UMLoquent Expression of AWACS Software Design

The UML is becoming the standard palette used by software designers to paint their thoughts.

ALEX E. BELL AND RYAN W. SCHMIDT

The primary objective of Boeing's NATO Midterm Modernisation Programme (NMT) is to produce a next-generation Airborne Warning and Control System (AWACS) that is computationally distributed, functionally scalable, and more technologically advanced than its predecessors. What makes this objective particularly challenging is that our group of 50-plus software engineers on the NMT project are generally very new to a number of the technologies being used to develop this system: distributed architecture, the 4+1 architectural model [4], object-oriented design, iterative development, Ada, and CORBA.

Another technology new to the NMT software engineering staff is the Unified Modeling Language, which is being used to express the software design. While the previous sentence may not seem overly profound upon first glance, the UML's tentacles reach into virtually all of the technologies previously mentioned as being used on NMT. It is the intent of this article to share the NMT program's usage of the UML and to relate the UML's connection to several of the technologies used in our program.

As an architecture-centric and use case driven program adopting an iterative development methodology, the UML is a very good choice for NMT to use in modeling its design. The UML is a standard language with what appears to be steadily increasing popularity. A diverse selection of UML-related tools and publications are available to improve the productivity of our software engineering staff. Using a popular, standardized modeling language to express our design increases the probability that new engineers coming to

NMT from other programs already know the UML and that less time will be required for them to become productive. Because the UML met NMT's programmatic and technical needs, was endorsed by a number of software engineering's critical mass, and appeared to be the standard by which future designs would be expressed, we did not conduct an exhaustive study investigating alternative techniques for capturing our software design.

Using the UML to Capture Different Levels of Abstraction

There are a number of different stakeholders in the NMT program who have a vested interest in the software under development but from different perspectives. Architects, requirements experts, developers, and management teams are interested in the same system's design but from different views and levels of granularity. The UML is an excellent vehicle for communicating the system design among these various stakeholders.

High-level diagrams contained in a domain model are used to describe the system design to the customer and other stakeholders interested in understanding its basic concepts but who have little interest in low-level details. It is the application model that contains more detailed information required by architects to assess a design and by developers to implement it. Should a domain model stakeholder desire further design detail, it is possible to navigate into the corresponding application model within the context of our design tool.

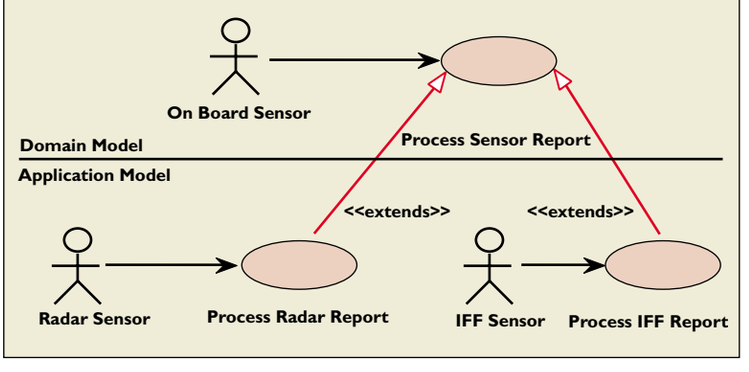
A domain level use case model captures the basic functionality of the NMT system from the perspective of its various actors. The application use case model, however, extends domain level use cases for the purposes of providing more granular detail. For example, the Process_Sensor_Report use case in the domain model addresses generalized sensor processing while the Process_Radar_Report and Process_IFF_Report use cases in the application model are the corresponding extensions that address the details of individual sensors. An example extension of a domain model use case is illustrated in Figure 1.

The varying levels of abstraction captured on the NMT program are also applicable to class diagrams. Domain level class diagrams capture NMT's primary abstractions and their corresponding relationships. These diagrams are composed of key classes plucked from the appropriate application class model with fine-grained abstractions typically ignored. Similar to NMT's use case model approach, a multilayered class model allows a stakeholder to view the design from a general perspective but supports the ability to optionally navigate into the detailed application model where all classes are available for inspection.

Mapping the UML to 4+1 Views of Architecture

It is through the methodical techniques of software architecture that NMT engineers are able to consider thousands of requirements representing the AWACS domain space without being overwhelmed by their number. The complexity of the domain is considerably reduced by decomposing it into hierarchical, abstract pieces whose details are hidden until a time in the software life cycle when it is appropriate for them to be addressed. These pieces must be viewed from a number of perspectives and a variety of implications must be considered that include their mutual relationships, scaleability, and reusability.

