Synthesizing Executable Models of Object Oriented Architectures

Lee W. Wagenhals, Sajjad Haider, and Alexander H. Levis
Systems Architecture Laboratory, C3I Center, Mail Stop 4B5,
George Mason University, Fairfax, VA 22030-4444, USA
{lwagenha, shaider1, alevis} @ gmu.edu

Abstract

The United States Department of Defense (DoD) has mandated the development of Command, Control, Communications Computers, Intelligence, Surveillance, and Reconnaissance (C4ISR) Architectures to support the acquisition of systems that are interoperable and will meet the needs of military coalitions. This paper provides a general description of an architecting process based on the object oriented Unified Modeling Language (UML) that includes three phases: analysis, synthesis, and evaluation. It then provides a rationale for style constraints on the use of UML artifacts for representing DoD C4ISR architectures. Finally the paper describes both a mapping between the UML artifacts and an executable model based on Colored Petri nets that can be used for logical, behavioral, and performance evaluation of the architecture. A procedure for the conversion is also provided.

Keywords: C4ISR Architectures, Object Orientation, UML, Colored Petri Nets

1 Introduction

Over the past decade, military organizations have mandated the use of architectures to address increased uncertainty about requirements, rapid changes in technology, changes in organizational structures, and a widening spectrum of missions and operations. Today, military organizations must respond to a variety of situations by quickly assembling and organizing coalitions from different components. These organizations must have the agility and flexibility to adapt to rapidly changing circumstances and bring about desired outcomes. This "plug and play" concept requires an unprecedented level of interoperability in the information systems that support the various units of a coalition. To achieve this flexibility, the US Department of Defense has looked to information architectures to provide current or future descriptions of various "domains" composed of components and their interconnections, actions or activities those components perform, and rules or constraints for those activities. One of the main areas of emphasis is the information exchange that will take place between elements of the architecture. The department has published the C4ISR Architecture Framework [1997] that provides common definitions, data, and references, and describes a set of products that comprise three views of an architecture. While this framework provides a standardized format for describing architectures, it does not provide a procedure for developing the artifacts that are used in the description. The lack of a definitive process has posed a challenge to those who are responsible for developing architectures that are compliant with the framework. Indeed, the issue of how to implement to the framework has been an area of research by the authors.

Responding to this requirement, two fundamental approaches have been used to implement the C4ISR Architecture Framework: structured analysis and object orientation [Levis and Wagenhals, 2000; Wagenhals, et al., 2000]. Which ever approach is used, the fundamental goals in developing the architectures are (1) to obtain a description of a C4ISR architecture, based on three architecture views, that answers the users' and operators' (warfighters') questions and supports the acquisition community's needs, (2) to develop an implementable process that uses current knowledge and tools to produce the desired description, (3) to develop an executable model that enables behavioral analysis and performance evaluation, and (4) to be able to carry out that evaluation. To support these goals, architecture design processes have been partitioned into three phases: analysis, synthesis, and evaluation. The analysis phases involves the development of static, dynamic, and implementation representations of the architecture. Note that these representations are static views, they contain a great deal of descriptive material including descriptions of dynamic behavior, but they cannot execute and generate dynamic behavior. In the synthesis phase, the elements of information developed in the analysis phase are converted to an executable model. These models are capability of generating dynamic behavior. In the evaluation phase, the executable model is used to verify the logical, behavioral, and performance aspects of the architecture and compare them with user requirements.

In order to use Object Orientation for architectures, research has been undertaken to address three problems; (1) can an Object Oriented process be developed and used to design an architecture, (2) what is the set of UML [Object Modeling Group, 1999] diagrams that should be used to represent a complete architecture for information or C4ISR systems, and (3) can a process for converting the object oriented representation of an architecture to an executable model be developed, thus formalizing the synthesis phase of the overall process? In Bienvenu et al., [2000] the artifacts of object orientation and their use in designing C4ISR architectures conformant to the C4ISR Architecture Framework, version 2.0, were discussed. While that work established the fact that an architecture could be developed using object orientation and that the resulting design could be mapped to the requisite C4ISR

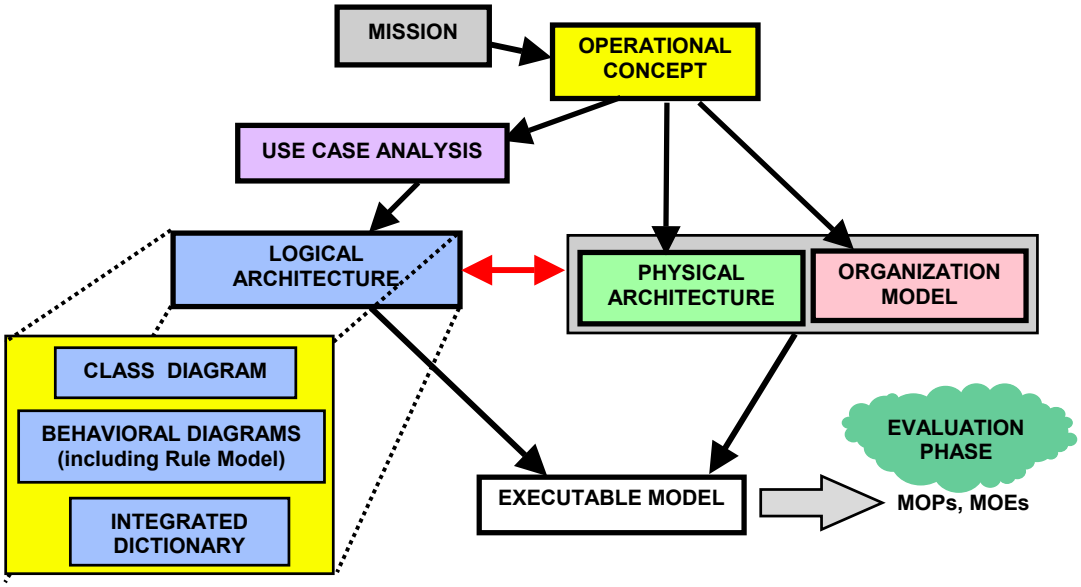

Figure 1. Top Level View of the Architecting Process.

products (problems 1 and 2), it stopped short of addressing problem 3. This paper introduces a new formulation of the architecture design problem that leads to an executable model (problem 3).

