Modeling and Prototyping of Real-Time Embedded Software Architectural Designs with Colored Petri Nets

Robert G. Pettit IV[1], Hassan Gomaa[2], and Julie S. Fant[1]

[1] The Aerospace Corporation,
Chantilly, Virginia, USA
{robert.g.pettit, julie.s.fant}@aero.org
[2] George Mason University
Fairfax, Virginia, USA
{hgomaa}@gmu.edu

Abstract. This paper describes an approach for constructing rapid prototypes to assess the behavioral characteristics of real-time embedded software architecture designs. Starting with a software architecture design nominally developed the using COMET concurrent object-oriented design method, an executable Colored Petri Net (CPN) prototype of the software architecture is developed. This prototype allows an engineer / analyst to explore behavioral and performance properties of a software architecture design prior to implementation. This approach is suitable both for the engineering team developing the software architecture as well as independent assessors responsible for oversight of the software architecture design.

Keywords: UML, rapid prototyping. coloured Petri-nets, real-time, embedded, concurrent, software architecture.

1 Introduction

The increasing complexity of software-intensive real-time embedded systems, particularly with respect to the behavior of concurrently executing software tasks, requires a thorough understanding of software architecture behavioral properties and tradeoffs among design decisions. Analyzing and understanding the concurrent behavior of real-time embedded software architectures during the early design stages is imperative to the successful and cost-effective development of the system. To address this issue, we present an approach for constructing rapid prototypes of embedded systems to assess the behavioral characteristics of concurrent software architecture designs. The approach leverages software design nominally developed using the COMET concurrent object-oriented design method [1] and reusable Colored Petri Net (CPN) [2] templates and components to rapidly prototype a concurrent software architecture. The goal of the CPN prototype is to compare and assess concurrent software architecture behavior to determine if the software architecture is feasible before spending valuable resources on hardware purchase, development, testing, etc. This paper expands on previous work [2] by specifically focusing on rapid prototyping / independent analysis of concurrent software architectures using reusable CPN components and templates. The complete set of CPN templates for the Unified Modeling Language (UML) [23] behavioral patterns used in this approach were defined in [2]. The resulting approach should provide the ability to quickly develop prototypes of software architecture.

1.1 Related Research

Prototyping the concurrent behavior of a real-time embedded system at design time is important to determine whether the system, with its set of concurrent tasks, behaves as desired both in terms of functionality and performance. If potential problems can be detected early in the life cycle, steps can be taken to overcome them.

Typical modeling and analysis methods include event sequence and queuing modeling [1, 3]; simulation modeling [4]; and scheduling analysis [3, 5, 6]. In recent years, there has been an increased effort to construct executable models of software designs and thus allow the logic of the design to be simulated and tested before the design is implemented. Existing modeling tools such as IBM® Rational® Rose® Technical Developer [7] and Ilogix Rhapsody [8] frequently use statecharts [9] as the key underlying mechanism for dynamic model execution. An alternative approach is to model concurrent object behavior using Petri Nets [10-14]. Our efforts [2, 14] have specifically focused on a Colored Petri Net (CPN) approach in which behavioral patterns are identified for objects via UML stereotypes in the software architecture and then modeled with CPN templates matching the behavioral patterns. We have chosen this approach since CPNs provide excellent modeling, analysis, and simulation capabilities for concurrent systems. Additionally, our approach supports independent assessments of the software architecture without requiring the software architect to adapt to a new paradigm. Furthermore, while our method for constructing architecture and design models is based on the COMET [1] approach, any design method that provides guidance on identification and classification of object roles and the structuring of concurrent tasks would be sufficient for our CPN modeling and analysis approach. With respect to COMET, we specifically use the stereotyped behavioral patterns for class roles, including input/output classes; control classes; entity classes; and algorithmic classes. We also use COMET's strategies for structuring concurrent tasks using UML active objects.

2 Rapid Prototyping Approach

The purpose of this paper is to describe an approach leveraging executable CPNs for the rapid prototyping of the behavior of communicating, concurrent tasks that make up the software architecture design of a real-time embedded. The purpose of the CPN prototypes proposed in this approach is to simulate the concurrently executing software tasks and to enable analysis and understanding of the concurrent behavior during the early design stages.

The proposed rapid prototyping approach has four major steps that are: 1) Develop the platform independent software architecture 2) Create the platform specific software architecture 3) Construct the CPN prototype 4) Execute and analyze the CPN prototype. Each step is described below in more detail.

