# Modeling Behavioral Design Patterns of Concurrent Objects

Robert G. Pettit IV
The Aerospace Corporation
15049 Conference Center Dr
Chantilly, Virginia (USA)
+1-703-324-8937
rob.pettit@aero.org

Hassan Gomaa
George Mason University
4400 University Dr
Fairfax, Virginia (USA)
+1-703-993-1652
hgomaa@gmu.edu

## ABSTRACT
Object-oriented software development practices are being rapidly adopted within increasingly complex systems, including reactive, real-time and concurrent system applications. While data modeling is performed very well under current object-oriented development practices, behavioral modeling necessary to capture critical information in real-time, reactive, and concurrent systems is often lacking. Addressing this deficiency, we offer an approach for modeling and analyzing concurrent object-oriented software designs through the use of behavioral design patterns, allowing us to map stereotyped UML objects to colored Petri net (CPN) representations in the form of reusable templates. The resulting CPNs are then used to model and analyze behavioral properties of the software architecture, applying the results of the analysis to the original software design.

## Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Object-oriented design methods, Petri nets.*

## General Terms
Performance, Design, Reliability

## Keywords
Software Architecture, COMET, Colored Petri Nets, Behavioral Design Patterns

## 1. Introduction
Object-oriented software development practices are being rapidly adopted within increasingly complex systems, including reactive, real-time and concurrent system applications. In practice, though, object-oriented software design techniques are still predominantly focused on the creation of static class models. Dynamic architectural models capturing the overall behavioral properties of the software system are often constructed using ad hoc techniques with little consideration given to the resulting performance or reliability implications until the project reaches implementation. Efforts to analyze behavioral issues of these architectures occur through opportunistic rather than systematic approaches and are inherently cumbersome, unreliable, and unrepeatable.

One means of improving the behavioral modeling capabilities of object-oriented architecture designs is to integrate formalisms with the object-oriented specifications. Using this technique, object-oriented design artifacts are captured in a format such as the Unified Modeling Language (UML) [1], which is intuitive to the software architect. The native object-oriented design is then augmented by integrating an underlying formal representation capable of providing the necessary analytical tools. The particular method used in this research [2] is to integrate colored Petri nets (CPNs) [3] with object-oriented architecture designs captured in terms of UML communication diagrams. Specifically, this paper will present a method to systematically translate a UML software architecture design into an underlying CPN model using a set of pre-defined CPN templates based on a set of object behavioral roles. These behavioral roles are based on the object structuring criteria found in the COMET method [4], but are not dependent on any given method and are applicable across application domains. This paper will also demonstrate some of the analytical benefits provided by constructing a CPN representation of the UML software architecture. After a survey of related research, Section 2 descries the concept of behavioral design pattern templates for modeling concurrent objects. Section 3 discusses how we construct an overall CPN model of the concurrent software architecture by interconnecting the individual behavioral design pattern templates. Section 4 describes the validation of the approach.

### 1.1 Related Research
There are many existing works dealing with the use of Petri nets for describing software behavior. As they relate to this paper, the existing works can be broadly categorized into the modeling of software code and the modeling of software designs. In this research, the focus is on improving reliability of object-oriented software designs rather than delaying detection to the software code. In terms of object-oriented design, the related Petri net research can be categorized as new development methodologies [5-8]; object-oriented extensions to Petri nets [9-12]; and the integration of Petri nets with existing object-oriented methodologies [13-20]. Since one of the goals of this research effort is to provide a method that requires no additional tools or language constructs beyond those currently available for the UML and CPN definitions, this approach [2,21-25] falls into the last category of integrating Petri nets with existing methodologies. The main features that distinguish this approach from other related works are a focus on the concurrent software

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*ICSE'06*, May 20–28, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005...$5.00.

architecture design and the use of consistent, reusable CPN templates to model the behavior of concurrent objects and their interactions. This paper also extends our more recent works [25] by specifically focusing on the behavioral design patterns of individual concurrent objects and applying these patterns to construct an underlying representation of the concurrent software design architecture.

## 2. Modeling Behavioral Design Patterns

To model concurrent object behavioral design patterns with CPNs, our approach starts with a concurrent software architecture model captured in UML. For the construction of this architecture model, we identify a set of behavioral design patterns used to categorize the objects along with a set of specification requirements necessary to correctly model the concurrent behavior with the underlying CPN model. Each of the identified behavioral design patterns then has a corresponding template, represented as a CPN segment, which is paired with the UML object and is instantiated to capture specific behavioral characteristics based on the object specifications. The following sections describe the object architecture definition along with the concept of behavioral pattern templates for modeling concurrent objects. Section 3 will then discuss how we construct an overall CPN model of the concurrent object architecture by connecting the individual behavioral pattern templates.

### 2.1 Concurrent Object Modeling

Our approach uses a UML communication diagram to capture the concurrent software architecture. Depending on the desired level of modeling, this architecture model can be constructed for an entire software system or for one or more individual subsystems. This communication diagram contains a collection of concurrent (active) and passive objects along with the message communication that occurs between the objects. Using our approach, objects within the concurrent software architecture are organized using the notion of components and connectors. Under this paradigm, concurrent objects are treated as components that can be connected through passive message communication objects and entity objects. In keeping with the COMET object structuring criteria, each object is assigned a UML stereotype to indicate its behavioral design pattern. Objects are broadly divided into application objects, which perform the work, and connector objects, which provide the means of communicating between application objects. For application objects, we use six stereotyped behavioral design patterns as illustrated in Figure 1: interface, entity, coordinator, state-dependent, timer, and algorithm. Additionally, connector objects can take the roles of: queue, buffer, or buffer-with-response, corresponding to asynchronous, synchronous, and return messages. These patterns are not intended to be an exhaustive list, but rather are intended to represent sufficient variety to model concurrent systems across a wide range of domains while also allowing these patterns to be extended as necessary for future applications.

