Validation of Dynamic Behavior in UML 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@isse.gmu.edu

Abstract. This paper describes an approach for modeling the behavioral characteristics of concurrent object-oriented designs using the Petri net formalism. Specifically, this paper describes an approach for integrating colored Petri nets with concurrent object architecture designs created with the COMET method and specified in the Unified Modeling Language (UML). This work is part of an on-going effort to automate the behavioral analysis of concurrent and real-time object-oriented software designs.

1 Introduction

This paper presents an approach for using colored Petri nets to model and subsequently validate the behavioral characteristics of concurrent object architectures represented in the Unified Modeling Language (UML). 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. Secondly, 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 design architecture could provide 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 [1]. 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 [2]. 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 the basic Petri net.

2 Modeling UML Dynamic Behavior Using Colored Petri Nets

The approach taken in this paper is to use a CPN model to augment the behavioral specifications of concurrent object-oriented design architectures created with 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 [3]. As the UML is now the standardized notation for describing object-oriented models [4;5], COMET 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 Design Modeling phase, an Architectural Design Model is developed. Subsystem structuring criteria are provided to design the overall software architecture. Each concurrent subsystem is then designed in terms of active objects and passive objects. Inter-object communication and synchronization interfaces are also defined at this point. This architectural design model (captured with UML collaboration diagrams) serves as the focal point for the UML to CPN mapping. Specifically, for capturing (and subsequently validating) the dynamic behavior of concurrent and real-time systems we are interested in modeling such architectural design features as the asynchronous or periodic behavior of concurrent objects, message communication between objects, and mutually exclusive access to shared data objects. Fig. 1 provides an example UML collaboration diagram illustrating these architectural design features.

In this example, an actor initiates some event on the system. The first active (asynchronous) object performs some processing on the input event and sends an asynchronous message to the active periodic object and a synchronous message to the second active asynchronous object. There is also an entity object that encapsulates data and provides operations to access the data. Since the entity object is being read by and written to by two active objects, it must also provide mutually exclusive access controls, which must in turn be represented by the corresponding CPN model. The following sections further discuss the roles of these elements in terms of the COMET method and discuss the mapping between these UML elements and the corresponding CPN segments.

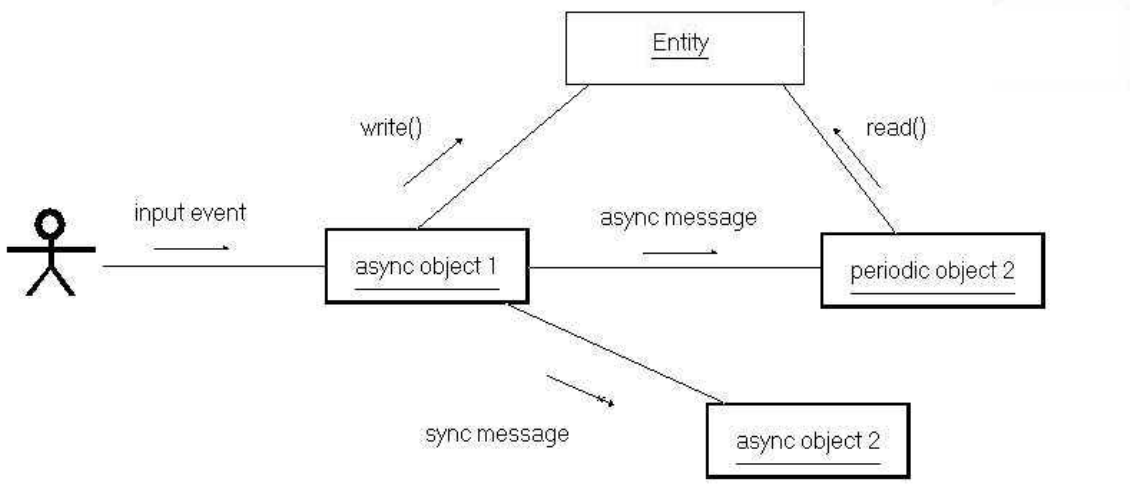

Fig. 1. Example UML Collaboration Diagram

2.1 Mapping Active Objects to Colored Petri Nets

Active objects form the basis of concurrency within the UML. Active objects may be found in interfaces or processing objects and may operate in either asynchronous or periodic modes. For the purposes of this paper, interface objects and processing objects will be treated the same in terms of creating Petri net templates. The only difference is that interface objects receive *events* from external sources, whereas processing objects receive *messages* from other objects within the system.

