UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis

John Anil Saldhana and Sol M. Shatz
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
{jsaldhan,shatz}@eecs.uic.edu

Abstract. UML, being the industry standard as a common OO modeling language needs a well-defined semantic base for its notation. Formalization of the graphical notation enables automated processing and analysis tasks. Object Petri nets (OPN) can provide a formal semantic framework for the UML notations plus the behavioral modeling/ analysis strength needed by system designers. This paper describes a methodology to develop a Petri net model of a system, by deriving a form of OPN called as Object Petri Net Models (OPMs) from UML Statechart diagrams and connecting them using UML Collaboration diagrams. Then, the single system-level Petri net can be analyzed by formal Petri net analysis techniques.

1. Introduction

The Unified Modeling Language (UML) specifies a modeling language that incorporates the object-oriented community's consensus on core modeling concepts. The behavioral specifications in UML are based on State Charts [Harel 87]. Statechart diagrams [UML99] in UML specify the sequences of states an object goes through during its lifetime in response to events, together with its responses to events. A Statechart diagram models the behavior of a single object over its lifetime [UML 99]. An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages (or events) that may be dispatched among them. A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.

Petri nets [Murata 89] are a formal and graphical appealing language, appropriate for modeling systems with concurrency. Colored Petri nets (CPNs) [Jensen 92] are a generalization of ordinary PNs, allowing convenient definition and manipulation of data values. CPNs also have a formal, mathematical representation with a well-defined syntax and semantics.

In this paper, we describe a methodology to derive a system-level CPN from UML models. The purpose is to create a formal model that can be subjected to various Petri net analysis and simulation techniques. This can aid in the validation of UML behavioral specifications to detect concurrency-related properties such as deadlocks.

2. Motivation and Related Work

Effective modeling of complex concurrent systems requires a formalism that can capture essential properties such as nondeterminism, synchronization and parallelism. Petri nets offer a clean formalism for concurrency, however lack thorough modularization techniques. Object orientation offers formalism for highly reusable and modular systems, but lacks general concurrency features. There have been a number of attempts to combine Petri nets with Object Oriented concepts to profit from the strengths of both approaches (e.g., [Shatz 98], [Lakos 94], [Deng 93]).

Object Petri nets [Lakos 94] provide a formalism that extensively adopts object oriented structuring into Petri nets. The intention is to increase further the expressive comfort of Petri nets and thus make feasible the modeling of complex systems, and at the same time reaping the benefits of object orientation including clean interfaces, reusable software components and extensible component libraries. Object Petri nets retain the important property of being transformable into behaviorally equivalent CPNs so that the analysis techniques developed for CPNs can be applied.

We suggest that design knowledge can be captured and formalized by the methodology outlined in Fig. 1. The methodology can enable a UML designer to verify UML models. In this paper, we focus on the key step of deriving an Object Petri net (OPN) from UML diagrams. We start with UML models as created by a system designer (appropriate UML editors can be used to develop UML Statechart and Collaboration diagrams). In our methodology, Statechart diagrams are first converted to flat state machines. These state machines are then converted to a form of OPN called Object Petri Net Models (discussed in Section 3). Then the UML collaboration diagrams are used to connect these Object models to derive a single CPN for the system under study. Any standard CPN analyzer can be used to support analysis and simulation of the resulting CPN. This framework has the advantage of exploiting the mature

- This material is based upon work supported by the U.S. Army Research Office under grant number DAAD19-99-0350.

theory and tools for Petri nets and essentially hiding these details from the end–user.

A related research effort for validating UML models is the work on a tool called vUML [Lilius 99]. This tool uses SPIN, a model checker. A disadvantage is that open models (models that react to external entities) cannot be verified by the model checker. Hence vUML tries to convert these open models to closed models, if the external events do not carry any parameters. Our approach works well with external events with parameters because of the token structure provided. If events are not completely defined, vUML has event generators to generate the events. Our methodology uses similar event generators.

An attempt has been made at modeling interactive systems with hierarchical colored Petri nets [Elkoutbi 98]. Here the use cases define behavioral specifications and are converted to hierarchical colored Petri nets. Finally, in the area of performance estimation of systems, [Pooley 99] uses UML to derive Stochastic Petri net models.

3. Generating and Linking Object Petri Net Models

As mentioned earlier, our goal is to define a process for the generation of Petri nets from UML specifications. More specifically, we focus here on two key steps: 1) Generation of Object Petri Net Models (OPMs) of individual objects or components, and 2) Linking these object models to create a system-level model. A "standard" CPN represents the final system-level model. We assume that the objects respond to events created internally within the object and to events created externally by other objects in the environment. The collaboration diagrams indicate such event flows between objects. The Statechart diagram represents the lifetime of an object. An algorithm to model the lifetime of an object is given in [UML 99].

An object has a unique event-based behavior in relation to other objects in the environment. As we will see, this behavior can be modeled as a CPN. Also an object has a well-defined interface with its environment. To fully incorporate both these features, we propose a model for an object called Object Petri Net Model (OPM). The general architecture of an OPM model is shown in Fig. 2. We describe the structure of an OPM in the following paragraphs.

