *m* states and *n* instances. Before compression, the number of possible marking is m^n. After compression, the number is *((n+m-1)! / ((m-1)! * n!))*.

Proof:

Without compression, we consider instances to be distinguishable. So, if there are *n* instances and each instance can be in any of *m* different states, there are m^n different composite states.

With symmetry-based compression we consider instances to now be indistinguishable. If we have *n* indistinguishable instances and each instance can be in any of *m* different states, then the "original" set of m^n composite states can be mapped into a smaller set of compressed states. Any two composite states that are equivalent when instance identities are ignored map to the same compressed state.

The problem of determining the number of different compressed states is equivalent to the problem of determining the number of possible ways one can distribute *n* indistinguishable items into *m* distinguishable groups. This is a classic combinatorial problem. To solve this, consider that each possible individual distribution of items to the set of *m* groups can be represented by a string consisting of *n* items plus *m-1* separators, i.e., a string of *n+m-1* symbols, where ^ is a special separator symbol. (For example, assuming four groups, the string "AB^C^DEF^G" can represent the following distribution: items A and B to group 1; item C to group 2; items D, E, and F to group 3; and item G to group 4. Likewise, the string "^^ABCD^EFG" can represent the distribution of items A, B, C, and D to group 3, and items E, F, and G to group 4.)

Our problem is to determine the number of combinations of such strings, where the item symbols are indistinguishable. First, we can note that the number of permutations of strings that consist of *n* items and *m-1* separators is equal to *(n+m-1)!*. But, among these permutations there are a number that represent identical distributions when the items are indistinguishable. In other words, for each combination (based on indistinguishable items), there are *k* corresponding permutations, where *k* is the number of permutations of strings that can be formed using symbols for the indistinguishable items and indistinguishable separators. In this case, *k* is equal to *n! * (m-1)!* Therefore we have *(n+m-1)! / n!*(m-1)!* combinations of strings, each representing a way to distribute *n* indistinguishable items into *m* distinguishable groups.

E.O.P. (End of Proof)

One natural question is how we can create a reachability graph consisting of compressed markings. The basic steps for creating such a reachability graph are similar to those for a standard reachability graph generation.

First, we identify all enable transitions of a giving marking. Then we compare each resultant marking (reached by firing an enabled transition) to the markings already in the reachability graph. If a marking is new (i.e., not previously generated), we add it to the reachability graph. Otherwise, we do not add it. This process is repeated until no new marking is added to the graph. Now, to generate a state space of compressed markings, we employ on-the-fly uncompression and compression of markings. Thus, for each generated marking, *M*, we store the marking in its compressed form. But, before we determine all of the associated enable transitions, we uncompress *M* to a general marking, and before we compare a resultant marking, *N*, with markings already in the reachability graph, we must first compress *N*, since all markings in the graph are in compressed form.

5. Design Methodology and Case Study

In the previous sections, we introduced our SBOPN model, instantiation rules for instance-level models, and the state marking compression technique associated with instantiation to help reduce the complexity of analysis. We now step back and describe how our modeling capability fits into the framework of a general design methodology. Then we can present a case study that combines the concepts of this paper.

Two key elements in the development process for object-oriented modeling are the micro development process and the macro development process [19]. The micro process serves as the framework for an iterative and incremental approach to development, while the macro process serves as the controlling framework for the micro process. The micro process tends to track the following activities:

- Identify the classes and objects at a given level of abstraction
- Identify the semantics of the classes and objects
- Identify the relationships among the classes and objects
- Specify the interface and then the implementation of these classes and objects

Our modeling work is intended as an aid for the micro process for object-oriented modeling. In this context, we present the following high-level steps that outline a design process and a mapping for model generation:

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

Step 2: Determine the states of each object. Then for each object, determine the messages that each object can accept in each state, and identify the next state(s) that is active following the acceptance of such messages. Accepted messages define methods, which are mapped into shared transitions.

Step 3: Create a SBO model for each object's basic control flow based on the information from Step 2.

Step 4: Combine all SBO models derived in Step 3 to produce the system-level SBOPN model. The SBO models are implicitly connected via matched shared transitions.

Step 5: Determine the number of instances of each object and the initial state of each instance.

Step 6: Use net simulation and state marking compression, and/or other state-space reduction techniques, as a basis for analysis of the modeled system.

5.1 Case Study: Model Development

In the recent years, the Internet has become a part of general life. At the same time, 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, we discuss a case study that focuses on design and analysis of a simple e-commerce architecture. In considering the design of a system, one interesting question is whether properties associated with a class-level design will be preserved in an instance-level design. If so, the analysis of the modeled systems can stop at the class-level. Unfortunately, the answer to this question is "no." Thus, it becomes necessary to support analysis of a system in terms of both class-level design and instantiation-level design. In this section, we will present an example, whose class-level design is deadlock-free, but which has a deadlock in the instance-level. Because of the instantiation rules of SBOPN, it is possible to do analysis for instance-level design.

On-line shopping is different from brick-and-mortar shopping, i.e., shopping in physical stores. Companies need to obtain not only order information from customers, but also other supplementary information, such as credit card information and shipping information. For convenience, most on-line companies, such as Amazon.com, allow customers to input this auxiliary information at various times during the order process, i.e.,

before or after product data selection. Of course, to complete a shopping transaction, both the basic order information and supplementary information are needed.

To model such a web-shopping system, a *user-interface* class is needed to handle user input/output, and an *information* class and an *order* class are needed to handle customer information and order information, respectively. Finally, a *cashier* class is needed to handle billing. To simplify our example, we omit login and logout processes because they do not significantly affect the properties of the system. As a result, we view customers are always ready to do shopping.

An SBO design of the *user-interface* class is shown in Figure 6. Note that customers can start by either beginning an order, or inputting user information. But only after both actions are taken, will this class be able to receive notification that a charge has occurred; then it can release the customer information and end the order. SBO models of the *information* class and *order* class are shown in Figure 7 and Figure 8, respectively. The initial state for both of these object classes is the *Ready* state. After receiving customer-specific information, the *information* class waits for a *Charge* method invocation. Then it waits for a release action from the *user-interface* class. After receiving an order, the *order* class waits for an invocation of its *Invoice* method. Then it waits to become released by the *user-interface* class. The *cashier* class, whose SBO model is shown in Figure 9, is also initially in the *Ready* state. After the *cashier* class receives an invoice, it charges the account, creates a bill, and then notifies the *user-interface* class that the charge is complete. The final system-level SBOPN model of the system design is shown in Figure 10.

