Formalization of Object Behavior and Interactions From UML Models[1]

John Anil Saldhana, Sol M. Shatz, and Zhaoxia Hu

Concurrent Software Systems Laboratory

Department of Computer Science

University of Illinois at Chicago

{jsaldhan, shatz, zhu}@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. This paper describes a methodology for synthesis of a Petri net model from UML diagrams. The approach is based on deriving Object Net Models from UML statechart diagrams and connecting these object models based on UML collaboration diagram information. The resulting system-level Petri net model can be used as a foundation for formal Petri net analysis and simulation techniques. The methodology is illustrated on some small examples and a larger case study. The case study reveals some unexpected invalid system-state situations.

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 [10]. Statechart diagrams [2] 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 [2]. 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 [17] are a formal and graphical appealing language, appropriate for modeling systems with concurrency. Colored Petri nets (CPNs) [14] 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.

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

2. Motivation and Related Work

Concurrent systems are systems composed of elements that can operate concurrently and communicate with each other. They are often part of safety-related or mission-critical systems. They are notably difficult to design. Such systems can usually exhibit an extremely large number of different behaviors. This is due to the combinatorial explosion resulting from all possible interactions between the different concurrent components of the system and the many possible race conditions that may arise between them. This situation makes the development of concurrent systems an extremely delicate task. Therefore, it is desirable to prove that the software programs driving these systems are *logically correct*. However, formal proof of system properties is difficult and verification of concurrent systems is further hindered by the *state explosion problem*, as the system's state space tends to grow exponentially with the number of processes. 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 basic Petri nets 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., [4],[15],[19]).

Object Petri nets [14] 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. Our work also considers a form of object Petri net, but our emphasis is on algorithmic synthesis of the net model.

Cheng [3] discuss the need to formalize the dynamic model of a system and integrate it within Object Modeling Technique (OMT), a graphical notation that was merged into UML. The researchers have used LOTOS, a formal specification language to formalize the dynamic (or behavioral) model of a system. LOTOS uses process algebra (an algebraic theory to formalize the notion of concurrent computation) and algebraic axioms. In contrast, our work uses the Petri net model, which has strength in its intuitive graphical notation. Also, our work targets UML, which is a superset of OMT with behavior being additionally represented using interaction diagrams (collaboration diagrams and sequence diagrams). Thus, we present a translation of UML statechart diagrams and derive message (or event) flow information from collaboration diagrams to yield a single system-level colored Petri net that represents the composite behavior of a set of objects. A validation process applied to this single comprehensive Petri net allows for a stronger behavioral validation of the software system.

Other works that apply Petri net modeling to UML specifications include the work of He, who has presented approaches to formally define class diagrams and use case diagrams [11], [12],[13]. Thus, this work is complementary to our work, although it uses a different type of Petri net called a Hierarchical Predicate Transition Net (HPrTN).

A related research effort specifically aimed at validating UML models is the work on a tool called vUML [16]. This tool uses SPIN, a model checker. One disadvantage is that open models (models that react to external entities) cannot be directly 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 appears to work 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 [7]. Here the use cases define behavioral specifications and are converted to hierarchical colored Petri nets. Finally, in the area of performance estimation of systems, [18] uses UML to derive Stochastic Petri net models. The approach does not focus on a process for derivation of a system-level model representing the complete behavior of a system.

3.   Generating and Linking Object Petri Net Models

3.1   Overview of the approach

As mentioned earlier, our goal is to define a process for the generation of Petri nets from UML behavioral specifications and thereby provide a formalization of behavior. More specifically, we focus here on two key steps: 1) Generation of Object Net Models (ONMs) 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 [2].

Before we present the specifics of our approach, including key definitions and algorithms, we would like to provide the reader with a preview of how the basic approach works. We do this by referring to some pieces of an example that will be discussed in more detail later. The example is that of a microwave oven system, which includes the following objects: A microwave oven (the cooking part of the oven), a power tube (that provides power for cooking), a light tube (that provides power to a light inside the oven), and a user. For now, let's just consider the oven object. Fig. 6 shows a statechart diagram for the oven. The oven object has six states and changes its state on the occurrence of events. For example, if the oven is in the state "Idle with door closed" when event V1 occurs, the oven changes to the state "Initial Cooking Period" – cooking begins. The event V1 indicates that the user has pushed the state button; the generation of this event can be seen in Fig. 5. When the oven enters the "Initial Cooking Period" state, the oven itself generates two new events (L1 and P1) that can trigger state changes in the light tube and the power tube, respectively.

Using our approach, the statechart for the oven object is converted into an Object Net Model (ONM), as shown in Fig. 8(a). This Petri net-based model models the internal state changes of the oven object as well as the routing of events (tokens) into and out of the object model. Event tokens (generated by other objects) enter the object model via the ITA port and event tokens (generated by this object) exit the object model via the OTA port. The ED place node routes the event tokens. Once we have generated ONMs for each of the objects, the ONMs are linked in a way that creates a system-level model. This can be seen in Fig 10(b). The final system-level model can then be used to support design simulation, analysis, code generation, etc. With this as an overview of the key steps in the approach, we can now proceed to a more detailed discussion on each of the steps.

3.2 Definitions and algorithms

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 an Object Net Model (ONM). The general architecture of an ONM model is shown in Fig. 2. We describe the structure of such a model in the following paragraphs.

Definition: An *Object 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). These provide an interface for the object with the external environment. In our model, all object interactions are assumed to be via explicit messages. Thus, the interface arcs allow the flow of event tokens into, and out of, an object model. This will be discussed further below.

To model the dynamic behavior of a system represented by a Petri net, we need to introduce tokens. In our methodology, we use 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 ONM, we need a few definitions.

Definition: An *Event-Place* is a place of the ONM 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 ONM. 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 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. 8(a), 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 Net Model, 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 using a set of algorithmic transformations (sketched below). Then, given a collaboration diagram showing the connection between objects, we connect the ONMs, to yield a single system-level CPN. Thus, the top-level view of our approach is defined by the following conceptually simple 2-step process:

Formalization of UML object model behavior:

Input: A set of UML statechart diagrams *S* and a UML collaboration diagram *C*.

Output: a system-level colored Petri net.

BEGIN

1. For each statechart diagram *s* ε *S*, convert *s* into an Object Net Model.

2. Using the collaboration diagram *C*, connect the ONM's obtained in step 1.

END

First, we describe the generation of ONMs from UML statechart diagrams. A statechart diagram [2] contains states (simple and composite) and transitions (events and actions). Statechart transitions are not to be confused with Petri net transitions; statechart transitions are denoted by standard finite state machine arcs that define a change from one state to a successor state. 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). Note that in our Object Net Models, we treat all events as deferred events.

A statechart 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 must evaluate to true to allow the transition to fire), *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

6

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 [20], 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 ONM. Basically, state machine states are mapped onto Petri net places and state machine transitions are mapped onto Petri net transitions. This process will be clarified by an algorithm presented shortly.

To aid the modeling of actions and events defined in a statechart, we introduce the concept of an event generator and some associated conversion rules.

Definition: An *Event Generator* is a function that generates an event – when used in a Petri net, an event generator generates an event token. An event generator that generates the specific event E is denoted as GEN(E).

Conversion 1: Actions that are associated with statechart transitions are mapped onto arc event generators. So, an event Y appearing as an arc label in some statechart is mapped to a function GEN(Y) associated with the arc.

Conversion 2: Statechart *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.

We introduce two special event generator functions that can be associated with a state, rather than a transition arc. First, if a state S contains the function GEN(X), it means that the event X is generated upon entry to state S (via any transition). Likewise, if a state S contains the function GEN'(Y), it means that event Y is generated upon exit from state S.

Conversion 3: The *entry* and *exit* actions of a statechart state can be mapped onto events that are generated before the object enters the state (i.e., of the form GEN(X)) and after it leaves the state (i.e., GEN'(X)), respectively.

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 when a triggered transition fires. A concurrent composite state can move to the next state only when the concurrent subpaths within the composite state join into one flow.

We now formalize the conversion concepts that we have been discussing in terms of an algorithmic process.

Algorithm 1: Convert a statechart diagram into an Object Net Model.

Input: A UML statechart diagram S.

Output: An Object Net Model M.

BEGIN

1. Let M be an ONM with an empty LM component (i.e., no places or transitions in LM). Note that by definition, M does contain other ONM components such as the IP and ED places.

2. Let sm be a flat state machine containing all the states of diagram S excluding the composite states.

3. For all $s\varepsilon S$, where s is a sequential composite state, convert s into a flat state machine fsm using Algorithm 2 (given below). Merge fsm into sm.

4. For all $s\varepsilon S$, where s is a concurrent composite state, convert s into a flat state machine fsm using Algorithm 3 (given below). Merge fsm into sm.

5. Map all actions that are associated with arcs in S onto event generators for the arc in sm.

6. Map all entry actions of a state in S onto event generators GEN for the corresponding state in sm.

7. Map all exit actions of a state in S onto event generators GEN' for the corresponding state in sm.

8. Map all the states, and arcs, of sm onto places, and transitions, in M, respectively.

9. For each transition in M generate an input arc from the place IP if the corresponding arc in S represents a state transition on the occurrence of an event.

10. For each transition t in M, generate an output arc from t to the place ED if either of the following two conditions holds: 1) a state (in sm), corresponding to one of t's input places, has an associated exit event (i.e., GEN'), 2) a state (in sm), corresponding to one of t's ouput places, has an associated entry event (i.e., GEN).

END

In Fig. 4(a), we show the conversion of a sequential composite state X into a flat state machine. For the composite state X, the internal substate B is the initial state. Since states B, C, and D are substates of X, when the system is in states B, C or D, we can identify the state as X/B, X/C or X/D, respectively. The state transition from composite state X to state E is possible from any of the substates B, C and D. Hence there are four arcs leading to state E. The corresponding Petri net is also shown.

Algorithm 2: Convert a sequential composite state into a flat state machine.

Input: A UML statechart diagram S containing a sequential composite state scs.

Output: Flat state machine O.

BEGIN

1. Let O be a flat state machine with zero states and zero transitions.

2. For all states $s \varepsilon S$ and $s \neq scs$, add state s and all its transition arcs into O.

3. If $s \varepsilon S$ and $s = scs$,

• Add each state sx ε scs and all its transition arcs into O.

- For each transition from a state $s \neq scs$ to a state sx ε *scs*, add a transition from s ε O to sx ε O.

- For each transition from a state sx ε *scs* to a state $s \neq scs$, add a transition from sx ε O to s ε O.

- For an entry transition from a state s ε S and $s \neq scs$ to state *scs*, add a transition from s ε O to sx ε O such that sx corresponds to an initial state of state *scs*.

- For an exit transition from state *scs* to a state s ε S and $s \neq scs$, if the transition is triggered by an event, add a transition from each state sx ε O to s ε O such that sx ε *scs*.

- For an exit transition from state *scs* to a state s ε S and $s \neq scs$, if the transition is triggerless, add a transition from state sx ε O to s ε O such that sx corresponds to an ending state of state *scs*.

END

In Fig. 4(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 the special states *fork* and *join* in the state machine diagram.

Definition: A *fork* state denotes parallel activation of all immediate successor states.
Definition: A *join* state denotes synchronized activation of an immediate successor state, with respect to the join state's source state.

It should be quite clear from Fig 4(b) that the *fork* state concurrently invokes both of the states A and C, and the *join* state will cause a transition to the state Q when states B and D have been reached. In the corresponding Petri net, each of these special *fork* and *join* states is mapped onto a Petri net transition, as is also shown in the figure.

Algorithm 3: Convert a concurrent composite state into a flat state machine.