Figure 1. Domain model use case example.



The NMT program uses a Software Architecture Document (SAD) to capture the objectives and principles influencing its architectural evolution. An executable architecture [1] implementing NMT's primary scenarios is used to validate and exercise the architecture's ability to accommodate new and evolving functionality. The 4+1 View of Architecture model is being used on NMT to express its software architecture. The 4+1 model evolves a system's software architecture from four different views and ties those views together with use cases. While this approach of expressing software architecture is not new, the NMT program uniquely uses the UML to express a number of the artifacts associated with the various views of the 4+1 model.

Implementation view. The implementation view describes the organization of the components being used to build our system. The most primitive artifact

associated with NMT's implementation view is a subsystem. A subsystem contains a collection of functionally cohesive classes that are methodically grouped together to accommodate motivations of reuse and configuration management. Based on criteria that include domain independence and need to know, each subsystem belongs to one of six well-defined layers as further detailed in the sidebar "NMT Implementation View." The NMT implementation view is modeled with class diagrams using stereotyped packages that represent its layers and underlying subsystems. The NMT program models only the static aspects of this view with the UML.

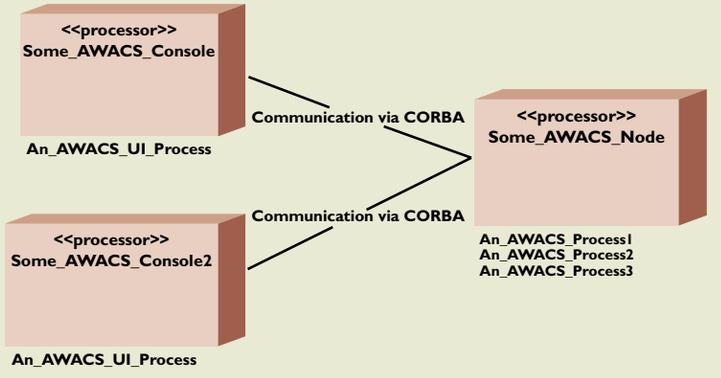
Process view. The process view is used to describe the threads and processes necessary to support a particular system's operational objective. The NMT program uses a combination of diagrams to express its process view.

The UML's deployment diagram is used to map NMT's processes to processing nodes. Because the deployment diagram does not provide the desired visibility into NMT's threading implementation, the class diagram in Figure 2a is instantiated to yield an object diagram that captures the mapping of threads to processes. The resulting object diagram contains all NMT processes and identifies the threads contained therein.

Figure 2a. Class diagram.

Figure 2b. Collaboration diagram.

Figure 3. Deployment diagram.



The UML's collaboration diagram is used to capture NMT's concept of class role. The concept of role is not applicable to all classes, but where necessary, is used to abstract where a class performs the services requested by its clients. Specifically, classes modeled in the logical view may act in different roles depending upon the particular process in which they are resident. A class with the role of server actually performs the work delegated to it by instances of the same class in other processes with a role of service_interface.

The collaboration diagram in Figure 2b is a simplistic example showing the means by which class roles are mapped to NMT processes. This example shows the journal class as having a role of server (S) in the Distributed_Services_Process while the Some_Client_Process contains an instance of the journal class in the role of service_interface (SI). It is an implementation detail of the service_interface to decide whether or not client requests need to be delegated to the journal server or accommodated locally. The actual diagram from which the figure is excerpted captures all NMT processes and identifies the roles of applicable class instances contained therein.

Deployment view. The processing nodes contributing to the NMT deployment view are captured using the UML's deployment diagram. Each node is modeled with its corresponding name under a «processor» stereotype and all applicable processes residing on the node are identified. The deployment diagram in Figure 3 is representative of the means by which NMT's deployment view is modeled.

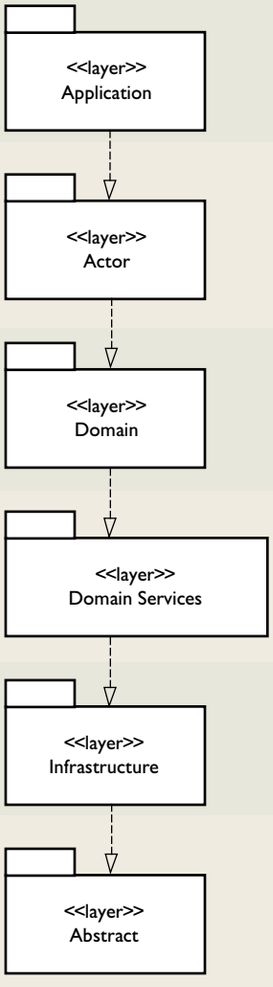
Logical view. The logical view expresses a system's functional requirements and is primarily filled with classes whose names are derived from artifacts of the domain space. Most of the NMT program's design effort is spent elaborating its logical view with the motivation of reaping the well-known rewards of building crisply abstracted, object-oriented, and layered components. The NMT program uses class diagrams to capture the static aspects of its logical view, sequence diagrams to express the dynamics occurring