5.2 Case Study: Design Analysis

First, we discuss analysis for a design that uses only one object for each of the class models – recall that we refer to this as a class-level model. The reachability graph of the web-shopping system is shown in Figure 11. For each marking, the object states are with respect to the following object ordering: *user-interface, information, order*, and *cashier*. From the graph, it is obvious that the system is both live and safe for the class-level design. Of course, existing Petri net algorithms can be used to verify such properties in the case of a more complex example.

Now, we would like to perform analysis on the system design in terms of multiple instances. To illustrate this, let us assume that the *user-interface* class has two instances and the other classes have only one instance. With this information, we can apply the instantiation rules introduced in Section 4 to the SBOs of each class to

get the instance-level SBOs. Figure 12 shows the SBO model corresponding to two instances of the *user-interface* class. The other SBO models are similar.

To analyze the system-level SBOPN model, we use the direct reachability graph approach. Note that the algorithm for reachability graph generation is still suitable for SBOPNs with multiple instances. In Section 4, compact markings were introduced to reduce the complexity of such reachability graphs. Here, we use this compact marking idea to simplify the analysis. The initial marking is *((R, R), R, R, R)*, which means that both instances of the *user-interface* class are in *Ready* state, as are the instances of the other classes. This state is uncompressed, resulting in the marking *(((ID1, R), (ID2, R)), R, R, R)*. The enabled transitions are *Input Information* and *Begin Order*. If these two transitions fire under the control of the first instance of the *user-interface* class, then the resultant markings are *(((ID1, INO), (ID2, R)), H, R, R)* and *(((ID1, ONI), (ID2, R)), H, R, R)*, respectively. If, instead, these transitions fire under the control of the second instance, the resultant markings become *(((ID1, R), (ID2, INO)), H, R, R)* and *(((ID1, R), (ID2, ONI)), H, R, R)*, respectively. Both *(((ID1, INO), (ID2, R)), H, R, R)* and *(((ID1, R), (ID2, INO)), H, R, R)* can be compressed to create *((R, INO), H, R, R)*. Likewise, both *(((ID1, ONI), (ID2, R)), H, R, R)* and *(((ID1, R), (ID2, ONI)), H, R, R)* can be compressed to create *((R, ONI), R, H, R)*. Let us focus on the marking *M=((R, ONI), R, H, R)*. The first component of M, *(R, ONI)*, identifies that one instance of the *user-interface* class is in the *Ready* state, while the other instance is in the *OrderNoInformation* (*ONI*) state. *ONI* represents that a customer has begun an order, but has not yet input the customer information. The "*R, H, R*" component of marking *M* identifies that the instances of the *information* class, the *order* class, and the *cashier* class are in the states *Ready*, *Hold*, and *Ready*, respectively. Now, the enable transitions are *Input Information* and *Invoice*. Since there is only one instance of the *order* class and the *cashier* class, the marking that results from firing *Invoice* is clear: *((R, ONI), R, WE, RC)*. But, the transition *Input Information* can be enabled by both instances of the *user-interface* class, since both of their states are elements in the state filter *{(UI_x, R), (UI_x, ONI)}*. Therefore, there are two possible resultant markings as a result of firing the transition *Input Information*: 1) *((INO, ONI), H, H, R)*, if the *user-interface* instance in the *Ready* state is involved in the firing, and 2) *((R, OAI), H, H, R)*, if the *user-interface* instance in the *ONI* state is involved. After repeating a similar process to all new markings, we obtain the reachability graph shown in Figure 13. As we can see in Figure 13, there are 13 markings in the reachability graph generated by the compact marking technique. Twelve of the markings (system states) have different states for the two instances of the *user-interface* class. Because the two instances are totally symmetric and the other classes only have one instance, for any compact marking M, there will be two standard (uncompressed) markings. Thus, for this example, there will be 25 system states in the "standard" reachability graph and the ratio of compact marking states to standard states is almost 2:1.

For this design, the reachability graph shows that there is a deadlock in the system. Thus, we have identified a potential for deadlock when there are two instances of the *user-interface* class and one instance of the other classes. The deadlock marking is *((INO, ONI), WR, WE, RC)*. In this marking, one instance of the *user-interface* has already input customer information, but not yet input order information; while the other instance has input order information, but not yet input customer information. However, neither instance can continue the shopping transaction because both the *information* instance and the *order* instance are waiting to be released, i.e., they are not in the *Ready* state. One potential solution to resolve this deadlock is to fix the order of inputting customer information and order information. But this reduces the convenience of the basic design. Another possible solution is to bind one *information* instance and one *order* instance together and assign them to one *user-interface* instance**.** This solution reduces the reuse of class models, but it can prevent deadlock and support convenience as in the basic design.

6. Conclusion and Future Work

In this paper, we presented a Petri net based notation for expressing high-level design of distributed-object software. The formal notation incorporates general instantiation as an aid to development of scalable design specifications. But, the use of multiple object instances will tend to increase the state space of the model, further adding to the complexity of analysis. Fortunately, we can exploit the symmetry among object instances to reduce the state space, while retaining important properties of the model. We presented this concept in terms of a marking compression technique.

As we showed in the case study, a system can be live in terms of its class-level design, but yet have deadlock in the case of multiple-object instances. Thus, it is important to analyze a system not only at the class level, but also the instance level. Most other net-based object-oriented formalisms do not directly support design or analysis for generalized instantiation. One open problem is how to effectively determine the number of instances that are necessary to analyze some instance-level design. This is similar in flavor to the standard problem of test case selection in software testing.

References

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

[2] Burch, J. R., Clarke, E. M., McMillan, K., and Dill, D., "Symbolic Model Checking: 1020 States and Beyond," *Informatics Computing*, Vol. 98, No. 2, June 1992, pp. 142-170.

[3] 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.

[4] D. Buchs and N. Guelfi, "CO-OPN: A Concurrent Object-Oriented Petri Net Approach," *Proceedings of the12th Int. Conf. on the Application and Theory of Petri Nets*, Denmark, in Lecture Notes in Computer Science, Springer-Verlag, 1991.

[5] 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.

[6] M. V. 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.

[7] C. A. Lakos and C. D. Keen, "LOOPN - Language for Object-Oriented Petri Nets," *Proceedings of the SCS Multiconference on Object-Oriented Simulation, Simulation Series*, Vol. 23, No. 3, Anaheim, California, 1991, pp. 22-30.

[8] A. Perkusich and J. de Figueiredo, "G-Nets: A Petri Net Based Approach for Logical and Timing Analysis of Complex Software Systems," *Journal of Systems and Software*, Vol. 39, No. 1, 1997, pp. 39-59.

