Modeling State-Dependent Objects using Colored Petri Nets

Robert G. Pettit IV[1] and Hassan Gomaa[2]

[1]The Aerospace Corporation, 15049 Conference Center Drive,
Chantilly, Virginia, USA 20151
rob.pettit@aero.org

[2]George Mason University, Department of Information and Software Engineering,
Fairfax, Virginia, USA 22030-4444
hgomaa@gmu.edu

Abstract. This paper describes an approach for using Petri nets to model and analyze the behavioral characteristics of state-dependent objects represented in the Unified Modeling Language (UML). Specifically, this paper describes an approach for systematically mapping UML state-dependent objects and their corresponding statecharts into colored Petri nets. This work is part of an on-going effort to automate the behavioral analysis of concurrent and real-time object-oriented software designs. The benefit of this approach is that by providing a systematic means for modeling state-dependent objects using colored Petri nets and relating this to the larger research effort, the overall concurrent architecture may then be modeled and analyzed using a single, cohesive technique.

Keywords: UML; Statechart; Colored Petri Net; Behavioral Analysis; COMET

1 Introduction

This paper presents an approach for using colored Petri nets to model and subsequently validate the behavioral characteristics of state-dependent objects represented in the Unified Modeling Language (UML) [1;2]. This research represents part of a larger effort [3] to integrate colored Petri nets with UML software architectures created using the COMET method [4]. The goal of this overall effort is to provide a systematic and seamless integration of CPNs with object-oriented software architectures in order to effectively model and analyze the behavioral properties of an architecture prior to implementation.

In COMET (Concurrent Object Modeling and Architectural Design Method), state-dependent objects are active (asynchronous) objects that react to stimuli based on the current state of the object. Each state-dependent object has an associated statechart that defines the state-based behavior for that object. Typically, a state-dependent object will have one input interface (operation) for processing events. When an event is received, the state-dependent object will then use the encapsulated statechart information to determine the appropriate behavior to execute. Based on the input that was received and the current state of the object, this behavior could range from simply changing states; executing some action; sending a message to other objects; or no behavior at all.

There are several tools currently available to simulate statechart execution. The benefit of this approach is that by providing a systematic means for modeling state-dependent objects using colored Petri nets and relating this to the larger research effort mentioned above, the overall concurrent architecture may then be modeled and analyzed using a single, cohesive technique.

The Petri net formalism was chosen based on its modeling and analytical power for concurrent systems. There are three general characteristics of Petri nets that make them

interesting in capturing concurrent, object-oriented behavioral specifications. First, Petri nets allow the modeling of concurrency, synchronization, and resource sharing behavior of a system. Second, there are many theoretical results associated with Petri nets for the analysis of such issues as deadlock detection and performance analysis. Finally, the integration of Petri nets with an object-oriented software architecture provides a means for automating behavioral analysis.

The basic notation for Petri nets is a bipartite graph consisting of places and transitions that alternate on a path and are connected by directional arcs [5]. In general, circles represent places, whereas bars or boxes represent transitions. Tokens are used to mark places, and under certain enabling conditions, transitions are allowed to *fire*, thus causing a change in the placement of tokens.

A colored Petri net (CPN) is a special case of Petri net in which the tokens have identifying attributes; in this case the color of the token [6]. At first, colored Petri nets seem less intuitive than the basic Petri net. However, by allowing the tokens to have an associated attribute, colored Petri nets scale to large problems much better than basic Petri nets.

2 The COMET Method

COMET is a Concurrent Object Modeling and Architectural Design Method for the development of concurrent applications, in particular distributed and real-time applications [4]. As the UML is now the standardized notation for describing object-oriented models [1,2], the COMET method uses the UML notation throughout.

The COMET Object-Oriented Software Life Cycle is highly iterative. In the Requirements Modeling phase, a use case model is developed in which the functional requirements of the system are defined in terms of actors and use cases.

In the Analysis Modeling phase, static and dynamic models of the system are developed. The static model defines the structural relationships among problem domain classes. Object structuring criteria are used to determine the objects to be considered for the analysis model. A dynamic model is then developed in which the use cases from the requirements model are refined to show the objects that participate in each use case and how they interact with each other. In the dynamic model, state dependent objects are defined using statecharts.

In the Design Modeling phase, an Architectural Design Model is developed. Subsystem structuring criteria are provided to design the overall software architecture. For distributed applications, a component based development approach is taken, in which each subsystem is designed as a distributed self-contained component. The emphasis is on the division of responsibility between clients and servers, including issues concerning the centralization vs. distribution of data and control, and the design of message communication interfaces, including synchronous, asynchronous, brokered, and group communication. Each concurrent subsystem is then designed, in terms of active objects (tasks) and passive objects. Task communication and synchronization interfaces are defined. The performance of real-time designs is estimated using an approach based on rate monotonic analysis.

The aspect of the COMET method that is specifically addressed in this paper is the emphasis on dynamic modeling, in the form of both object interaction modeling and finite state machine modeling, describing in detail how object collaborations and statecharts [8,9,10] work together.

3 Statecharts in the COMET Method

3.1 State Transition Diagrams and Statecharts

Traditionally, finite state machines were modeled by means of state transition diagrams or state transition tables. One of the potential problems of state transition diagrams is the proliferation of states and transitions, thereby making the state transition diagram very cluttered and difficult to read. A very important way of simplifying state transition diagrams was made by Harel with the introduction of statecharts [8,10], which increases the modeling power of state transition diagrams by introducing superstates and the hierarchical decomposition of superstates.

Significant simplification of statecharts can often be achieved through the use of hierarchical decomposition of states, where a superstate is decomposed into two or more interconnected substates. This decomposition is sometimes referred to as the **or** decomposition, because being in the superstate means that the statechart is in one and only one of the substates. The hierarchical statechart notation also allows a transition out of every one of the substates on a statechart to be aggregated into a transition out of the superstate. Careful use of this feature can significantly reduce the number of state transitions on a statechart.

Another kind of hierarchical state decomposition supported is the **and** decomposition. That is, a state on one statechart can be decomposed into two or more concurrent statecharts. When the higher-level statechart is in the superstate, it is simultaneously in one of the substates on the first lower-level concurrent statechart *and* in one of the substates on the second lower-level concurrent statechart.

Although the name concurrent statechart implies that there is concurrent activity within the object containing the statechart, the **and** decomposition can be used to show different aspects of the same object, which are not concurrent. This latter approach is used in the COMET method.