between the corresponding class instances, and statechart diagrams to describe the dynamic aspects of a particular class. Because sequence diagrams emphasize the time order of messages, they are the generally preferred alternative to collaboration diagrams on the NMT program. The fragment in Figure 4a is indicative of NMT's class diagram usage at the domain level and captures a small portion of the surveillance design.

Use case view. The use case view contains the storyboards expressing a system's behavior from an external perspective. A program's initial use cases are those capturing its primary functional objectives and are used as a vehicle for touching upon all of the other architectural views simultaneously with the motivation of establishing a system's architectural foundation. Subsequent use cases are added to support requirements coverage and to stress an evolving architecture's ability to withstand potential shockwaves resulting from extreme requirements. The use case view serves not only as a vehicle to populate the other

NMT Implementation View

In the context of the NMT design environment, each of the implementation view «layers» is expandable with a double mouse click to reveal a subimplementation view exposing the subsystems that meet residency requirements of the selected layer. The general criteria for assigning a subsystem to a particular layer are described in the text to the right of each layer depicted in the diagram appearing here. An important characteristic of the NMT implementation view layers is that visibility is defined only in the downward direction to mitigate the effects of coupling and to increase the likelihood of software reuse.



The Application Layer contains only the glue subsystem whose artifacts define the main programs contributing to NMT's operational system.

The Actor Layer contains those subsystems whose classes translate the demands of actors (operators and external systems, for example) into actions upon domain classes. Similarly, classes in this layer are sensitive to notifications of domain object state changes in response to anonymous subscriptions. Many of the classes in this layer fit the Embassy[1] pattern described in this article.

The Domain Layer contains the subsystems whose classes embody the heart and soul of a system's primary domain space. In the case of NMT, this layer embodies the surveillance, identification, and battle management classes.

The Domain Services Layer contains classes that are germane to a particular domain space (C3I or air traffic control, for example) and are intended to be reused by similar domain layer instances. Examples of NMT classes residing in this layer include airbase, geometry, and radar observation with no changes expected to support future AWACS derivatives.

The Infrastructure Layer is the first of two layers comprising NMT's Domain Independent Distributed Infrastructure. A number of classes providing domain independent distributed services, such as recording and system management, are located here.

The Abstract Layer is the other half of NMT's Domain Independent Distributed Infrastructure. It contains abstract classes such as lists, tables, and queues. It also contains basic classes that include support for event reporting and time management.

[1]The Embassy pattern was invented on the Canadian Automated Air Traffic System (CAATS) program in 1994 and extended on NMT to consider ORB-related implications.

Figure 4a. Domain level usage of class diagram.

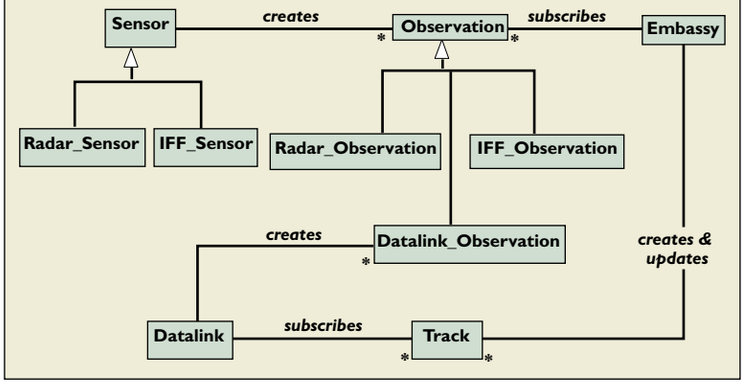
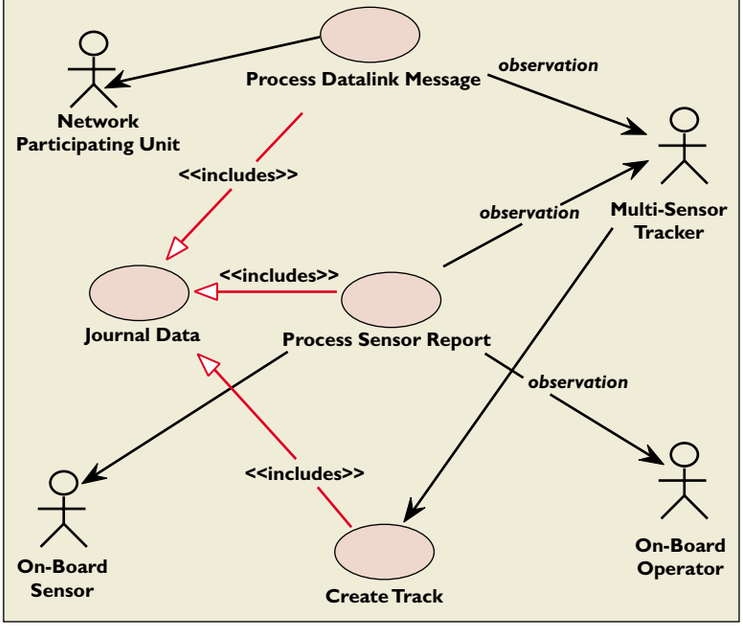