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

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

[11] D. Hammer, A. Hanish, and T. Dillon, "Modeling Behavior and Dependability of Object-Oriented Real-Time Systems," *Journal of Computer Systems Science and Engineering*, Vol. 13, No. 3, May 1998, pp. 139-150.

[12] A. Hanish and T. Dillon, "A Tool for Object-Oriented Dynamic Modeling," *Proceedings of the 2nd International Symposium on Object-Oriented Real-Time Distributed Computing* (ISORC), 1999, pp. 111-116.

[13] 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.

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

[15] 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.

[16] E. Clarke, O. Grumberg and D. Long, “Verification Tools for Finite-State Concurrent Systems,” In *A Decade of Concurrency - Reflections and Perspective,* Lecture Notes in Computer Science, Vol. 803, 1994.

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

[18] P. H. Starke, “Reachability Analysis of Petri Nets Using Symmetries,” *Syst. Anal. Model. Simul*., Vol. 8, 1991, pp. 293-303.

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

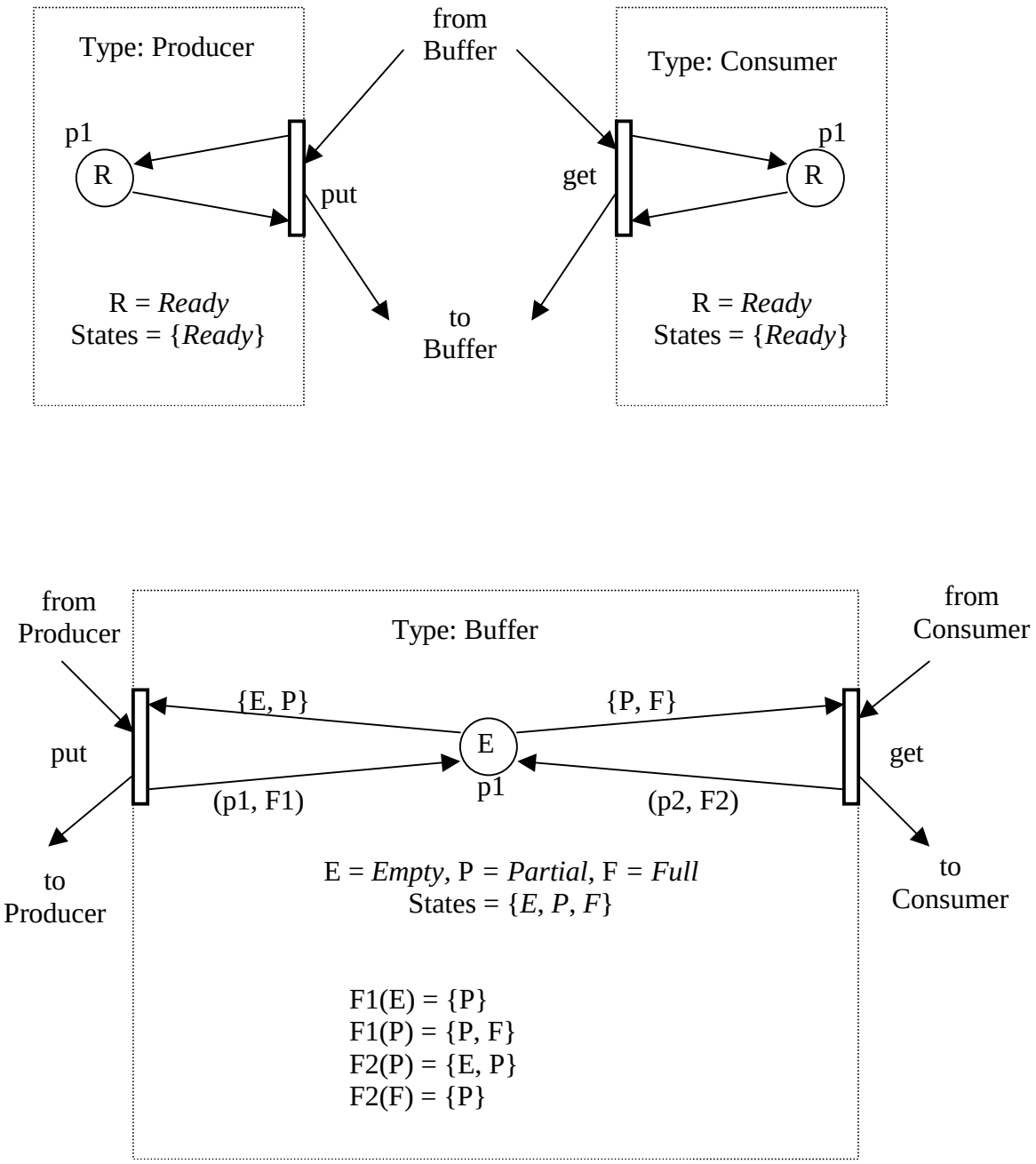

Figure 1. A SBOPN for the producer, consumer, and buffer system

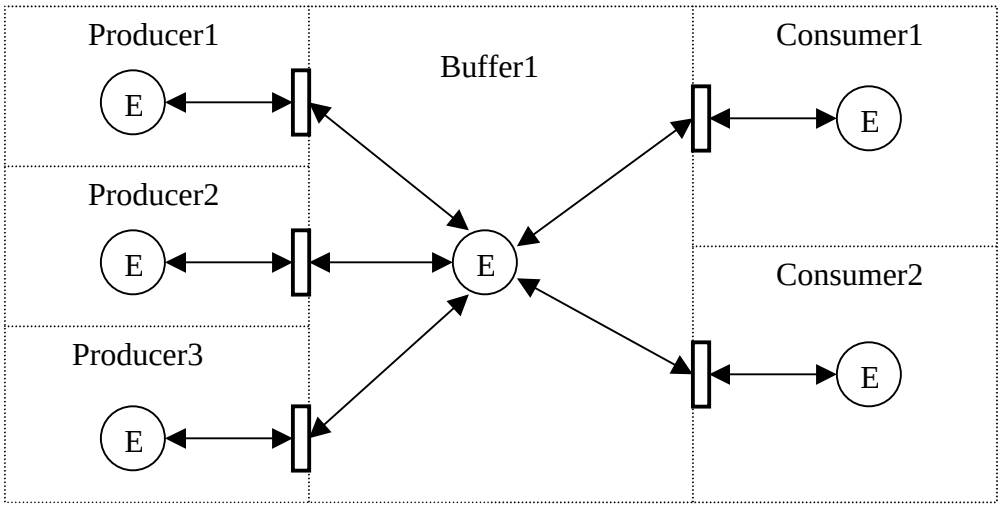

Figure 2. Multiple Producers and Consumers with replicated shared transitions