Definition: An Object Petri Net Model is a three tuple (LM, EGM, IA). LM is a model for the object's lifetime behavior (this model is specified by a CPN). EGM is the Event Generation /Management mechanism and IA is a set of interface arcs.

The *event generation/management* (EGM) mechanism supports the generation and routing of events in the object. The EGM mechanism defines three places - *in-place* (IP), *out-place* (OP) and *event-dispatcher* (ED) place. The IP models the flow of events into the object and OP models the flow of events out of the object. The ED has two functions – 1) Creating internal event tokens and 2) Routing the event tokens. The set IA defines two arcs. The *Input Transition Arc* (ITA) and *Output Transition Arc* (OTA) provide the interface of the object with the external environment.

The dynamic behavior of a system represented by any Petri net needs a token. In our methodology, we have an event token. The structure of the event token is of the form <*type, flag*>, where *type* denotes the event type (may contain a structure to represent parameters passed by external events) and *flag* is a variable to tell if an event is an internal event or an external event; flag equals *in* or *ex* respectively. The possibility of a multiple structured *type* of an event token helps us to handle events with parameters from external entities. To fully describe the EGM mechanism in OPM, we need a few definitions.

Definition**:** An *Event-Place* is a place of the OPM that can generate or store an event token.
Definition**:** An *Event-Dispatcher place* is an event-place that can generate event tokens of the form <*type, in*> and store event tokens of the form <*type, ex*>.
Definition**:** An *in-place* is an event-place that can store event tokens of the form <*type, in*> or <*type, ex*>.
Definition**:** An *out-place* is an event-place that can store event tokens of the form <*type, ex*>.

Once the *event-dispatcher place* generates an internal event token, it moves this token to the *in-place*, which interfaces to the lifetime model of the OPM. Also, since external event tokens are to be routed to the environment of the object, the event-dispatcher place forwards these tokens to the *out-place*. The *in-place* stores all event tokens that are used by the object and the *out-place* stores the event tokens that are to be sent to the environment of the system. All event tokens that are generated by the object (within the LM component of the model) are forwarded to the *event-dispatcher place* for appropriate dispatching. Because of the *event-dispatcher place*, internal event tokens of an object are used only internally and are never passed to the external environment.

An arc exists from the *in-place* to all transitions in the LM that model actions initiated by any events. An arc exists from a transition *t* in the LM to the *event-dispatcher place*, if an event is generated by the transition. For example, in Fig. 9, the transition from the "Initial Cooking Period" state to the "Cooking Interrupted" state takes place if, and when, the external event V3 has occurred. This transition generates an external event P2. The ITA carries external event tokens (<*type, ex*>) that

get deposited into the *in-place*. The OTA carries external event tokens that are removed from the *out-place*.

Now that we have defined and described the Object Petri Net Model (OPM), we proceed to describe the generation of such models from UML diagrams. In our approach, we derive the LM from the Statechart diagram of the object. Then, given a collaboration diagram showing the connection between objects, we connect the OPMs, to yield a single system-level CPN.

First, we describe the generation of OPMs from UML Statechart diagrams. A Statechart diagram [UML 99] contains states (simple and composite) and transitions (events and actions). A state has several parts, namely **Name** (textual string for identification, can be anonymous), **Entry/exit actions** (actions executed on entering and exiting the state respectively), **Internal transitions** (transitions that are handled without causing a change in state), **Substates** (nested structure of a state, can be sequentially active or concurrent substates) and **Deferred events** (a list of events that are not handled in that state, but are postponed and queued for handling by the object in another state).

A transition has five parts, namely **Source state** (state affected by the transition), **Event Trigger** (event whose reception makes the transition fireable), **Guard Condition** (boolean expression that is evaluated before a transition fires. The transition can fire only if the condition evaluates to true), **Action** (executable atomic computation), and **Target state** (state that becomes active after the completion of the transition).

A Statechart diagram example is shown in Fig. 3. In this diagram, the darkened ovals denote the *initial states*. The ovals labeled as *A, C, D, E* and *Receiving* are *states*. *Receiving* is an example of a *composite state*, and states C and D are nested states. There is a *triggerless* transition from state D to state A. On an *event* error, there is a state *transition* from state A to state E and the *action* associated with the transition is *printReport*. There is an *entry* action and an *exit* action for the state *Receiving*.

Since Statechart diagrams may contain hierarchical or nested states, effective conversion to Petri nets requires that the nested states be "flattened". Given a Statechart diagram that models the lifetime of an object, one can generate a Shlaer-Mellor Object life cycle [Shlaer 92], which is a flat state machine (containing just simple states and arcs). Then these flat state machines can be converted into a CPN that forms the LM of the OPM. Basically, state machine states are mapped onto Petri net places and state machine transitions are mapped onto Petri net transitions.

For the conversion of Statechart diagrams into flat state machines, we use the following conversion rules.

Definition: An *Event Generator* is associated with a state and is a function that generates an event token. An event generator that generates the specific event E is denoted as GEN(E).

***Conversion 1*:** The actions that are part of the transitions in the Statechart diagrams are mapped onto event generators. This conversion is illustrated in Fig. 4(a).