This paper is structured as follows. Section 2 describes the basic UML constructs that are used to render an architecture and describes a process for creating an architecture that is composed of those constructs. As part of that process, a specific style is imposed so that the conversion of the architecture to the executable model can be readily accomplished. Section 3 describes the mapping, and thus a conversion process from the UML artifacts used in the architecture to the executable model. In this case, the Colored Petri net (CP-net) [Jensen, 1997] formulism is used for the executable. Finally, Section 4 provides the conclusions and directions for further research. The authors assume that the reader has some familiarity with the main UML artifacts, and a basic understanding of colored Petri nets, in particular the elements of their graphical representation.

2 Object Orientation for Architecting

The Uniform Modeling Language (UML) has become the standard for visualizing, specifying, constructing, and documenting systems. It is a modeling language that has a vocabulary (symbols), semantics, and syntax. Within the language, there are two classes of modeling constructs or views, called diagrams. The structural diagrams, i.e. class, object, component, and deployment diagrams, document the static aspects of the system being modeled. Behavioral diagrams, i.e. activity, collaboration, sequence, state chart, and use case diagrams portray the dynamic behavior of the system. When used to specify a system, each of these diagrams represents a specific aspect of the same system.

UML is a rich language that can be used to represent architectures of information systems, including C4ISR systems, using multiple views. Two problems must be addressed: what is the necessary and sufficient set of UML artifacts to represent an architecture in the analysis phase and what is the process by which these views will be created. The fact that we want to be able to synthesize an executable model in the form of a Colored Petri (CP) net helps answer the first question because the UML views will need to contain all of the information necessary to full specify the CP-net. The conversion process requires that all of the artifacts are consistent with one another. This means that the process must include a formal procedure for model concordance.

Figure 1 shows a high level depiction of the process. One starts with a mission, which, in the context of C4ISR architectures, is usually is a military mission, function, or task. From this mission, an operational concept is created that describes how the mission will be carried out. The operational concept then drives the architecture design process. The architecture can be viewed as having two perspectives, a logical perspective and a physical perspective. The logical perspective details the activities and information flows that will accomplish the operational concept. The physical perspective describes the physical nodes and links that will be instantiated to carry out the activities of the logical perspective. It may also involve a description of the organizational structure that will be associated with the physical systems and the roles of the elements of the organization that will be involved in carrying out the activities. When using UML, Use Cases offer a means for refining the operational concept and can act as a bridge between it and the logical perspective. We call the logical perspective the logical architecture view. Elements of the logical architecture can be allocated to the elements of the physical architecture to complete the architecture design. The two-headed arrow between the logical and the physical perspectives in Figure 1 illustrates this approach. From a UML perspective, the logical architecture is composed of both static structural diagrams including class and object diagrams, and behavioral representation, including activity, sequence, collaboration, and state chart diagrams. The physical architecture can be represented using either class diagrams or implementation diagrams.