The identification of stereotyped behavioral roles allows us to select a specific CPN template to model each object (further described in Section 3.2). These behavioral stereotypes are generic across applications, so we also capture specific application information using the following tagged values:

- Execution Type. Each object must be declared as either passive or concurrent and for concurrent objects, further specified to be asynchronous or periodic.
- IO Mapping. Input-output message pairings must be specified for each object
- Communication Type. Indicate whether message communication occurs through asynchronous or synchronous means.
- Activation Time. The period of activation must be specified for each periodic concurrent object.
- Processing Time. Estimated processing times for completing an execution cycle should be assigned to each object if timing is to be accounted for in performance analysis.
- Operation Type. Indicate whether operations on entity objects perform “reader” or “writer” functionality.
- Statechart. For each state-dependent object, a UML statechart is used to capture the state behavior for that object. A detailed discussion of how the statechart is translated into the CPN model is provided in Pettit and Gomaa [24].

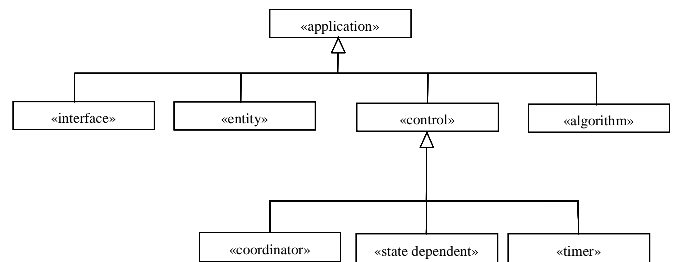


**Figure 1. Stereotype Hierarchy for Application Objects**

### 2.2 Defining Behavioral Pattern Templates

The basis for our approach to modeling concurrent object behavior lies in the notion of a behavioral design pattern (BDP) template, which represents concurrent objects according to their role along with associated message communication constructs. For each BDP template, we employ a self-contained CPN segment that, through its places, transitions, and tokens, models a given stereotyped behavioral pattern. Each template is generic in the sense that it provides us with the basic behavioral pattern and component connections for the stereotyped object but does not contain any application-specific information. The connections provided by each template are consistent across the set of templates and allow concurrent objects to be connected to passive objects (entities or message communication) in any order.

We provide a BDP template for each object type identified in the previous section. Since each of these templates captures a generic behavioral design pattern, when a template is assigned to a specific object, we then augment that template with the information captured in the tagged values for the object. For the resulting CPN representation, this affects the color properties of the tokens (e.g. to represent specific messages) and the rules for processing tokens (e.g. to account for periodic processing or special algorithms). The following sections describe a subset of our behavioral templates for both concurrent object components and their connectors.

### 2.2.1 Asynchronous Interface Object Template

Consider the case of an asynchronous, input-only interface object. The template for this behavioral design pattern is given in Figure 2.

This template represents a concurrent object, that is, an object that executes its own thread of control concurrently with other objects in the software system. While this template models relatively simple behavior (wait for input; process input; wait for next input), it features characteristics found throughout the concurrent object templates. First, to model the thread of control within a concurrent object, a control token (CTRL) is assigned to each concurrent object. For this template, a control token is initially present in the Ready place. Thus, this template is initialized in a state whereby it is ready to receive an input at the ProcessInput transition. As an input arrives (and given that the control token is in the Ready place), ProcessInput is allowed to fire, simulating the processing of the external input and the behavior of the asynchronous input interface object. ProcessInput consumes both a token representing the external input as well as the control token representing the executable thread of control. An output arc from ProcessInput uses a function, processInput (Input_event) to generate the appropriate token representing an internal message passed to another object within the system. The exact behavior of the processInput function (as with any arc-inscription functions throughout the templates) is determined from the object specification when a template is instantiated for a specific object. Finally, to complete the behavioral pattern for this template, the control token is passed to the MessageSent place and eventually back to the Ready place, enabling the template to process the next input.

### 2.2.2 Periodic Algorithm Object Template

The asynchronous interface template addressed asynchronous behavior for a concurrent object, where the object is activated on demand by the receipt of a message or an external stimulus (as in the case of the interface example). For periodic behavior, where an object is activated on a regular periodic interval, consider the template for a concurrent periodic algorithm object given in Figure 3.

Algorithm objects are internal concurrent objects that encapsulate algorithms, which may be activated or deactivated on demand. In the case of the periodic algorithm object, once the algorithm is enabled, it awakens on its activation period, performs the desired algorithmic task, and then returns to a sleep state until the next activation period.

Looking at the periodic algorithm template from Figure 3, you should notice that, like the previous concurrent object template, there is Ready place with a control token that indicates when the object is ready to start its next processing cycle and models the thread of execution. This is common across all concurrent object templates. To model the ability for an algorithm object to be enabled or disabled, the input interface to this template occurs through the Enable_Alg and Disable_Alg transitions. (Note that we maintain the use of transitions as the interface points for all concurrent objects.) Thus, in addition to the control token being present on the Ready place, an Enable token must also be present on the Alg_Enabled place in order for the Perform_Alg transition to be enabled and subsequently fired. The actual behavior performed by the algorithm is captured by decomposing the Perform_Alg transition. The resulting decomposition uses one or more place-transition paths to model the behavior performed within the algorithm. The information necessary to derive the CPN algorithm model may be contained in the UML class specification for the algorithm object or, for more complex algorithms, may be captured in supporting UML artifacts such as the activity diagram. Multiple algorithms may be encapsulated within the same algorithm object. In these cases, the enable/disable transitions, enabled place, and processing transition are repeated for each encapsulated algorithm. However, there will only ever be one control token and ready place in a single concurrent object as our approach does not allow for multi-threaded concurrent objects.

Finally, to capture the periodic nature of this template, a Sleep place along with Wakeup and Timeout transitions have been added to the basic asynchronous object template. This place-transition pair will be common to all periodic templates. In this case, the periodic algorithm starts in the Sleep place rather than Ready. After the desired sleep time (indicating the activation period of the object) has elapsed, the Wakeup transition is enabled and, when fired, removes the CTRL token from the Sleep place and places it in the Ready place. This now enables the template to perform any enabled algorithms. If one or more algorithms are enabled, the template proceeds in the same manner as the previous asynchronous algorithm template. However, if no algorithms are enabled when the template wakes up, the Timeout transition will fire and return the Control token to the Sleep place and wait for the next period of activation.