Input: A UML statechart diagram S containing a concurrent composite state *ccs*.

Output: Flat state machine O.

BEGIN

1. Let O be a flat state machine with zero states and zero transitions.

2. For all states s ε S and $s \neq ccs$, add state s and all its transition arcs into O.

3. If s ε S and $s = ccs$,

- Add all the states along each distinct path in s into O.

- Add a state *fork* into O and add a transition from *fork* to each state in O that corresponds to an initial state on a distinct path in s.

- Add a state *join* into O and add a transition to *join* from each state in O that corresponds to an ending state on a distinct path in s.

4. For all states $s \, \varepsilon \, S$ such that $s \neq ccs$ and there is a transition from s to ccs, add a transition from $s \, \varepsilon \, O$ to state *fork* $\varepsilon \, O$.

5. For all states $s \, \varepsilon \, S$ such that $s \neq ccs$ and there is a transition from ccs to s, if the transition is triggerless add a transition from state *join* $\varepsilon \, O$ to $s \, \varepsilon \, O$.

END

For a transition from a concurrent composite state, if the transition is triggered by an event, similar conversion rules can be derived based on the ideas of Algorithm 3.

As mentioned earlier, our goal is to generate ONMs for objects in a system and then connect them, using UML collaboration diagrams. If the naming of events is kept uniform between statechart diagrams and collaboration diagrams, we can connect the ONMs for each object to create a system-level CPN for the whole system. We present two examples to illustrate the ONM generation and linking process.

3.3 Examples

Example 1: Consider an example of a one-minute microwave oven adapted from [20]. The oven is powered by a Power Tube and contains a Light Tube. Four events are possible: V1 (user pushes the on-button), V2 (the time-out when the cooking period has expired), V3 (user opens the door) and V4 (user closes 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. 5 and 6. We can assume that the initial states are Idle (for user), Idle with door open (for oven), Deenergized (for power tube), and Off (for light tube). 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). For the oven design being considered, the state Cooking Interrupted is distinct from the state Idle with door open since entry to the state Cooking Interrupted does not require the light to be turned on (it is already on). The object life cycle models of the user, oven, power tube and the light bulb are shown in Fig. 7.

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 ONMs for the microwave oven example are shown in Figs. 8(a), 8(b) and 9. 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 (Fig.10(a)), the ONMs of the various objects can be combined together to form a system-level Petri net, as shown in Fig. 10(b). 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 ITA and ITA arcs of the ONMs to the general event place representing the environment. The resultant Petri net is a CPN, with event tokens as defined earlier in the section.

ONMs are linked by a place called an *Intelligent Linking Place (ILP)*. Based on the UML collaboration diagram, a token tag is attached to each ITA (input transition arc) of an ONM 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 ONM 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 ONM of the object in consideration.

Example 2**:** Consider an example of an ATM machine, dispensing cash to a user [2]. The statechart diagrams and the collaboration diagram are shown in Figs. 11 and 12, respectively. The description of the problem 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.

The ATM behavior is modeled using a sequential composite state (See Fig. 11(a), the statechart diagram of the ATM machine). Using the conversion rules described earlier, we can obtain the object life cycle model for the ATM machine shown in Fig. 13. Finally, the ONMs for the two objects are shown in Fig. 14. Note that events "Maintain", "readCard" and "ejectCard" are all internal events to the ATM and are denoted as I1, I2 and I3, respectively. 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 system-level CPN obtained after linking of the ONMs is shown in Fig. 15. If the system is a *closed system* with no interaction with the external environment, we do not need the super place (labeled at Event) representing the environment. For a closed system, any event that is of interest to some specified object is generated by one of the specified objects. In this case, all event tokens that are routed by the ILP place originate from one of the specified object models (ONMs); thus the EVENT place, which represents the environment, can be removed.

4.   Case Study - Spacecraft Control System

Consider a Spacecraft Control system that consists of the cooperating subsystems defined in Table 1. The description of the problem is based on an example described in [5]. The Mission Control subsystem issues a *goal* to a Trajectory Planner subsystem. A goal is basically of the form "Get the Spacecraft from here to there." The Planner builds a *trajectory* (a path to be taken by the spacecraft) and passes it to a Motion Coordinator subsystem. The Motion Coordinator actually gets a command of the form GoTo(x,y,z,t,a,b,c) where x,y,z are the positions, a,b,c are rotations along the X,Y and Z axes, respectively, and t is the time variable. Using a Controller, the Coordinator coordinates the *position* (a coordinate position in space relative to a fixed coordinate origin (x,y,z,t) and *attitude* (amount of rotation in three axes) adjustment of the Rocket powering the Spacecraft. The Controller issues commands to the Rocket and a Sensor that monitors the Rocket's position and attitude. The Sensor reports results to the Controller, which are used to apply corrections to the Rocket.

Classes	Roles
Mission Control	Main control system that generates goals. A goal being "Get the spacecraft from location x to location y".
Trajectory Planner	Plans the full trajectory, including positional and attitudinal changes.
Movement Coordinator	Applies a trajectory received from the Trajectory Planner. It sends a command to the Controller to adjust the position and attitude of the Spacecraft.
Rocket	Gives thrust to the Spacecraft according to a trajectory.
Controller	Controls the Spacecraft's position/attitude by controlling the Rocket.
Sensor	Ensures that the Spacecraft is moving correctly within the six-degree-of-freedom frame of reference.

Table 1. Subsystems in the Spacecraft Control example and their roles

The various events that are generated in the subsystems are listed in Table 2. Please note that event tokens in the methodology can contain passed parameters. For example, the event token from the Trajectory Planner to the Motion Coordinator subsystem may contain parameters x, y, z, t, a, b and c.

Sub System	Event	Short Name	Type	Role
Mission Control	evAbort	M1	External	Abort the carrying of the current goal
	evCancel	M2	Internal	Signal generated to abort the goal
	evGoal	TP1	External	Ask Trajectory Planner to carry this goal
	evApplyGoal	TP2	External	Signal Planner to carry on the built plan
Trajectory Planner	evApplyTrajectory	MC1	External	Signal Movement Coordinator to execute the trajectory
Movement Coordinator	evAdjust	C1	External	Ask Controller to apply adjustment
Controller	evControllerFinish	C2	External	Signal Coordinator that it has completed the goal
	evRocketStartUp	R1	External	Signal Rocket to boot
	evRocketBegin	R2	External	Signal Rocket to begin the process
	evRocketAdjust	R3	External	Signal Rocket to start the adjustment process
	evRocketFlameOn	R4	External	Signal Rocket to begin flaming
	evRocketShutDown	R5	External	Signal Rocket to shut down flaming
	evSensorStartUp	S1	External	Signal Sensor to start up
	evSensorBegin	S2	External	Signal Sensor to begin sensing
	evSensorShutDown	S3	External	Signal Sensor to shut down sensing
Rocket	evRocketDone	R6	External	Signal Controller that the goal has been carried out
	evStop	R7	Internal	Internal signal indicating a problem
	tm(FlameTime)	R8	Internal	End of a timer set for a specific flaming period
Sensor	evSensorData	S4	External	Pass sensory data to Controller
	tm(SenseTime)	S5	Internal	Internal event when the timer set to sensing period has expired

Table 2. Events generated in the Spacecraft Control example

The statechart diagram of the Controller is shown in Fig. 16. We see that the diagram contains a nested state called 'Adjusting' when the Controller is in the 'Controlling' state. This is an instance of a sequential composite state. When the Controller receives an event evAbort, it can exit from any of the states in the composite

state and become idle. To convert the statechart diagram into an Object Net Model, first we convert it into an object life cycle applying Algorithm 2 (from Section 3). The object life cycle model is shown in Fig 17. The ONM of the Controller is a direct mapping of the states in the object life cycle model to Petri net states and the arcs to Petri net transitions, in accordance with the procedure described in Section 3. The ONM of the Controller is not shown due to lack of space, but the mapping from the object life cycle to an ONM is based on the algorithm described in Section 3.

The statechart diagram of the Movement Coordinator, given in Fig. 18, shows that there is a concurrent composite state when the Coordinator is in the 'Coordinating' state. In order to convert this concurrent composite state into a flat state machine, we apply an algorithm that is derived from, but a bit more complex than, Algorithm 3 (from Section 3). In this case, we must deal with the triggered transitions from the concurrent composite state. The Coordinator moves to 'idle' state from the 'Coordinating' state when there is an event *evAbort* or the Controller sends an event *evControllerFinish* indicating the success of the goal. As discussed in Section 3, we use two special states *fork* and *join*. The fork state replicates the token it receives to equal the number of outgoing arcs and the join state waits until it receives the tokens from all incoming arcs and then reduces the number of tokens to one. In this case, the fork state replicates the number of event tokens MC1 (*evApplyTrajectory*) to two, as there are two outgoing arcs. The corresponding ONM of the Movement Coordinator object is shown in Fig. 19.

The statechart diagram of the Rocket object is shown in Fig. 20. The Rocket gets the signal events from the Controller. When the Rocket is flaming, the Controller sends the event *evRocketAdjust* to apply new correction to the Rocket. If there is any internal event *evStop*, the Rocket becomes idle, waiting for the Controller to send an event *evRocketFlameOn*. At the expiration of the Flame Time timer, an internal event R8 is generated. Then the Rocket stops flaming and waits for the Controller's signal to start flaming again or the goal to be aborted. To save space in this presentation, we do not explicitly show the object life cycle model or ONM for the Rocket object.

The statechart diagram of the Sensor is shown in Fig. 21. When the Sensor is in 'Active' state, the Sensor waits for a finite amount of time (*SenseTime*). When an internal event S5 signaling the expiration of *SenseTime* timer is generated, the Sensor starts the 'Sensing' state. Once finished with sensing, the Sensor generated an external event *evSensorData*, for the Controller to accept the sensory data. After applying Algorithm 2 to reduce the sequential composite state 'Active', the object life cycle model of the Sensor is shown in Fig 22. The ONM of the Sensor is a direct mapping of the states in the object life cycle model to Petri net states and the arcs to Petri net transitions, in accordance with the procedure described in Section 3. The ONM of the Sensor is not shown to save space.

The statechart diagram of the Trajectory Planner is shown in Fig. 23. The resulting ONM of the Trajectory Planner object is shown in Fig. 24. The statechart diagram of the Mission Control object is shown in Fig. 25. When there is an internal event *evCancel*, the Mission Control subsystem aborts the goal process and sends an event *evAbort* to all the other subsystems. The corresponding ONM is shown in Fig. 26.

Now that we have generated the ONMs of the individual objects, the linking process described earlier is applied, using the collaboration diagram shown in Fig. 27. The resulting system-level Petri net is shown in Fig. 28. This synthesized system-level Petri net was subjected to a manual analysis and simulation process. The results indicate that there is a possibility for the system to enter some invalid states – in 3 cases. These invalid system-wide state situations are not mentioned in [5], but are summarized below.

Invalid state 1：Consider the scenario when the Trajectory Planner is in the state 'Trajectory Ready' and an event *evAbort* (i.e., <M1,ex>) (generated by the Mission Control subsystem) is received by all the subsystems. All the subsystems reset and become inactive. But the Planner subsystem remains in the current state. During the next cycle, the system will enter an invalid state. The Planner is in state 'Trajectory Ready' and waiting for an event *evApplyGoal* (i.e., <TP2,ex>) from the Mission Control object. A new goal is issued by the Mission Control object when it issues an event *evGoal* (i.e., <TP1,ex>) and enters state 'Active | Sending Goal.' When the Mission Control object issues an event *evApplyGoal* (i.e., <TP2, ex>), indicating a command to apply the new goal, the Planner will respond to this event, but apply the previous goal that has already been aborted. The invalid system-state is detected because the Mission Control object is in its 'Inactive' state, but the Planner object is in its 'Trajectory Ready' state. The solution to this invalid system state is to change the design of the statechart diagram of the Trajectory Planner object by adding a transition from state 'Trajectory Ready' to 'Idle' on an event *evAbort*.