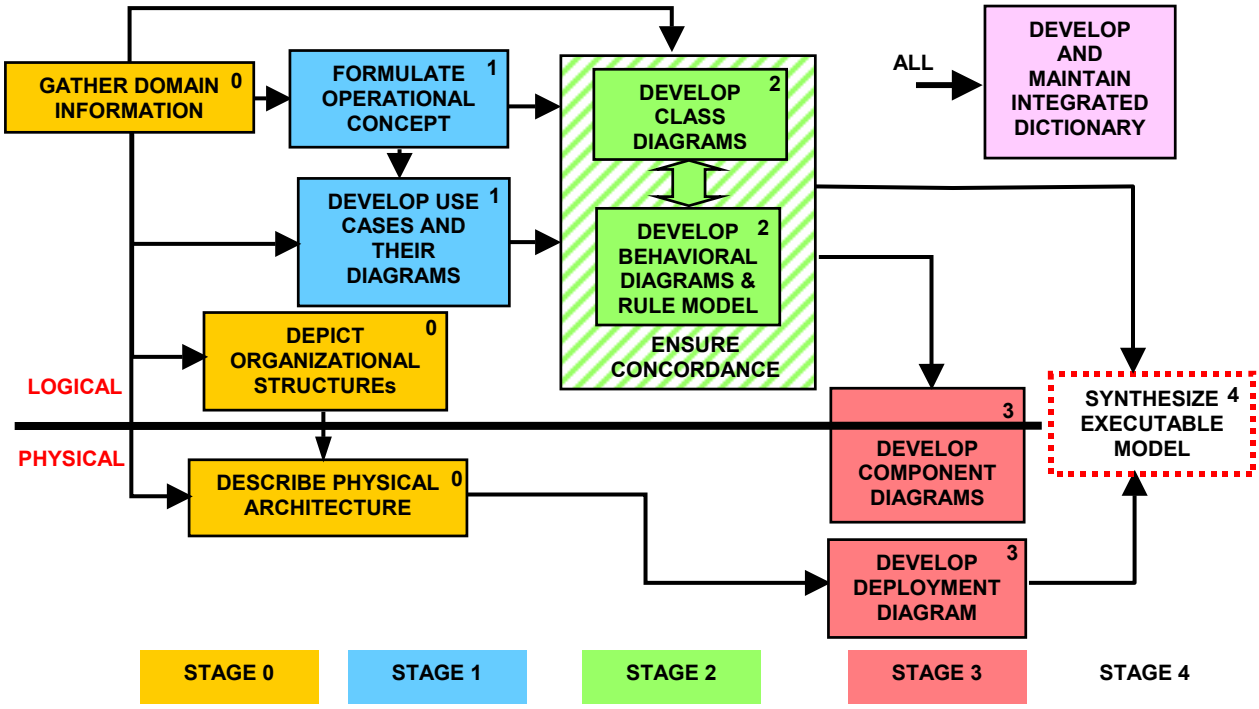

Figure 2. UML based architecture design process

The development of the Logical, Physical, and Organizational models constitutes the analysis phase of the architecting process. Once completed, it must be possible to synthesize the executable model from these artifacts. This constitutes the synthesis phase of the architecting process. Once the executable has been constructed and tested, it can be used to support the evaluation phase. In this phase, the executable model is used compare the behavior of the architecture against user requirements and to generate data to compute measures of performance (MOPs) and measures of effectiveness (MOEs). It should be noted that it is possible to develop an executable model from the information contained in the logical architecture view only. Such an executable can be used to evaluate logical and behavioral aspects of the architecture. .

A five stage process has been developed that uses the UML and its associated diagrams to design and represent architectures of C4ISR systems (Figure 2). In stage 0, all of the domain information needed to design the architecture is collected. This includes descriptions of the operational concept, functions, and tactics, techniques, and procedures as well as physical systems and organizational structures that will be included in the architecture. In the second stage, the operational concept is formulated and depicted as an operational concept graphic. In addition, use cases and their diagrams are created to describe scenarios between users and the system for which the architecture is being developed. A scenario is a sequence of interactions between a user and a system. Once the required operation of the system has been defined, the logical architecture for carrying out the use cases is designed in stage 2. The architect decides what activities and information flows will accomplish the operational concept as defined by each Use Case, allocates those activities to classes, determines the attributes that each class needs to carry out its activities (operations), and develops the rules for each operation. Once the Logical Architecture View is completed, the Architect can show how the logical construct will be allocated to the physical aspects of the design in stage 3. There are at least two ways to do this. In one method, the architect can use component diagrams that reflect the actual components that will reside inside the physical nodes of the system and deployment diagrams that reflect the allocation of the components to physical nodes. Alternatively, the architect can use class diagrams to represent physical system nodes, messages, and components.

Throughout the stages, the architect must develop and maintain an integrated dictionary, a single repository of definitions and descriptions of all elements of every diagram in the architecture. In addition, the architect must ensure that consistency is maintained between all of the views. We call this maintaining concordance, and it is crucial throughout this process. Because of the general nature of their use, modern tools that support object oriented analysis do not support concordance in the manner required to design an architecture. However, tool vendors` e.g. Popkin Software, Inc. and Ptech, Inc. are responding to the need by providing special extensions to their basic tools that support the C4ISR Architecture Framework.

The concordances concepts are reflected in Figure 3. It shows activity, sequence and collaboration diagrams as well as a class diagram. The activities in the activity diagram use the same name as the equivalent operations in the collaboration diagram. The arrows in the activity diagram correspond to messages or events associated with links in the collaboration diagram. The message or event descriptions represent the operation being called by an object to include any parameters that are contained in the message. These messages descriptions are the same on both the collaboration and the sequence diagrams.

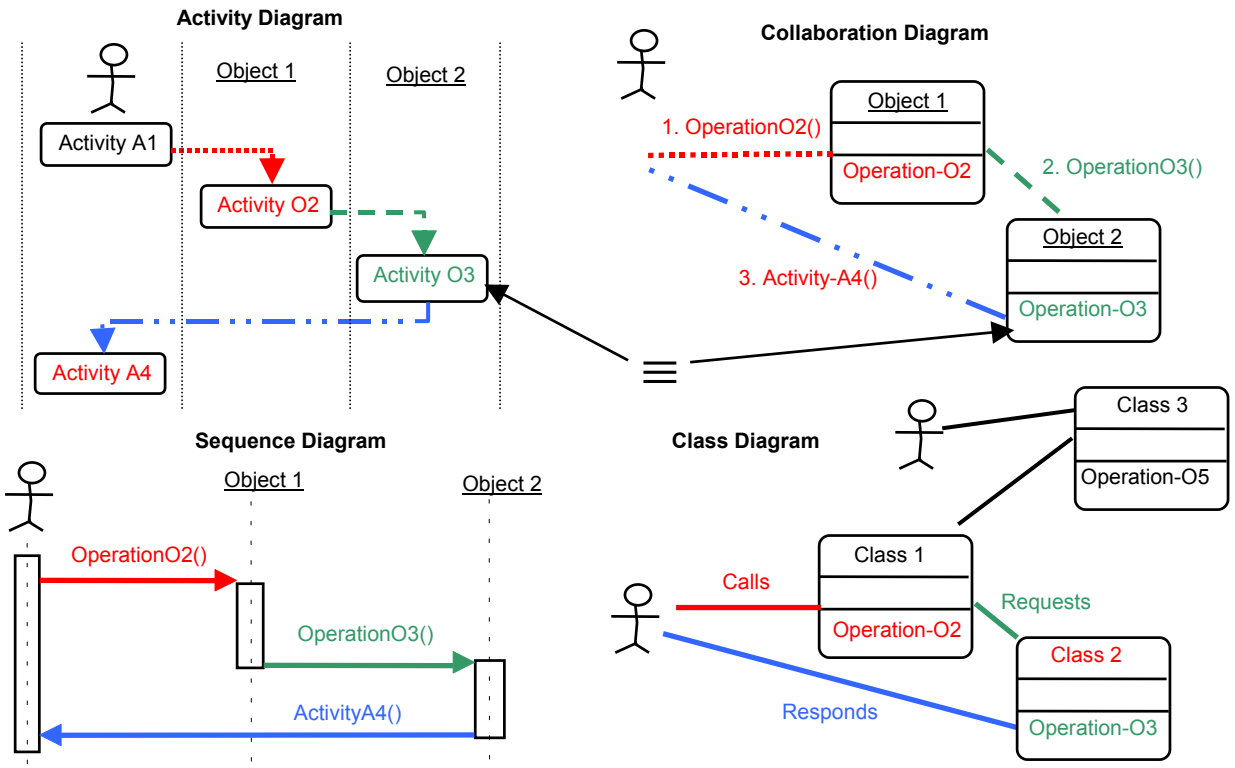

Figure 3. Concordance between activity, collaboration, sequence, and class diagrams.