3.2 State Dependent Objects and Statecharts

State dependent objects are control objects whose behavior depends not only on the input received, but also on the current state. In COMET, a state dependent object is described by means of a statechart. A complex system can have many state dependent objects and hence several statecharts. COMET encourages the design of objects with only one thread of control; to address concurrency, multiple concurrent objects can be used. The Cruise Control system described in this paper has one state dependent object and hence one statechart. However, in other case studies given in [4], there are examples of distributed control in which the control aspects of the system are distributed among many state dependent objects, each described by its own statechart. If there are many objects of the same type, then each object will execute an instance of the same statechart.

4 Modeling UML State-Dependent Objects Using Colored Petri Nets

Using the COMET method, state-dependent objects are defined to encapsulate the behavior specified by a statechart. These state-dependent objects provide an interface for receiving events and then perform some behavior based on the input event, the current state, and the state-transition specifications of the encapsulated statechart.

To fit within the context of modeling a large-scale concurrent software architecture, the approach used in this paper to specifically model state-dependent objects must address both the high-level behavioral structure of state-dependent objects as well as the specific state-dependent behavior of the associated statechart. To capture both of these aspects, the resulting CPN model is structured using a series of hierarchical decompositions as described in the following sections.

4.1 CPN Structural Model of State-Dependent Objects

Using our approach, the top-level CPN model captures the high-level state-dependent object as a whole. This top-level model is illustrated in Figure 1. At this level, we can see the interface between the state-dependent object and its surrounding environment, including input events, output actions, the current state, and the high-level control flow within the object.

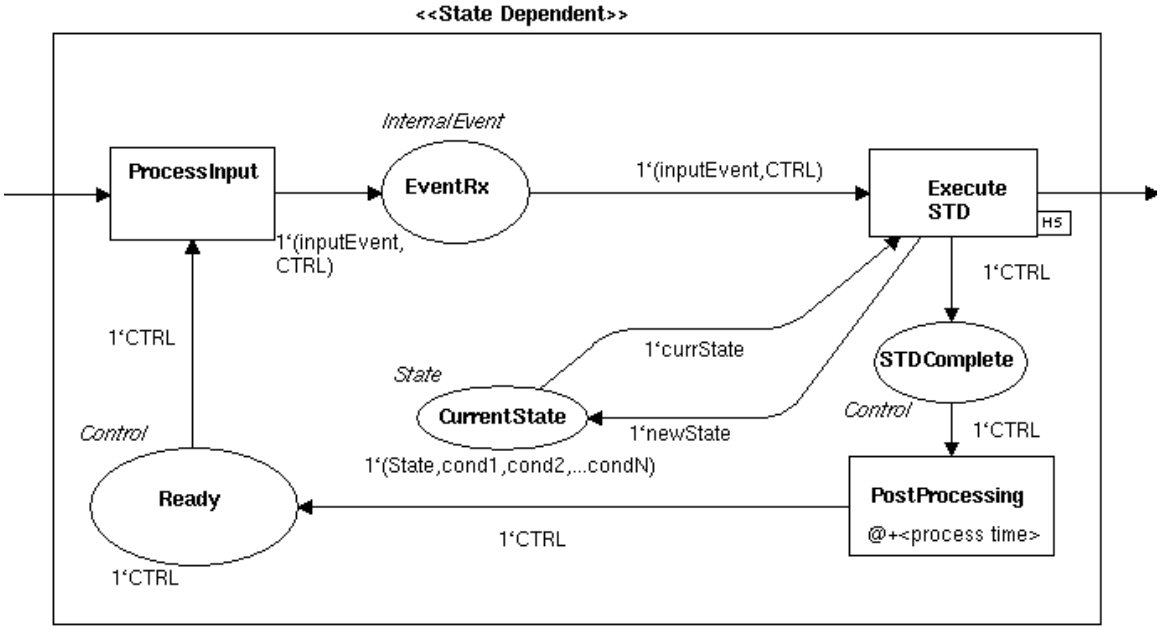

Figure 1. Top-Level CPN Model for State-Dependent Objects

In the COMET method, state-dependent objects are modeled as *active* objects in UML, thus indicating that they operate concurrently with their own thread of control. In our CPN model, this concept is maintained by using a control token (labeled *CTRL* in the CPN diagrams) to model the flow of control within the object. Each active object is modeled with its own control token, which, in addition to modeling the flow of control, is used to model synchronization with other objects and (with a timestamp) is used to simulate the progression of time for each object.

A state-dependent object modeled with the CPN segment from Figure 1 would begin its lifecycle with a control token residing in the *Ready* place. Thus, the *ProcessInput* transition would then be enabled by the presence of an input event. Upon firing, *ProcessInput* would pair the input event with the control token and pass this tuple to the *EventRx* place, indicating that an event has been received and needs to be processed through the state transition diagram. The *ExecuteSTD* transition is decomposed to model the specific statechart behavior encapsulated by the state-dependent object. This decomposition is described in Section 4.2.

Once the statechart model has completed its execution (including any output actions), the control token is passed to the *STDComplete* place. The *PostProcessing* transition then increments the time tag on the control token (to simulate the processing time for the state-dependent object to handle an event) and returns the control token to the *Ready* place to await the next event.

One final note to be aware of at this level is the definition of state. The current state is maintained in the *CurrentState* place. The *ExecuteSTD* transition (and its subsequent decompositions) remove the current state from this place with each input event and return the modified (new) state upon completion. The state of the object itself is actually represented by a tuple stored at *CurrentState*. This tuple not only contains the "state" of the system as we would normally consider it, but it also contains the current status of system conditions used to make transition determinations in the statechart. Use of these conditions combined with the state is further illustrated in Section 5 with an example from the Cruise Control System.

4.2 Top-Level CPN Statechart Model

To begin modeling, the actual state-dependent behavior is first converted to a "flattened" representation so that all state transitions are explicitly captured between leaf states. Any necessary conditions that were captured in the hierarchical statechart (e.g. device status) are then represented as conditions on the state transition and are also captured as part of the conditions in the *CurrentState* tuple. Furthermore, "entry" actions for UML states are mapped to actions on the corresponding entry transition(s) to those states. Similarly, "exit" actions are mapped to actions on the corresponding exit transitions(s). Activities that are persistent within a given state (represented by the UML "Do" keyword) are also mapped to actions on the entry transition(s) as these persistent activities are implemented by separate active objects in the COMET method [4].

Working now from a flattened statechart, the *ExecuteSTD* transition is then decomposed into a lower-level CPN. This first-level decomposition provides one CPN transition for each (leaf) state in the object's statechart. Furthermore, the places providing tokens to and receiving tokens from *ExecuteSTD* are shown at this level as *port* places. A port place is used as a connector between hierarchical levels of CPNs. Thus, the *EventReceived* place shown at the root level providing input to *ExecuteSTD* is identical to all subsequent instances of *EventReceived* in the various levels of *ExecuteSTD's* decompositions.