Invalid state 2: This scenario is similar to the previous one. Consider when the Rocket object is in the state 'Preparing' when the event *evAbort* (i.e., <M1, ex>) is received by all the subsystems. All the subsystems reset and become inactive. But the Rocket subsystem remains in the current state. During the next cycle, the system will enter an invalid state. The Rocket is in state 'Preparing' and waiting for an event *evRocketBegin* (i.e., <R2,ex>) from the Controller. A new goal is generated by the Controller when it issues an event *evRocketStartUp* (i.e., <R1,ex>) and enters state 'Controlling | Preparing.' When the Controller issues an event *evRocketBegin* (i.e., <R2,ex>)**,** indicating starting the new goal, the Rocket will respond to this event and start the previous goal that has already been aborted. The solution to this invalid system state is to change the design of the statechart diagram of the Rocket object by adding a transition from state 'Preparing' to 'Inactive' on an event *evAbort*.

Invalid state 3: Again, this scenario is similar in cause to the other two. Consider when the Sensor object is in the state 'Preparing' and an event *evAbort* (i.e., <M1, ex>) is received by all the subsystems. All the subsystems reset and become inactive. But the Sensor subsystem remains in the current state. During the next cycle, the system will enter an invalid state. The Sensor is in state 'Preparing' and waiting for an event *evSensorBegin* (i.e., <S2,ex>) from the Controller. A new goal is generated by the Controller when it issues an event *evSensorStartUp* (i.e., <S1,ex>) and enters state 'Controlling | Preparing'. When the Controller issues an event *evSensorBegin* (i.e., <S2,ex>), indicating starting the new goal, the Sensor will respond to this event and start the previous goal that has already been aborted. Hence this is a very dangerous situation because the position and the attitude of the Spacecraft may be wrong. The solution to this problem is to change the design of the statechart diagram of the Sensor object by adding a transition from state 'Preparing' to 'Inactive' on an event *evAbort*.

The results of the analysis and simulation phase point to the need for verification and validation of concurrent computing systems and the need for formalizing a notation like UML. Thus the methodology described in this paper has the potential to aid UML designers in identifying design flaws.

5.   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 object Petri net, called an Object Net Model (ONM), and discussed two key activities: 1) Generation of ONMs of individual objects or components, and 2) Linking these object models to create a system-level model. The method was defined algorithmically and illustrated on some small examples, as well as on a large case study.

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. This can involve integration with existing work in this direction, as mentioned in Section 2. Another direction for future work is to develop a set of tools to support the presented methodology. This will involve the creation of custom tools to carry out the net synthesis process and possible integration with existing Petri net tools for net analysis and display. One other related direction for 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. Perhaps this can be aided by development and use of a special form of query language, along the same lines as previous work done for Ada analysis [6], [1].

Acknowledgment

We thank Dr. G. Richter, of the GMD Institute for Autonomous Intelligent Systems (Germany), for his comments on an earlier version of this paper.

6. References

[1] C. Black, S. M. Shatz, and S. Tu, "A Query Language for Automated General Analysis of Concurrent Ada Programs," *Int. Journal of Computer Syst. Science and Engineering*, March 1998, Vol. 13, No. 2, pp. 83-95.

[2] G. Booch, I. Jacobson and J. Rumbaugh, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[3] B. Cheng, et.al., "Formalizing and Integrating the Dynamic Model within OMT," *Proceedings of the 19th International Conference on Software Engineering*, May 17-23, 1997, Boston, MA.

[4] Y. Deng, et.al, "Integrating Software Engineering Methods and Petri Nets for the Specification and Prototyping of Complex Information Systems," *Proc. Of the 14th Int. Conf. On Application and Theory of Petri Nets*, 1993, Chicago, Illinois, June 1993. Also, Lecture Notes in Computer Science, Springer-Verlag, Ed: Marco Ajmone Marsan.

[5] B. P. Douglass, *Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*, Addison Wesley, 1999.

[6] 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 Methodology*, Vol. 3, No. 4, Oct. 1994, pp. 340-380.

[7] 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.

[8] R. France, et.al., "Towards a Formalization of UML Class Structures in Z," *Proceedings of the 21st Annual International Computer Software and Applications Conference*, Washington D.C, August 11-15, 1997.

[9] M. Gogolla and F. Presicce, "State Diagrams in UML: A Formal Semantics using Graph Transformations," *Proceedings of the Workshop on Precise Semantics of Modeling Techniques* (PSMT '98), pp. 55-72. Technical University of Munich, technical Report TUM-I9803, 1998.

[10] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* Vol. 8 (1987*)*, pp. 231-274.

[11] X. He, "Formalizing Class Diagrams Using Hierarchical Predicate Transition Nets," *Proc. of the 24th International Computer Software and Application Conference*, Taiwan, Oct., 2000.

[12]   X. He, "Defining UML Class Diagrams using Hierarchical Predicate Transition Nets," *Proc. of the Workshop on Defining Precise UML Semantics in ECOOP'2000* (European Conf. on Object Oriented Programming).

[13]   X. He, "Formalizing Use Case Diagrams in Hierarchical Predicate Transition Nets," *Proc. of* the *IFIP 16th World Computer Congress*, Beijing, China, August, 2000, pp. 484-491.

[14]   K. Jensen., *Coloured Petri Nets, Vol 1: Basic Concepts*, Springer-Verlag 1992.

[15]   C. Lakos, "Object Petri Nets - Definition and Relationship to Colored Petri Nets," Technical Report TR94-3, Computer Science Department, University of Tasmania, 1994.

[16]   J. Lilius and I. Paltor, "vUML: A Tool for Verifying UML Models," Technical Report 272, Turku Centre for Computer Science, TUCS, 1999.

[17]   T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, Vol. 77, No. 4 pp. 541-580, April 1989.

[18]   R. Pooley and P. King, "Using UML to Derive Stochastic Petri Net Models," *Proceedings of the 15th Annual UK Performance Engineering Workshop*, pp. 45-56, July 1999.

[19]   S. Shatz and A. Khetarpal, "Applying an Object-Based Petri Net to the Modeling of Communication Primitives for Distributed Software," *Proc. of the Conf. on High Performance Computing*, 1998.

[20]   S. Shlaer and S. J. Mellor*, Object Life Cycles – Modeling the World in States*, Yourdon Press, Prentice Hall. 1992.
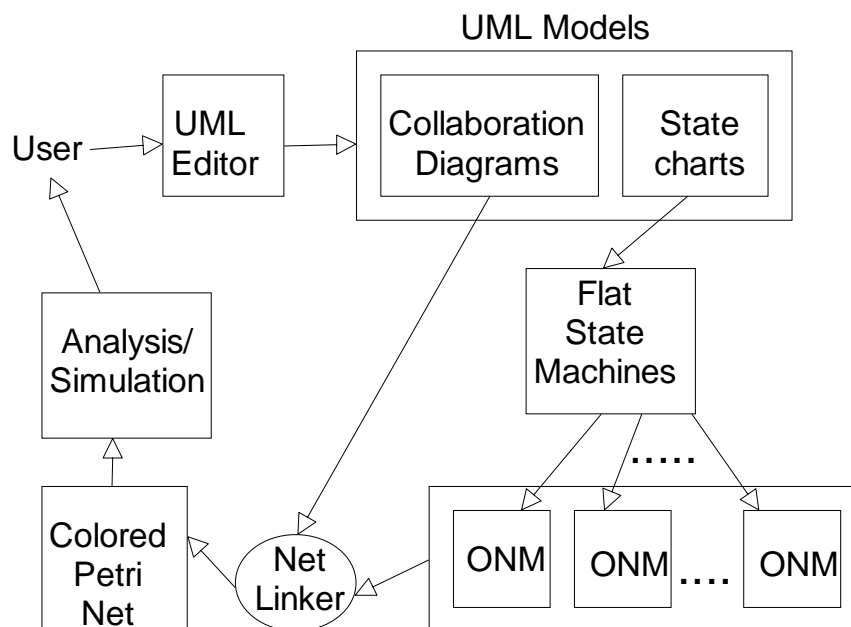
Figure 1. Block diagram of the proposed methodology

Object

In-place

ITA

<type,ex>

incoming
external events

...

LM
of
the
Object

Event
Dispatcher

<type,in>

.....

Out-place

<type,ex>

<type,ex>

<type,ex>

OTA

<type,ex>

Outgoing
external events
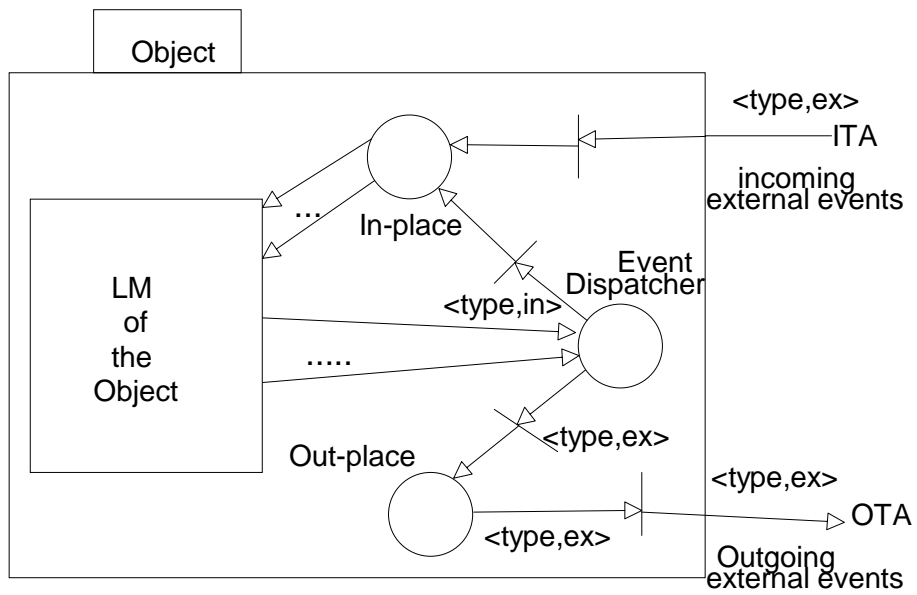
Figure 2. A generic Object Net Model (ONM)