***Conversion 2*:** *Guard conditions* of the form **when(X)** or **after(X)** are mapped, by the environment or the object in question, onto an event token that is created when the condition becomes true. For example, **when(temp is greater than 200F)** or **after(2secs)** are all mapped to events that are generated when the named condition holds. *Internal events* and *deferred events* of a state in a UML diagram are ignored during conversion, as they do not contribute to the control flow of the object.

***Definition*:** GEN(X) and GEN'(Y), associated with state S, are event generators that generate an event X and event Y, respectively. GEN(X) generates the event X just before state S is entered and GEN'(Y) generates the event Y just after the state S is exited.

***Conversion 3*:** The *entry* and *exit* actions of a state can be mapped onto internal events that are generated before the object enters the state and after it leaves the state, respectively. See Fig. 4(b).

***Conversion 4*:** Composite states may involve sequential or concurrent substates. Sequential substates have a distinct feature that the state can exit from any of the substates. A concurrent composite state can move to the next state only when the concurrent sub paths within it are executed fully.

In Fig. 5 (a), we show the conversion of a sequential composite state X into a flat state machine. When the system is in states B, C or D, it is actually in states X/B, X/C or X/D. The state transition from composite state X to state E is possible from any of the substates B, C and D. Hence there are three arcs leading to state E. The corresponding Petri net is also shown. For the conversion of concurrent composite state, we need a few definitions.

***Definition*:** A *transition-state* in a finite state machine is a state that denotes a transition. It is equivalent to a Petri net transition.

***Definition*:** A *fork* state is a transition-state that splits an incoming transition into two or more transitions **terminating** in orthogonal target states.

***Definition*:** A *join* state is a transition-state that merges several transitions emanating from source states in different orthogonal regions, causing the execution to synchronize.

In Fig. 5(b), we show the conversion of a concurrent composite state X into a flat state machine. Here the state X doesn't transit to state Q until the goal states for both paths have been reached. This is modeled

by special transition-states *fork* and *join* in the Statechart diagram. In the corresponding Petri net, these states are mapped onto a transition. The corresponding Petri net is also shown.

If the naming of events is kept uniform in the Statechart diagrams and collaboration diagrams, we can connect the OPMs for each object to create a system-level CPN for the whole system.

As mentioned earlier, our goal is to generate OPMs for objects in a system and then connect them, using UML collaboration diagrams. We describe two examples to illustrate our approach.

Example 1: Consider an example of a one-minute microwave oven adapted from [Shlaer 92]. The oven is powered by a Power Tube and contains a Light Tube. Four events are possible: V1 (generated when the user pushes the on-button), V2 (the time-out when the cooking period has expired), V3 (opening the door) and V4 (closing the door). The arrival of V1 during a not-yet-completed cooking period has the effect of extending the period by another minute. We consider V2 as an internal event created in the Oven on a time-out. For the power tube, events are P1 (Energize) and P2 (Deenergize). For the light tube, the events are L1 (Turn On) and L2 (Turn Off). The state diagrams depicting the entire life cycle of the oven, the user, the light and the Power tube objects are shown in Figs. 6 and 7. Note that the event generators within a state of the form GEN(X) denote the entry action and those of the form GEN'(X) denote the exit action that occurs in a state (as given by conversion 3). The object life cycle models of the user, oven, power tube and the light bulb are shown in Fig. 8.

In our approach, we model the environment as a "super place" storing all the events generated by a subsystem to be used in other subsystems. Each object (subsystem) in the environment has a place (event-dispatcher place) for generating and storing tokens representing internal events. The corresponding OPMs for the microwave oven example are shown in Figs. 9 and 10. The collaboration diagram shows the sequences of events that are passed between the objects. In our approach, we assume for simplicity that the designer maintains uniformity in the naming of events in the Statechart diagrams and the collaboration diagrams. An event *e* in a Collaboration diagram corresponds to an event *e* in a Statechart diagram (associated with an event generator, denoted GEN(e)).

Given the connection between the objects by the collaboration diagram, the OPMs of the various objects can be combined together to form a system-level Petri net, as shown in Fig. 11. The complete details of the system-level Petri net are not shown because of the complexity of the diagram. The system-level CPN is obtained by connecting the input and output transition arcs of the OPMs to the general event place representing the environment. The resultant Petri net is a CPN, with event tokens as defined earlier in the section.

OPMs are linked by a place called as **Intelligent Linking Place (ILP).** Based on the UML collaboration diagram, a token tag is attached to each ITA of an OPM in the system. The token tag is of the form <TK>, where TK is a set of event types. Each token tag denotes the event tokens that the input transition arc can carry. In the case of the microwave example, the ITA of the OPM for the microwave oven will have the tag <{V1,V3,V4}>. A token tag of the form <{}> denotes that the ITA can carry no event tokens. Also, the ILP attaches an extra parameter, an *ObjectID* (a unique identifier assigned to each object in the system) to the token. The intent behind adding this identifier is to allow the appropriate tokens to be routed to appropriate objects. Let us consider the case when two or more events of the same type need to be sent to more than one unique object. To solve this problem, the ILP replicates the event tokens and based on the information from the collaboration diagram, attaches the *ObjectID* as described earlier.