An **asynchronous** active object is activated by an asynchronous stimulus (e.g. message or interrupt) rather than a timer event. The CPN representation of an asynchronous active object consists of a series of places and transitions that use a control token to represent the flow of control within the object. Fig. 2 shows an example of modeling an asynchronous object with a CPN. In this CPN template, event tokens enter from an external source (external device, system, or application object, depending on whether we are dealing with an interface or a processing object). When the CPN segment is ready to process events as indicated by the presence of a control token in the *Async Control* place, the transition is fired and the event and control tokens are given to the *Event Received* place. Notice the "@" notation on the *Process Event* transition. This indicates that all timed tokens are incremented by some arbitrary processing time in order to simulate the real-time nature of execution. In the process described by this paper, all control tokens are timed.

Next, the internal event and control tokens are passed to the Send Msg transition to be translated (using special guard or code segments) to the appropriate CPN segment, representing the receiving object in the UML model. This message may be sent immediately in the case of asynchronous communication or may be blocked until the receiver is ready in the case of synchronous communication. Communication mechanisms are discussed in more detail in Section 0. Once the message has been sent, the control token is returned to the *Async Control* Place and the CPN segment is ready to process the next event or message.

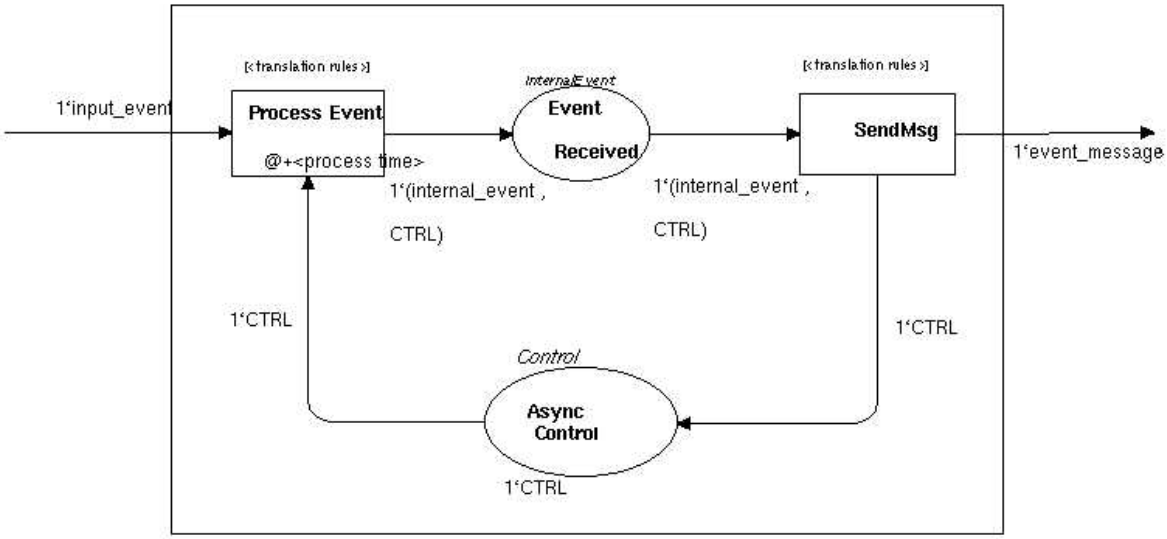

Fig. 2. CPN Segment for Asynchronous Active Objects

A **periodic** active object is activated at regular time intervals rather than on demand. To represent periodic objects with CPNs, a *Wakeup* transition is introduced that will delay control tokens from returning to the *Periodic Control* place by the desired period of activation. Fig. 3 provides the CPN segment that represents periodic objects.
In this example, the object being modeled starts execution with a control token in the *Sleep* place. The *Wakeup* transition to move from the *Sleep* place has a duration of <sleep time> associated with it. After the specified duration has been reached, the object essentially "wakes up" and acts on any waiting events or messages in the same manner as the asynchronous CPN segment.