### 2.2.3 Entity Object Template

In contrast to concurrent objects, passive objects do not execute their own thread of control and must rely on operation calls from a concurrent object. Using our approach, the entity objects from Figure 1 are passive objects. The purpose of an entity object is to store persistent data. Entities provide operations to access and manipulate the data stored within the object. These operations provide the interface to the entity object. To account for the possibility of multiple concurrent objects accessing a single entity object, our approach stipulates that each operation be tagged as having "read" or "write" access and for the object to be tagged with "mutually exclusive" or "multiple-reader/single-writer" rules for access control. This allows us to apply the appropriate template with the desired mutual exclusion protection for the encapsulated object attributes. The behavioral design pattern template representing an entity object with mutually exclusive access is shown in Figure 4.

In this template, attributes are modeled with a CPN place containing tokens representing the attribute values. The underlying functionality of each operation is captured in an "idmOperation" transition that can be further decomposed as necessary to implement more complex functions. When instantiated for a specific entity object, the "idm" tag is replaced with a specific identifier for each operation. Finally, the interface to each operation is provided by a pair of CPN places – one place for the operation call and another for the return. Collectively, these places form the interface to the entity object. As opposed to concurrent objects, all passive objects and message connectors will use CPN places for their interface, allowing concurrent objects to be connected through their transition interfaces. Thus, for performing an operation call, a

concurrent object places its control token and any necessary parameter tokens on the calling place and then waits for the control token to be returned along with any additional operation results at the call return place. Recall that entity objects do not have their own thread of control, thus they become part of the calling object's thread of control for the duration of the operation call.

### 2.2.4 *Message Communication Templates*

Finally, in addition to application object templates, our method also provides templates for connector objects representing message communication. These connectors may represent asynchronous or synchronous message communication between two concurrent objects.