Tokens that are generated in the environment (represented by an Event place in a system-level CPN) are passed to the ILP for appropriate dispatching. Any events that are shown by the user in a Statechart diagram, but not in a collaboration diagram, are considered as events local to that object and are created by the event dispatcher place of the OPM of the object in consideration.

Example 2: Consider an example of an ATM machine, dispensing cash to a user [UML 99]. The Statechart diagrams and the collaboration diagram are shown in Figs. 12 and 13. The description of the problem [UML 99] is as follows:

An ATM machine has three basic states: Idle (waiting for customer interaction), Active (handling a customer transaction) and Maintenance (perhaps having a cash store replenished). While active, the behavior of the ATM follows a simple path: Validate the customer, select a transaction, process the transaction and then print a receipt. After printing, the ATM returns to the idle state. While in the active state, the user might any time cancel the transaction, returning to the ATM to the idle state.

This behavior is modeled as a sequential composite state as in Fig. 13(a), which also shows the Statechart diagram of the ATM user. Using the conversion rules, described earlier, we obtain the object life cycle models shown in Fig. 14. Finally, the OPMs for the objects are shown in Fig. 15. The collaboration diagram shows how the objects interact with each other. All named events in the Statechart diagrams, which do not show up on the collaboration diagram, form the internal events of that respective object. The collaboration diagram is shown in Fig. 16. The system-level CPN obtained after linking of

the OPMs is shown in Fig. 17.

If the system is a *closed system* with no interaction with the external environment, we do not need the super place representing the environment. For a closed system, the EVENT place that represents the environment is deleted.

4. Conclusions and Future Work

In this paper, we have presented a methodology to support formal validation of UML specifications. The main idea is to generate a PN model for UML components to allow use of existing net analysis techniques. We defined a generic form of an OPN model, Object Petri Net Models (OPM), and discussed two key activities: 1) Generation of OPMs of individual objects or components, and 2) Linking these object models to create a system-level model. As described, the OPMs help us to model Open Systems as well as Closed Systems, which is lacking in earlier related research.

One area for future research is to investigate the use of UML use case diagrams in our approach to strengthen the behavioral modeling and analysis. Another direction for future research is to explore ways to present the system-level analysis and simulation results to the user. Since the methodology calls for a UML architect to provide the input specifications, it is only reasonable for the output results to be in a form that is meaningful to that user. Thus, we will investigate ways to map from the CPN analysis results back to the UML specifications.

5. References

1. [Deng 93] Y.Deng et.al. "Integrating Software Engg Methods and Petri nets for the specification and prototyping of complex information systems". *Application and Theory of Petri nets 1993, 14th International Conference proceedings*, Chicago, pp 203-223, June 1993.
2. [Elkoutbi 98] M.Elkoutbi and R.F.Keller. "Modeling Interactive Systems with Hierarchical Colored Petri Nets". *Proc. of the Conference on High Performance Computing*, 1998. April 6-9, Boston.
3. [Harel 87] David Harel. "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming 8 (1987)*, pp. 231-274.
4. [Jensen 92] K. Jensen, *Coloured Petri Nets*, Vol 1: Basic Concepts, Springer-Verlag 1992.
5. [Lakos 94] C.A.Lakos. *Object Petri nets – Definition and relationship to colored nets*. Tech Report TR 94-3, Computer Science Dept, University of Tasmania.
6. [Lilius 99] Johan Lilius and Ivan Paltor. *vUML a tool for verifying UML models*. Technical Report 272, Turku Centre for Computer Science TUCS, 1999.
7. [Murata 89]. T. Murata, "Petri Nets: Properties, Analysis and Applications", *Proceedings of the IEEE*, Vol.77, No.4 pp.541-580, April 1989.
8. [Pooley 99] Rob Pooley and Peter King. "Using UML to derive Stochastic Petri net models". *Proceedings of the fifteenth annual UK Performance Engineering Workshop*, pp 45-56, July 1999.
9. [Shatz 98] Shatz et.al. "Applying an Object-Based Petri Net to the modeling of Communication Primitives for Distributed Software". *Proc. of the Conf. on High Performance Computing*, 1998.
10. [Shlaer 92] Shlaer and Mellor. *Object Life Cycles – Modeling the world in states*, Yourdon Press, Prentice Hall. 1992.
11. [UML 99] Booch et.al. *The Unified Modeling Language User Guide*, Addison-Wesley.