Figure 2 provides an illustration of this first-level decomposition. Note that all transitions (representing the states of the object's statechart) are connected to the *EventReceived*, *STDComplete*, and *CurrentState* places. Additionally, if a given state generates an external action, appropriate links will also be added to handle those cases. Furthermore, each CPN "*State*" transition in Figure 2 uses an arc inscription on the input arc to enable the transition only if the current state corresponds to the state being modeled by that transition. Thus, the *State1* transition is only enabled when the current state is equal to *State1* and so forth.

4.3 Modeling Behavior for Individual States

To model the specific state-dependent behavior, each of these CPN *State* transitions from Figure 2 is again decomposed into a lower-level CPN segment. This final level of decomposition is shown in Figure 3.

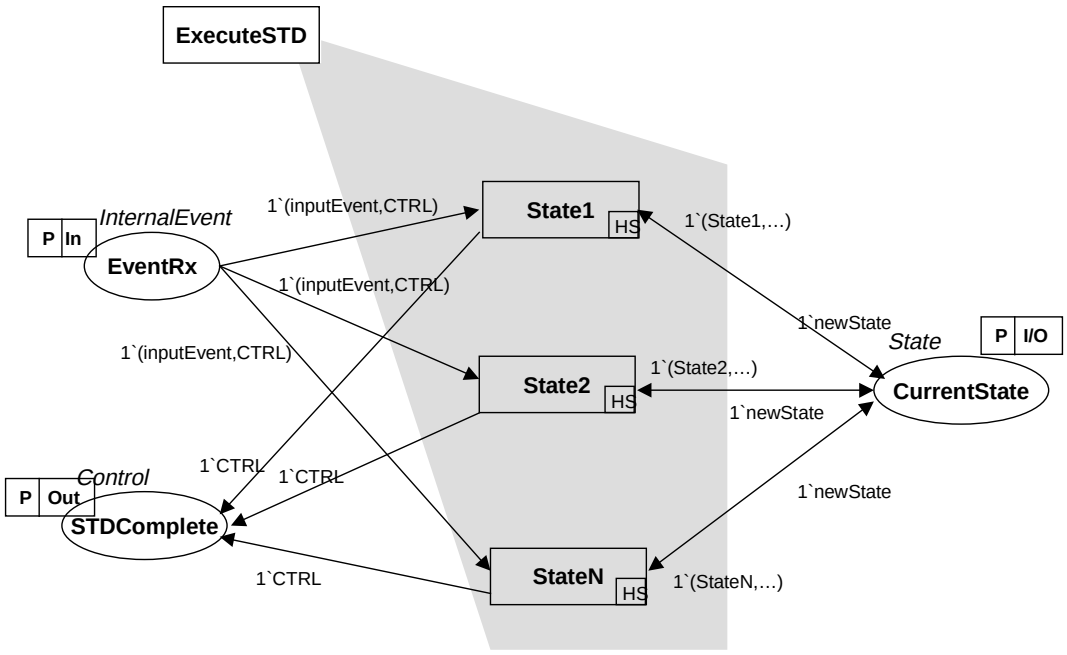

Figure 2. First Level Decomposition of CPN Statechart Model

At this level, there is one transition (*State-n-DetSt*) to determine the next state and the particular action branch that needs to be taken. Once the current state is determined and the appropriate *DetSt* transition is enabled (using the arc inscription from *CurrentState*), this transition executes a code segment to determine the next state (*newState*) and the action branch (*branch*) based on the current state and the input event (*inputEvent*). This code segment simply consists of a case statement in the ML language that specifies the desired output of new state and action branch for each possible input event.

After the *newState* and *branch* values are determined, a tuple containing these values is passed to the *State-n-Branch* place. From this branch place, one branch is chosen based on the value of the *branch* variable. Each branch is a series of alternating transitions and places that model the behavior of the action to be performed. Each transition in a branch is labeled by the name "Action", followed by the branch number (0..n), followed by the step number (a..z). In our approach, branch zero (0) will always be used when no action (other than a possible state change) is required.

For branches with more than one step, an interim place is used to connect each transition in the path. This interim place contains the new state tuple and is labeled by the name, "*State*" followed by the branch number, followed by the previous and next step numbers. For example, in Figure 3, the interim state along branch 1 between steps "a" and "b" is labeled, "*State-n-State1ab*".

Along these action paths, the CPN model may generate tokens to be passed as messages, events, or operation calls to other objects. In the case of asynchronous actions, the branch may be continued uninterrupted. However, in the case of synchronous processing, the next step must wait for the previous step to complete before continuing. This is accomplished by having the next transition enabled by both the interim state place and a return place from the synchronizing action. An example of a synchronous message can be found in the Cruise Control example of Section 5.