2.1 Develop the Platform Independent Software Architecture Model

The first step in our approach is to develop the platform independent software architecture model (PIM). The purpose of the PIM is to capture the concurrent object behavior in the form of concurrent behavioral design patterns (BDP), which in subsequent steps will be mapped to CPN templates or components [2]. As discussed in previous work, each BDP represents the behavior of concurrent objects together with associated message communication constructs, and is depicted on a UML concurrent interaction diagram. Each object is assigned a behavioral role (such as I/O, entity, or control) which is given by the COMET concurrent object structuring criteria [1] and depicted by a UML stereotype. An example of a behavioral design pattern for an asynchronous device input concurrent object is given in Figure 1. Note that these behavioral patterns are commonly seen across the UML community as «bondary», «entity», and «control». With out approach using the COMET method, however, additional details are provided for such things as specifying input and output, identifying concurrency properties, and defining state-dependent behaviors.

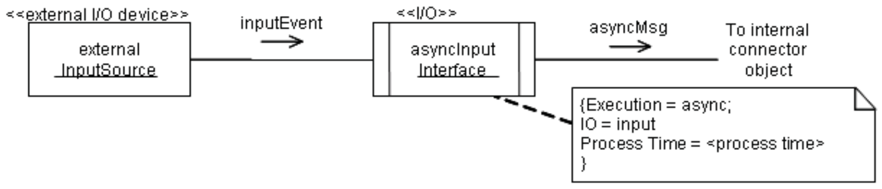

Fig. 1. Asynchronous input concurrent object behavioral design pattern

2.2 Create the Platform Specific Software Architecture Model

The second step in our rapid prototyping approach is to develop the platform specific software architecture model (PSM). The purpose of the PSM is to capture the performance characteristics of how the software architecture will perform if implemented on a specific platform. To enable fast construction of PSMs, the UML PIM model should be annotated with platform specific characteristics. This is quicker than creating a separate or external PSM model.

Platform specific characteristics and values can then be directly added to the UML software architecture model using a UML Profile such as the UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) [21]. The MARTE Profile provides the ability to capture non-functional performance characteristics directly in UML models. For example, tagging a message in an interaction diagram with the <<paStep>> stereotype indicates that it is a step in sequence that uses resources. The specific platform specific values, such as execution time, can be captured in the stereotype's tags like *hostDemand*.

Performance values can be determined from published information about the platform, as well as through measurement. Note that multiple PSMs can be applied to a given architecture, supporting prototypes for tradeoff analyses. These models may also be constructed at varying levels of fidelity depending on available information. As the development efforts mature, so then can the prototypes of the architecture. During this process, collaboration with domain experts and systems engineers is highly recommended in order to capture the most realistic and complete set of platform specifications. Section 3 of this paper illustrates both a PIM and PSM for a robot controller case study.

2.3 Construct the CPN Prototype

After the PSM is developed, an executable CPN prototype from the PSM can be systematically constructed. For each BDP in the PSM (identified by a UML stereotype), a self-contained CPN template is required, which by means of its places, transitions, and tokens, models a given concurrent behavioral pattern. A set of existing reusable CPN templates can be found in [2]. These templates include: I/O (boundary); entity; control; and algorithm. As an example, Figure 2 is the CPN template for an asynchronous device input concurrent object shown in Figure 1.

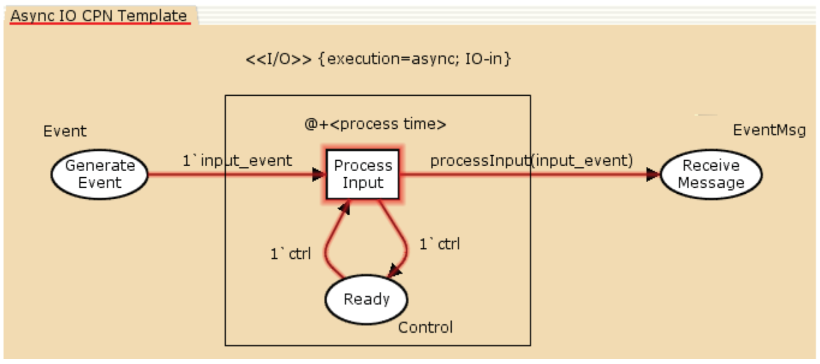

Fig. 2. Asynchronous input concurrent object CPN template

To instantiate the templates for each specific object, an analyst using our approach must provide a certain set of architectural parameters captured by following tagged values:

-<u>Execution Type</u>: passive, asynchronous, or periodic
-<u>IO</u>: input, output, or I/O
-<u>Communication Type</u>: synchronous or asynchronous
-<u>Activation Time</u>: periodic activation rate
-<u>Processing Time</u>: estimated execution time for one cycle
-<u>Operation Type</u>: read or write
-<u>Statechart</u>: for each «state dependent» object.