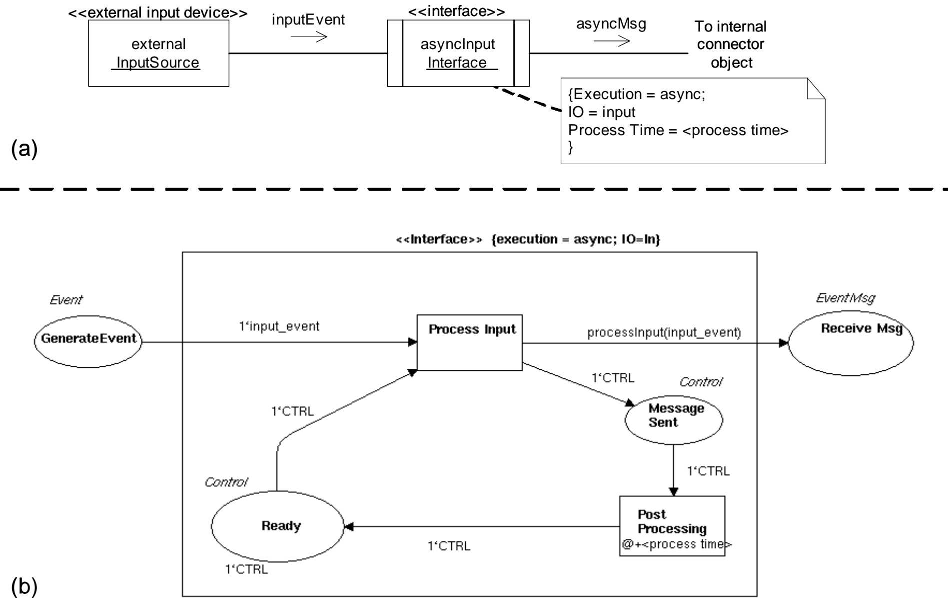


**Figure 2. Asynchronous Input-Only Interface Object: (a) UML (2.0); (b) CPN Template**

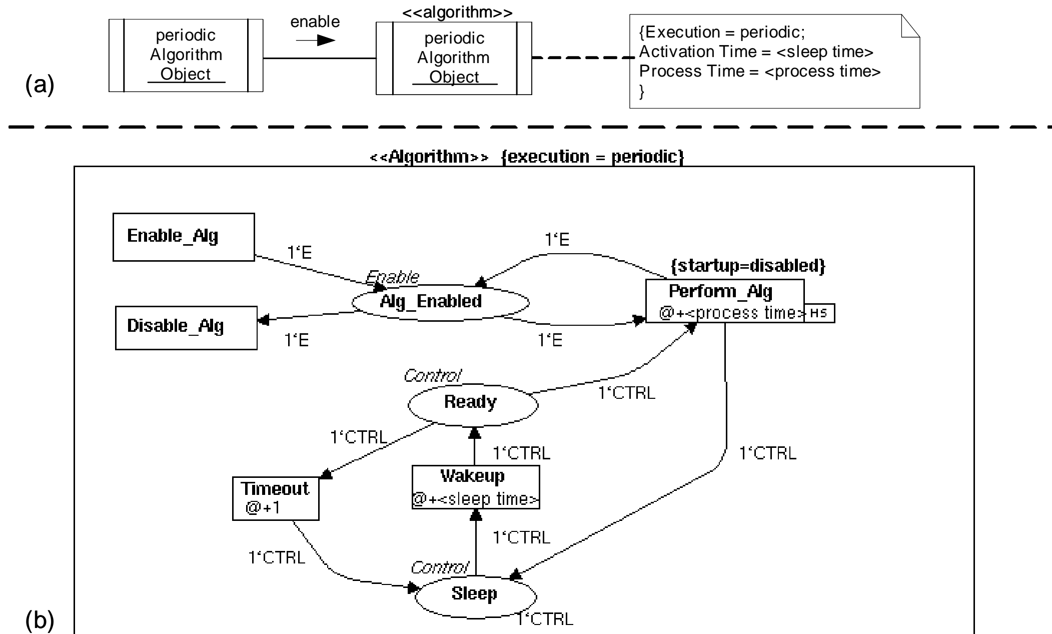


**Figure 3. Periodic Algorithm Template: (a) UML; (b) CPN Representation**

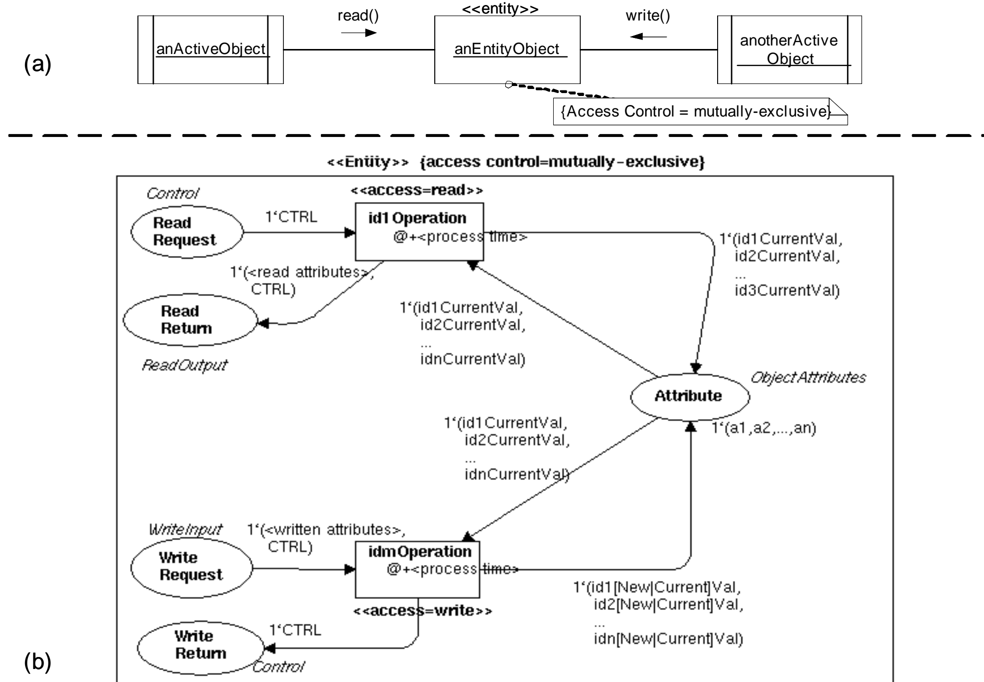


**Figure 4. Passive Entity Template: (a) UML; (b) CPN Representation**

Consider the message buffer template shown in Figure 5. Notice that, as with passive entity objects, the interface to connector objects always occurs through a place rather than a transition, thus allowing concurrent object interfaces to be linked with connector interfaces while still enforcing the Petri net connection rules of only allowing arcs to occur between transitions and places. The message buffer template models synchronous message communication between two concurrent objects. Thus, only one message may be passed through the buffer at a time and both the producer (sender) and consumer (receiver) are blocked until the message communication has completed. The behavior of synchronous message communication is modeled through this template by first having the producer wait until the buffer is free as indicated by the presence of a "free" token in the buffer. The producer then places a message token in the buffer and removes the free token, indicating that the buffer is in use. Conversely, the consumer waits for a message token to appear in the buffer. After retrieving the message token, the consumer sets the buffer once again to free and places a token in the "Return" place, indicating to the producer that the communication has completed.

Asynchronous message connector templates continue to employ places for their interfaces. However, asynchronous message communication, which involves the potential for queuing of messages, is more involved than the simple synchronous message buffer and must therefore add a transition to handle this behavior. The corresponding template is shown in Figure 6.

With asynchronous communication the sender is not blocked awaiting acknowledgement that the sent message has been received and a message queue is allowed to form for the object receiving the asynchronous messages. In this template, the ManageQueue transition is decomposed into a subnet that implements the FIFO placement and retrieval of messages in the queue [26]. To send an asynchronous message, a concurrent object places a message token on the Enqueue place. The subnet under ManageQueue would then add this message token to the tail of the queue. Another concurrent object receiving the asynchronous message would wait for a message token to be available in the Dequeue place (representing the head of the queue). It would then remove the message token from Dequeue and signal DequeueComplete in a similar manner to the operation calls previously described for entities. This signals the queue that a message token has been removed from the head of the queue and that the remaining messages need to be advanced.

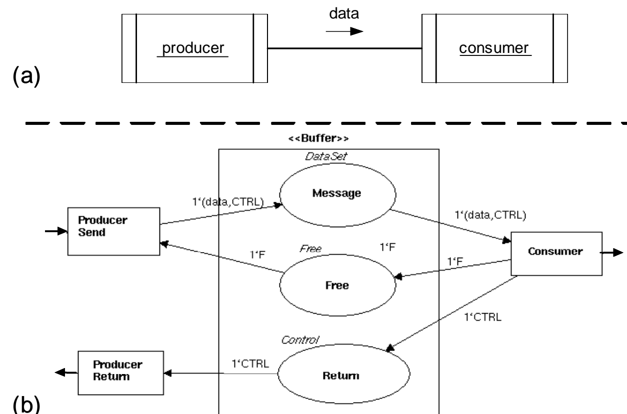


**Figure 5. Synchronous Message Buffer Connector Template: (a) UML; (b) CPN Representation**

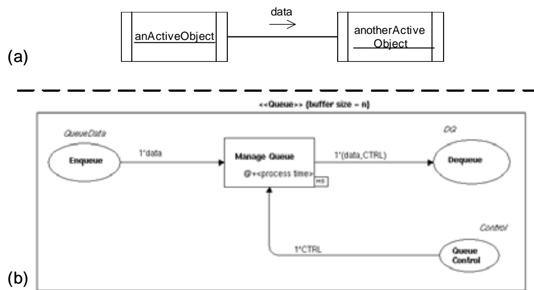


**Figure 6. Asynchronous Message Queue Template**

## 3. Constructing CPN Models from UML

Up to this point, we have just discussed individual CPN templates being used to model behavioral design patterns of concurrent objects, passive «entity» objects, and message communication mechanisms. This section presents our method for constructing a CPN model of the concurrent software architecture by applying and interconnecting these templates. The basic construction process consists of the following steps:

1. Construct a concurrent software architecture model using a UML communication diagram to show all concurrent and passive objects participating in the (sub) system to be analyzed along with their message communication.
2. Begin constructing the CPN model by first developing a context-level CPN model showing the system as a single CPN substitution (hierarchically structured) transition and the external interfaces as CPN places. Using a series of hierarchically structured transitions allows us to work with the CPN representation at varying levels of abstraction, from a completely black-box view, a concurrent software architecture view (in the next step), or within an individual object as desired for the level of analysis being applied to the model.
3. Decompose the system transition of the CPN context-level model, populating an architecture-level model with the appropriate CPN templates representing the objects from the concurrent software architecture.
4. Elaborate each instance of CPN template to account for the specific behavioral properties of the object it models.
5. Connect the templates, forming a connected graph between concurrent object templates and passive entity objects or message communication mechanisms.