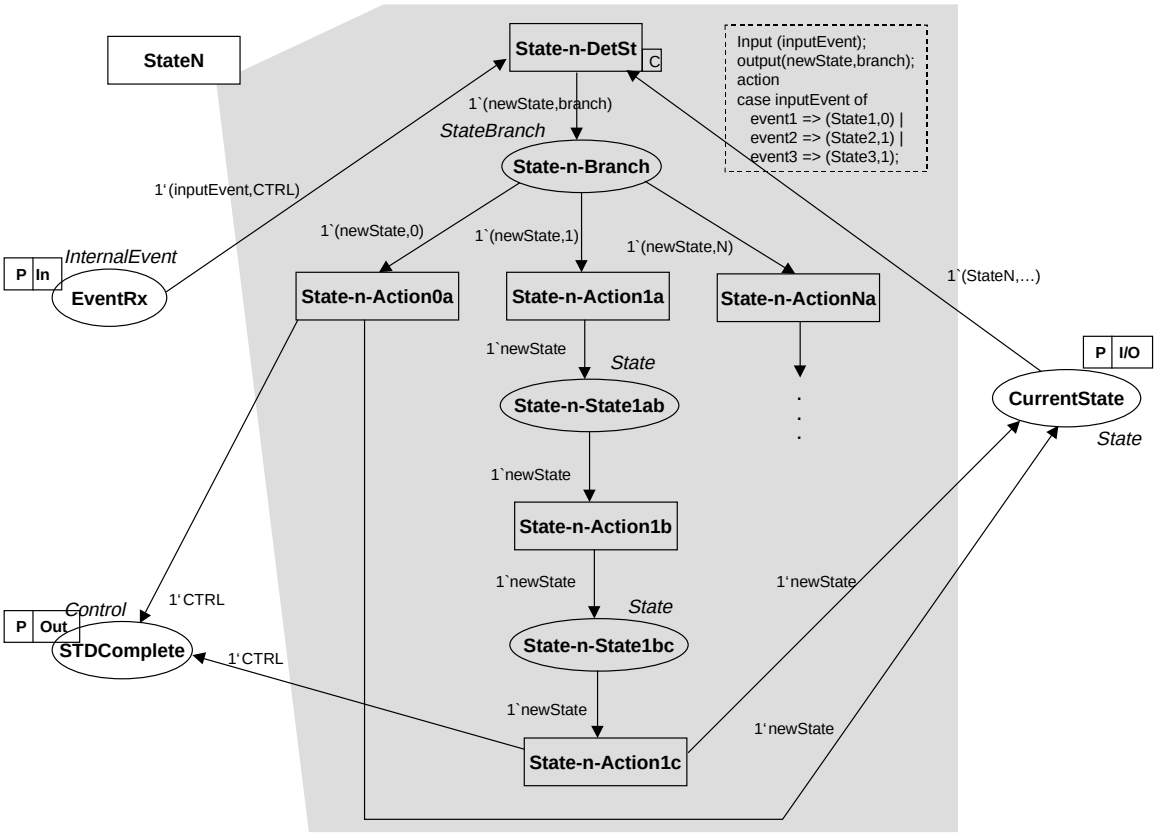

Figure 3. Detailed State Decomposition

Finally, when a given path has completed, it's last action is to update the *CurrentState* place with the new state and to pass a control token to the *STDComplete* place indicating that the current input event has now been processed.

The *STDComplete* place then essentially returns to the top level CPN diagram and passes the control token to the *PostProcessing* transition. This transition performs any necessary post processing actions and uses its time delay to simulate the processing time for the state-dependent object. When this transition completes, the control token is returned to *Ready* place and the model is ready to process the next input event.

At this point, the CPN model for the state-dependent object may be executed through a simulator or analyzed by constructing an occurrence graph of the state-space using tools such as DesignCPN [7]. The state-dependent object may be analyzed in isolation using CPN places for the input and output stubs or it may be integrated into the CPN model of the overall concurrent software architecture.

In the following section, we use an example from the Cruise Control System to illustrate how a specific state-dependent object within a software architecture may be modeled using the approach described in the above sections.

5 Case Study: Cruise Control System

The Cruise Control System [4] is a real-time control system that manages the speed of an automobile based on inputs from the driver (via a lever on the steering column). The behavior of the cruise control is state-dependent in that the executed actions correspond not only to the driver input, but also on the current state of the system and with the status of the engine and the brake.

To illustrate the modeling of state-dependent objects using CPNs, we will use the state-dependent ":*CruiseControl*" object and corresponding statechart from the Cruise Control System. In this example, the :*CruiseControl* object accepts external inputs/events and executes the cruise control statechart. Figure 4 provides a partial collaboration diagram illustrating this state-dependent object along with its interfaces to the rest of the cruise control system. As can be seen from this figure, the *:CruiseControl* object accepts event inputs from the cruise control lever interface indicating whether the driver has selected to accelerate (*Accel*), engaged the cruise (*Cruise*), turned the cruise off (*Cruise Off)*, or requested that cruising be resumed (*Resume*). Furthermore, the *:CruiseControl* object must accept messages from the *SpeedControl* object indicating that the cruising speed has been reached and accept inputs indicating the current status of the brake (*Pressed* or *Released*) and engine (*On* or *Off*). Finally, in terms of output actions, the :*CruiseControl* object must set (or clear) the desired speed and must send appropriate (state-based) commands to the *SpeedControl* object that controls the throttle output and monitors the current speed.

The statechart for the :*CruiseControl* object is shown in Figure 5. The leaf (lowest-level) states for *:CruiseControl* are: Idle, Initial, Accelerating, Cruising, Resuming, and Cruising Off. It is these low-level states that will be used for the CPN state-dependent modeling.

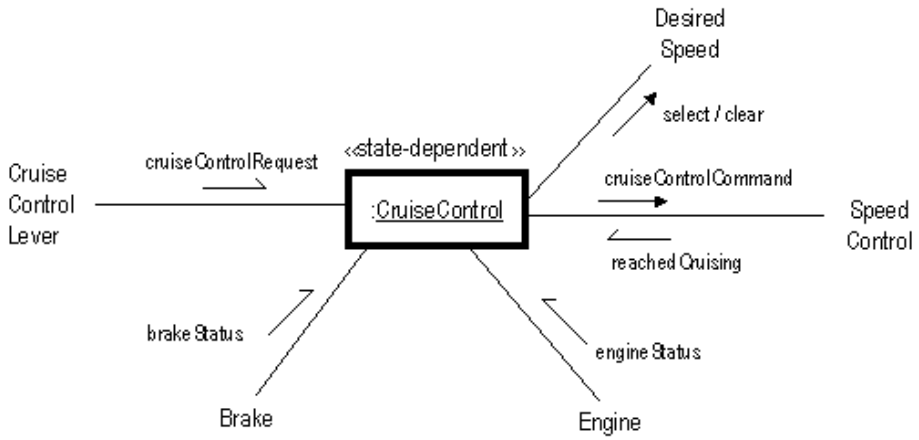

Figure 4. Cruise Control State-Dependent Object

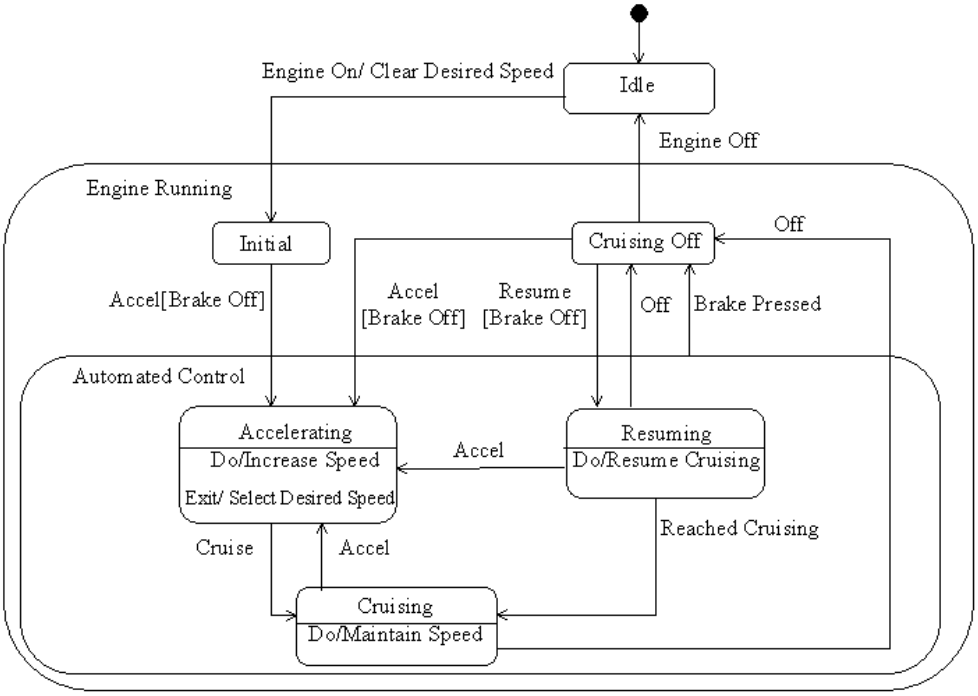

Figure 5. Cruise Control Statechart
Figure extracted from [4] with permission of Addison-Wesley Publishing.