Finally, the Class diagram reflects a composite of the collaboration diagrams plus other collaboration diagrams not shown.

Since it is a requirement that we be able to synthesize an executable model from the static views developed using UML, it was important to consider both the requirements of the architecture design and the target executable modeling formalism, CP-nets. CP-nets are not inherently object oriented. They have a fixed structure that does not change dynamically. On the other hand, objects can have relatively short life times, that is, they can be created and destroyed during the course of a scenario. The challenge is how to represent the types of objects that come and go in at CP-net. Of course, CP-nets have tokens that can be created and destroyed within the structure. Thus, one approach is to use tokens to represent objects. It is easy to understand that tokens can represent the attributes of objects. However, representing the operations in a manner that supports the visualization of objects is difficult.

C4ISR architectures are composed of a combination of fixed structures and objects that have short life cycles. The fixed structures are related to the physical components of the architecture. They include operational nodes and elements and system nodes, systems, system components, and communications links. Because of the emphasis on interoperability, C4ISR Architectures emphasize the information exchange requirements (IERs) between the static elements of the architecture. These IERs represent messages and data. This understanding induces the concept that the architecture is composed of fixed, long-term objects (the nodes and links) and the short term, transitory objects, the messages and data. Thus the fundamental concept for the conversion is that fixed elements of the architecture will be represented by the fixed structure of the CP-net (transition, places, and arcs), and the transitory messages and data will be represented by tokens.

To implement this fundamental concept for the conversion of the UML artifacts to the CP-net executable model, two style restrictions have been incorporated in the analysis process. In the first style constraint, the architect must partition the classes into those that represent the fixed structure and those that represent the transient messages and data. Messages will be passed between the fixed objects. Since messages or data will be represented by tokens in the CP-net, messages and data will be represented as classes that have only attributes. Thus, the messages or events that are described in the collaboration and sequence diagrams are rendered as association classes on the class diagrams between classes that represent the fixed classes of the architecture. Figure 4 shows a fragment of an activity diagram and the associated fragment of a class diagram. This architecture, that represents a billing system for a gasoline station[1], has two fixed objects: a pump and a gas station. Messages are exchanged between these two objects based on the operations of each. The activity diagram shows that a message is passed from an activity that is performed by a *Pump* object operation, *ProvideSalesInfo*, to an activity performed by a *Gas_Station* object operation, *ComputeCostOfSale*. The content of the message, which

[1] An example of a gas station credit card billing system is used in this paper rather than a C4ISR system because its operational concept is will understood by a general audience.

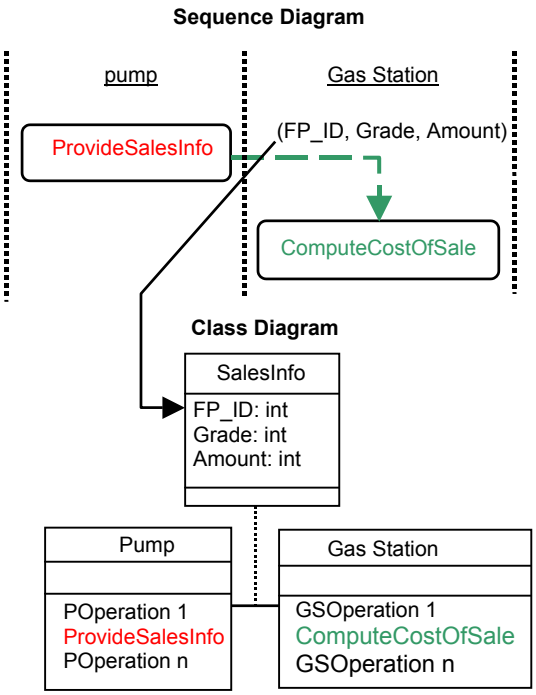

Figure 4 Association classes specify messages or events.

will be specified in the collaboration diagram, is a triple, (FP_ID, Grade, Amount). The corresponding class diagram has a *Pump* class and a *Gas_Station* class, each with assigned operations including the two shown in the activity diagram fragment. The message is manifested as an association class between the *Pump* and *Gas_Station* classes and has the three attributes that constitute the content of the message. As we will see in section 3, this will simplify the conversion of the information in the class diagram to the CP-net structure.

In the second style constraint, all non-association classes, which represent the fix elements of the architecture, will be converted into sets of classes that contain either operations or attributes, but not both. This can be done using the aggregation form of association. Any non-association class that has both operations and attributes is converted to a "parent" object that only contains the operations and one or more "child" objects that contain the attributes and have the aggregation association with the parent object. Figure 5 illustrates this conversion.