To illustrate the application of this approach, consider a partial example from the well-known Cruise Control System [4]. This example was chosen for this paper as it requires little explanation for the UML model and allows us to focus on the use of behavioral design pattern templates and the CPN representations. Figure 7 provides a partial communication diagram of the Cruise Control System concurrent software architecture.

To begin, focus on the input events being provided by the Cruise Control Lever. (We will return to the brake and engine inputs later in this section.) Cruise control lever events enter the system via a concurrent «interface» object that sends an asynchronous message to the «state dependent control» object to process the requests based on rules defined in a corresponding statechart. Based on the state of the CruiseControl object, commands are given to a concurrent periodic «algorithm» object enabling it to compare speed values from two passive «entity» objects and determine the correct throttle values, which are then passed on to the periodic output interface, ThrottleInterface.

Given this concurrent software architecture, the second step in our process would construct the context-level CPN model shown in Figure 8. At this level, we see the system as a black-box represented as a single transition, "CruiseControlSystem". External input and output interfaces for the cruise control lever, brake, and engine devices are represented as places. The purpose of this context-level CPN model is to provide a central starting point for our modeling and analysis. By structuring the CPN model in this way, we can analyze the system as a black box, dealing only with external stimuli and observed results (corresponding to the tokens stored in these places) or we can use hierarchical decomposition to gain access to the individual object behavioral design pattern templates (and their detailed CPN implementation) by systematically decomposing the hierarchically structured transitions (indicated with the HS tag).

In the third step, the CruiseControlSystem transition from the context-level model is decomposed into an architecture-level model populated with the appropriate CPN behavioral design pattern template for each of the cruise control objects. Given the architecture design from Figure 7 (and continuing to ignore AutoSensors for the moment), we would need to instantiate two «interface» templates, two «entity» templates, one «state dependent control» template, and one «algorithm» template. We would also need to use «queue» and «buffer» templates for the asynchronous and synchronous message communication respectively.

Once the appropriate templates have been assigned to each object, the fourth step in the process is to elaborate each template to model a specific object. To illustrate, consider CruiseControlLeverInterface. This object is an asynchronous input-only interface that accepts events from the cruise control lever device and, based on the input event, generates the appropriate messages for the cruise control request queue. Applying the asynchronous input interface template from Figure 2, we arrive at the elaborated CPN segment for CruiseControlLeverInterface shown in Figure 9.

To elaborate the template for the CruiseControlLeverInterface, the place and transition names from the basic template have been appended with the object ID (1) for the specific object. The control token for this model has also been set to the specific control token for the CruiseControlLeverInterface object (CTRL1) and the time region for the PostProcessing_1 transition has been set to "@+100" to reflect the Process Time tagged value. The CruiseControlLeverInterface CPN representation is then connected to the software architecture by establishing an input arc from the CruiseControlLeverDevice place, representing the external input from the device, and an output arc to the Enqueue place, modeling the asynchronous message communication identified in the UML software architecture. Token types (colors) are then specifically created to represent the incoming event and outgoing messages. Finally, the processInput1() function is elaborated to generate the appropriate asynchronous message based on an incoming lever event. This elaboration process is similar for all templates.