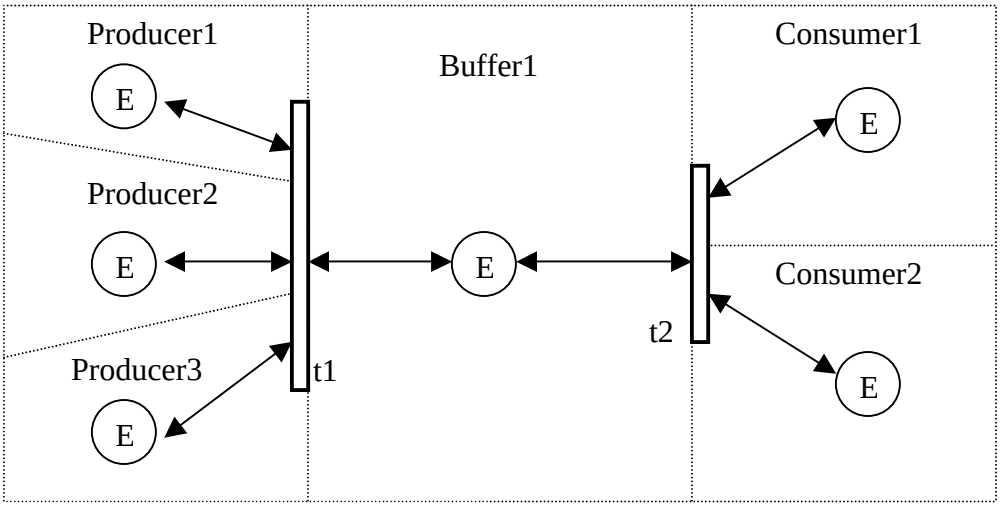

Figure 3. Multiple Producers and Consumers with common shared transitions

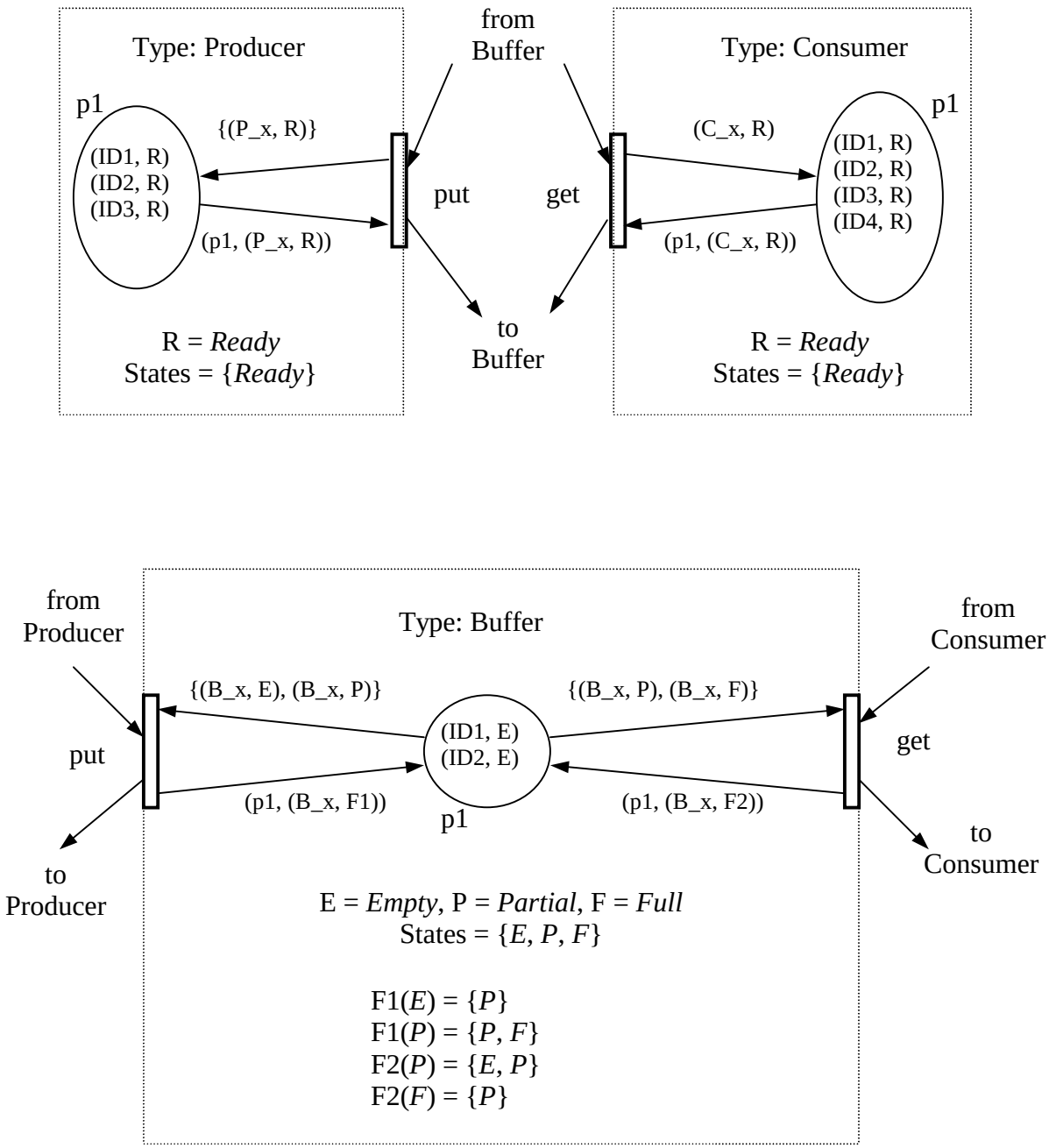

Figure 4. A SBOPN after applying instantiation rules

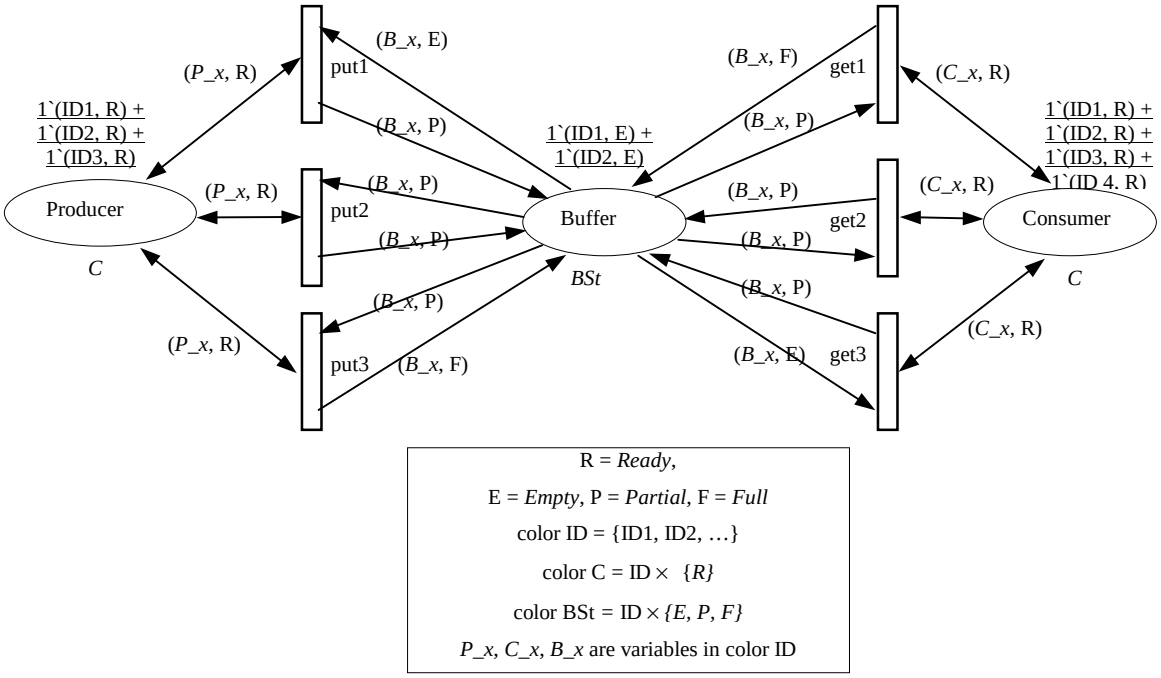

Figure 5. A CPN obtained by unfolding the model in Figure 4

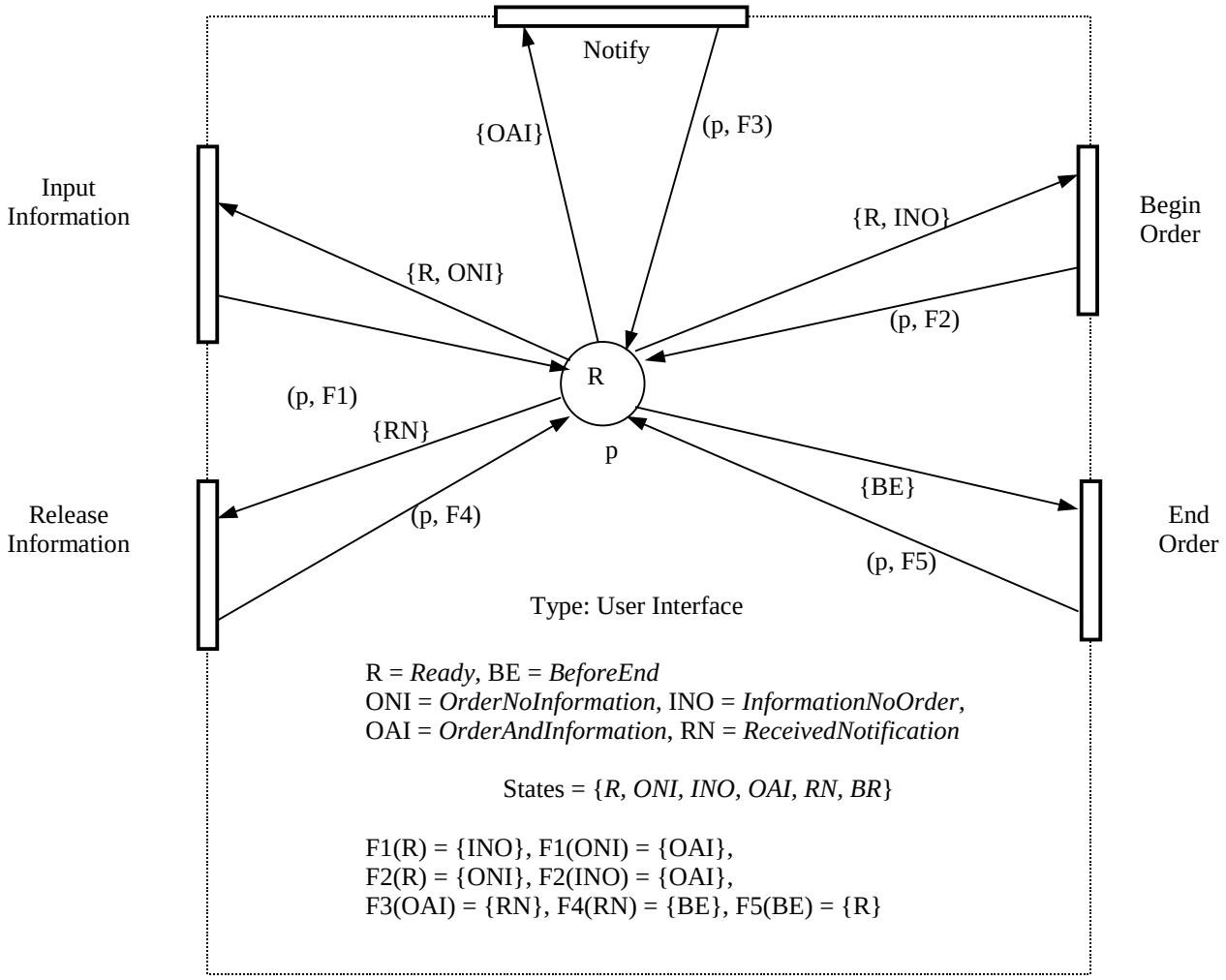

Figure 6. SBO of “User Interface” Class in the web-shopping system

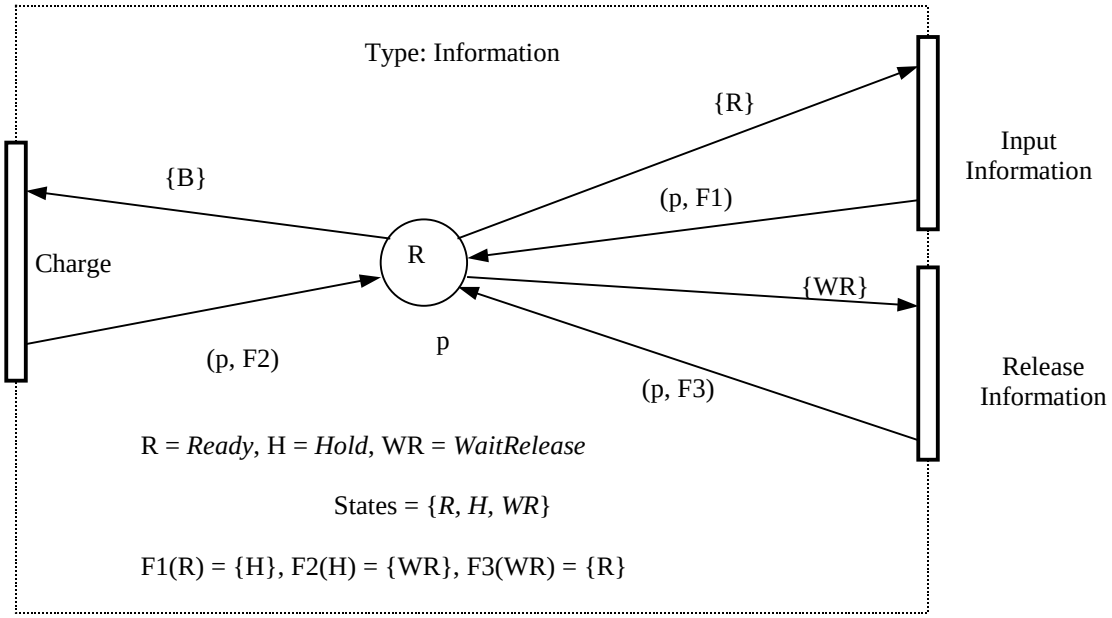