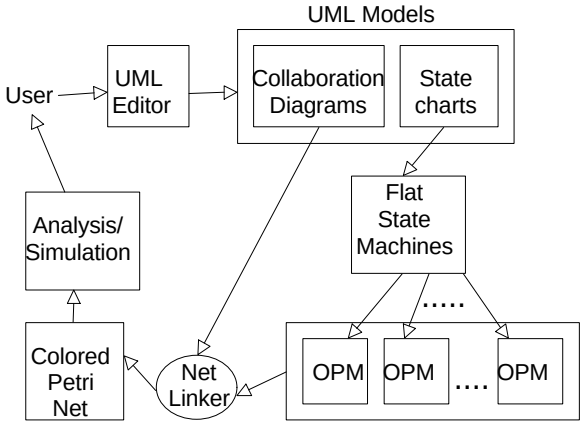


Fig. 1 Block diagram of the proposed methodology

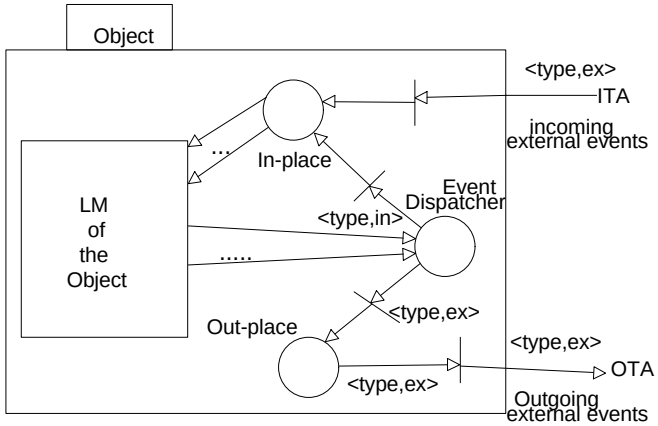


Fig. 2 Generic Object Petri Net Model (OPM).

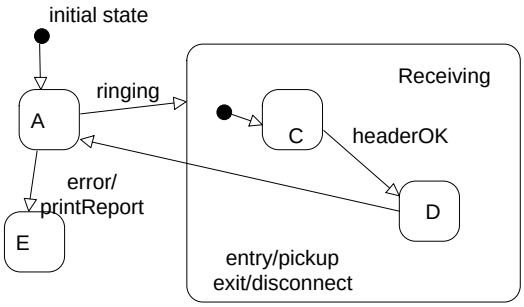


Fig. 3 A Statechart diagram example.

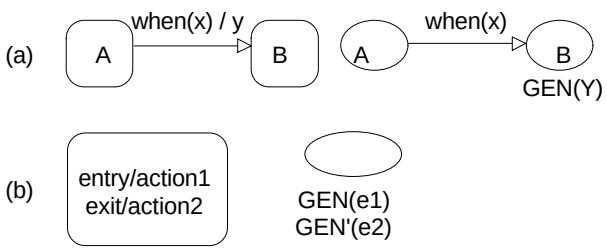


Fig. 4 (a) Conversion of action Y of an arc to GEN(Y).
(b) Mapping of entry/ exit actions to GEN / GEN'.

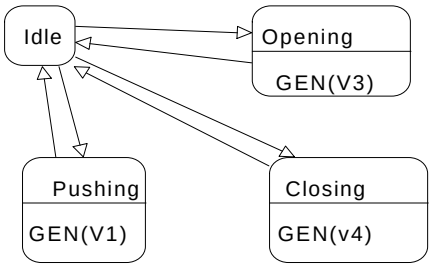


Fig. 6 Statechart diagram of the user, for the microwave oven example.

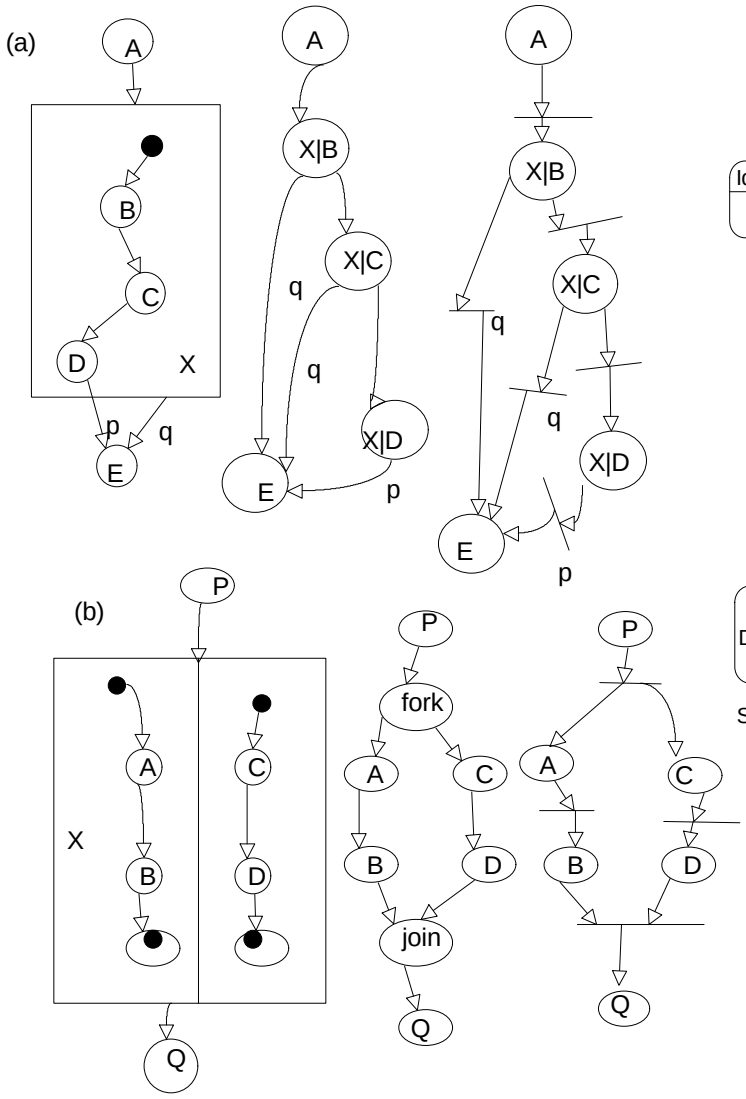


Fig.5(a) Conversion of { (a)sequential (b)concurrent } composite states X to flat state machines.