5.1 CPN Model for Cruise Control

To start modeling the behavior of the *:CruiseControl* state-dependent object, we begin by using a CPN segment to capture high-level behavioral properties of the state-dependent object and its connections to the rest of the software architecture. This CPN segment is illustrated in Figure 6.

At this level, the CPN models the general control flow of a state-dependent object. In the case of the *:CruiseControl* object, this segment has been tailored to accept cruise control messages as the input events via the *ProcessInput* transition. Additionally, the *ExecuteSTD* transition is connected to CPN places corresponding to the *Select* and *Clear* operations of the *DesiredSpeed* entity object as well as the places corresponding to the synchronous message buffer for the *SpeedControl* object. The *State* colorset must also be tailored to fit the *:CruiseControl* object and in this case is declared to be a tuple (specifically, a triple) consisting of tokens representing the cruise control leaf states; the engine status; and the brake status.

With the high-level CPN segment in place, we can now begin modeling the state-dependent behavior of *:CruiseControl* by decomposing the *ExecuteSTD* transition into a set of CPN transitions representing the leaf states of the cruise control statechart. This first level decomposition is shown in Figure 7. Using the approach presented in Section 2, one CPN transition is used for each of the six low-level states of the cruise control statechart: Idle, Initial, Accelerating, Cruising, Resuming, and CruisingOff. Each of these CPN transitions accepts a tuple containing the cruise control message and control token from the *EventReceived* place along with the current state from the *CurrentState* place. When tokens are available on both the *EventReceived* and *CurrentState* places, the transitions will then be enabled according to the arc inscription from *CurrentState*. Thus, the *Idle* transition is only enabled if the current state is Idle; the *Initial* transition is only enabled for a current state of Initial; and so forth. Finally, the output from each of these transitions updates the

CurrentState place with the *newState* token and sends a control token to the *STD Complete* place indicating that processing has been completed for the current input event.

The final step in modeling the state-dependent behavior is to decompose each of the transitions in Figure 7 into a CPN segment capturing the behavior for each of the cruise control states. To illustrate this final step, the CPN models for two of the cruise control states (Idle and Accelerating) are described in detail in the following sections. The remaining CPN models can be derived by applying this same systematic approach.

5.1.1 Idle State

The idle state is the starting point for the cruise control statechart. The CPN implementation of the cruise control idle state is shown in Figure 8. In the cruise control statechart shown in Figure 5, there is only one transition identified from the idle state – this occurs when the engine is turned on and the system transitions to the initial state. However, in the CPN implementation of the state-dependent behavior, changes in conditions that affect the cruise control system must also be accounted for. These conditions include changes to the brake and engine status. Thus, the implementation for the idle state not only checks for an "EngineOn" event, but must also check for the other possibilities of "BrakeOn," "BrakeOff," and "EngineOff". When one of these three "events" occurs, the code segment of the *IdleDetSt* transition sets the status variables accordingly, and sets the branch value to 0 (indicating that no further action is necessary). This new tuple is then passed to the *IdleBranch* place. From the *IdleBranch* place, arc inscriptions determine which branch (CPN transition) will be enabled. With the branch value set to 0, the *IdleAction0a* transition is enabled and, when fired, simply updates the *Current State* and sends a control token to the *STD Complete* place indicating that processing is complete for this input event.

The only input event that produces an action and state transition from the idle state is the change in engine status as indicated by a *ccMsg* token with the value, "EngineOn". When this occurs, the code segment of *IdleDetSt* sets the output tuple to be *(Initial,true,bs,1)* indicating that the state should be changed to the initial state; the engine is turned on; the brake status (*bs*) is unchanged; and the action branch is 1. This tuple is then given to the *IdleBranch* place and enables the *IdleAction1a* transition. According to the cruise control statechart, the action that occurs when transitioning from idle to initial states is to clear the desired speed. Thus, the *IdleAction1a* "calls" the clear operation by passing a control token to the *Clear_6* place and then placing the state tuple in the *IdleState1ab* place to continue down this action branch. (The *Clear_6* place is part of the CPN model for the *DesiredSpeed* entity object of the cruise control software architecture. The number six (6) appended to the place name represents the object identifier for *DesiredSpeed*.) Since we are simulating an operation call, we need to have some synchronization so that we wait for the operation to complete before continuing our processing. This is accomplished by having the next transition in the branch, *IdleAction1b*, wait for both a state tuple in the *IdleState1ab* place and a returned control token in the *ClearRtn_6* place (indicating that the clear desired speed operation has completed). Since this is the final step in the action branch, the *IdleAction1b* transition updates the *Current State* place with the new state and conditions and sends a control token to the *STD Complete* place, completing the processing of this input event.