To illustrate pairing these architectural parameters with BDPs, refer to Figures 1 and 2. Figure 1 is an active object, “asyncInputInterface” that implements the I/O behavioral pattern as indicated by its stereotype. Furthermore, tagged types are used to capture specific architectural properties of the object, namely that it executes asynchronously; handles only input; and has a yet-to-be specified processing time of <process time>. The resulting CPN representation in Figure 2 reflects these parameters with the selection of an asynchronous, input-only CPN template and by setting the time inscription on the Process Input transition to @+<process time>.

This <process time> parameter is an estimate of the time required by the object to complete one activation cycle. This information can be obtained directly from UML MARTE annotations in the PSM. For example, process time can be found in the <<paStep>> stereotype in the *hostDemand* tag.

Since CPN templates provide only the basic behavioral pattern and component connections, they must be refined to provide application specific behavior.

To rapidly support construction of the prototype, we recommend using a reuse repository of CPN components. A CPN component is an elaborated CPN template for a commonly used object. For example, if a company commonly uses a specific sensor, a CPN component can be created for the software controller for the particular sensor. This CPN component can then be reused quickly in multiple different prototypes. Reusing CPN components will ultimately reduce the time it takes to construct the CPN prototypes. This is critical in rapid prototyping environments.

After all the BDPs in the PSM have an associated CPN templates or CPN components, the CPN templates and components are then interconnected via connector templates to create a prototype of the software architecture. The CPN prototype is then executed using a CPN tool, thereby allowing the designer to analyze both the concurrent behavior of the CPN prototype, with a given external workload applied to it.

3. Case Study: Robot Control

We illustrate our rapid prototyping approach using a robot controller case study based on the Lego® Robotics Invention System™ (RIS), commonly known as Mindstorms™ [16]. The RIS platform was chosen based on the embedded nature of the platform with easily reconfigurable sensors and actuators [18].

The robot controller case study is an autonomous rover employing an infrared light sensor and two motors (actuators). The goal of the rover is to search an area for colored discs, while staying within the course boundary and avoiding obstacles. In this case study, the light sensor is the sole input sensor, responsible for detecting boundary markings, obstacle markings, and discs according to different color schemes. This case study was used as a term project for a graduate course on real-time embedded software engineering at George Mason University.

3.1 Robot Controller PIM

The architecture model for the autonomous rover is illustrated in Figure 3. In this particular scenario, we are interested in navigating the course; responding to changes from the light sensor; and taking the appropriate action based on the detection event.

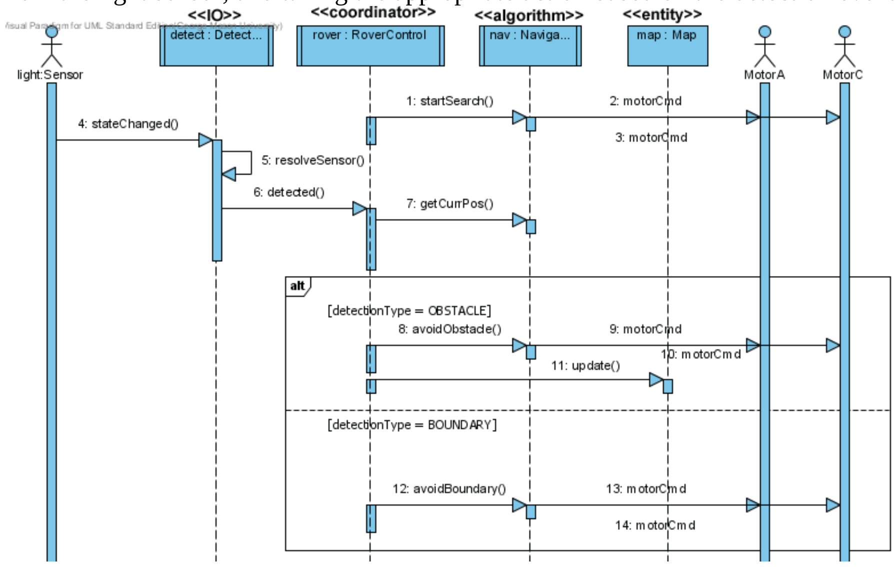

Fig. 3. Robot controller PIM interaction diagram

In this design, there are three active, concurrently executing objects (detect, rover, and nav) and one passive object (map). External I/O objects (depicted as actors in Figure 3) are also shown for receiving light sensor input and for modeling output to the two motors. Following object structuring guidelines from the COMET method, each of the objects in the system is stereotyped according to the hierarchy previously shown in Figure 1. These stereotypes indicate the behavioral design pattern (BDP) implemented by each object. Further details about the behavioral properties are augmented with the architectural parameters as follows:

The detect, rover, and nav objects all operate asynchronously and have an Execution Type tagged value of “async”. As the input interface for the light sensor, the detect object has an IO tagged value of “input”. All messages between the active objects have a Communication Type tagged value of “synchronous”, indicating synchronous, buffered communication. This particular design decision was made to decrease the risk of missing a boundary or obstacle detection event. Other design choices for this system would be to employ FIFO or priority queuing. The affects of these design decisions could also be analyzed using the techniques presented in this paper, but are not shown due to space limitations. Finally, the update() operation on the map object has an Operation Type tagged value of “writer”.