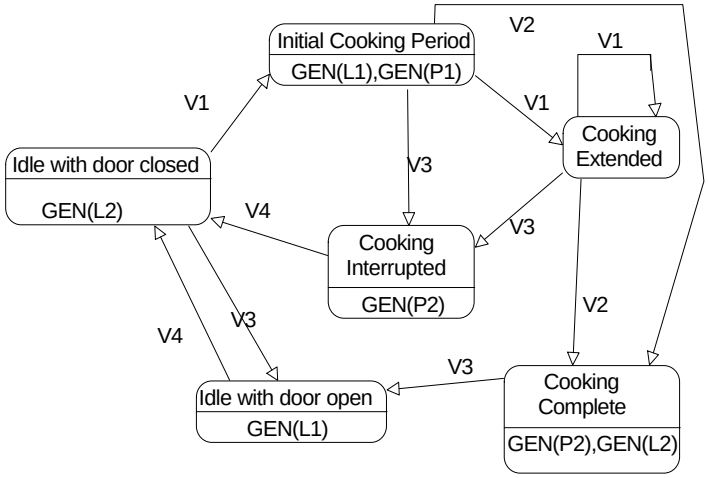


Fig. 7 Statechart diagrams for the microwave oven example.

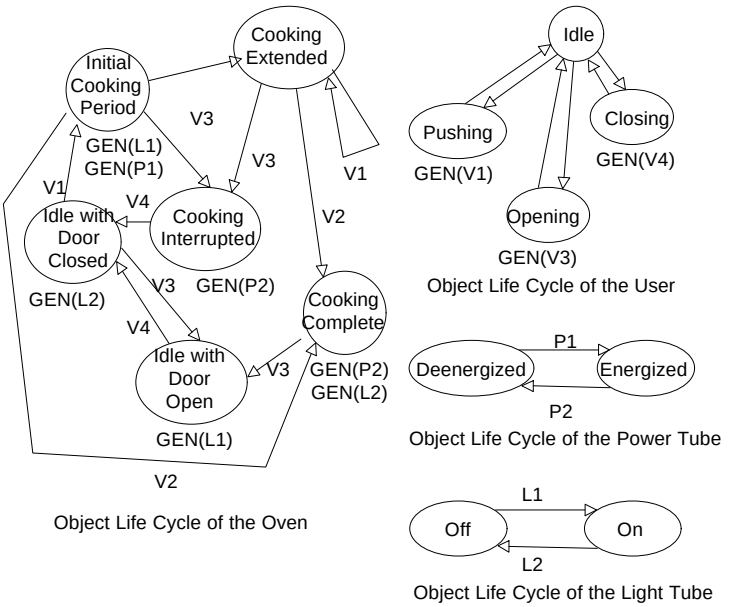


Fig. 8 Object life cycles for the microwave oven example.

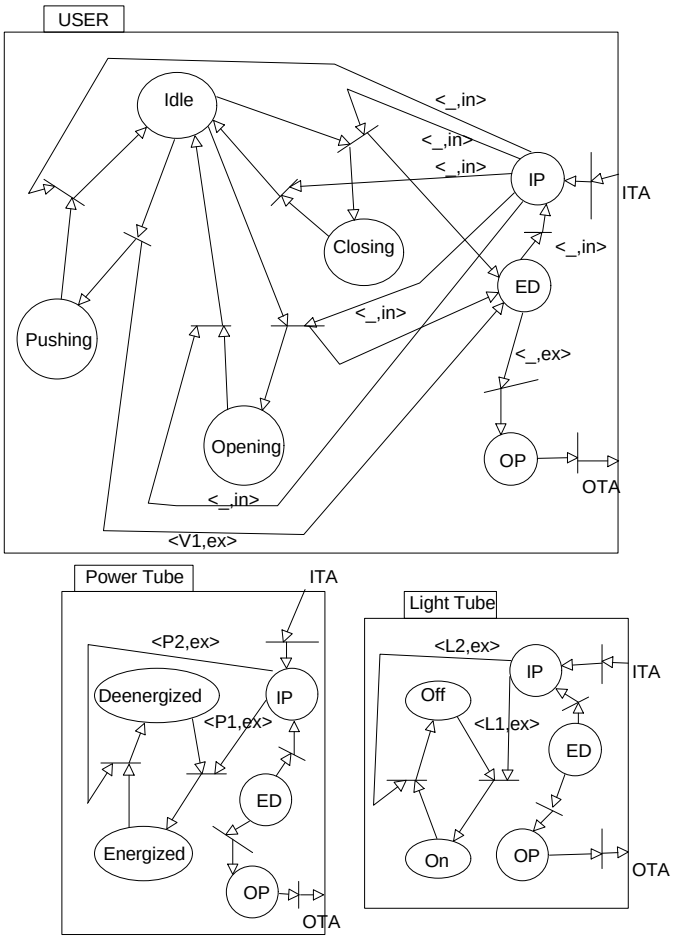


Fig. 10 OPMs for the microwave oven example.