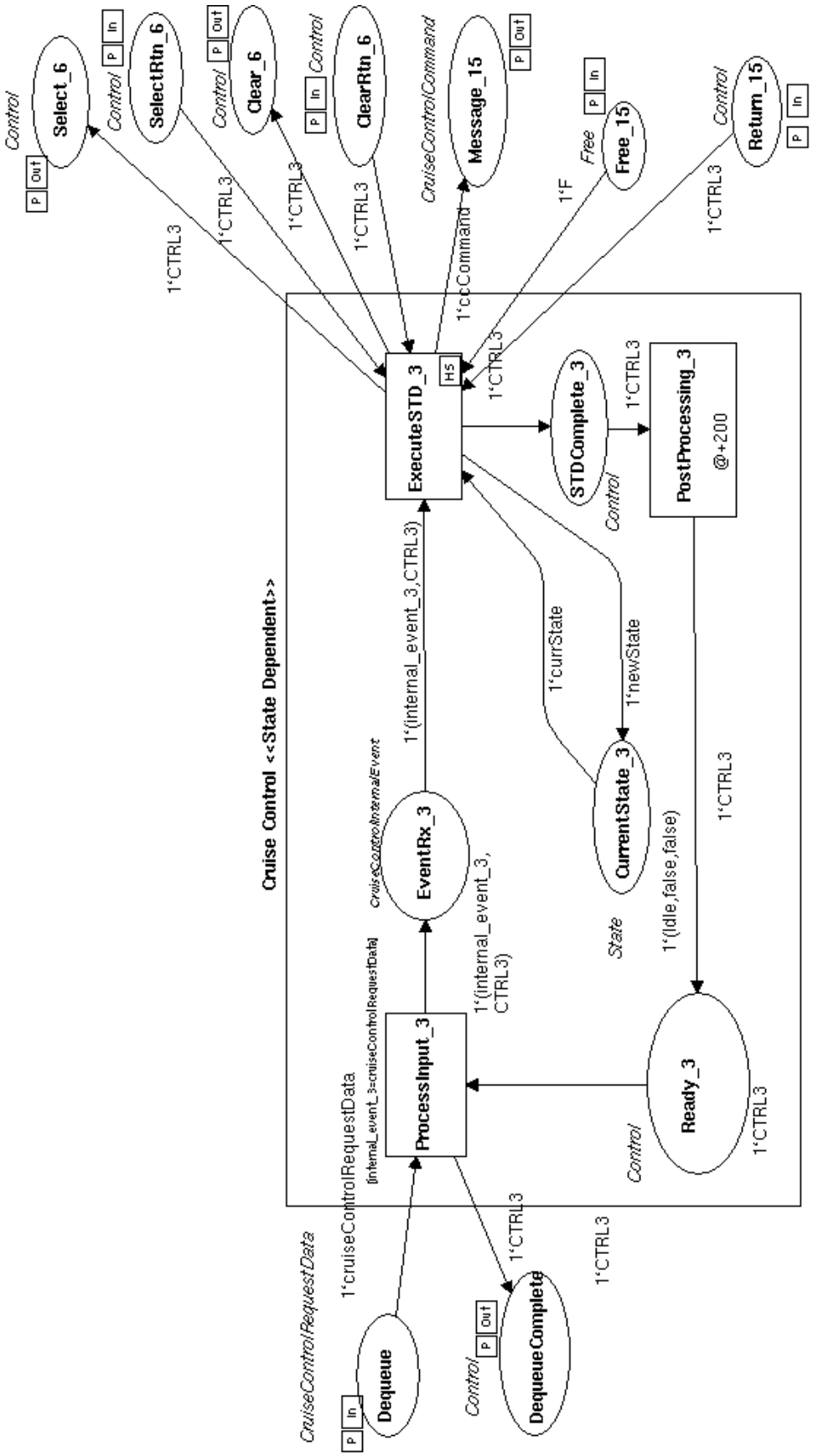

Figure 6. Cruise Control Object: Top-Level CPN Segment

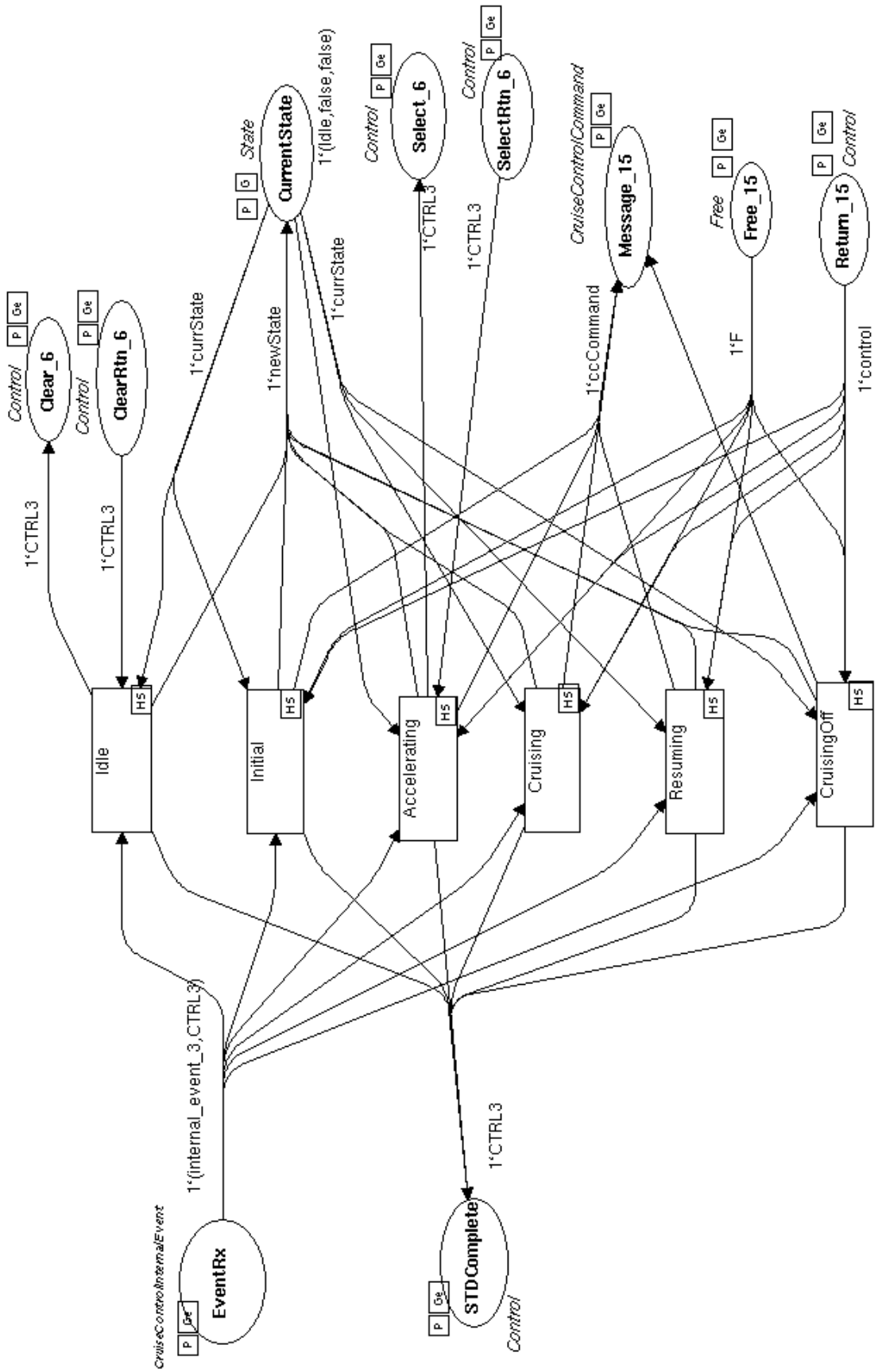

Figure 7. Cruise Control: First Level Decomposition

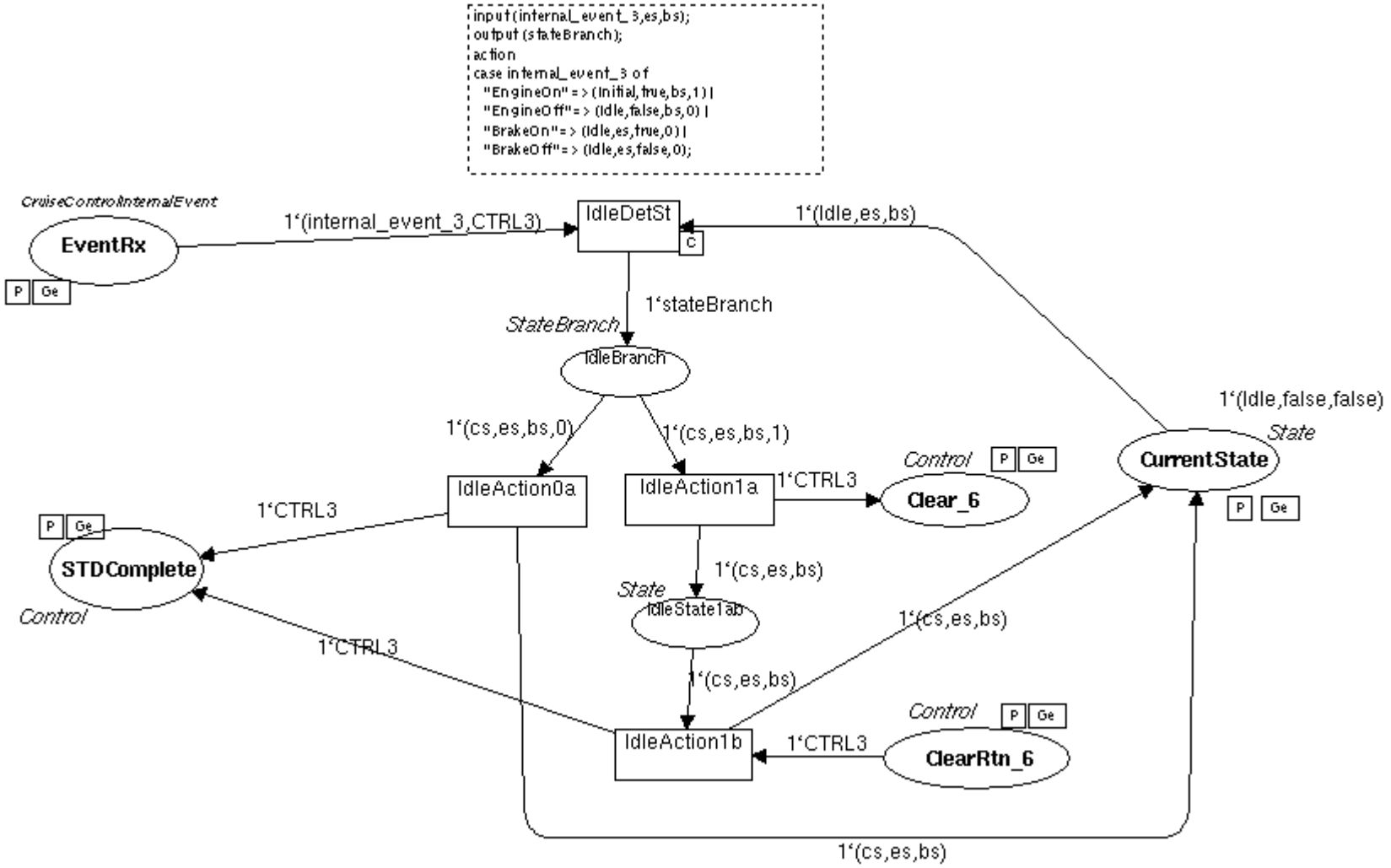

Figure 8. Cruise Control: Idle

5.1.2 Accelerating State

The behavior captured by the Accelerating state is slightly more significant than that provided by the Idle state. While in the Accelerating state, the automobile is accelerating to reach the desired cruising speed. When a "Cruise" request is detected, the system transitions from the Accelerating state to the Cruising state. During this transition, the Desired Speed is selected and (as indicated by the "Do" activity of the Cruising state) a "Maintain Speed" command is sent to the *SpeedControl* object. The other possible transitions from the Accelerating state are for the driver to press the brake, thus disengaging the cruise control and transitioning to the Cruising Off state or the engine stops and the system transitions to Idle.

The CPN model for the Accelerating state is shown in Figure 9. As with the Idle state, we again enter the model at a CPN transition (*AccelDetSt*) that uses a code segment to determine the new state and the action branch used to produce any desired actions. The only transition that produces actions from the Accelerating state is a transition to the Cruising state. Thus, this transition is assigned action branch 1 while all other inputs are assigned branch 0 and simply update the state and current engine and/or brake conditions.