initial state

ringing

A

Receiving

C

headerOK

D

error/
printReport

E

entry/pickup
exit/disconnect
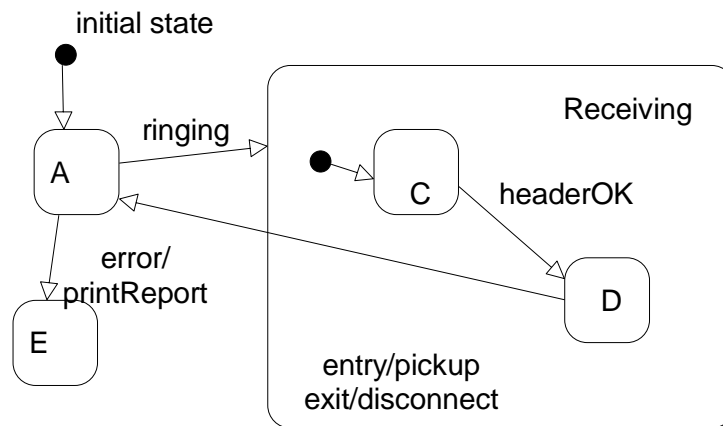
Figure 3. A statechart diagram example
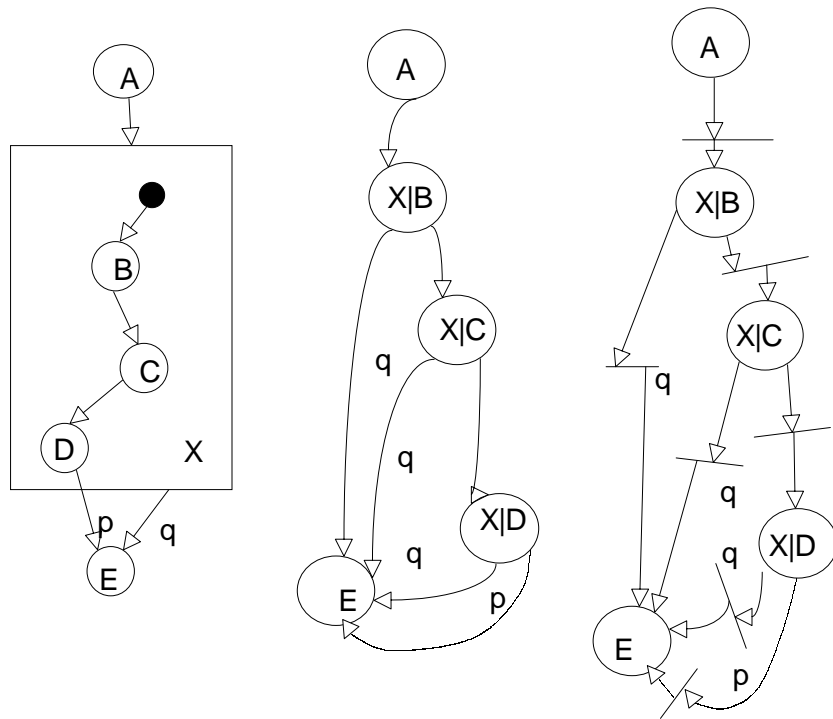
Figure 4(a). Conversion of a sequential composite state (X) to flat form and Petri net form
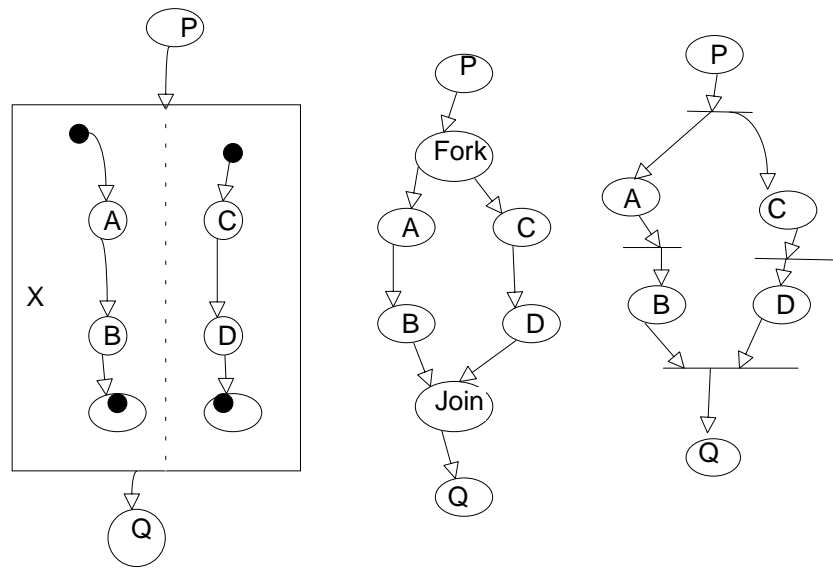
Figure 4(b). Conversion of a concurrent composite state (X) to flat form and Petri net form
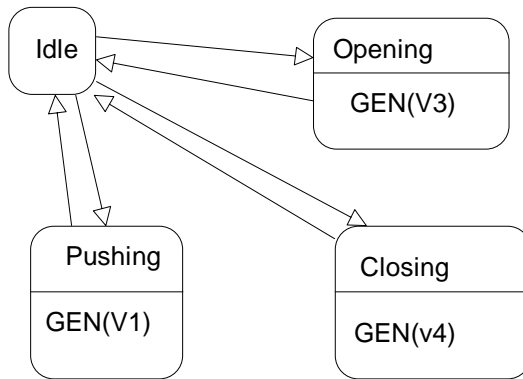
Figure 5. State diagram for a user object for the Microwave Oven example
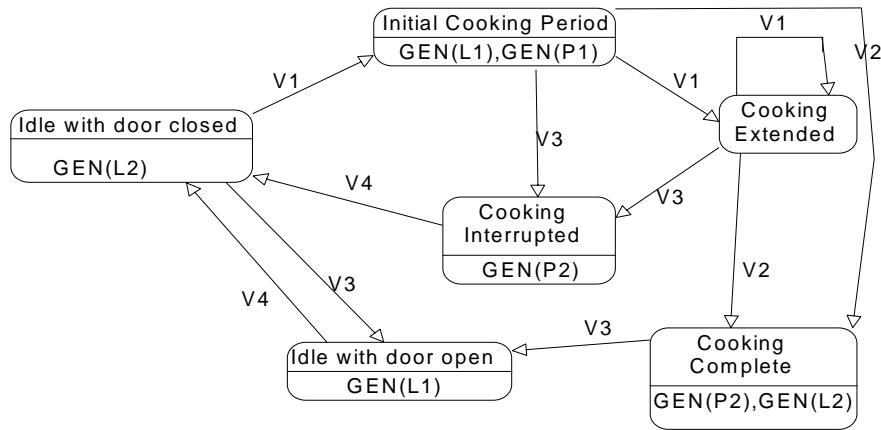
Statechart diagram of the Oven
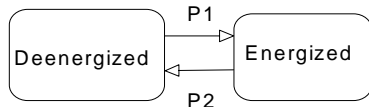
Statechart diagram of the Power Tube
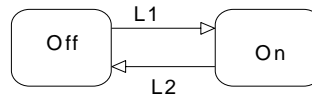
Statechart diagram of the Light Tube