3 Synthesis of the CP-net from the UML based architecture

An algorithm has been formulated to facilitate the conversion of the Object Oriented artifacts into the executable model. The algorithm requires a specific style, explained in Section 2, which restricts the construction of those artifacts. It is essential that all concordance errors between various artifacts be corrected before the conversion to the executable model. A discrete event dynamical system model is appropriate because it allows the modeling of concurrent, asynchronous, event driven systems, which characterize C4ISR systems. We have adopted Colored Petri Nets [Jensen, 1992] for

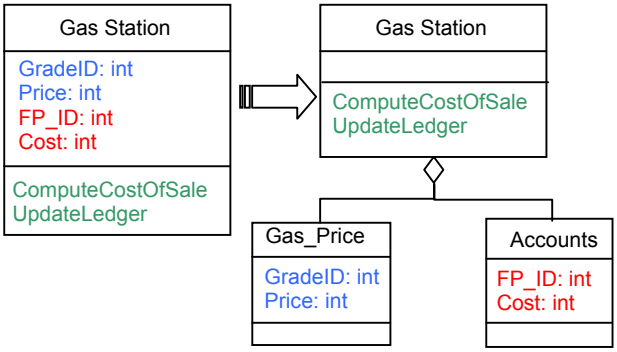

Figure 5. Aggregation association to implement style constraint.

representing the executable model because they can be used for mathematical analysis as well as for simulations.

Several approaches can be used to generate the executable model. For example, executable models can be derived from various behavioral diagrams (activity diagrams, state chart diagrams, etc.) or structural diagrams (class, object, implementation diagram, etc.). Our approach requires information from several concordant diagrams. We have chosen to use the class diagram to provide the basic structure for the conversion to a CP-net because it is the most general description of the object oriented design of the architecture. We assume the class diagram has been derived using the procedure described in Section 2 so that it contains information derived from activity, collaboration, and sequence diagrams. The conversion requires information from activity diagrams and rule models that have been specified for each operation of the classes.

In constructing the CP-net we need to create an unambiguous mapping between the elements of the various UML diagrams and the elements of the CP-net. This includes structural elements, i.e places, transition, input and output arcs, and logical elements, including color sets and variables in the global declaration node, the associations of color sets with places, arc inscriptions, guard functions, and code segments. Finally we need to determine the initial markings of the CP-net. We will illustrate the mapping with a series of figures. Figure 6 summarizes the algorithm.

Figures 7 through 10 will be used to illustrate algorithm for converting the class diagram to the CP-net using the architecture of the gasoline station. Figure 7 shows a fragment of a class diagram that has been created using the style constraints described in Section 2. Note that each class has either attributes or operations, but not both. The association classes present the messages that will be passed from one operation to another. The attributes of non-association classes have been captured in classes that have the aggregation association with the class that has the operations. In this example, a class called *Pump* will generate *SaleInfo* messages that are send to the *ComputeCostofSale* operation of an instance of the *Gas_Station* class. The *ComputeCostOfSale* operation will use the data contained in the *SaleInfo* message and data contained in the *GasPrice* class to generate an *Amount_Pumped* message.

1. Construct the Global Declaration Node using the attributes of all of the classes in the class diagram.
2. Construct the hierarchical CP net:
   - 2.1. Create a substitution transition for each interacting classes in the class diagram
   - 2.2. Create a place for each association and aggregated classes. Assign the appropriate colorset.
   - 2.3. Create arcs between the substitution transitions and the places using the activity diagram. There should be a one-to-one matching between the numbers of associations in the class diagram and the number of places between transitions in the executable diagram.
   - 2.4. Create a sub-page for each substitution transition.
     - 2.4.1. Create a transition for each operation.
     - 2.4.2. Assign the Input, Output, and I/O ports places
     - 2.4.3. Create the Arcs based on the activity diagram
     - 2.4.4. Add Arc inscriptions, guard functions, or code segments derived from the rules associated with each operation.
   - 2.5. Specify initial markings for each place that represents an aggregate class.

Figure 6. Procedure for synthesizing CP net from a UML based Architecture.

We begin the algorithm with step 1 by constructing the Global Declaration Node. The Global Declaration Node is constructed using information in the class diagram. "Atomic" color sets are defined for the different attributes of the classes in the class diagram. Each color set should have the same name as the attribute it represents. The domain of the attribute is specified in the class diagram and is used to define each color set. Variables are defined for each atomic color set. Color sets for the classes are defined as products of atomic color sets. ML records also could be used. These color sets are given the same name as the classes they represent and the color set region of each place is selected from the appropriate color set in the global declaration node.

Figure 8 illustrates step 1 of the algorithm, the generation of the Global Declaration node from a fragment of a class diagram of Figure 7. The class diagram shows two classes, a Pump and a Gas_Station class. The association class SalesInfo represents message that are passed from a *Pump* object to a *Gas_Station* object. The SalesInfo class has three attritubes, each of type integer: FP_ID, Grade_ID, and Amount. In the corresponding Global Declaration Node, each attribute is declared. In this example the color set SalesInfo is declared as a product of the three attribute color sets that comprise the SalesInfo class. Variables are also declared for the color sets. In a similar fashion products of the atomic color sets are used

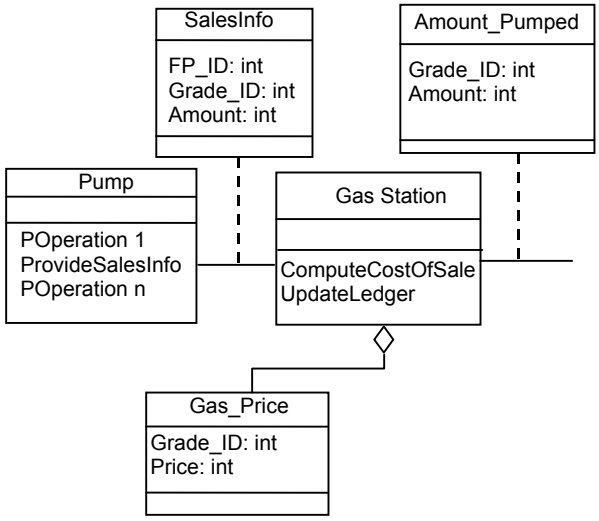

Figure 7. Fragment of Class Diagram.

```
Global Declaration Node
color FP_ID = int;
color Grade_ID = int;
color Amount = int;
color Price = int;
color SalesInfo = product FP_ID * Grade * Amount ;
color Gas_Price = product Grade_ID * Price ;
color Amount_Pumped = product FP_ID * Amount ;
var fpic : FP_ID ;
var gid : Grade_ID ;
var amt : Amount ;
var prc : Price ;
…
```

Figure 8. Fragment of Global Declaration Node.

to specify the color sets for the attributes of the non-association classes (which will correspond to tokens that represent the value of the attributes of the object with which the class is associated). In figure 8 the color set Gas_Price is declared as a product of Grade_ID and Price.