Proceeding through branch 1, the first action is to select the desired speed. Thus, the *AccelAction1a* transition sends a control token to the CPN place corresponding to the Select Desired Speed operation call. The internal state information is transitioned to the *AccelState1ab* place and the *AccelAction1b* transition then waits for the Select Desired Speed operation to return as indicated by a control token in the *Select_6* place. Once this has completed, the state information is passed to the *AccelState1bc* place and the *AccelAction1c* transition performs the next action by sending a "MaintainSpeed" synchronous message to the *SpeedControl* Message buffer. Since this is a synchronous message, the *AccelAction1c* transition must first retrieve a *Free* token from the buffer, thus indicating that the transition is free to place a message in the *Message* place of the buffer. The *AccelAction1c* transition then sends the *ccCommand* token (set to "MaintainSpeed") to the *Message* place and passes the internal state information to the *AccelState1cd* place. The *AccelAction1d* transition must now wait for *SpeedControl* to retrieve the message from the buffer (since this is modeling synchronous communication). Once *SpeedControl* retrieves the message, it places a control token in the *Return* place and then *AccelAction1d* can fire, sending the updated state information to the *Current State* place and the control token to the *STDComplete* place. At this point, the processing for the Accelerating state is now complete and the *:CruiseControl* object may process the next event.