Note that values for the Processing Time parameters are left unspecified at this point as we will set these parameters based on the PSM in the next section.

3.2 Robot Controller PSM

The next step in our approach is to create a platform specific model for the target platform. This is shown in Figure 4 using historical data and hardware specifications for the RIS system [18-20].

In this model, our rover is identified as the single node in the system and is based on the Robot Command eXplorer (RCX) platform. The RCX is the central component to any RIS system and houses the Hitachi H8 microcontroller with a 16 MHz CPU. Paired with the leJOS Java environment the execution speed of this CPU is documented to be 1750 operations per second (OPS).

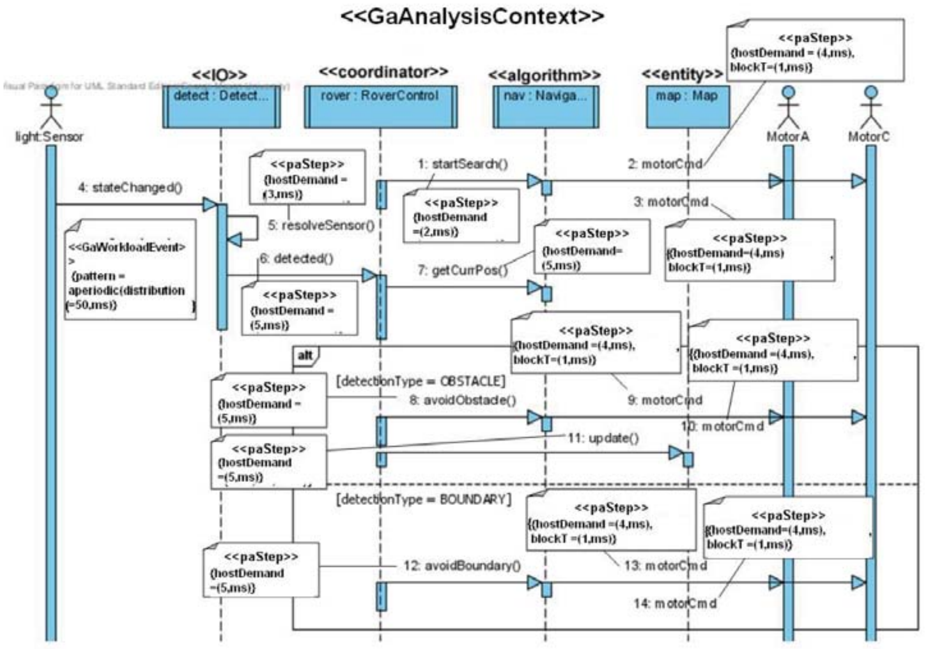
<<GaAnalysisContext>>
<<IO>>
<<coordinator>>
<<algorithm>>
<<entity>>
detect : Detect...
rover : RoverControl
nav : Naviga...
map : Map
light:Sensor
MotorA
MotorC
<<paStep>> {hostDemand = (4,ms), blockT=(1,ms)}
<<paStep>> {hostDemand = (3,ms)}
1: startSearch()
2: motorCmd
<<paStep>> {hostDemand =(2,ms)}
3: motorCmd
4: stateChanged()
5: resolveSensor()
<<GaWorkloadEvent>> {pattern = aperiodic(distribution =50,ms)}
6: detected()
<<paStep>> {hostDemand= (5,ms)}
<<paStep>> {hostDemand=(4,ms) blockT=(1,ms)}
7: getCurrPos()
<<paStep>> {hostDemand = (5,ms)}
<<paStep>> {hostDemand =(4,ms), blockT=(1,ms)}
<<paStep>> {hostDemand =(4,ms), blockT =(1,ms)}
alt
[detectionType = OBSTACLE]
<<paStep>> {hostDemand = (5,ms)}
8: avoidObstacle()
9: motorCmd
10: motorCmd
11: update()
<<paStep>> {hostDemand =(5,ms)}
[detectionType = BOUNDARY]
<<paStep>> {hostDemand =(4,ms), blockT =(1,ms)}
<<paStep>> {hostDemand=(4,ms), blockT =(1,ms)}
<<paStep>> {hostDemand =(5,ms)}
12: avoidBoundary()
13: motorCmd
14: motorCmd

Fig. 4. Robot controller PSM interaction diagram

Additionally, there are 16 KB of ROM and 28 KB of RAM available on the RCX of which, 17.5 KB of RAM are used by the leJOS operating system. The system clock resolution on the RCX, at 1ms, is longer than the time required for observed context switching between concurrent threads, thus the leJOS.overhead is set to zero. In our system, there are three physical devices attached: one light sensor at port S2 and two motors at ports A and C. Independent control of these motors is used to steer the rover; turning is achieved by rotating the left (Motor A) and right (Motor C) motors in opposite directions. Using historical data, the detection latency of the light sensor was set at 10.3 ms, while the output latency of the motors was set at 1 ms.