Figure 4b. Use case diagram usage.



views with their associated artifacts, but also as a means to validate the software architecture for fidelity and completeness. The diagram in Figure 4b is representative of NMT's use case diagram usage and models a portion of its surveillance design.

Using the UML to Express a Contract Across Systems

Three geographically dispersed companies are responsible for developing the NMT software. One develops the user interface, another develops the tracking software, while Boeing is the primary contractor developing the remaining functionality in addition to being responsible for software integration. A CORBA solution has been chosen to glue the various systems together in the most seamless and extensible manner possible. The contract defining the interfaces between these external systems is expressed using the Interface Definition Language (IDL) and is generated by forward engineering UML artifacts from the logical view.

The stubs and skeletons produced by preprocessing the forward-engineered IDL are used in support of one of NMT's most important design patterns, the Embassy pattern. The Embassy pattern is a hybrid of the Observer, Bridge, and Remote Proxy patterns [3] and has several important architectural objectives discussed here. The classes contributing to the Embassy pattern are captured in Figure 5.

The Embassy pattern is a metaphor for the notion of an embassy being used to represent another country's interests in a foreign land. Modeling the real world, NMT's embassy abstraction represents the interests of an external system within the confines of a foreign computer. The Embassy pattern uses the Bridge pattern to decouple external systems from the implementation of NMT's domain classes as well a Remote Proxy pattern to represent the interests of objects residing in different address spaces. The embassy abstraction is supported by two classes: the attaché and the correspondence. An embassy is composed of any number of attachés, each of which has a unique interest in the problem domain. An attaché is an observer of specific domain objects in which it has particular interest and embodies the rules of when and what is sent to the external system it represents in response to changes in those domain objects. The attaché also acts in the capacity of a bridge by transforming the desires of the external system it represents into actions upon NMT domain objects.

The correspondence class plays an important role in NMT's objectives of abstracting the fact that an ORB is being used to support its middleware needs. A correspondence object not only embodies the message content that is sent to or received from external systems but it also encapsulates the means by which the message is actually sent.

Difficulties Expressing the NMT Design

While the UML is strong in expressing the logical view of the NMT architecture, we have found it to be less powerful in its ability to describe our process and deployment views. Accurately capturing the allo-

THE NMT PROGRAM'S USAGE OF THE UML WILL EVOLVE PRECISELY IN THE SAME MANNER AS ITS SOFTWARE: IN A ROUND-TRIP, ITERATIVE FASHION.

cation of classes to processes and threads is critical to understanding our system design. Although the UML is capable of expressing process to node mappings through its deployment diagram, it does not provide the inherent capability we desire for mapping classes in the logical view to the processes or threads identified in the process view. As previously illustrated, we creatively use the UML to accommodate our modeling needs.

Some of the difficulties we have encountered in expressing our design cannot be blamed on the UML but are the fault of commercial design tools. We have used two leading design tools on NMT and have experienced a significant time lag between the declaration of standards in the UML and the ability to model those standards within the context of the tools. For example, as suggested in *The Unified Modeling Language User Guide* [2], a means by which to model processes and threads is to use active objects in the context of object or collaboration diagrams. These objects have stereotypes of «process» or «thread» and communicate via synchronous or asynchronous messages. Our tools do not support the annotation of the class and collaboration diagrams with message communication types or the identification of the physical location of the process. The design tool's deployment diagram does not provide the capability to allow physical nodes to contain component instances and components to contain objects.