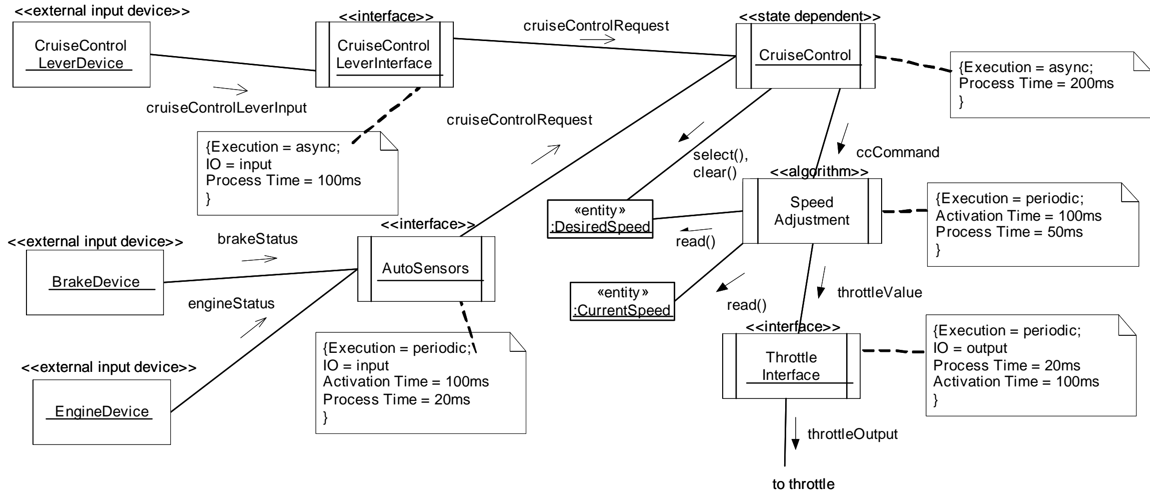


**Figure 7. Partial Concurrent Software Architecture for Cruise Control**

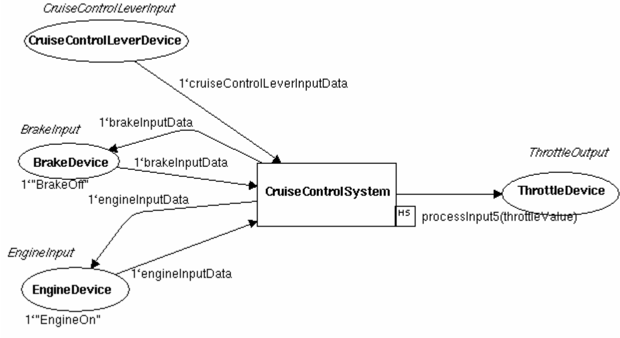


**Figure 8. CPN Context-Level Model for Cruise Control**

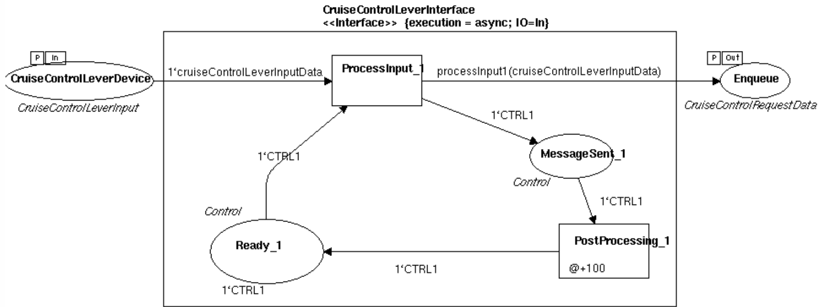


**Figure 9. Asynchronous Input-Only Interface Template Applied to CruiseControlLeverInterface**

Once all templates have been elaborated, our fifth and final step connects the templates to form a connected graph of the concurrent software architecture. The entire CPN architecture model for cruise control is too large for inclusion in this paper. However, Figure 10 illustrates the component connections between the CruiseControlLeverInterface and the CruiseControl templates using an asynchronous message queue connector. As can be seen from this figure, the two concurrent object templates communicate via the queue connector by establishing arcs between the interface transitions of the concurrent objects and the interface places of the queue connector. This component connection method applies to the entire software architecture using our approach of allowing concurrent objects to be connected to either passive entity objects or to a message communication connector.

To further illustrate the component-based approach used for constructing these CPN let us now consider expanding the model to include input from the brake and engine devices. In addition to the cruise control lever inputs, Figure 7 also shows brake and engine status messages arriving from the respective devices. These status messages are handled by the AutoSensors periodic interface object and are passed to CruiseControl via an asynchronous message through the same cruise control request queue already being used by CruiseControlLeverInterface. Using our component-based modeling approach, the AutoSensors object can be added to our CPN model by simply instantiating a CPN representation of the periodic input interface behavioral design pattern template using the specified characteristics for AutoSensors and then connecting it to the existing queue template. The resulting CPN model is given in Figure 11.

The addition of AutoSensors also illustrates another capability of the interface template. Whereas the cruise control lever is an asynchronous device, providing interrupts to CruiseControlLeverInterface, the brake and engine devices are passive devices that must be polled for their status. In Figure 11, every time AutoSensors is activated, it retrieves the status token from the brake and engine device places. After checking the status, the token is immediately returned to the device places, modeling persistence of device status information that can be polled as necessary. The remainder of the AutoSensors template should be familiar, being constructed of the standard Ready and ProcessInput place-transition pair for interface object templates (Section 2.2.1) and the Sleep and Wakeup place-transition pair included for periodic objects (Section 2.2.2).

As demonstrated in this section, the primary benefits of our component-based modeling approach are that connections can easily be added or modified as the architecture evolves or to provide rapid "what-if" modeling and analysis.

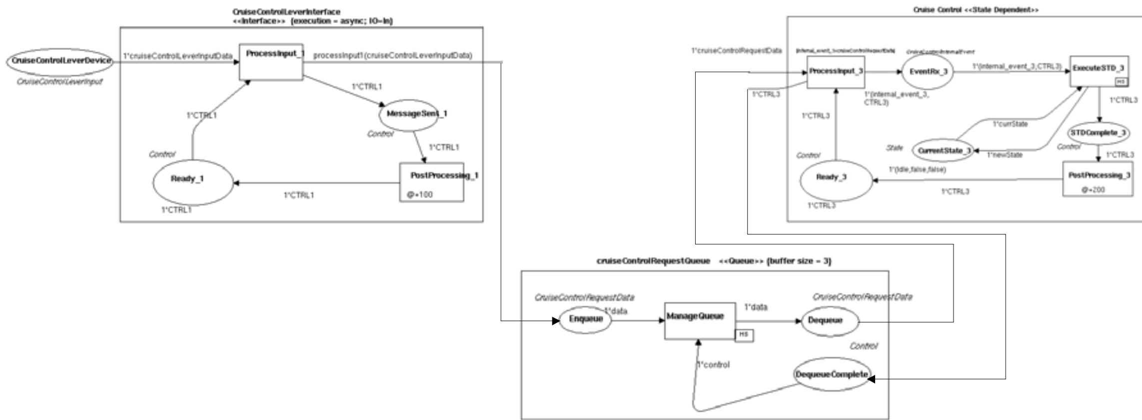


**Figure 10 Connecting CruiseControlLeverInterface and CruiseControl via Asynchronous Communication**

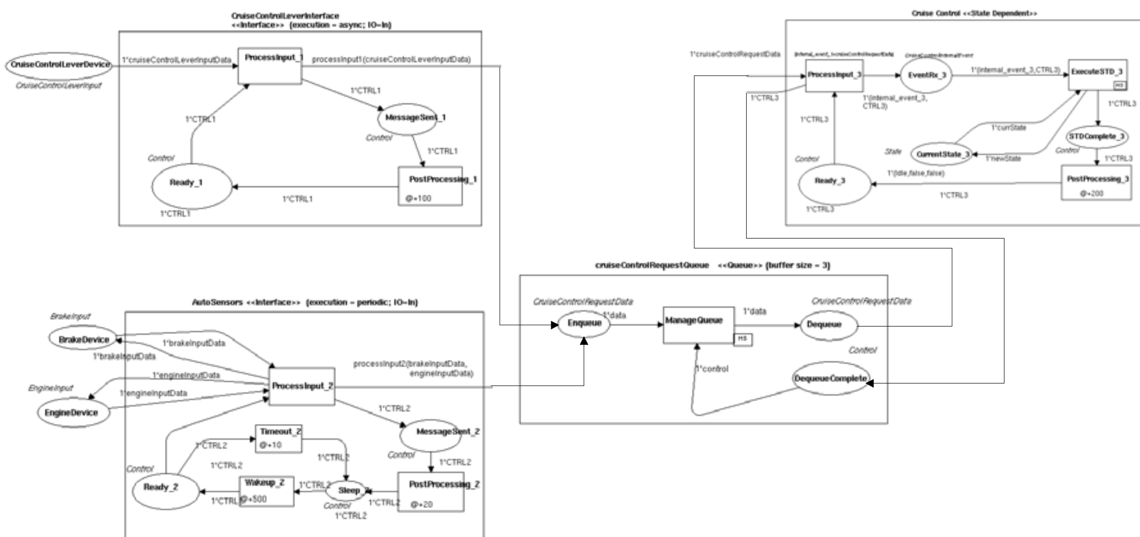


**Figure 11. Addition of AutoSensors to the CPN Architecture**

Furthermore, by maintaining the integrity between a CPN template and the object it represents, modeling and analysis results can readily be applied to the original UML software architecture model. Thus, while from a pure CPN perspective, our CPNs could be further optimized, we feel that it is of greater benefit to maintain a component-based architecture that closely represents the structure of our original UML design artifacts.