It would also be useful to combine the information from the PSM with historical data on software size. This type of information is commonly maintained by software development organizations and, in our case, we will rely on average software sizes across the set of student projects. From this data, we discover the following:

«IO» objects average 19 instructions (in Java bytecode) per execution cycle and have an average size of 1,182 bytes.
«coordinator» objects average 27 instructions and 2,722 bytes.
«algorithm» objects average 89 instructions and 1,015 bytes per algorithm.
«entity» objects average 1,400 bytes.

Now, using the combined historical sizing data and information from the rover, we can augment the PSM with this platform specific information. Prior to the CPU being available for performance measurement, an initial estimate of the execution time is computed by multiplying the estimated average number of instructions by the computational speed of the CPU (1750 OPS in our PSM example). These estimates are captured in the hostDemand tag.

3.3 CPN Prototype

Using the above PSM design information, we can now begin to construct a Colored Petri Net (CPN) prototype of the software architecture [2]. Using our approach, we start with a context level model, capturing the system as a black box (transition) and external sensors and actuators represented as places. This model, allowing us to focus on the highest level of abstraction with observed inputs and outputs is shown in Figure 5.

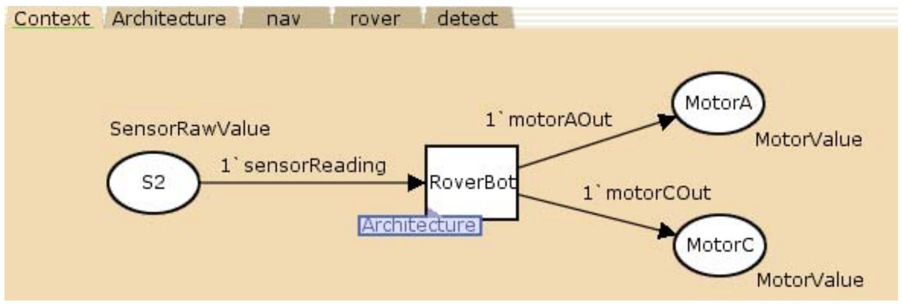

Fig. 5. Robot controller context level CPN

Moving forward, our second step is to decompose the RoverBot system-level transition into a layer of abstraction representing the concurrent object architecture. This architecture level model is shown in Figure 6. At this level, each of the active objects from is represented as its own transition (box) in the CPN prototype. Each of these will be further decomposed to implement the specific CPN template matching the objects behavioral design pattern or a CPN component if one exists for the object. We have also included the single «entity» object containing map data and it is represented by a place for the map data to be stored along with a transition and two places representing the behavior for calling the update() operation. Finally, as all message communication between active objects in the RoverBot system is synchronous, there is a CPN place modeling a buffer for the synchronous communication between detect and rover and between rover and nav. Notice that our external input and output places have also been carried down to this level as well.