Figure 6. Statechart diagrams for the Microwave Oven example
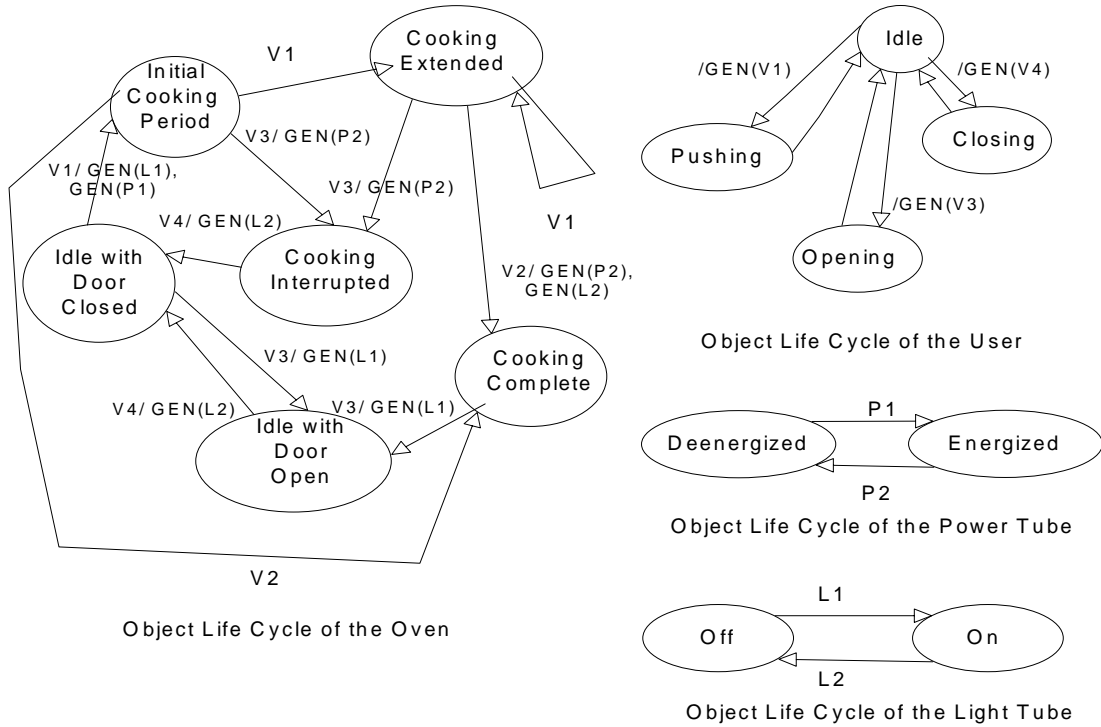
Figure 7. Object life cycle models for the Microwave Oven example
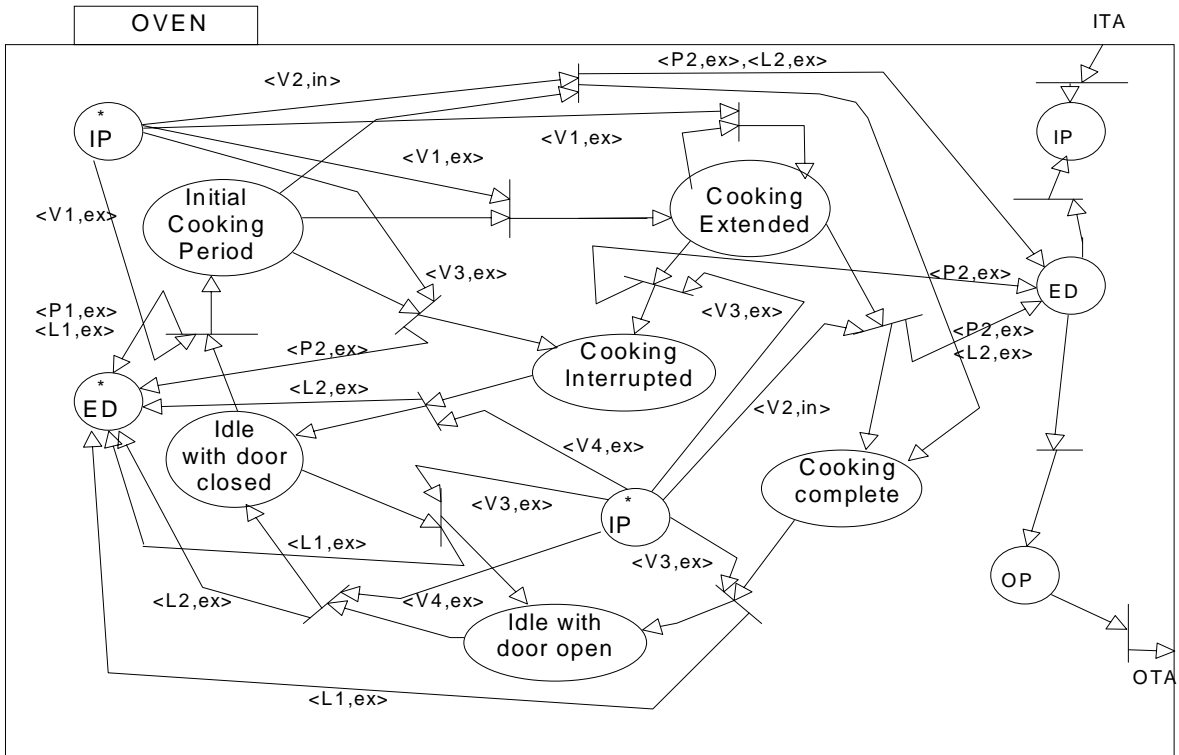
Figure 8(a). ONM for the Oven object [Places IP and ED (asterisked) are duplicated for clarity]
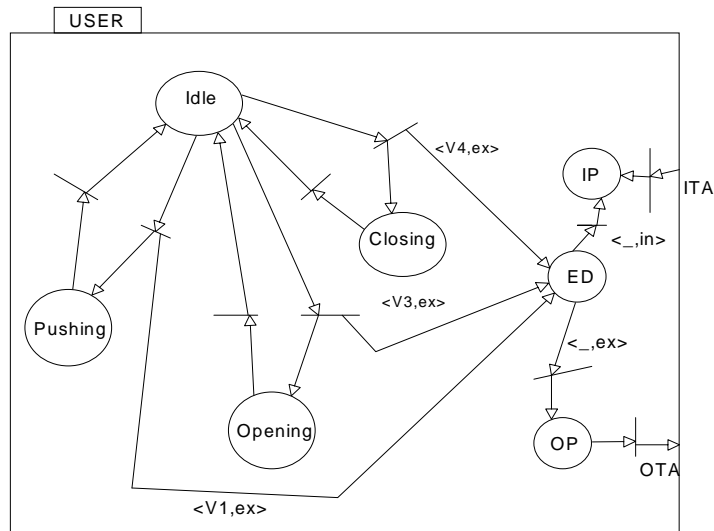
Figure 8(b). ONM for the User object
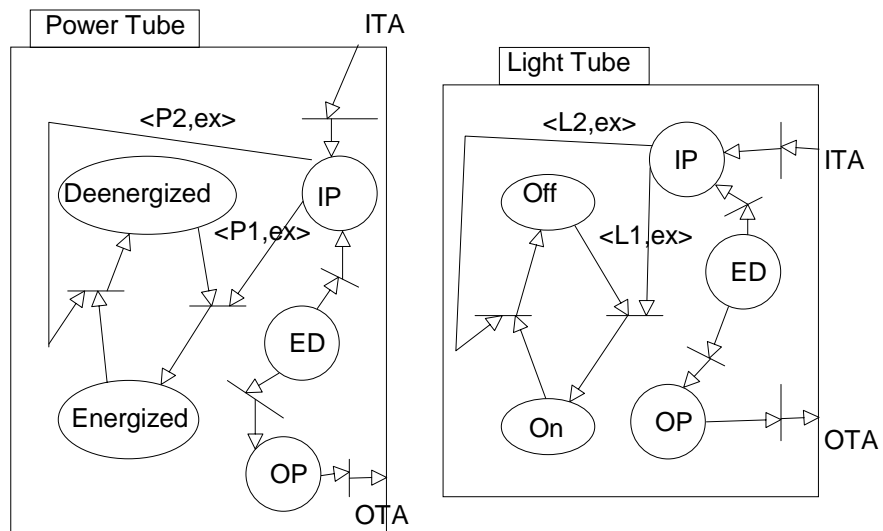
Figure 9. ONM's for the Power Tube and Light Tube objects
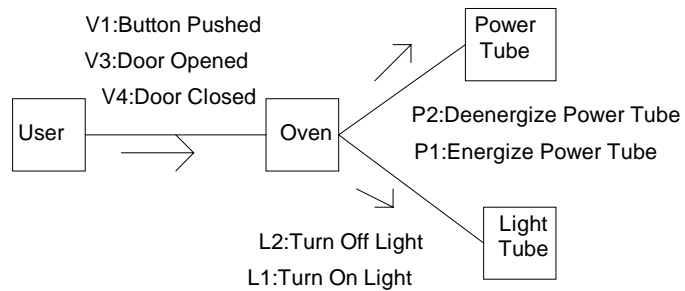
Figure 10(a). Collaboration diagram for the Microwave Oven example
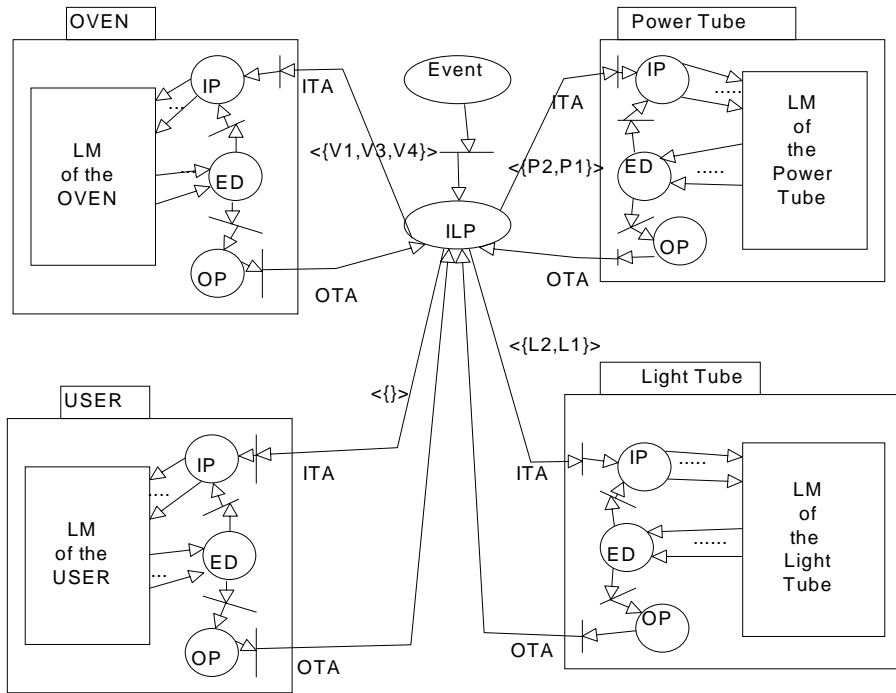
Figure 10(b). System-level colored Petri net architecture for the Microwave Oven example
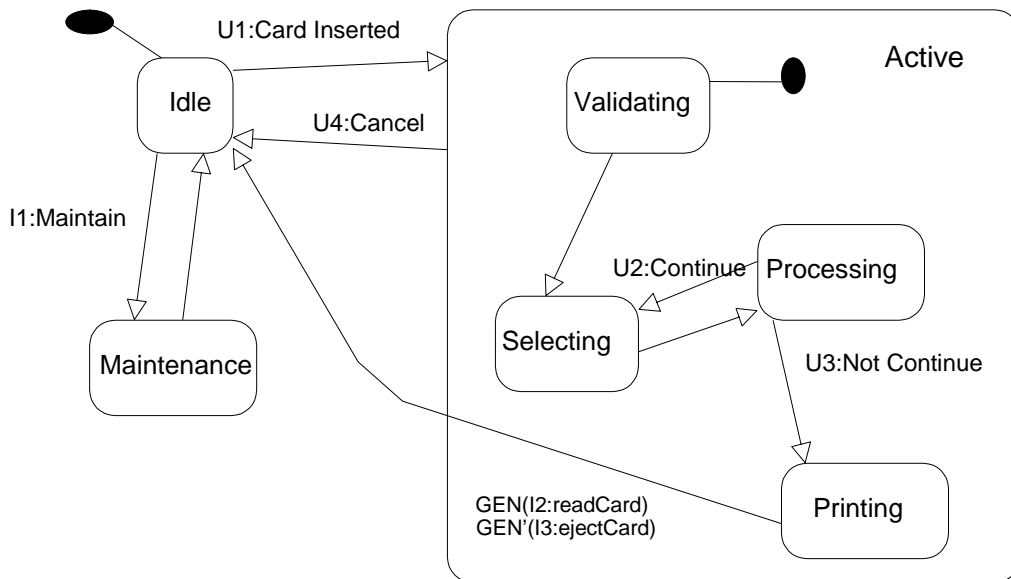
Figure 11(a). Statechart diagram for the ATM machine
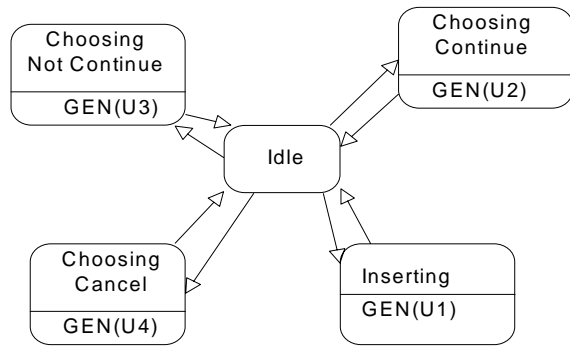
Figure 11(b). Statechart diagram for an ATM user
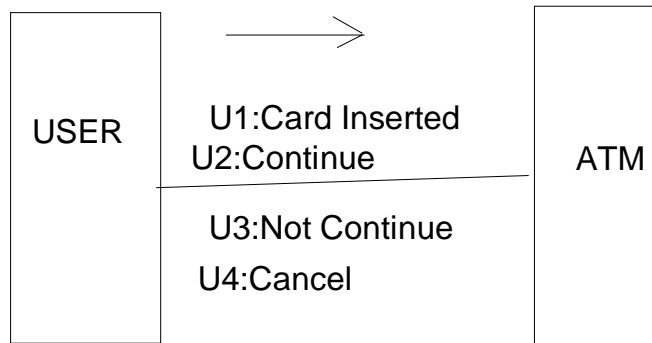
Figure 12. Collaboration diagram for the ATM machine example
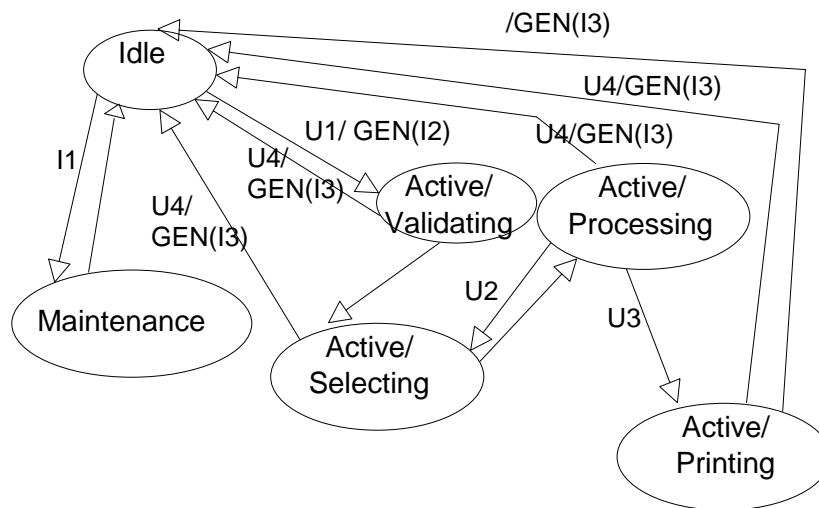
Figure 13. Object life cycle model of the ATM machine