Figure 7. SBO of “Information” Class in the web-shopping system

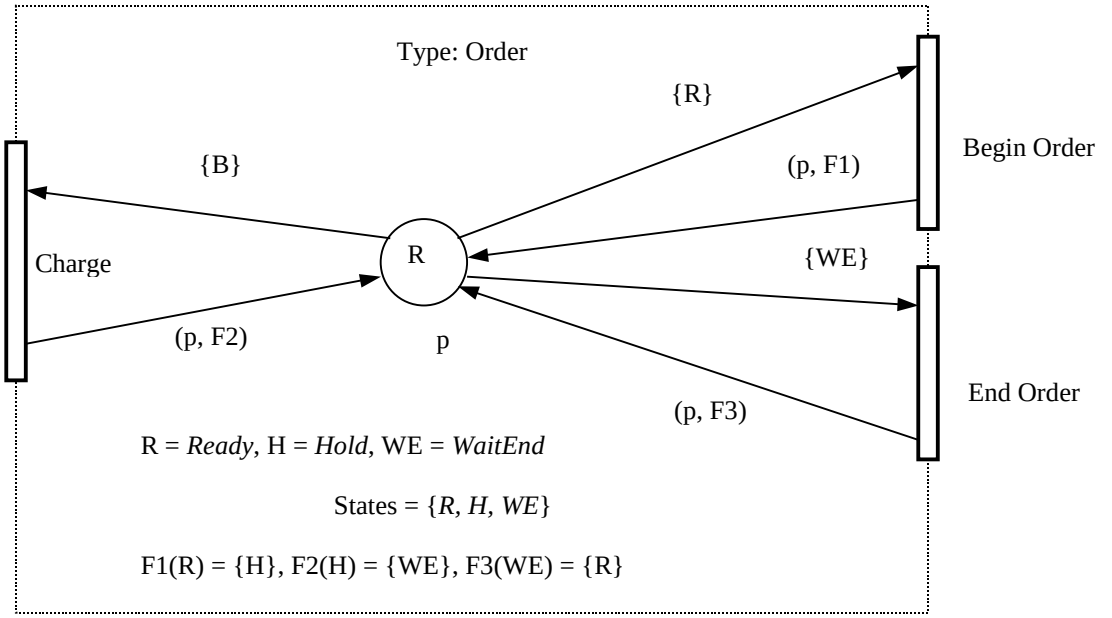

Figure 8. SBO of “Order” Class in the web-shopping system

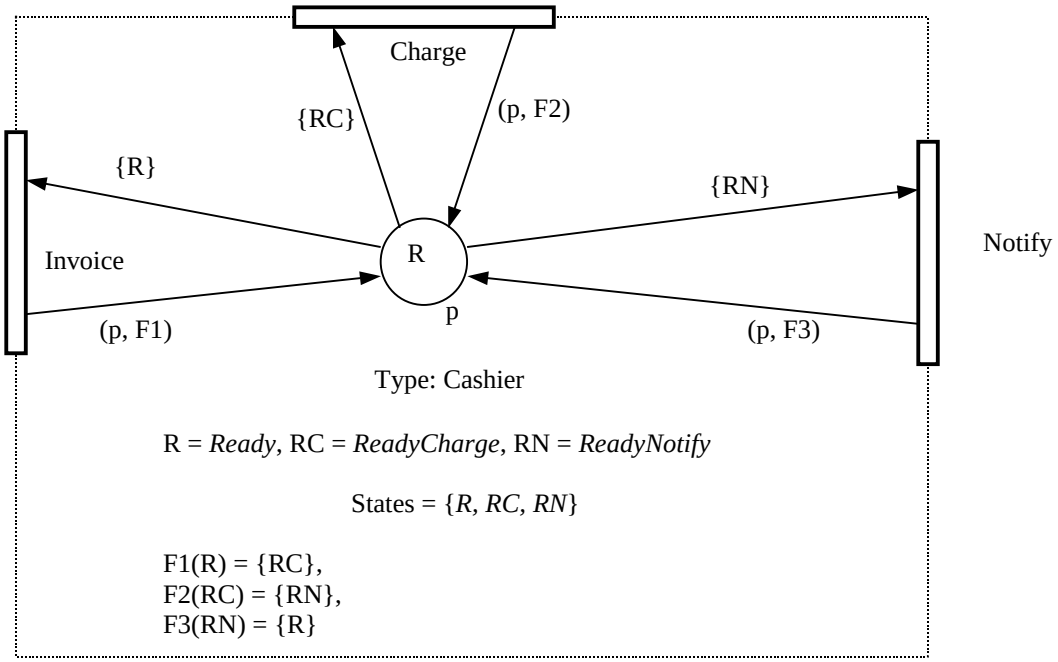

Figure 9. SBO of “Cashier” Class in the web-shopping system

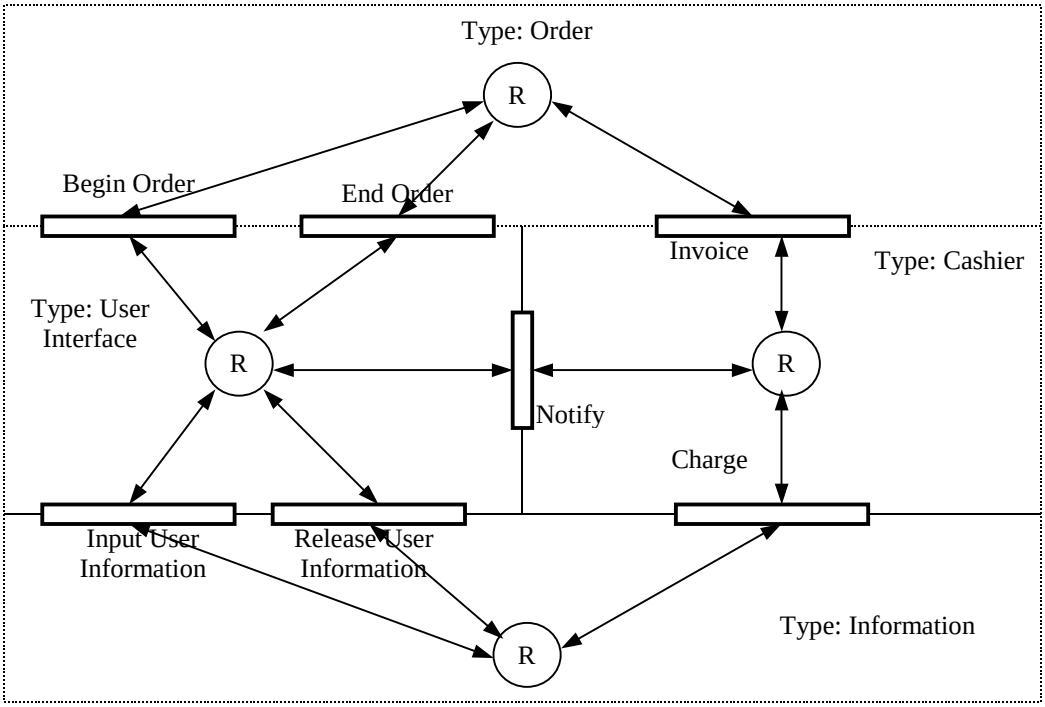

Figure 10. SBOPN of the web-shopping system

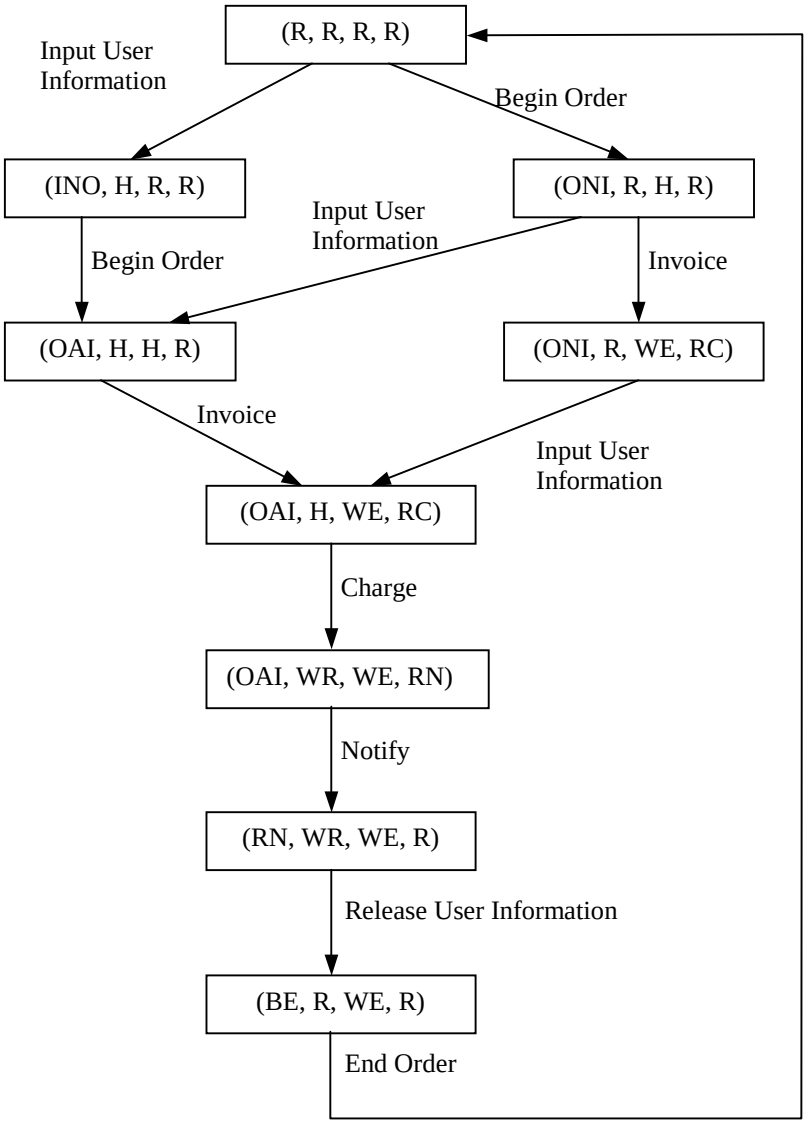

Figure 11. Reachability graph of the web-shopping system (class-level)