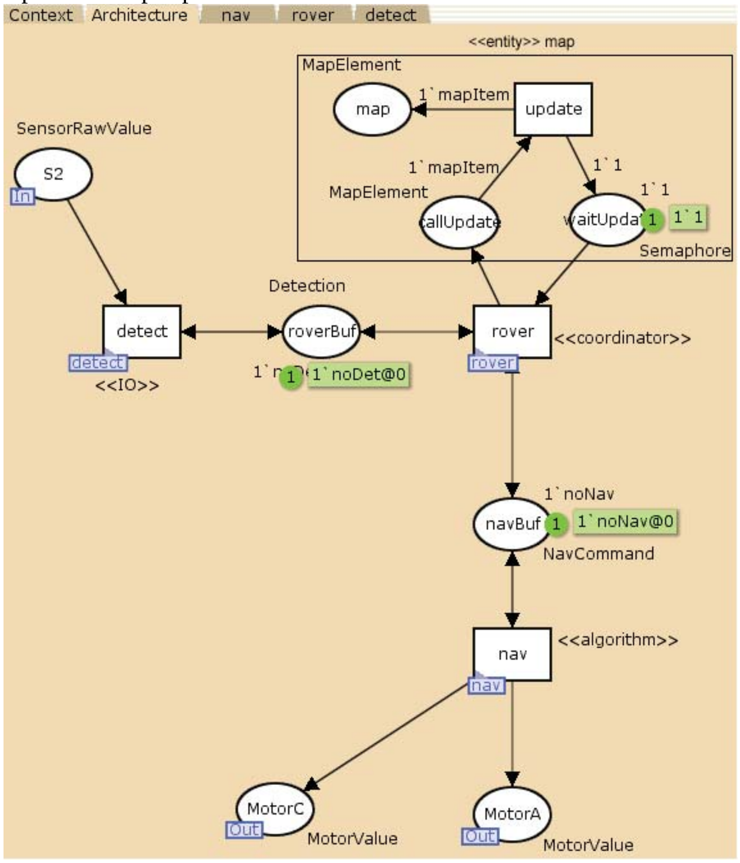

Fig. 6. Robot controller CPN architecture

Once an architecture-level model is established, each of the transitions representing an active object is then decomposed by applying the CPN template associated with the behavioral design pattern of that object. For the asynchronous, input-only «IO» object, "detect", this CPN object-level model is shown in Figure 7. Here, the CPN template has been inserted and instantiated specifically for the detect object by setting the object ID to "1" as seen by the number appended to place and transition names. The specific control token, C1 has also been added as has the function for processing detections, "detection (sensorReading)". To maintain consistency, the main transition of this template, Pin1, has also been connected to the sensor input place and to the roverBuf message buffer place.

Now, using the combined historical sizing data and information from the rover PSM, we can augment the architectural parameters within the CPN prototype to obtain further insights as to the behavioral and performance aspects that should be expected when matching the original platform independent design model with the actual platform characteristics of the target implementation.

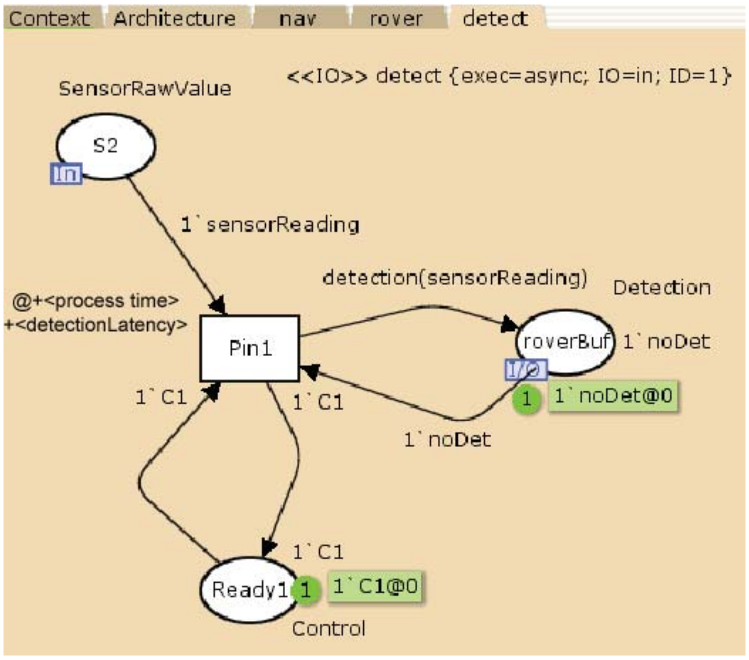

Fig. 7. CPN template for the "detect" object

To begin, we add a place, RAM, to our CPN prototype. This will model the available memory resources measured in bytes. The initial token value for the RAM place is calculated by subtracting the leJOS memory overhead along with the average RAM usage for the objects in our architecture from the total available RAM specified in the PSM.

Table 1. Calculating Memory Availability

Source	**Memory (Bytes)**
ram.sizeKB	28,672
lejos.kbMemOverhead	17,920
«IO» detect	1,182
«coordinator» rover	2,722
«algorithm» nav	2,030
«entity» map	1,400
Available RAM:	**3,418**

Once the rover system begins execution, the primary consumption of memory occurs when points are added to the map object. For each point added to the map, 16 bytes are used for x and y coordinates; detection event; and timestamp. To prototype this memory consumption, the RAM place from the CPN context level model is attached to the Update transition of the map object's CPN representation on the architecture level model. This is shown in Figure 8. Using this approach, 16 bytes are subtracted from the available RAM each time the update operation is called. If the system reaches a point where less than 16 bytes are available, then the CPN model will be suspended.

Next, the <process time> parameter will be updated for each active object template. This information can be obtained from hostDemand tag in the PSM's <<psStep>> stereotype. Additionally, each «IO» object template will add the detection or output latency to the <process time>. For example, the detect object, responsible for interfacing with the light sensor, would have a basic <process time> of 10.8ms. An additional 10.3ms are then added to account for the value of lightSensor.detectionLatency from the PSM, resulting in a total time delay 21ms. Once time values have been allocated to all objects, we can move forward with analyzing the prototype of the architecture as described in the next section.
These initial estimates can eventually be replaced with higher fidelity data as it becomes available, allowing an engineer to refine the behavioral analysis as desired.