5.1.3 Modeling the Remaining Cruise Control States

The remaining cruise control states can be modeled by decomposing the remaining CPN transitions from Figure 7 using the same systematic approach used to model the Idle and Accelerating states presented above. Once the remaining states have been captured at this level, the *:CruiseControl* object may be analyzed using a tool such as DesignCPN [7]. As mentioned earlier, this analysis may occur independently by using CPN places to simulate the inputs and outputs. Alternately, the models may be integrated with the larger cruise control architecture (by using the actual CPN places corresponding to message queues, operation calls, etc.) to complete the behavioral modeling and analysis of the system as a whole. This analysis may cover such aspects as deadlock detection, performance analysis against time constraints, or checking boundary conditions to, for example, determine if the set queue sizes can handle the observed volume of message traffic.

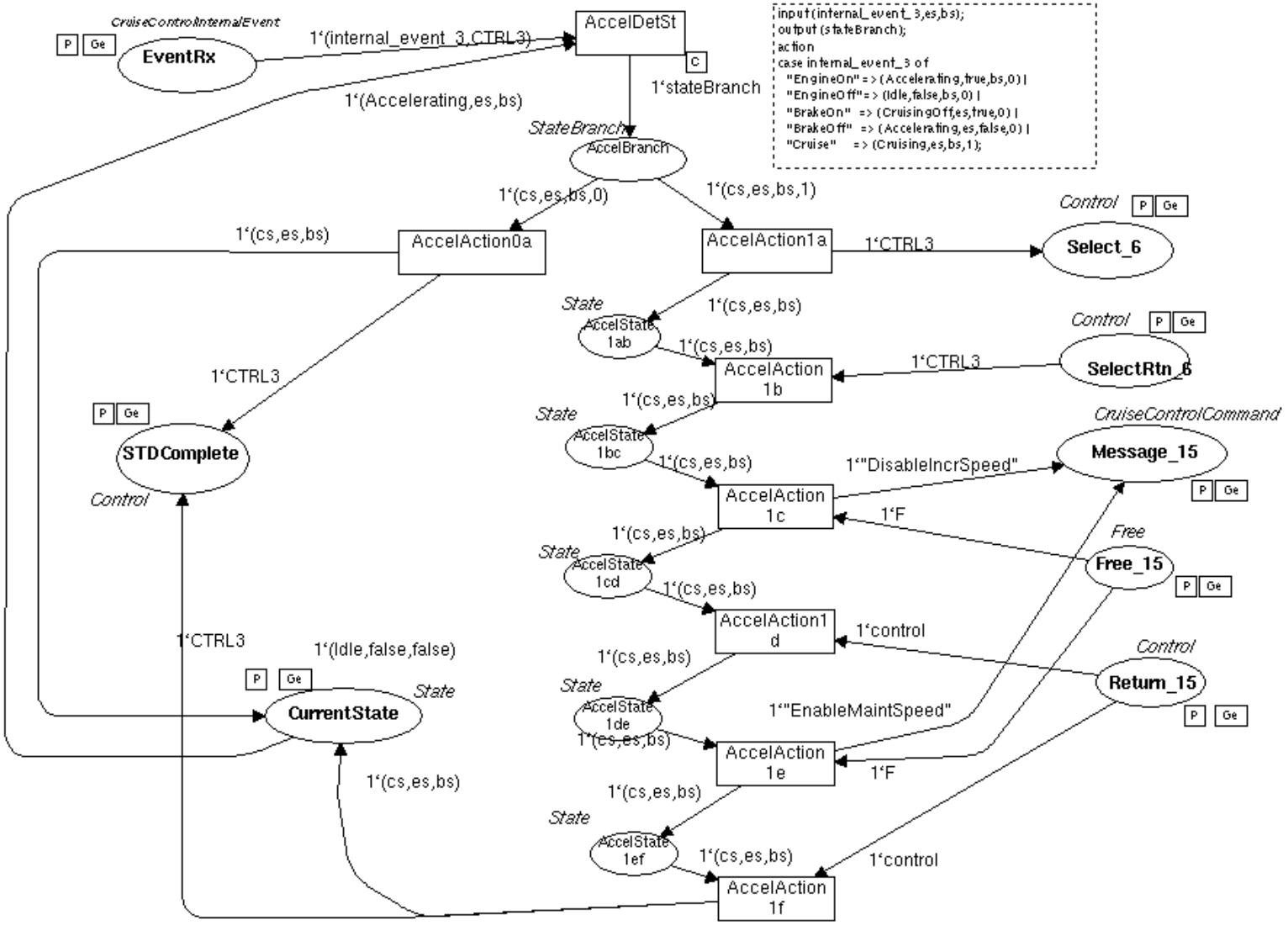

Figure 9. Cruise Control: Accelerating

6 Conclusions and Future Research

This paper presents a systematic, scaleable, and repeatable approach for using colored Petri nets to model the behavior of UML objects containing statecharts, which is capable of being automated. This approach may be integrated into the larger effort of modeling the overall concurrent software architecture using CPNs. The resulting CPN is then used to validate such dynamic properties as the absence of deadlock and starvation conditions as well as providing a timing and behavioral analysis of the architecture through simulation. This analysis through CPNs reduces the overall risk of software implementation by allowing behavioral characteristics to be validated from an architectural design rather than waiting for the system to be coded.

This paper represents on-going research efforts to integrate colored Petri nets with object-oriented software design methods for concurrent and real-time systems. Future research will explore the automatic generation of CPNs from UML. It is the goal of this continuing research to arrive at a set of CPN translation rules that can be effectively integrated with software design methods to provide increased reliability and analytical capabilities at multiple levels of abstraction.

7 References

[1] Rumbaugh, J., Jacobson, I., and Booch, G., *The Unified Modeling Language Reference Manual* Reading, Mass.: Addison-Wesley, 1999.

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

[3] Pettit, R. G. and Gomaa, H. “Validation of Dynamic Behavior in UML Using Colored Petri Nets.” *UML 2000 Behavioral Semantics Workshop.* York, England. October, 2000.

[4] Gomaa, H., *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison Wesley Object Technology Series, 2000, http://www.aw.com/cseng/titles/0-201-65793-7.

[5] David, R. and Alla, H., "Petri Nets for Modeling of Dynamic Systems: A Survey," *Automatica*, vol. 30, no. 2, pp. 175-202, 1994.

[6] Jensen, K., *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use* Berlin, Germany: Springer-Verlag, 1997.

[7] DesignCPN. http://www.daimi.aau.dk/designCPN/.

[8] Harel, D., "On Visual Formalisms." CACM 31, 5 (May 1988), 514-530.

[9] Harel, D. and E. Gary, "Executable Object Modeling with Statecharts", Proc. 18th International Conference on Software Engineering, Berlin, March 1996

[10] Harel, D. and M. Politi, "Modeling Reactive Systems with Statecharts", New York, NY: McGraw-Hill, 1998.