## 4. Validation

The validation of our approach was in three parts. First, there was the issue of whether our behavioral stereotypes and corresponding templates could be applied across domains and projects. This was demonstrated by successfully applying our process to two case studies, the cruise control system (a portion of which was shown in the previous sections) and the signal generator system [2]. Secondly, we performed validation to determine if the resulting CPN models provided a correct model of the concurrent software architecture. This was necessary to validate that our approach would result in an accurate representation of the original architecture and was by far the most tedious part of validation, as it required manual inspection and unit testing of each object and its corresponding CPN template representation for the two case studies. Finally, after determining that our template approach satisfied the modeling requirements for both case studies, we then sought to demonstrate the analytical capabilities gained from using CPNs to model concurrent software architectures. The behavioral analysis addresses both the functional behavior of the concurrent architecture as well as its performance, as described next. The detailed analytical results for both case studies are provided in [2].

### 4.1 Validating Functional Behavior

For functional analysis, the simulation capabilities of the DesignCPN tool are used to execute the model over a set of test cases. These test cases may be black-box tests in which we are only monitoring the context-level model in terms of input events and output results or they may be white-box tests in which we analyze one or more individual object representations. In our approach, black box test cases were derived from use cases while white box test cases were derived from object interactions, object specifications, and statecharts. In each of these cases, the

appropriate inputs for each test case were provided by placing tokens on the CPN places representing the external actors in the context model. The CPN model was then executed in the simulator and observed at the desired points to determine if the correct output was generated or if the correct logical paths were chosen.

Again, consider the cruise control system. Figure 12 illustrates a black-box simulation in which the driver has selected “Accelerate” from the cruise control lever (with the engine being on and the brake being released). Figure 12(a) shows the state of the system before the simulation run and Figure 12(b) illustrates the results of accelerating, namely a value being sent to the throttle. This form of simulation may be applied to as low or as high of a level of abstraction as desired in order to gain visibility into the desired behavior of the architecture. For example, one could choose to simply conduct black box testing by placing input tokens on actor places, executing the simulation, and then observing the resulting token values on output actor places. Alternatively, if a more detailed investigation is desired, the engineer may navigate the CPN hierarchical construction and observe such characteristics as the behavior of state changes within a state dependent object’s CPN representation. A detailed analysis of this state-dependent behavior is provided in [24].

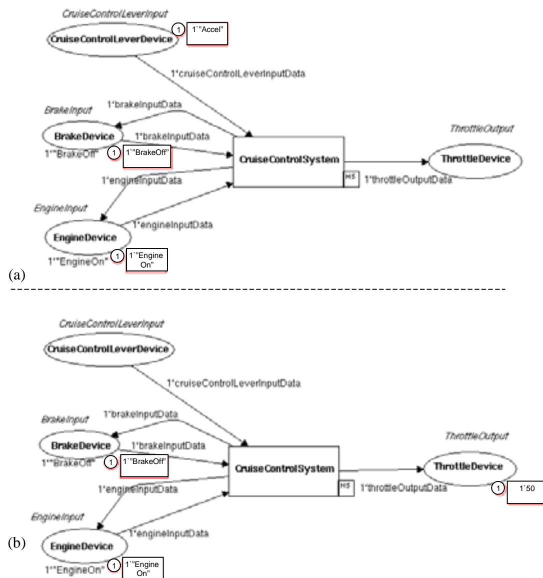


**Figure 12. Example Cruise Control Black-Box Simulation**

## 4.2 Validating Performance

In addition to simulation capabilities, the DesignCPN [27] tool used in this effort also has a very powerful performance tool [28] that can be employed to analyze performance aspects of the concurrent software architecture. This tool can be used to analyze such things as queue backlogs, system throughput, and end-to-end timing characteristics. As an example of the latter, we conducted a test to monitor the cruise control system response times to commands being input from the cruise control lever. To conduct this analysis, commands were issued to the cruise control system while the system was in a simulated state of operation with a speed of 60 miles per hour (100 kph). The performance tool was used to monitor changes in the throttle output and compare the time at an observed output change to the time the original command was issued.

The results from this analysis are shown in Figure 13. From this figure, we can see that all cruise control commands complete in less than one second (1000ms) and most complete in less than 500ms. Detailed performance requirements were not provided for our cruise control case study. However, if this cruise control system was an actual production system, an engineer could compare the analysis results against documented performance requirements to determine if the system in fact satisfies the necessary performance criteria. By being able to conduct this form of analysis from the concurrent software design, an engineer can both improve the reliability of the software architecture at the design level and correct problems prior to implementation.

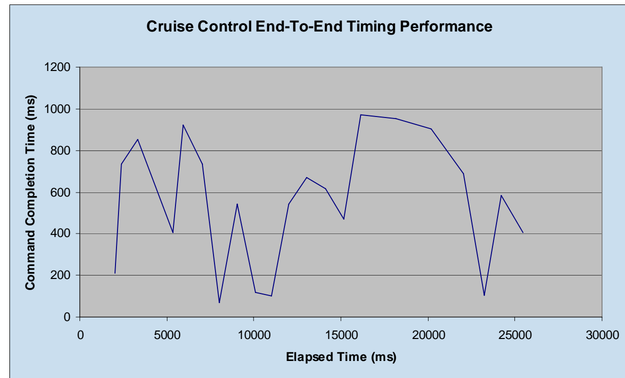


**Figure 13. Cruise Control End-to-End Timing Analysis**

# 5. Conclusions and Future Research

The long-term goal of this research effort is to provide an automated means of translating a UML concurrent software architecture design into an underlying CPN representation that can then be used to conduct behavioral analysis with results communicated in terms of the original UML model. To date, we have developed a method for systematically translating a UML software architecture into a CPN representation. This method employs reusable templates that model the behavior of a set of objects according to their stereotyped behavioral roles. Each template provides a consistent interface that allows templates to be interconnected as components of a larger system, thus creating the overall CPN representation. The resulting CPN model enables the analysis of both the functional and performance behavior of the concurrently executing objects. As the CPN representation mirrors the structure of the concurrent software architecture, the results can be readily applied to the original UML model.