In step 2 of the algorithm, a hierarchical structure is developed for the CP-net. The top-level page corresponds to the class diagram. In step 2.1, on the top-level diagram, substitution transitions are created for each (non association) class that is not the "part-of" aspect of an aggregation relationship. These are the "parent" classes that contain only operations. In step 2.2 the classes that are the "part-of" aspect (the ones with only attributes) become port places on the top page and are connected to the substitution transition that represents the parent class with an input/output port and socket. Their color sets will be defined from the set of attributes in the class. Places are created on the top-level page for each association class. Each is given the appropriate color set that was specified in the global declaration node for the association class. In step 2.3 arcs are created on the top-level page.

Figure 9 illustrates the conversion of the fragment of the class diagram in Figure 7 to the CP net structure. In step 2.1 substitution transitions are created to represent the Pump and Gas_Station classes. These are the classes that

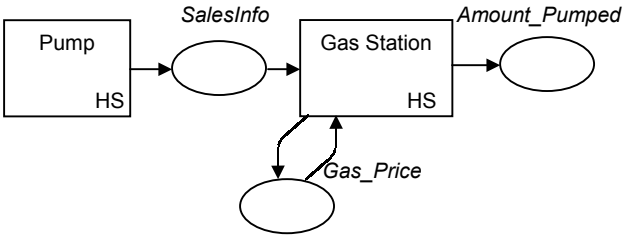

Figure 9. Fragment of Top-Level CP net Page.

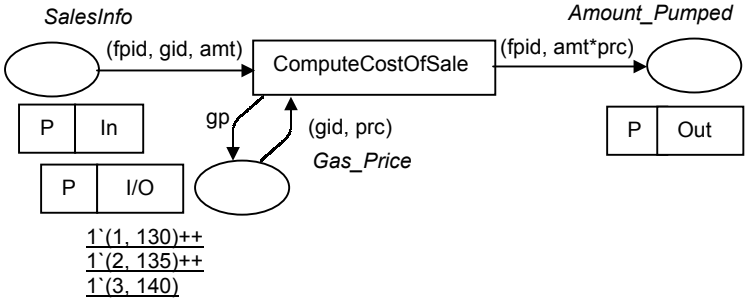

Figure 10. Fragment of Gas Station Substitution Transition Page.

have operations. In step 2.2, places are created for each association class (SalesInfo and Amount_Pumped). A place is created for the Gas_Price class that is "part-of" the Gas_Station. This place will carry a token that represents the values of the attributes of the Gas_Station. In step 2.3 arcs are created. The direction of each arc connecting a place and a transition is determined from the activity diagram (see Figure 4). Arcs are created from the Pump substitution transition to the SalesInfo place indicating the Pump generates (sends) a SalesInfo message (object). This message goes to the Gas_Station so an arc is created from the SalesInfo place to the Gas_Station substitution transition. Input/Output arcs are used between the non-association Gas_Price place and the corresponding Gas_Station substitution transition to allow the Gas_Station transition to access and update the value of its attributes.

In step 2.4, a sub-page is created for each substitution transition. Each operation for a class is represented by a transition that is placed on the sub-page of the substitution transition for the class to which it belongs. The places that represent the association classes and the classes with no operations that interact with the class are also placed on the sub-page of the substitution transition. They become input or output port places on the pages that represent the substitution transitions.

The direction of the arcs may be determined from the activity diagram or the collaboration diagrams, if they are not explicitly indicated on the class diagram. On the sub-pages of the substitution transitions, the activity diagram is used to determine how each port place is connected to the transitions that represent the operations of the class. Finally, the rule model [Levis and Wagenhals, 2002] is used to specify the arc inscriptions, guard functions, or code segments. These rules have the structure: **`if`** (set of input conditions) **`then`** (set of output conditions), **`else`** (set of output conditions).

Figure 10 shows a fragment of the Gas_Station sub-page featuring the transition that represents the *ComputeCostOfSale* operation. The activity diagram provided the information needed to connect the input and output port places to the *ComputeCostOfSale* operation. The rule model for the *ComputeCostOfSale* operation specified, "If (Sales_Info) then compute and send Amount_Pumped (FP_ID, (amount of gas * gas price per gallon)). (The Gas Price per gallon is specified for 3 grades of gasoline)." This rule can be implemented using the arc inscription shown. It could also be implemented using a code segment. The initial marking is provided to specify the value of the attributes of an instance of the *Gas_Price* class.

A more detailed example of this process has been developed and is used in academic classes. Lecture notes are available on line at Levis and Wagenhals, [2002] that contain details of both the UML based architectures and the synthesized CP-net.