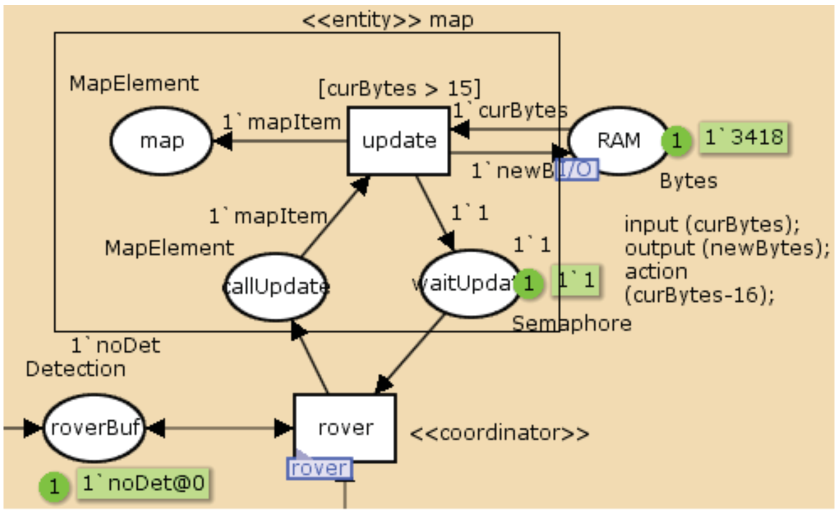

Fig. 8. Consumption of RAM by "map" object

3.4 Analyzing the Prototype

Recall from the sequence diagram of that the primary purpose of the autonomous rover system is to navigate an area, mapping objects discovered by the light sensor and taking evasive action when the light sensor detects obstacles or course boundaries. To begin analyzing this behavior with the corresponding CPN prototype, we use a test driver to provide simulated input events at random intervals. One of the first things we want to discover is how quickly the architecture responds to the detection of an obstacle or boundary. This can be analyzed from the context-level model by taking the difference in time stamps from the time an obstacle or boundary event arrives on the light sensor place to the time that a command is issued to the motors. For example, if the first obstacle was detected at time 6459 (all time is in milliseconds in this model). From the timestamps on the Motor places, we can see that from the time an input arrives to the time the system responded, there was an elapsed time of 31ms. This information could then be used, along with the speed of the rover, to determine if the reaction time is sufficient using this software architecture and this particular platform.

Other forms of analysis could include observing the memory usage over time or investigating the interaction among the concurrent objects as the simulated input rate varies. Analysis of physical architectural variations such as different light sensors or motors could also be conducted by applying different PSMs. Analysis of software architecture variations such as the use of different message communication mechanisms (e.g. FIFO or priority queuing) between the active objects could also be explored. These analyses are not shown due to space limitations in this paper.

3.5 Comparing Prototype with Observations

To validate our prototype, the rover design presented above was implemented in leJOS and the code was instrumented to capture timestamps. Execution of the rover when presented with boundaries or obstacles initially identified actual response times of 25-27ms from the point that the light sensor was presented with the boundary or obstacle to the point that the first motor command was output in response to the detection. This is slightly under the 31ms estimated by our analysis in the previous section. Interestingly, though, as we conducted tests with the rover over time, we observed response times increasing as battery power decreased. The above measurements of 25-27ms were observed with fully charged (9.0V) batteries. However, response times of up to 33ms were observed as the battery power was depleted to 8.2V. These results are summarized in Table 2 below. Replacing the depleted batteries with a fully charged set returned the response times to the initially observed 25-27ms. Thus, we believe that future research should include a power source with the embedded platform specific model.

Table 2. Response Time Results

Team	Response Time in Milliseconds			
	Run 1	**Run 2**	**Run 3**	**Run 4**
1	27	27	29	33
2	25	26	27	27
3	26	25	26	28
4	26	27	29	32
5	25	25	26	26

4 Conclusions and Future Research

In this paper, we have presented an approach to combine information from platform-independent and platform-specific models to construct prototypes of software architectures for embedded systems. This approach allows an engineer / analyst to examine behavioral and performance properties of a software architecture design paired with a candidate implementation architecture. The underlying CPN prototype is particularly useful in modeling concurrent object architectures in event-driven, real-time embedded systems. Applying the behavioral design patterns in the UML-based design along with corresponding CPN templates and components, the results from the analyses can be directly mapped back to the original design artifacts. Furthermore, by employing architectural parameters such as processing time, the CPN analysis model can be rapidly modified to account for different candidate architectures.