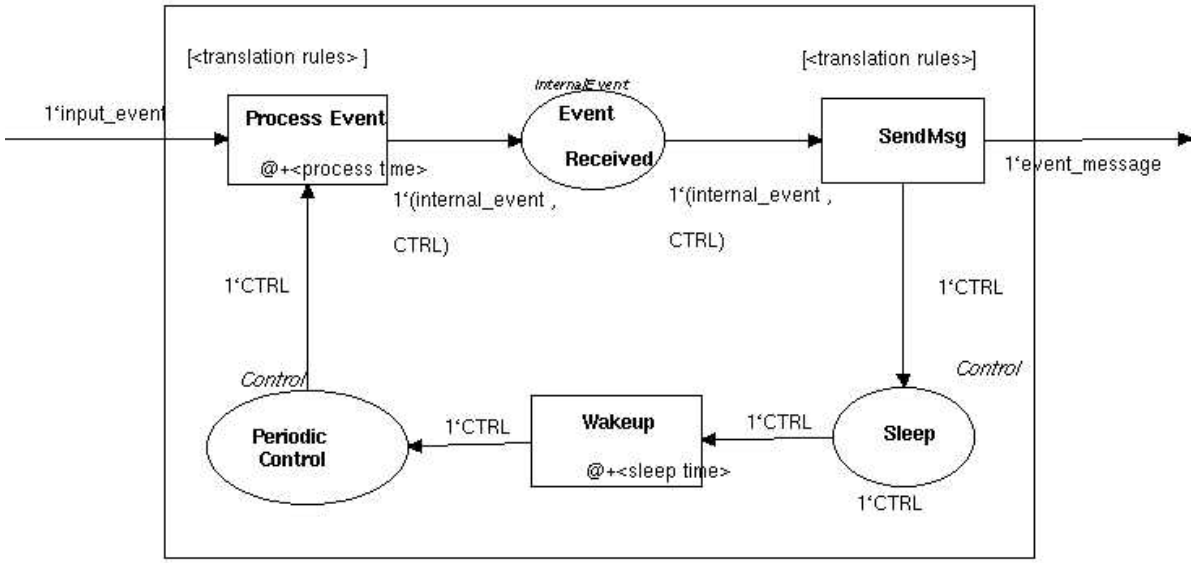

Fig. 3. CPN Segment for Periodic Active Objects

2.2 Mapping Passive Objects to Colored Petri Nets

In COMET, *entity* objects are passive objects that provide mechanisms to encapsulate or store data that needs to be accessed by other objects within the system. These entity objects must also provide the protection mechanisms to enforce mutual exclusion rules necessitated by the passive objects being accessed by multiple active objects. The general CPN segment for entity objects with mutually exclusive access protection is illustrated in Fig. 4.
When this generic segment is instantiated, there will be one *ReadOp* and *WriteOp* transition for each read and write operation for each attribute accessed from the object interface. To use the read and write operations, two places per read or write operations are needed – one place for the request and one for the response/return. In addition to the

read and write places, there is one *Free* place per unit of protection (e.g. attribute) that is used to enforce the mutual exclusion rules. This Free place contains one token indicating if the attribute is in use. If the Free token is available (i.e. the attribute is not in use), the corresponding read or write transitions are allowed to fire, thus allowing the attribute token to be retrieved or modified.