It is possible to generalize the mapping between the UML artifacts and the CP-net. Figures 11 through 16 show the generalized mapping. They are similar to Figures 7 through 10, but contain generic classes, attributes and operations. They also contain the generic activity diagram and a generic rule associated with the one of the operations. Figures 11, 12 and 13 show related fragments of a class and activity diagram and a rule associated with Operation 2.1 of Class 1. Figure 11 and 14 reveal that there is a direct mapping from the class diagram to the Global Declaration Node. The structure of the CP-net is derived from both the class diagram and the activity diagram. The top page of the structure (Figure 15) comes directly from the class diagram, but the activity diagram is needed to specify the input and output arcs in the CP-net, both on the top page and on each sub-page of each substitution transition (Figure 16). Finally, there is a need for rules to be specified for each operation. These rules can be specified using structured language, (if…then…else). The rules are needed to specify the arc inscriptions of the CP-net and any guard or code segments that may be used. The rules must be consistent with the messages that invoke operations, any attributes of the object that are used to produce messages, and attributes whose value are changed by the incoming message as a result of the operation.

4 Evaluation of Architectures

Evaluating the effectiveness of C4ISR systems is a difficult undertaking (Sproles, 2001; Levis, 1997). Evaluating the architecture on the basis of which C4ISR systems will be designed and built is an even harder one. This is why the executable model is essential if evaluation and ultimately comparison of different architectures is to be accomplished. Once the executable model has been constructed, it can be used in three forms of evaluation: logical, behavioral, and performance. The first step is to validate the logic of the model.

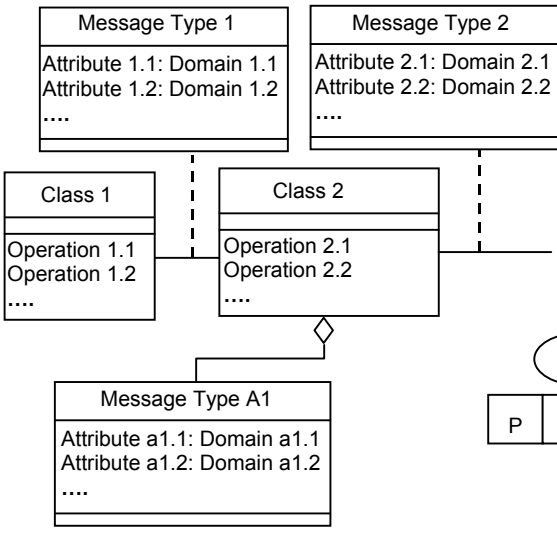

Figure 11. Fragment of Class Diagram.

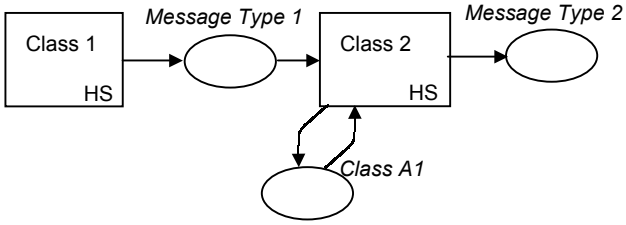

Figure 15. Fragment of General Top-Level CP net Page.

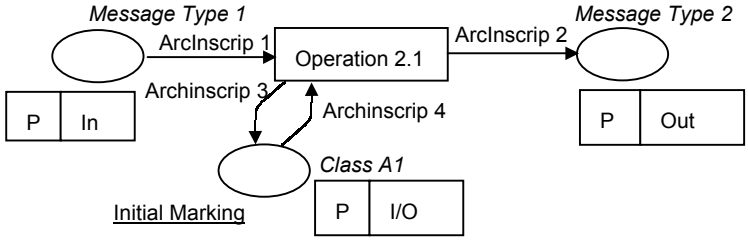

Figure 16. Fragment of Generalized Substitution Transition Page.

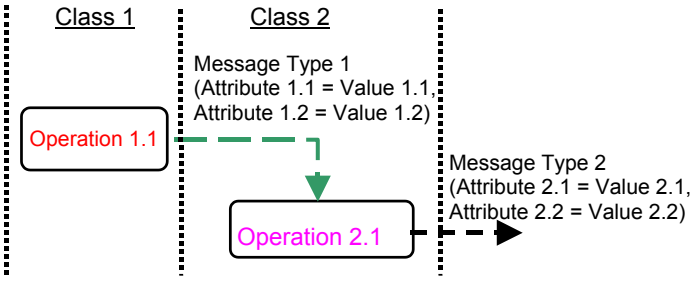

Figure 12. Generalized Activity Diagram.

```
Rule Operation 2.1

If  Message Type 1
(Attribute 1.1 = Value 1.1 and
Attribute 1.2 = Value 1.2)
then Message Type 2
(Attribute 2.1 = Compute(arg a1.1, Value 1.1) = Value 2.1,
 Attribute 2.2 = Compute(arg a1.2, Value 1.2 ) = Value 2.2)
Attribute a1.1 = Compute(arg a1.2, Value 1.1) = Value a1.1)
...
Else ....
```

Figure 13. Generalized Rule.

```
Global Declaration Node
color Attribute 1.1 = Domain 1.1;
color Attribute 1.2 = Domain 1.2;
color Attribute 2.1 = Domain 2.1;
color Attribute 2.2 = Domain 2.2;
color Attribute a1.1 = Domain a1.1;
color Attribute a1.2 = Domain a1.2;
color Message Type 1 = product Attribute 1.1 * Attribute 1.2 ;
color Message Type 2 = product Attribute 2.1 * Attribute 2.2 ;
color Class A1 = product Attribute a1.1 * Attribute a1.2 ;
var va111 : Attribute 1.1 ;
var va112 : Attribute 1.2 ;
var va121 : Attribute 2.1 ;
var va122 : Attribute 2.2 ;
var va1a11 : Attribute a1.1 ;
var va1a12 : Attribute a1.2 ;
```

Figure 14. Fragment of Global Declaration Node.