USER

Choosing
Not Continue

Choosing
Continue

IP

ITA

Choosing
Cancel

Idle

<U2,ex>

<U1,ex>

ED

Inserting

OP

OTA

<U4,ex>

<U3,ex>

ATM

ITA

7   4

1

IP

Idle

Active/
Validating

Active/
Processing

7

7

5

3

6

7

ED

Maintain

7

Active/
Selecting

ED*

Active/
Printing

IP*

4

4

7

4

ED*

2

OTA

OP

IP*
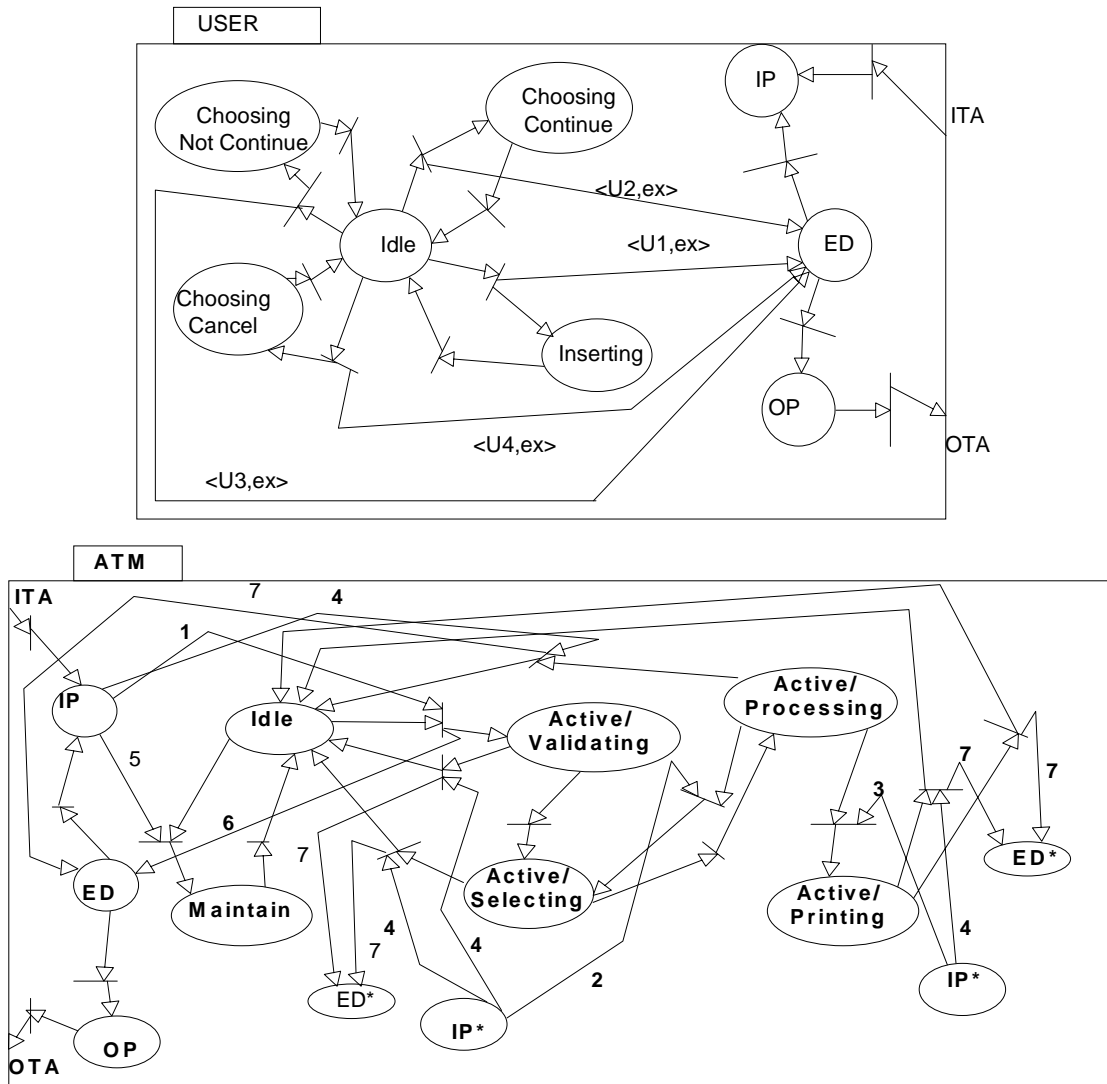
Legend: 1:<U1, ex> 2:<U2, ex> 3:<U3, ex> 4:<U4, ex> 5:<I1, in> 6:<I2, in> 7:<I3, in>

Figure 14. ONM's for the user and ATM machine example objects [IP is duplicated as IP* for clarity]

Event

<{U1,U2,U3,U4}>

ATM

USER

IP

ITA

ILP

<{}>

IP

ITA

.....

.....

LM
of
the ATM

...

ED

ITA

ILP

ITA

ED

....