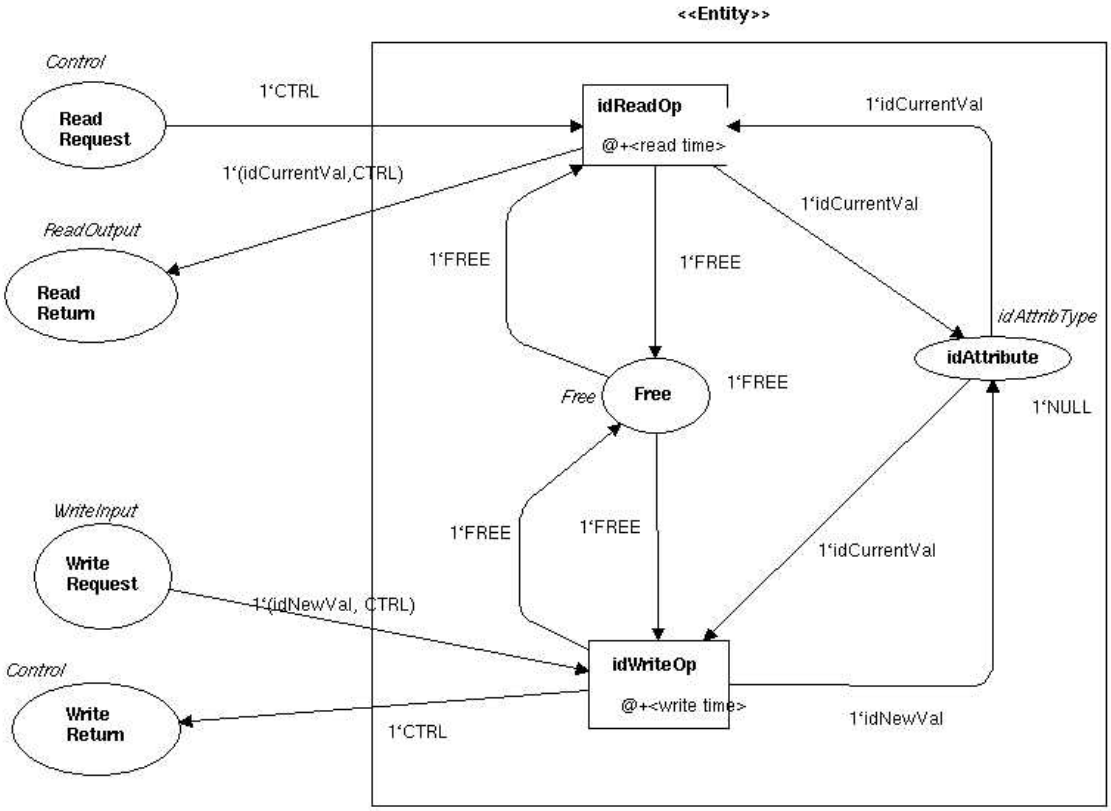

Fig. 4. CPN Segment for Entity Objects

2.3 Mapping Message Communication to Colored Petri Nets

There are two general forms of message communication that can occur between concurrent active objects: *asynchronous* and *synchronous*. With **asynchronous communication**, a producer object places a message on a queue and then continues its processing. A consumer object would then retrieve the first message from the queue, do some processing based on the message, and then retrieve the next message from the queue (if any).
Modeling FIFO queuing behavior using CPNs can be complex for large buffer sizes. There are currently no CPN formalisms to enforce ordered placement or retrieval of tokens to and from a given place. In the absence of ordered token placement and retrieval capabilities to and from a single place, *2n* places and *n+1* transitions are needed to model a queue of *n* elements (one place for storage and one place to indicate whether a place contains a token). Given that a queue has at least one free space, a producer would first place a token on the end place of the CPN queue. Through the series of *n+1* transitions and *n* free place indicators, the enqueued token will advance to the furthest available slot in the queue. To dequeue an element (i.e. retrieve a message from the queue), a consumer would remove a token from the place representing the head of the queue. Tokens are then shifted to the right as one is removed.
In the case of **synchronous** communication, a producer object sends a message to the consumer object but instead of continuing with its processing, it will wait for the message to be received by the consumer. This form of communication is handled simply by passing a token to a CPN segment (e.g. the *input_event* of Fig. 2) and then having a *Control* token returned to the sender either after the token has been received (at the first transition of the CPN segment) or after a return token (message) has been generated.

3 Validating Dynamic Behavior

The first step to validating the dynamic behavior of a UML architecture using CPNs is to translate the concurrent object architecture model (represented by a UML collaboration diagram) into a corresponding CPN network. This is accomplished by replacing each object and message communication element by the appropriate CPN segment as partially illustrated in the previous section. (Several more specific CPN segments were used in the actual research effort.)

Once the UML architectural model has been translated to a CPN, an occurrence graph [2]is generated to construct a graph of all reachable markings for the CPN. These graphs can be extremely large and complex, but are capable of being automated using tools such as DesignCPN[6], which was applied for this research. Based on these graphs, Petri net theory may be applied to validate the absence of deadlock or starvation conditions as well as providing statistical analysis of the architectural usage. Furthermore, DesignCPN also provides a timed simulation capability that allows architectural timing constraints to be evaluated.

4 Conclusions and Future Research

This paper outlines an approach for using CPN segments to model the dynamic behavior of concurrent object architectures expressed in the UML. Given a concurrent architecture and the CPN segments, an engineer may proceed with behavioral analysis by first mapping the UML architectural elements into a CPN representation. 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 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 model 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.

5 References

1. David, R. and Alla, H., "Petri Nets for Modeling of Dynamic Systems: A Survey," *Automatica*, vol. 30, no. 2, pp. 175-202, 1994.
2. Jensen, K., *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use* Berlin, Germany: Springer-Verlag, 1997.
3. Gomaa, H., *Designing Concurrent, Distributed, and Real-Time Applications with UML*, , Reading, Mass.: Addison-Wesley, 2000.
4. Rumbaugh, J., Jacobson, I., and Booch, G., *The Unified Modeling Language Reference Manual*, Reading, Mass.: Addison-Wesley, 1999.
5. Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*, Reading, Mass.: Addison-Wesley, 1999.
6. Jensen, K. *DesignCPN*, version 4.0. Aarhus, Denmark, University of Aarhus, 1999.