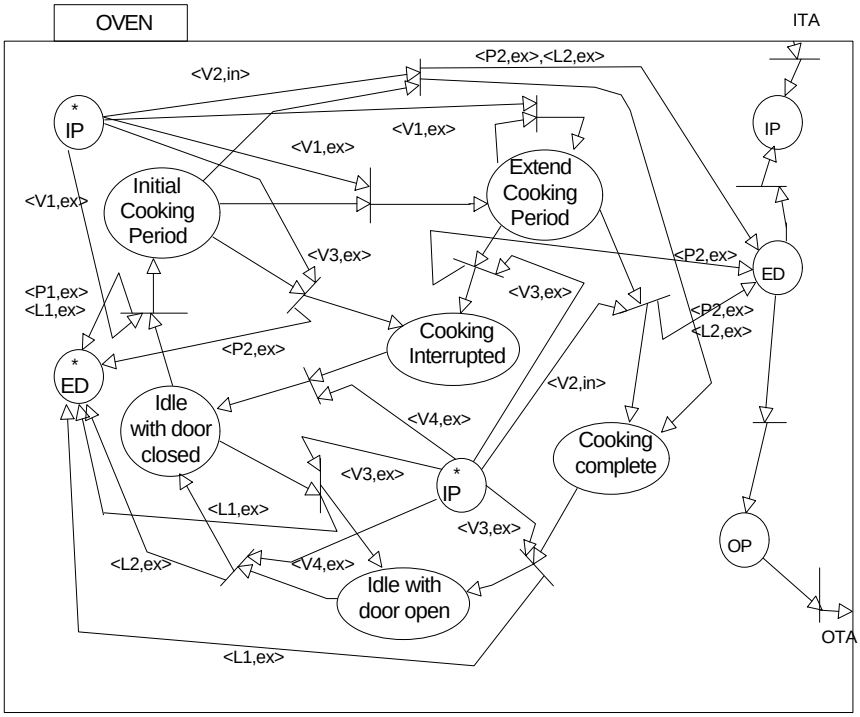


Fig. 9 OPM of the oven object in the microwave oven example.

Places IP and ED are duplicated (identified with *) to maintain clarity.

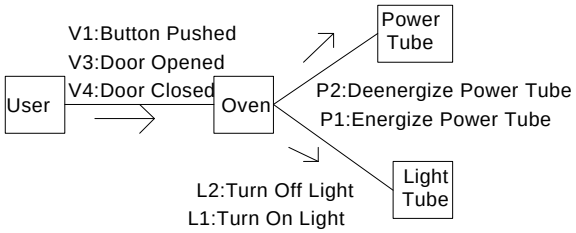


Fig. 11 Collaboration diagram for the microwave oven example.

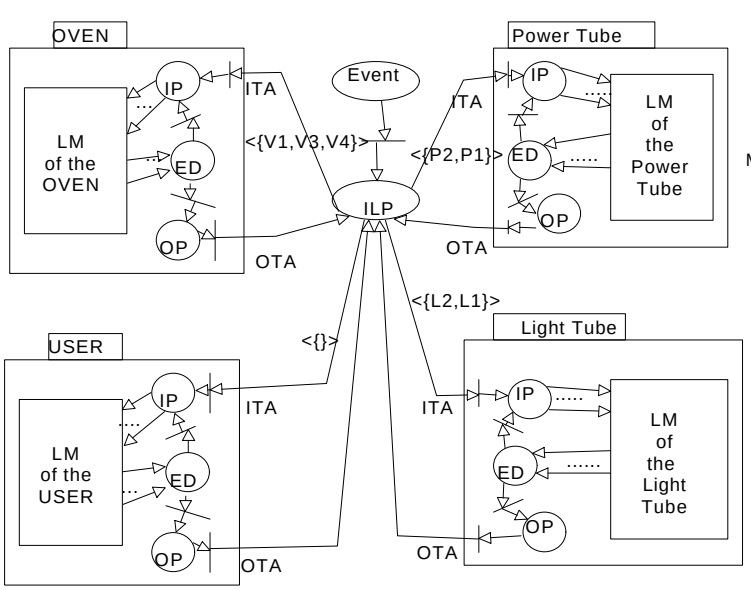


Fig. 12 The system-level Petri net for the microwave oven example.

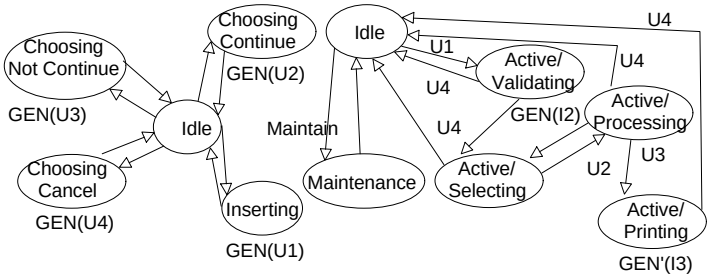


Fig. 14 Object Life Cycles of the User and the ATM machine, in the ATM example.