Future research in this area will need to investigate approaches to facilitate the automated translation from a UML model into a CPN model that can be read by a tool such as DesignCPN. Additional research also needs to be conducted to investigate the scalability of this approach to larger systems, including distributed applications and providing behavioral templates for the COMET distributed components [4]. Finally, the use of state space analysis should be investigated further. Most of the analysis conducted with this research effort has focused on the use of simulations for functional analysis and on the performance tool for performance analysis. State space analysis

could also be used to further refine deadlock detection as well as to analyze system-wide state changes.

# 6. References


[1] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. 2nd Edition. Addison-Wesley, 2005.

[2] R. G. Pettit, *Analyzing Dynamic Behavior of Concurrent Object-Oriented Software Designs*, Ph.D., School of IT&E, George Mason University, 2003.

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

[4] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, 2000.

[5] M. Baldassari, G. Bruno, and A. Castella, "PROTOB: an Object-Oriented CASE Tool for Modeling and Prototyping Distributed Systems," *Software-Practice & Experience*, v.21, pp. 823-44, 1991.

[6] B. Mikolajczak and C. A. Sefranek, "Integrating Object Oriented Design with Concurrency Using Petri Nets," *IEEE International Conference on Systems, Man and Cybernetics*, Piscataway, NJ, USA, 2001.

[7] R. Aihua, "An Integrated Development Environment for Concurrent Software Developing Based on Object Oriented Petri Nets," *Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region.*, Los Alamitos, CA, USA, 2000.

[8] X. He and Y. Ding, "Object Orientation in Hierarchical Predicate Transition Nets," *Concurrent Object-Oriented Programming and Petri Nets. Advances in Petri Nets*, Berlin: Springer-Verlag, 2001, pp. 196-215.

[9] O. Biberstein, D. Buchs, and N. Guelfi, "Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism," *Concurrent Object-Oriented Programming and Petri Nets. Advances in Petri Nets*, Berlin: Springer-Verlag, 2001, pp. 73-130.

[10] S. Chachkov and D. Buchs, "From Formal Specifications to Ready-to-Use Software Components: The Concurrent Object Oriented Petri Net Approach," *Second International Conference on Application of Concurrency to System Design*, Los Alamitos, CA, USA, 2001.

[11] A. Camurri, P. Franchi, and M. Vitale, "Extending High-Level Petri Nets for Object-Oriented Design," *IEEE International Conference on Systems, Man and Cybernetics*, New York, NY, USA, 1992.

[12] J. E. Hong and D. H. Bae, "Software Modeling and Analysis Using a Hierarchical Object-Oriented Petri Net," *Information Sciences*, v.130, pp. 133-64, 2000.

[13] D. Azzopardi and D. J. Holding, "Petri Nets and OMT for Modeling and Analysis of DEDS," *Control Engineering Practices*, v.5, pp. 1407-1415, 1997.

[14] C. Lakos, "Object Oriented Modeling With Object Petri Nets," *Concurrent Object-Oriented Programming and Petri Nets. Advances in Petri Nets*, Berlin: Springer-Verlag, 2001, pp. 1-37.

[15] C. Maier and D. Moldt, "Object Coloured Petri Nets- A Formal Technique for Object Oriented Modelling," *Concurrent Object-Oriented Programming and Petri Nets. Advances in Petri Nets*, Berlin: Springer-Verlag, 2001, pp. 406-27.

[16] J. A. Saldhana, S. M. Shatz, and H. Zhaoxia, "Formalization of Object Behavior and Interactions from UML Models," *International Journal of Software Engineering & Knowledge Engineering*, v.11, pp. 643-73, 2001.

[17] L. Baresi and M. Pezze, "On Formalizing UML with High-Level Petri Nets," *Concurrent Object-Oriented Programming and Petri Nets. Advances in Petri Nets*, Berlin: Springer-Verlag, 2001, pp. 276-304.

[18] K. M. Hansen, "Towards a Coloured Petri Net Profile for the Unified Modeling" Centre for Object Technology, Aarhus, Denmark, Technical Report COT/2-52-V0.1 (DRAFT), 2001.

[19] J. B. Jørgensen, "Coloured Petri Nets in UML-Based Software Development - Designing Middleware for Pervasive Healthcare," *CPN '02*, Aarhus, Denmark, 2002.

[20] B. Bordbar, L. Giacomini, and D. J. Holding, "UML and Petri Nets for Design and Analysis of Distributed Systems," *International Conference on Control Applications, Anchorage*, Alaska, USA, 2000.

[21] R. G. Pettit and H. Gomaa, "Integrating Petri Nets with Design Methods for Concurrent and Real-Time Systems," *Real Time Applications Workshop,* Montreal, Canada, 1996.

[22] R. G. Pettit, "Modeling Object-Oriented Behavior Using Petri Nets," *OOPSLA Workshop on Behavioral Specification*, 1999.

[23] R. G. Pettit and H. Gomaa, "Validation of Dynamic Behavior in UML Using Colored Petri Nets," *UML* 2000, York, England, 2000.

[24] R. G. Pettit and H. Gomaa, "Modeling State-Dependent Objects Using Colored Petri Nets," *CPN 01 Workshop on Modeling of Objects, Components, and Agents*, Aarhus, Denmark, 2001.

[25] R.G. Pettit and H. Gomaa, "Modeling Behavioral Patterns of Concurrent Software Architectures Using Petri Nets." *Working IEEE/IFIP Conference on Software Architectures*, Oslo, Norway, 2004.

[26] R. David and H. Alla, "Petri Nets for Modeling of Dynamic Systems: A Survey." *Automatica* v.30(2). Pp. 175-202. 1994.

[27] K. Jensen, "DesignCPN," 4.0 ed. Aarhus, Denmark: University of Aarhus, 1999.

[28] B. Lindstrom and L. Wells, "Design/CPN Performance Tool Manual," University of Aarhus, Aarhus, Denmark September 1999.