The static views describe the structure, the data, and the rules that manipulate that data to accomplish tasks. We need to verify that the combination of rules, data, and structure "works", e.g. the rules are consistent and complete. This can be accomplished by executing the model to be sure that it runs properly. In a sense we are "debugging" the architecture. Any errors found must be corrected in the appropriate static views to preserve traceability between the executable model and the architecture artifacts. One method that will do this is to test a single thread in the model and to examine each step of the execution to ensure that the model is following the logic desired. Any "flaws" will result in either an incorrect response or a deadlock. The execution should match the behavioral diagram models (sequence or collaboration diagrams). Once flaws are corrected, we know that the executable model runs. We know that the rules, structure, and data logically work together.

The next step is to examine the behavior of the architecture; this is an examination of the functionality of the architecture. The behavior of the executable model and the dynamics models, i.e., the activity, sequence, and collaboration diagrams, should correlate. This behavior evaluation has several facets. Does the architecture produce the correct behavior for a given stimulus? Does the information (or messages) arrive at the right functions in the right sequence, i.e., are the inputs processed in the required way? The behavior of the architecture should be compared to the user's requirements.

Because the executable model is a CP-net, the evaluation of the architecture can be supported by both simulation and analysis [Kristensen et al., 1998]. In using simulation, the behavior of the architecture can be examined by using inputs consistent with the operational concept. State space analysis allows behavioral properties to be verified by analysis without resorting to simulation. These techniques can compliment the multiple running of the model in simulation to reveal overall properties. The techniques can reveal dead locks (conditions in which the architecture stops executing),

infinite cycles (generally not desirable; they may indicate the circulation of messages without resolution) and the lower and upper bounds of the number of tokens (i.e., the queues and their properties) that can occur in any place in the architecture.

Some behavioral evaluation can be accomplished using an executable model derived only from the functional (or operational) architecture view. Single stimulus/response analysis can show that the architecture does what it is supposed to. Once the architecture has the desired behavior for single stimulus/response it can be evaluated for abnormal behaviors on the part of the external systems. This can reveal errors/omissions in the model. Additional behavioral evaluation can be accomplished when some aspects of the physical architecture are included. Processors, communications links, and their associated delays may affect the behavior of the architecture (e.g. sequencing of events). The impact of time delays and processing times can be evaluated. With the physical or systems architecture view providing performance parameters to the executable model, the latter can be used for performance evaluation**.**

5 Conclusion

There has been a growing need to be able to develop and evaluate architectures for C4ISR systems. The Department of Defense has released an Architecture Framework that specifies how architectures should be represented. Logical processes that are based on structure analysis have been developed and tested. There is a great deal to be gained if a sound process can be developed that uses UML constructs to build these architectures. We believe that such a process must guarantee that an executable model can be synthesized from the information contained in the UML artifacts.

We informally have shown such a process. It is based on imposing style constraints on the rendering of the architecture. It also requires strict adherence to the rules of concordance. The process has been demonstrated on a specific example of an information system. We believe the process can be generalized and then formally specified. Once this is done, it can be used by organizations that are tasked with producing and evaluating architectures. Such a formal specification can also be used by tool vendors to develop extension to their products that will automate the concordance process and the automatic conversion of an architecture created using UML to an executable model using CP-nets. Such automation currently does not exist in commercial tools so the conversion is a totally manual process.

The executable model can be a powerful tool for determining the potential behavior and performance of systems that are designed to be conformant to the architecture. They can take the community of architects from the process of just building architectures to the process of using architectures to support operational and investment decisions. Executable models of architectures that are timed phased can be used to generate the data needed to compute measures of performance and effectives that can be viewed over time. This can assist in the review and selection of investment strategies that are based on MOPs and MOEs. Thus, architectures can be a valuable tool for the management of change in an environment of uncertainty.

Acknowledgments:

This work was supported by the Office of Naval Research under grant No. N00014-00-1-0267. The contributions of Daesik Kim and Asma Ali of the System Architectures Laboratory at George Mason University are gratefully acknowledged.

References:

Bienvenu, M., I. Shin, and A. H. Levis, C4ISR Architectures III: An Object-Oriented Approach for Architecture Design, *Journal of Systems Engineering*, Vol. 3, No. 4, 2000

C4ISR Architecture Framework Version 2.0. C4ISR Architecture Working Group. Department of Defense, December 18. 1997.

Jensen, K., *Coloured Petri nets*, *Basic Concepts, Analysis Methods, and Practical Use. Volumes 1-3.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1997

Kristensen, L. M., S. Christensen, and K. Jensen, The Practitioner's Guide to Coloured Petri Nets, *International Journal for Software Tools for Technology Transfer*, 2(2): 98-132 Springer_Verlag, 1998.

Levis, A. H. "Measuring the Effectiveness of C4I Architectures," *Proc. Int'l Symposium on Defense Information Systems,* KIDIS, Seoul, Korea, June 1997.

Levis, A. H. and L. Wagenhals, L., C4ISR Architectures I: Developing a Process for C4ISR Architecture Design, *Journal of Systems Engineering*, Vol. 3, No. 4, 2000

Levis, A. H. and L. Wagenhals, C4ISR Architecture Framework Implementation, Course Notes for AFCEA Course 503, AFCEA, Fairfax, VA, 2002 available online: http://viking.gmu.edu/http/503_v/index.html

Levis, A. H. and L. Wagenhals, Advances in C4ISR Architectures, Course Notes for AFCEA Course 504, AFCEA, Fairfax, VA 2002 available online: http://viking.gmu.edu/504g/index.html

Object Modeling Group, Inc, Unified Modeling Language Specification, version 1.3, June 1999.

Sproles, N, The Difficult Problem of Establishing Measures of Effectiveness for Command and Control: A Systems Engineering Perspective, *Journal of Systems Engineering,* Vol. 4, No. 2, 2001

Wagenhals, L., I. Shin, D. Kim, and A. H. Levis, C4ISR Architectures II: Structured Analysis Approach for Architecture Design, *Journal of Systems Engineering*, Vol. 3, No. 4, 2000