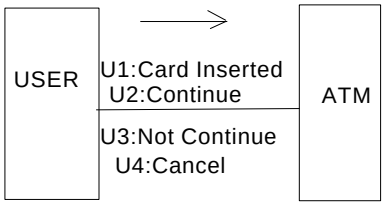


Fig. 16 Collaboration diagram for the ATM machine example.

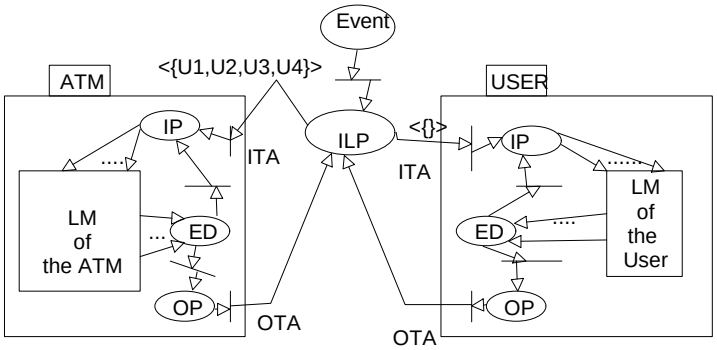


Fig. 17 System-level Petri net for the ATM machine example.

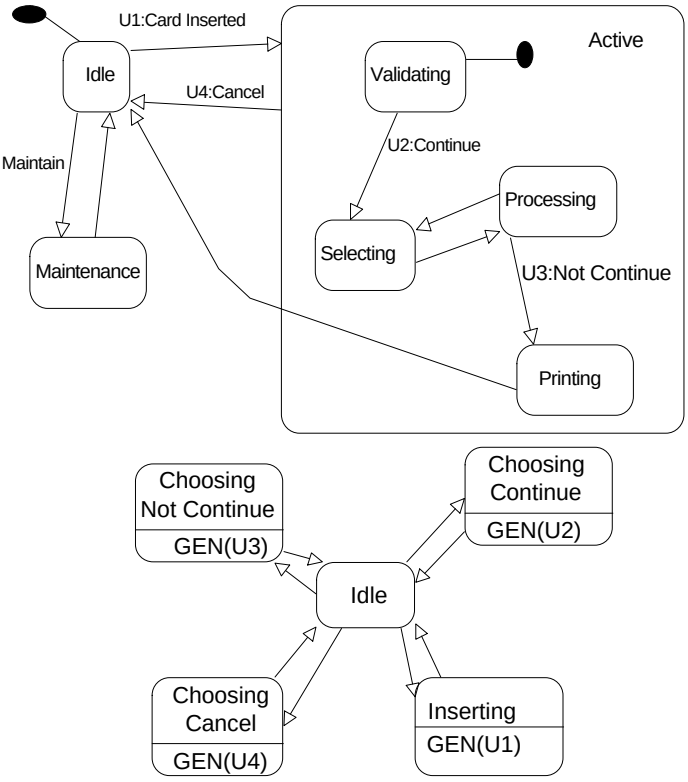
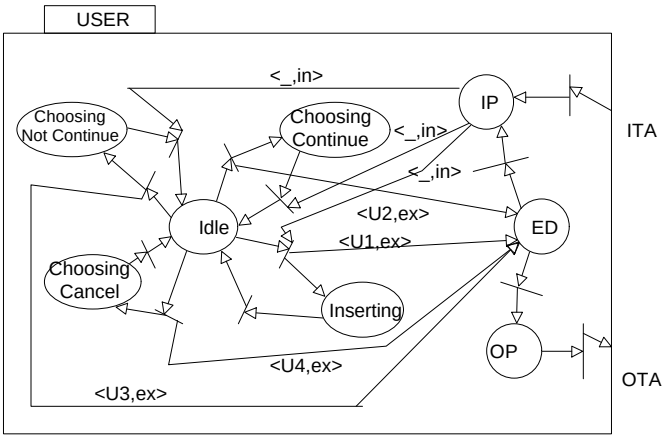


Fig. 13 Statechart diagrams of
(a) ATM machine and (b) ATM user.



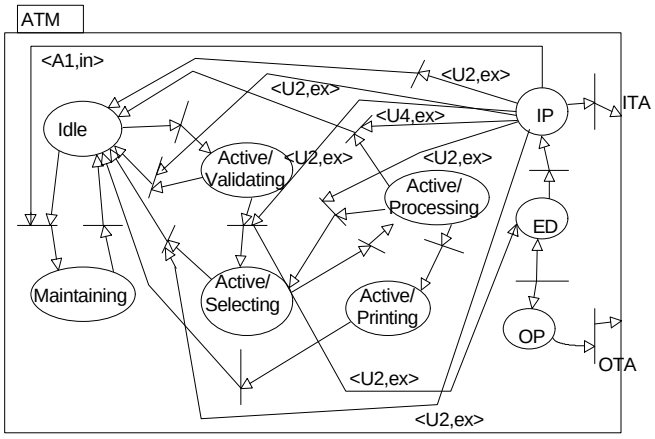


Fig. 15 OPMs of ATM user and the ATM machine.