LM
of
the
User

OP

OTA

OTA

OP
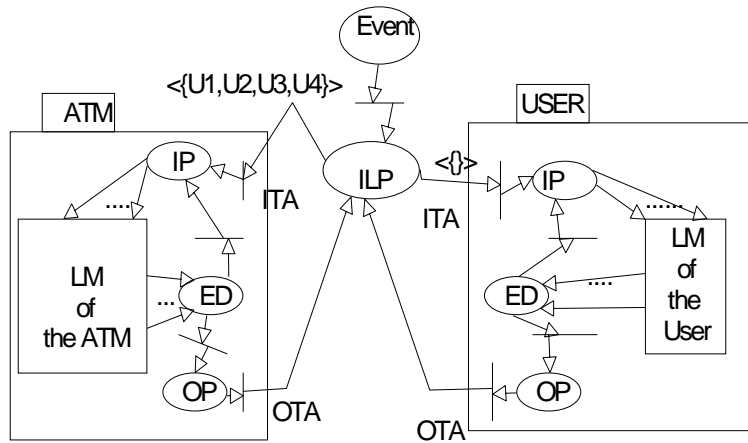
Figure 15. System-level colored Petri net for the ATM machine example
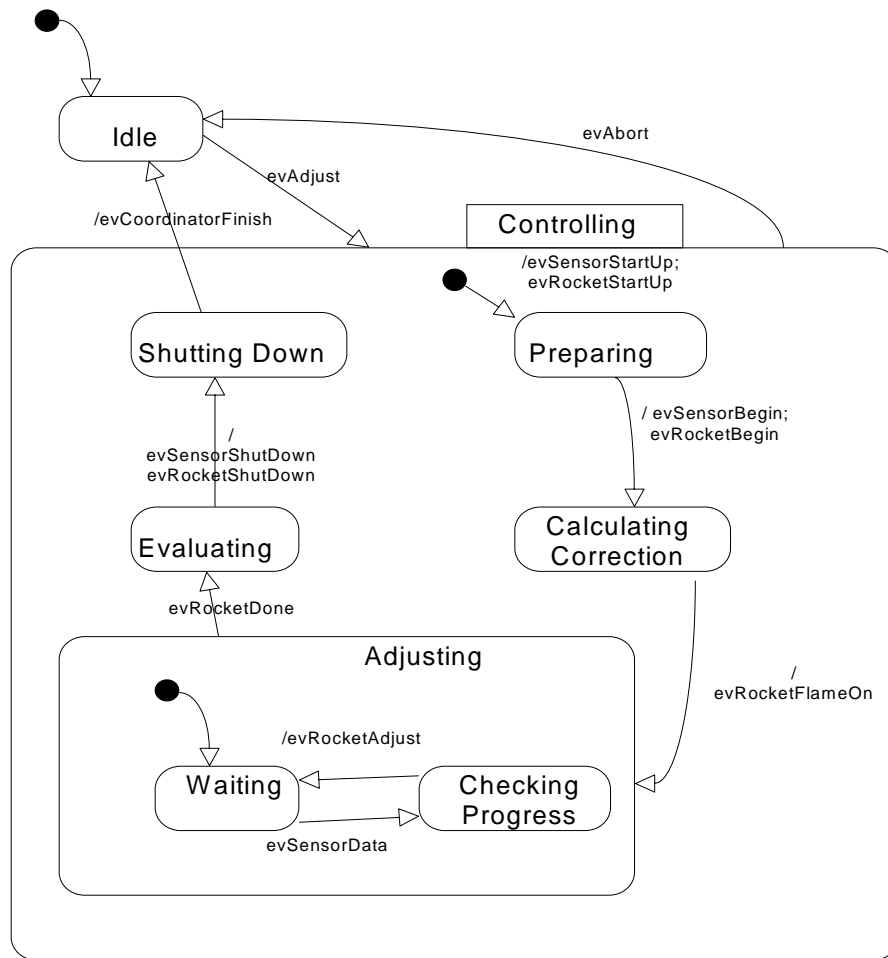
Figure 16. Statechart diagram of the Controller
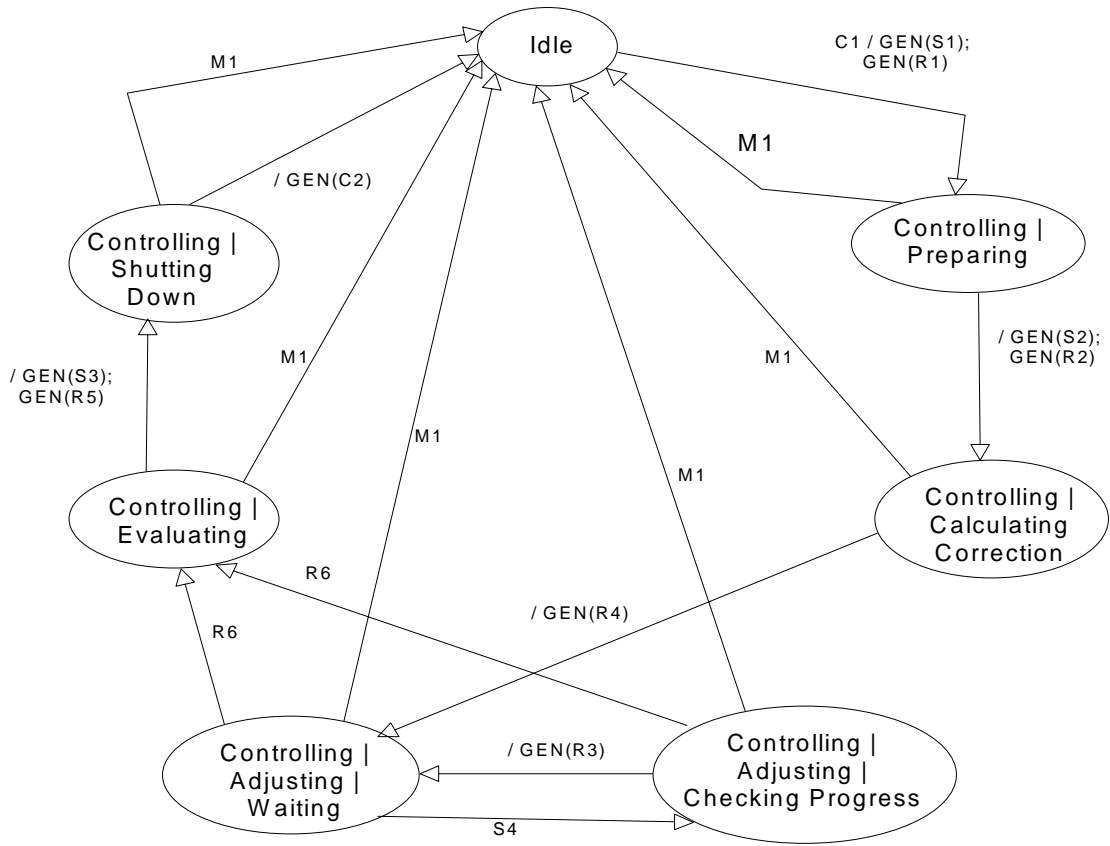
Figure 17. Object life cycle model of the Controller object
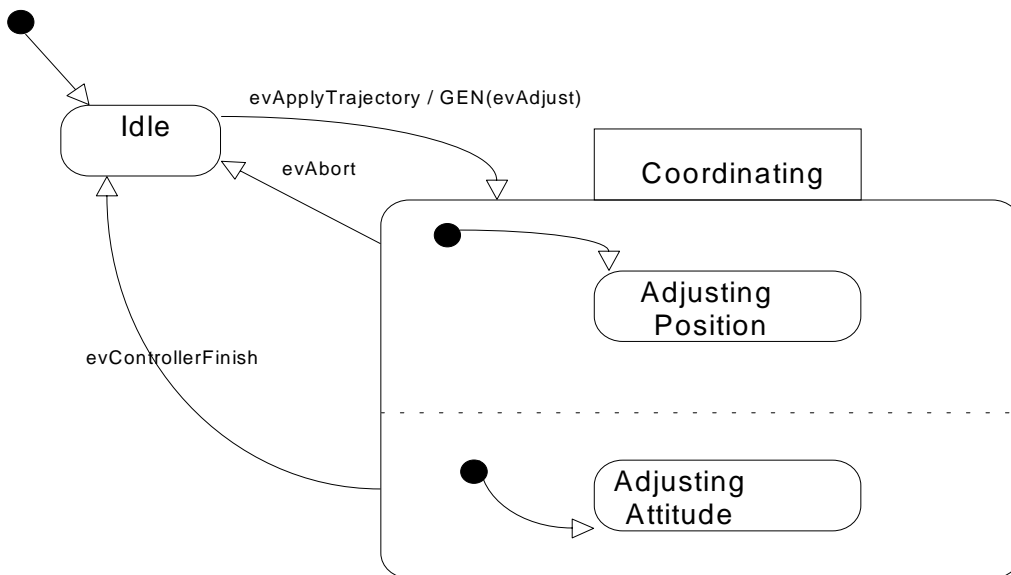
Figure 18. Statechart diagram of the Movement Coordinator
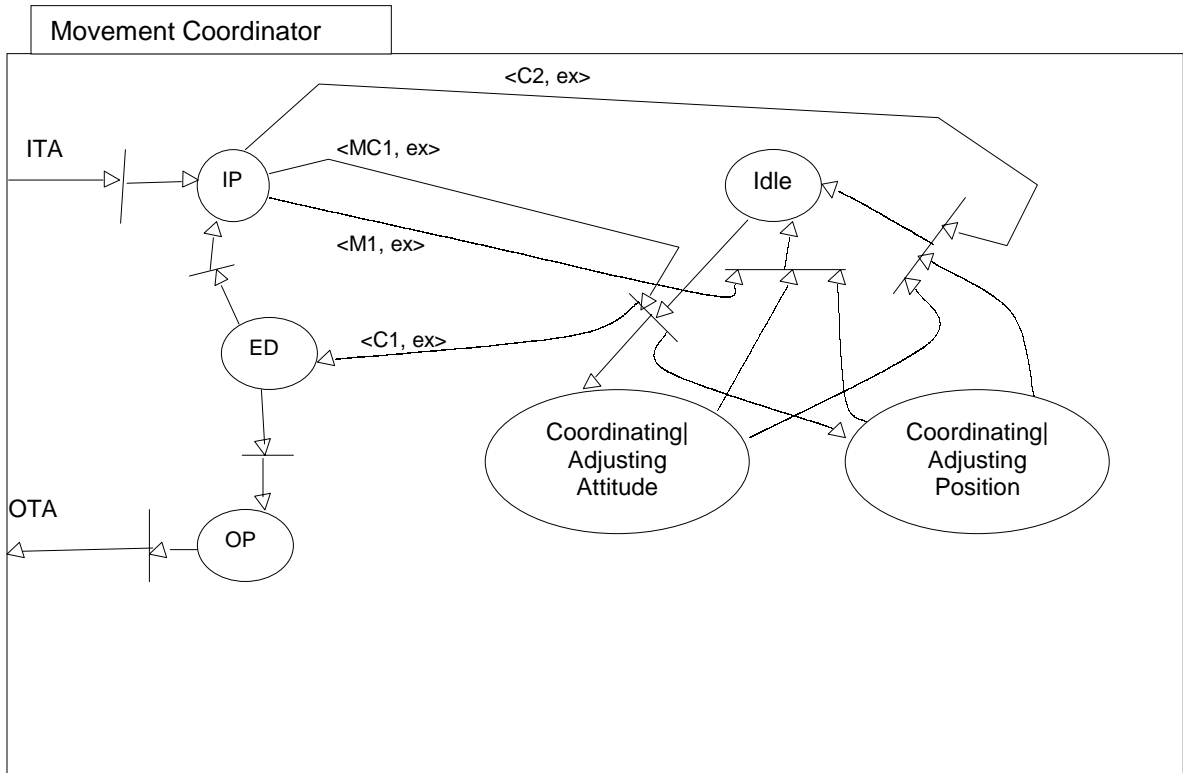
Figure 19 ONM of the Movement Coordinator object
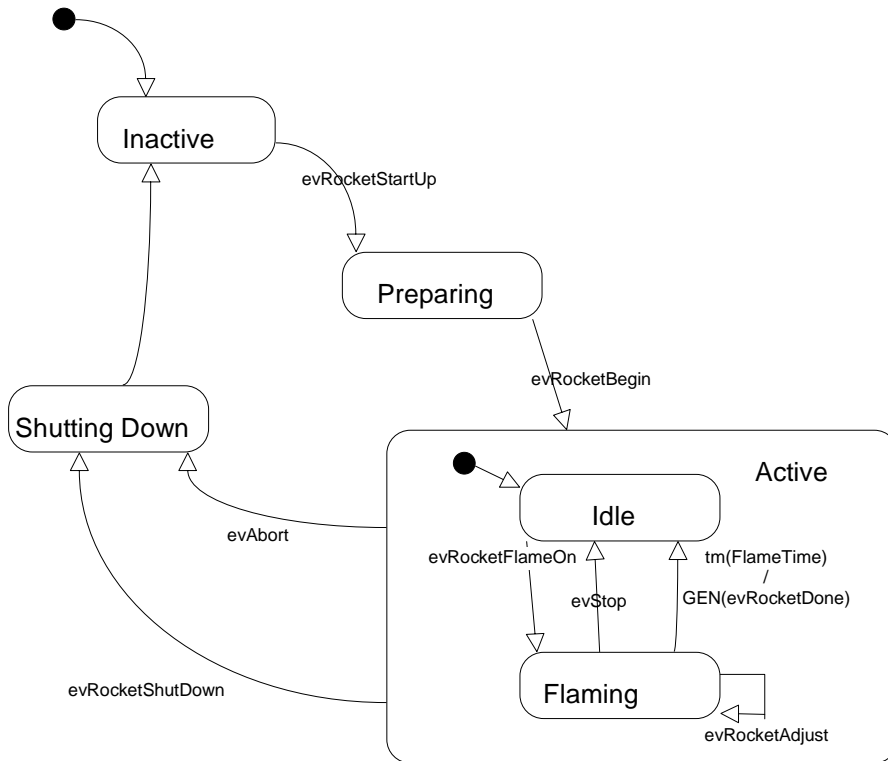
Figure 20. Statechart diagram of the Rocket
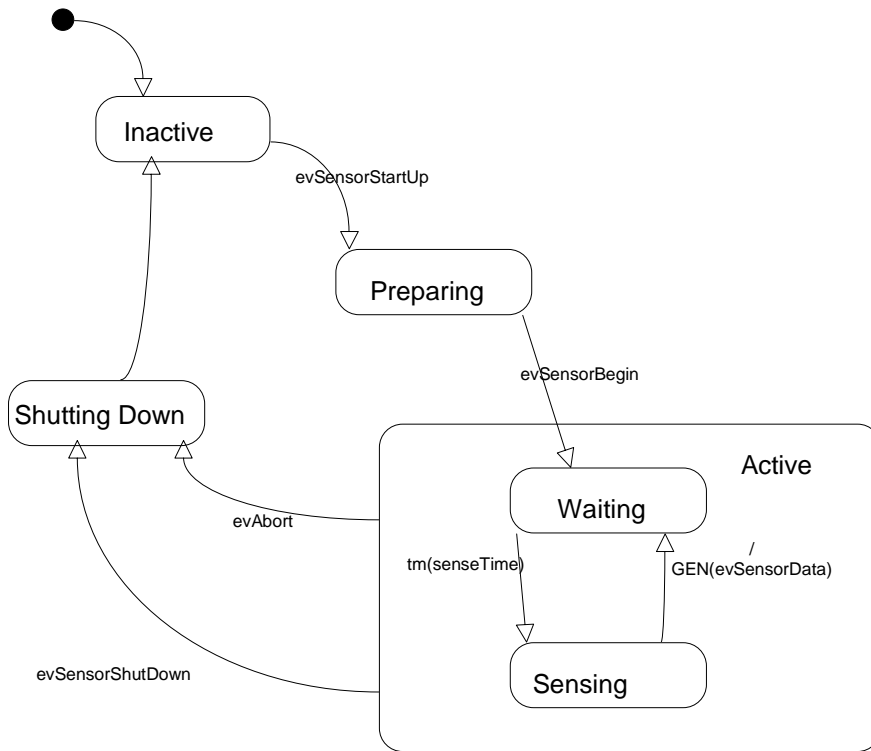
Figure 21. Statechart diagram of the Sensor
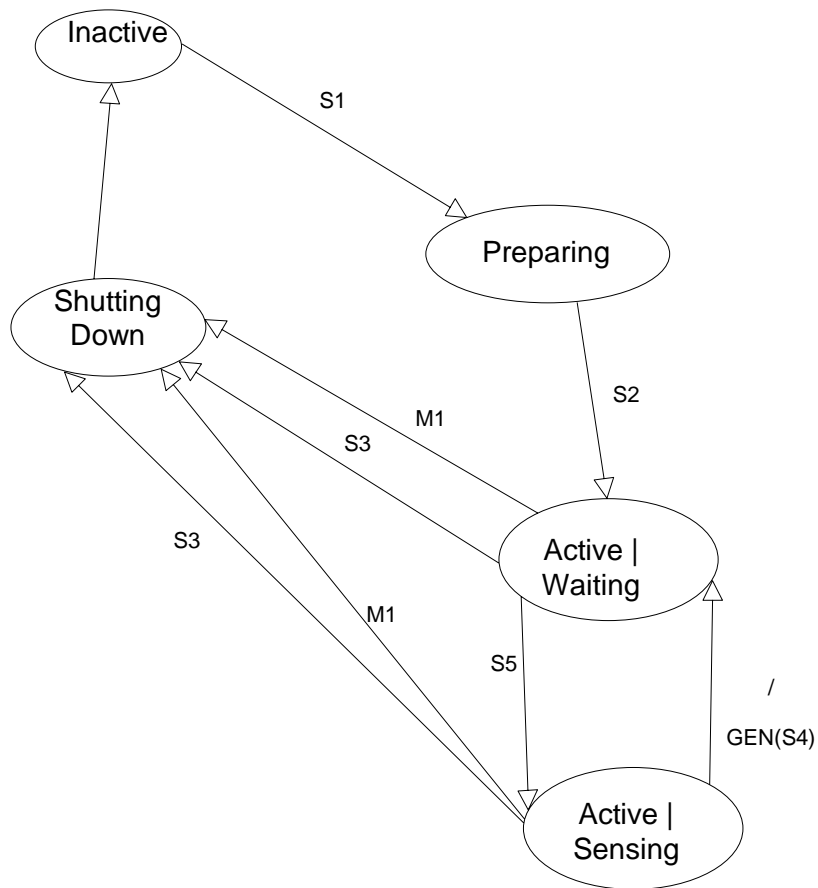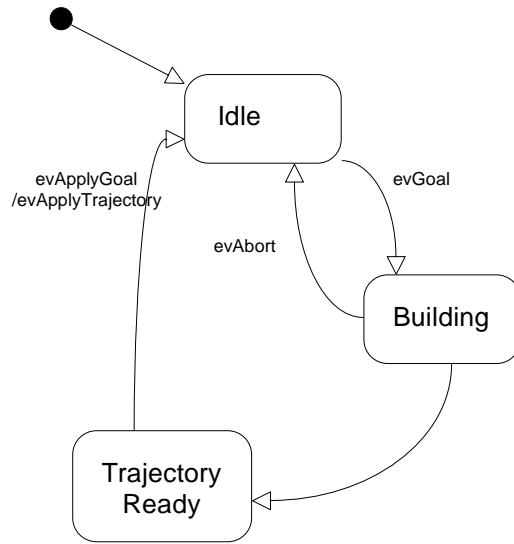