There are other tools available for constructing executable models of software designs such as the Rhapsody tool by IBM® [24]. While these tools are certainly useful, we feel there are certain advantages to our CPN approach. Modeling and prototyping of the architecture design is possible without depending on a particular UML modeling tool or design method – we only require that the basic behavioral patterns be identified. In fact, while we illustrate this approach with the UML, there is really no need to enforce the use of UML as long as the patterns can be identified for individual software abstractions. One such study applying this approach to a non-UML design is currently underway and will be published in the future. Furthermore, our CPN approach is more tolerant to varying levels of fidelity than other executable modeling tools. Using Rhapsody as an example again, each object must have detailed specifications (typically in the form of a state machine) in order for the model to be executed. With the CPN approach, even the most basic architecture designs can be simulated to show concurrent interactions, with increasing levels of fidelity as more specifications are added.

Future research in this area must continue to examine properties that should be captured and the most effective ways in which to capture them. In comparing our observed results to our analyses, the inclusion of a power model would obviously be desired in an embedded system. Additionally, future work should consider the ability to model distributed software designs configured to execute on multiple distributed embedded nodes and the communication between them. Finally, as mentioned above, work is also underway to provide a more generic approach to the executable CPN approach that allows for flexibility in the origin of the software design, whether that is captured in UML or other modeling languages.

References

1. H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, 1 ed: Addison-Wesley, 2000.
2. R. Pettit and H. Gomaa, "Modeling behavioral design patterns of concurrent objects," presented at ICSE 2006, Shanghai, China, 2006.
3. L. Sha and J. B. Goodenough, "Real-Time Scheduling Theory and Ada," *IEEE Computer*, vol. 23, pp. 53-62, 1990.
4. C. W. Smith, *Performance Engineering of Software Systems*: Addison Wesley, 1990.
5. H. Gomaa and D. Menascé, "Performance Engineering of Component-Based Distributed Software Systems," in *Performance Engineering 2001*, *LNCS*: Springer, 2001, pp. 40-55.

6. SEI, *A Practioner's Handbook for Real-Time Analysis - Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston: Kluwer, 1993.
7. IBM, "IBM Technical Developer," 2006.
8. Ilogix, "Ilogix Rhapsody," Ilogix, 2006.
9. D. Harel and E. Gery, "Executable Object Modeling with Statecharts," 1996.
10. M. Baldassari, G. Bruno, and A. Castella, "PROTOB: an Object-oriented CASE Tool for Modelling and Prototyping Distributed Systems," *Software-Practice & Experience*, vol. 21, pp. 823-44, 1991.
11. O. Biberstein, D. Buchs, and N. Guelfi, "Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism," in *COPN, Advances in Petri Nets*, *LNCS*: Springer-Verlag, 2001, pp. 73-130.
12. L. Baresi and M. Pezze, "On Formalizing UML with High-Level Petri Nets," in *COPN, Advances in Petri Nets*, *LNCS*. Berlin: Springer-Verlag, 2001, pp. 276-304.
13. K. M. Hansen, "Towards a Coloured Petri Net Profile for the Unified Modeling Language - Issues, Definition, and Implementation," Centre for Object Technology, Aarhus, Denmark, Technical Report COT/2-52-V0.1, 2001.
14. R. Pettit and H. Gomaa, "Modeling Behavioral Patterns of Concurrent Software Architectures Using Petri Nets," presented at 4th WICSA, Oslo, Norway, 2004.
15. B. Huber, R. Obermaisser, P. Peti, and C. E. Salloum, "Resource Specification of the DECOS Integrated Architecture," TU Wien, Vienna, Austria, Technical Report October 12, 2005 2005.
16. Lego, "Lego Mindstorms - http://mindstorms.lego.com."
17. R. Pettit, "SWE 626: Software Project Lab for Real-Time and Embedded Systems," George Mason University, 2006.
18. B. Bagnall, *Core LEGO MINDSTORMS Programming: Unleash the Power of the Java Platform*: Prentice Hall, 2002.
19. K. Proudfoot, "RCX Internals - http://graphics.stanford.edu/~kekoa/rcx/," 1999.
20. N. S. Andersen and M. N. Kjærgaard, "Advanced programming of the LEGO RCX for education," Technical University of Denmark, 2001.
21. UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) Beta 2, OMG In, June 2008, http://www.omg.org/cgi-bin/doc?ptc/2008-06-08
22. UML Profile for Schedulability, Performance and Time 1.1, February 2005, OMG Inc., http://www.omg.org/technology/documents/formal/schedulability.htm
23. Unified Modeling Language (UML), Version 2.2, February 2009, OMG, http://www.uml.org.
24. IBM® Rational® Rhapsody, http://www-01.ibm.com/software/awdtools/rhapsody/