Other examples in which design tools lag in the support of the UML include failure to support activity diagrams and inability to place notational components such as Actor figures on all desired diagrams.

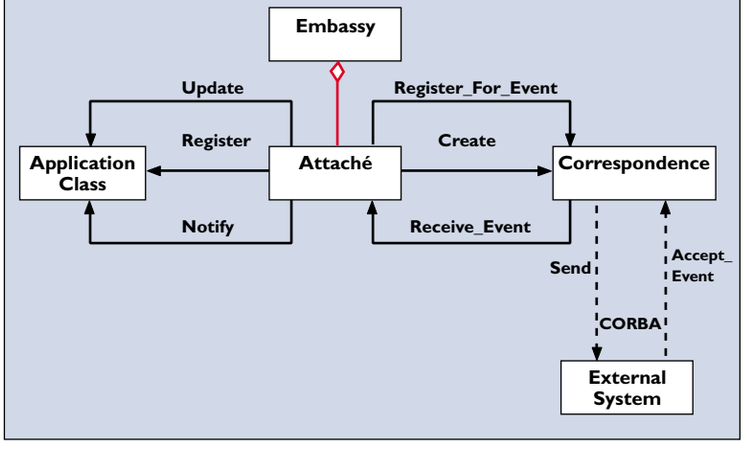
Using the UML to Support Round-Trip Engineering

Independent of how a program represents its software design, an objective not to be overlooked is how to prevent divergence between its design and implementation spaces. Unless a program has the capability to generate its design space artifacts from the implementation space, and vice versa, a program is at risk of divergence as a result of schedule pressures, budgetary concerns, and diligence issues. As the thumbscrews tighten, developers will modify their code in the interest of pain management but may defer making the corresponding changes in the design model. Volumes can be written on this subject, the following paragraphs will focus on the issues as applicable to the NMT program.

The Boeing Company's contribution to the NMT program is being written in the Ada programming language. Attempts to forward engineer classes contained in NMT's logical view into Ada specifications have been of little value because of the fuzzy semantics associated with mapping classes represented with the UML into Ada. For example, many of the NMT classes are implemented using Ada child packages for reasons of improved encapsulation and configuration management. There is currently no known means by which to annotate our classes or persuade our forward engineering toolset to create our Ada artifacts with the structure we would like them to have.

A similar problem exists with regard to reverse engineering our class artifacts from Ada specifications. All of our child packages are undesirably reverse engineered into classes when they are really just implementation details associated with the parent class. We know of no means by which to annotate our Ada artifacts to influence our reverse engineering toolset to

Figure 5. Embassy pattern classes.



behave as we would like it to do.

Because the problems experienced at the class level present major obstacles to NMT's effective usage of round-trip engineering, we have not investigated its usage to support consistency of our implementation space with other UML artifacts. Until we are able to find a means by which to automate consistency between our design and implementation spaces en masse, it is through diligent adherence to our software development processes that divergence between these spaces is minimized.

Conclusion

The UML has been very capable of supporting the evolution and expression of the NMT program's software design. In the areas we have had some trouble modeling our design, such as in the concept of class roles, we invoked artistic license to create diagrams that accommodated our needs. We also have a variety of UML-centric tools at our disposal with which to alternatively express, measure the progress of, and validate the fidelity of our designs.

The NMT program's usage of the UML will evolve precisely in the same manner as its software: in a round-trip, iterative fashion. Our diagram selection, level of abstraction, and breadth of stakeholder views will appropriately respond to perceived value added and level of need.

We would like to share how we envision our UML usage evolving on the NMT program, but have simply not yet taken enough round trips to do so. As we complete a software life cycle's worth of UML lessons learned, however, our currently imperfect hindsight will most certainly approach the 20/20 level. We intend to leverage our hindsight in the future so that we use the UML most efficiently and intelligently on potential derivatives of the NMT program. ■

References

1. Booch, G. *Object Solutions: Managing the Object-Oriented Project*. Addison Wesley, Reading, MA, 1995.
2. Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, MA, 1999.
3. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
4. Kruchten, P. The 4+1 view of architecture. *IEEE Software 12*, 6 (Nov. 1995), 45–50.

Alex E. Bell (alex.e.bell@boeing.com) is a software architect at The Boeing Company in Seattle, WA.

Ryan W. Schmidt (rschmidt@s1.com) is a senior consultant with Insight Technology Group in St. Louis, MO.