Figure 22. Object life cycle model of the Sensor object

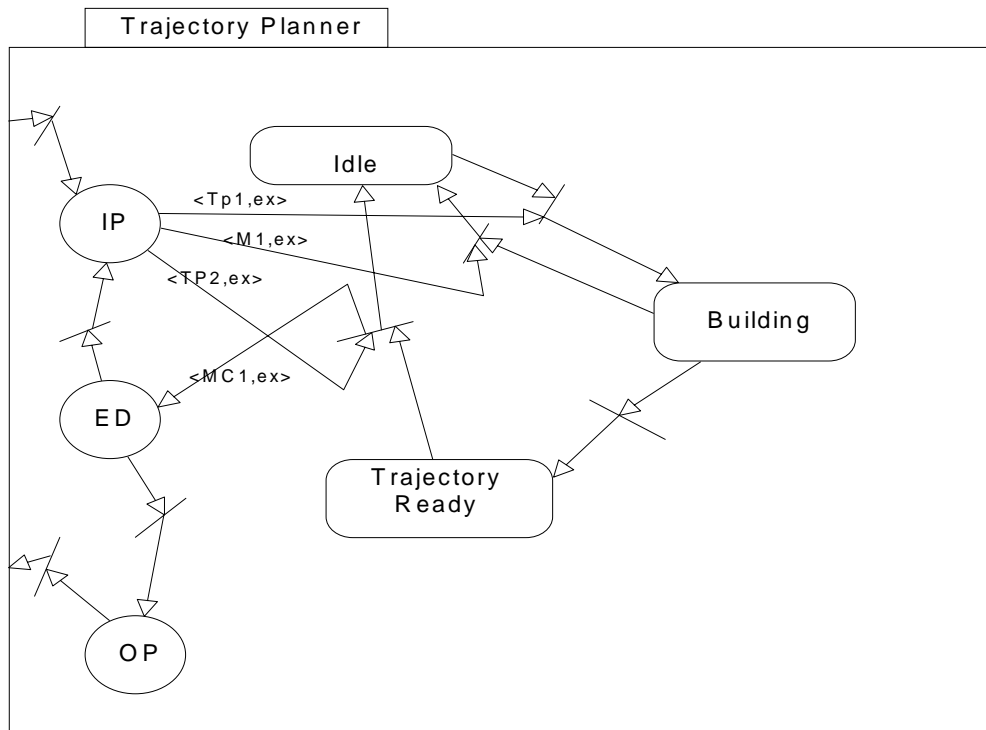Figure 23. Statechart diagram of the Trajectory Planner Object
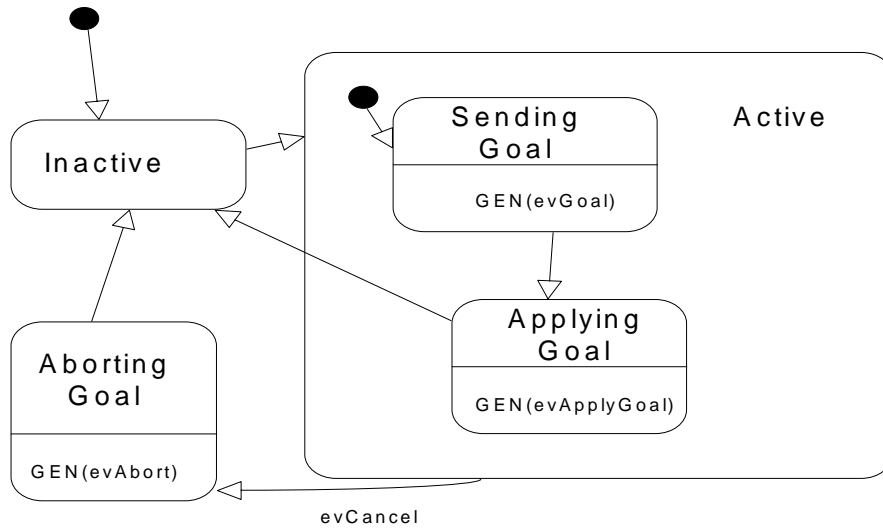

Figure 24. ONM of the Trajectory Planner object

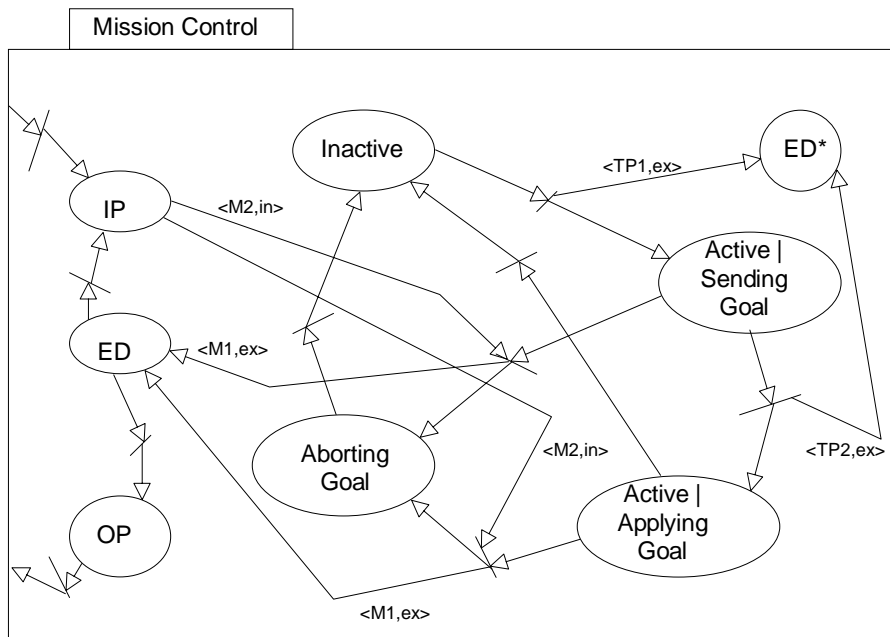Figure 25. Statechart diagram of the Mission Control object

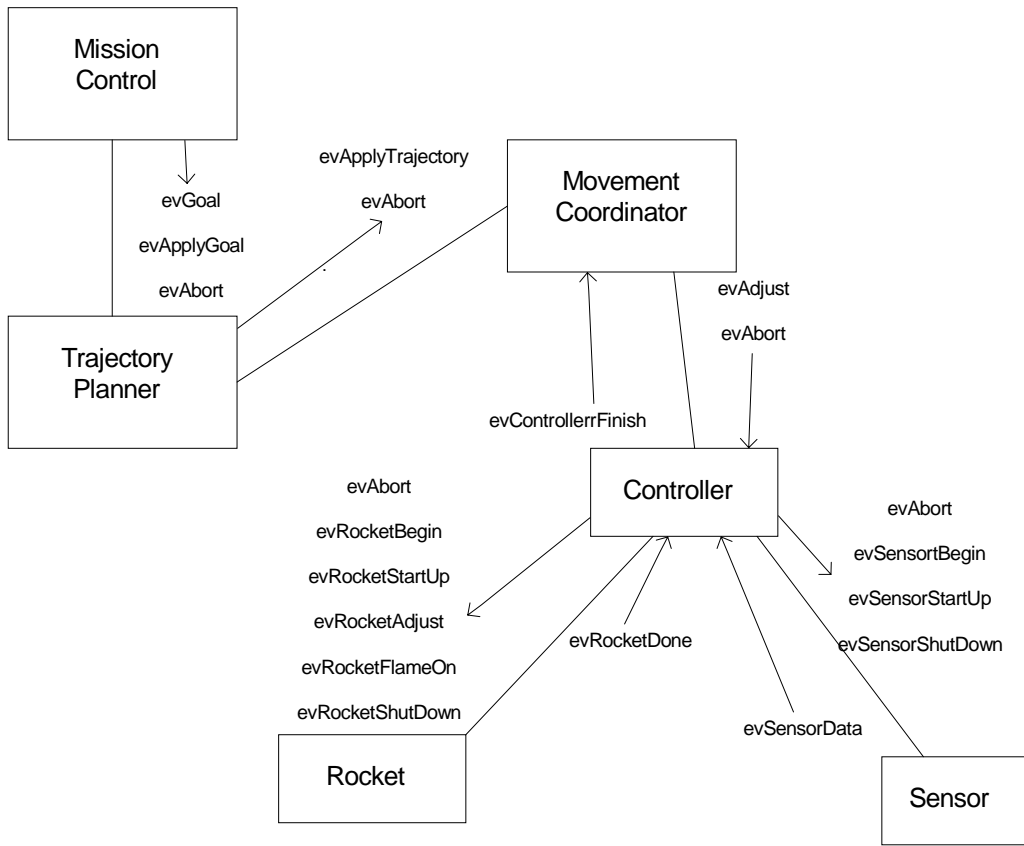Figure 26. ONM of the Mission Control object [ED is duplicated as ED* for clarity]

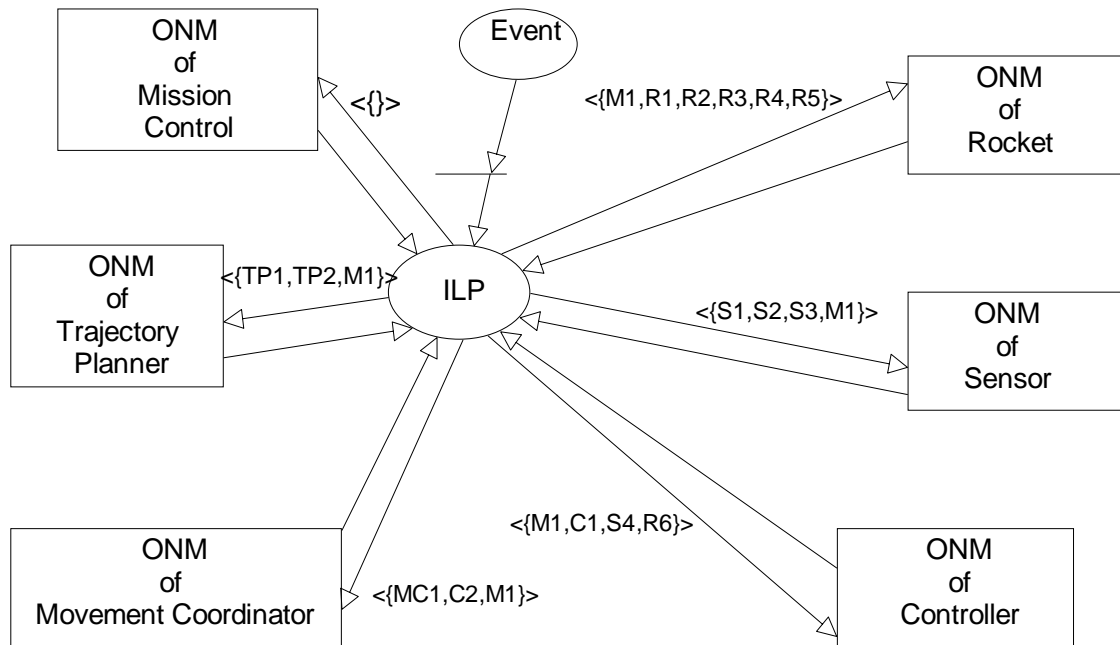Figure 27. Collaboration diagram for the Spacecraft Control example

Figure 28. System-level Petri net for the Spacecraft Control example