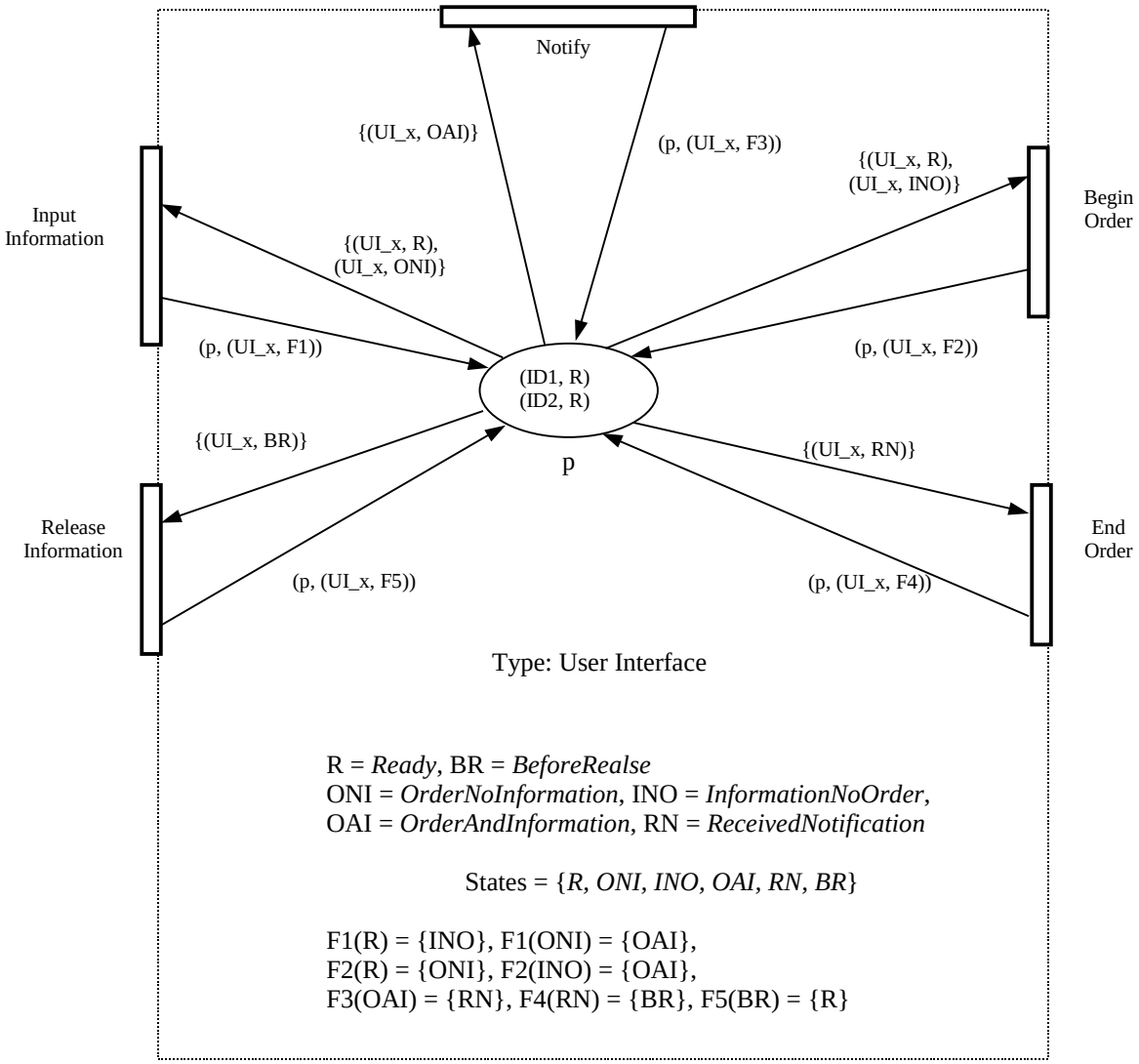

Figure 12. SBO of two instances of “User Interface” Class

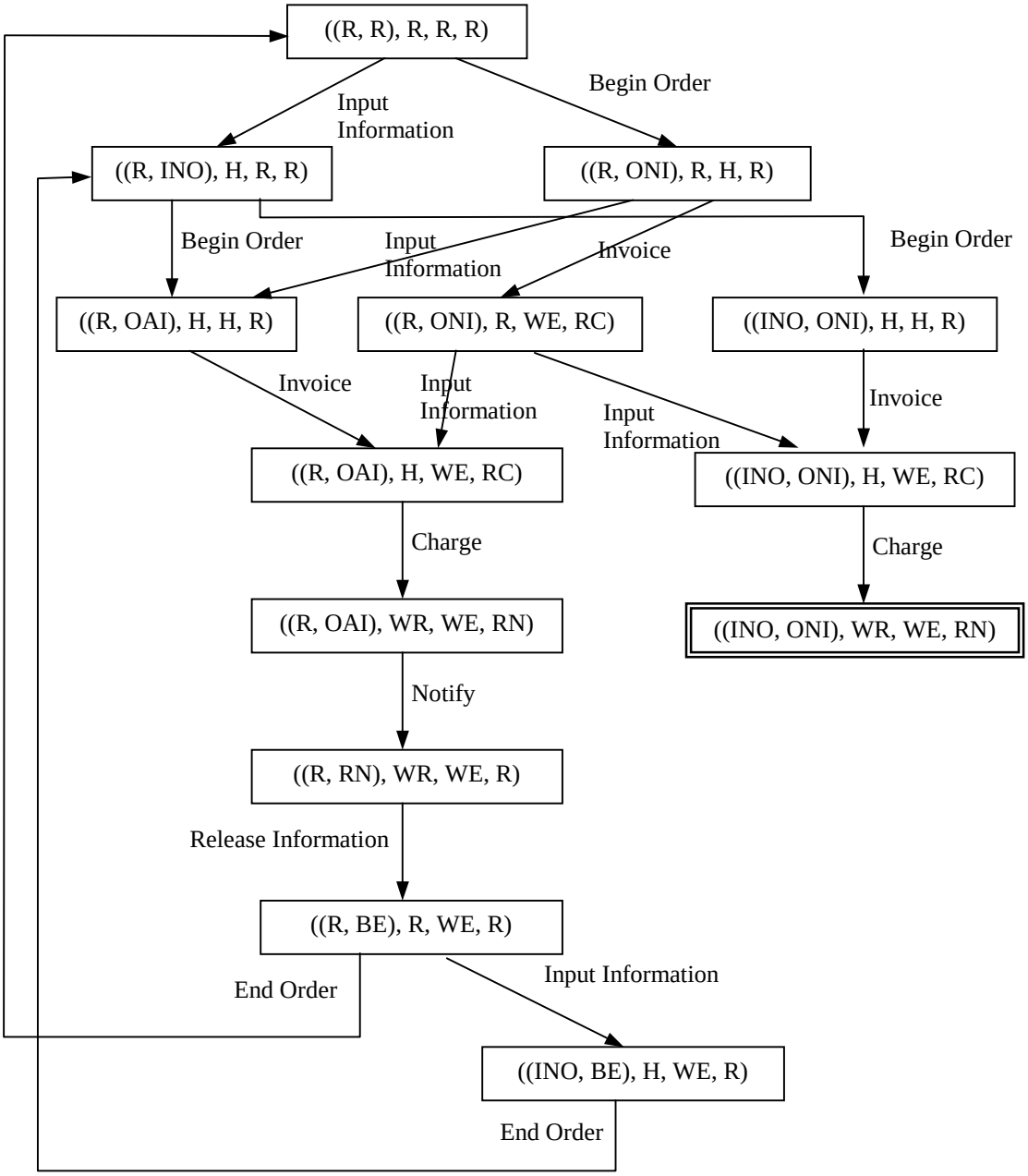

Figure 13. Reachability graph of the web-shopping system with two instances of the “User Interface” Class