2006-01-1557

Using UML 2.0 to Create Executable Code from Requirements Capture and Consistent Requirement Specifications for Real-Time Automotive Software Development

Brendan Jackman
Waterford Institute of Technology, Ireland.

Shepherd Sanyanga
Ford (Europe), United Kingdom.

ABSTRACT

The development of vehicle control systems has evolved to become an exercise in the design and integration of complex, distributed hardware and software components. The various components are typically developed by geographically dispersed, multi-cultural teams from both OEMs and suppliers.

This paper gives a brief overview of using the Unified Modelling Language (UML) as a means of capturing the requirements of real-time distributed systems in a graphical notation shared by all team members. UML is commonly used to model system concepts, albeit typically as system "sketches" without any formal definition of the model's semantics. This paper specifically addresses the additions to the latest version of UML that supports higher levels of abstraction, model-based development, executable models and the specification of non-functional requirements. These improvements to UML make it more semantically complete, which means that a UML model can unambiguously describe a system, resulting in simpler automatic model verification and automatic code generation. The modelling of automotive network management requirements in a typical vehicle application is used to illustrate the benefits of the UML model development approach.

INTRODUCTION

UML has its origins in the early object-oriented analysis and design methodologies of Grady Booch (the Booch Methodology), James Rumbaugh and associates (Object Modeling Technique) and Ivar Jacobson (Objectory). After many years of working in parallel, these three approaches converged to become the first version of UML, resulting in UML V1.1 being standardized by the Object Management Group (OMG) in 1997. Since then UML has become the *de facto* standard for software and system modeling. The widespread use of object-oriented development languages (Java, C++, C#) and platforms (J2EE, COM+, .NET) in mainstream computing has increased the requirement for a semantically-rich modeling notation that supports component- and model-based development. The latest version of UML, UML 2.0, addresses the shortcomings of previous versions by providing additional support for expressing system structure, constraints and behavior. These latest enhancements also address the limitations of previous versions of UML in describing real-time distributed systems.

VEHICLE SYSTEMS SPECIFICATION

The requirements specifications for today's vehicle systems are usually expressed in natural language, perhaps supplemented with some informal diagrams such as state transition diagrams and flowcharts. The problem with these documentation methods is that they can be ambiguous, inconsistent and incomplete. When modules of a system are being developed by different organizations, much development and testing time is wasted trying to resolve differences in understanding of what the system requirements are.

A common approach to the management of complexity has traditionally been standardization, the use of interchangeable parts that provide compatible interfaces and services. The use of standard modules and protocols also helps the requirements specification effort because standard implementations also imply a commonly understood reusable specification.

There has been a move to model-based development in recent times which has brought with it a range of diagramming and modeling notations that developers can use instead of natural language specifications. To be effective however, the modeling notations used must have well-defined semantics and be more than just an informal diagramming notation. UML is a well-defined modeling notation that can be used to document system requirements.

UML SYSTEM VIEWS

UML currently has over a dozen different diagrams to model various aspects of a system. Fortunately not all of them need to be used in a development project. Developers are free to select the models that best express the important aspects of their system. While each diagram expresses either a behavioral or structural aspect of the system, it is useful to organize the models into a 4+1 view of the system, that is, there are four distinct views of a system, plus one view that describes how the overall system fits together. These views are shown in Figure 1.

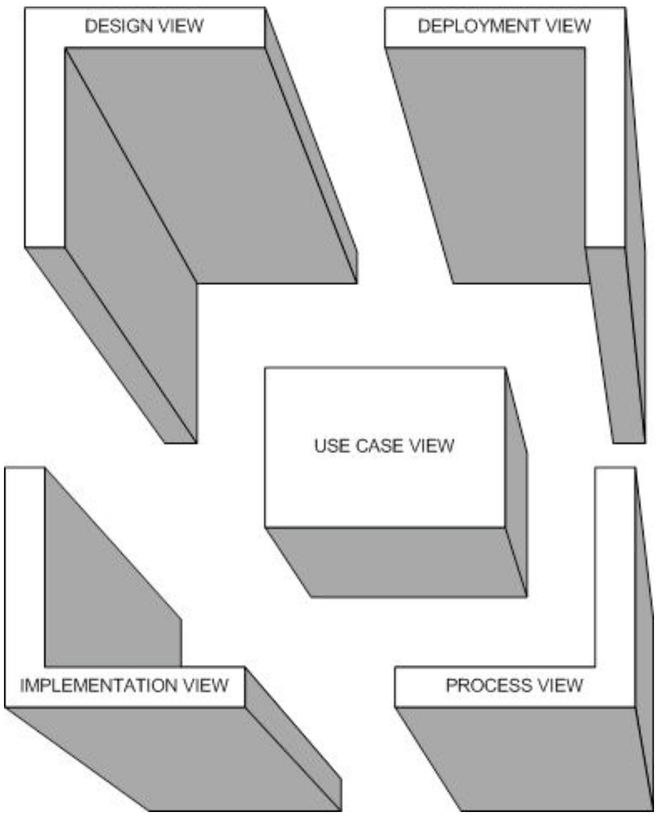

Figure 1. System Views

USE CASE VIEW

In UML the overall system functionality required by the users is described by a Use Case diagram. Use Cases are pieces of functionality that are invoked by Actors, which can be end-users, or in the case of real-time systems, external sensors or other system components. Use Cases can have dependencies on other Use Cases, allowing functions to be reused. Use Cases can also be special cases of other Use Cases using generalization or inheritance. This allows standard functions to be tailored to suit special requirements. Figure 2 shows a Use Case diagram for a power window system.

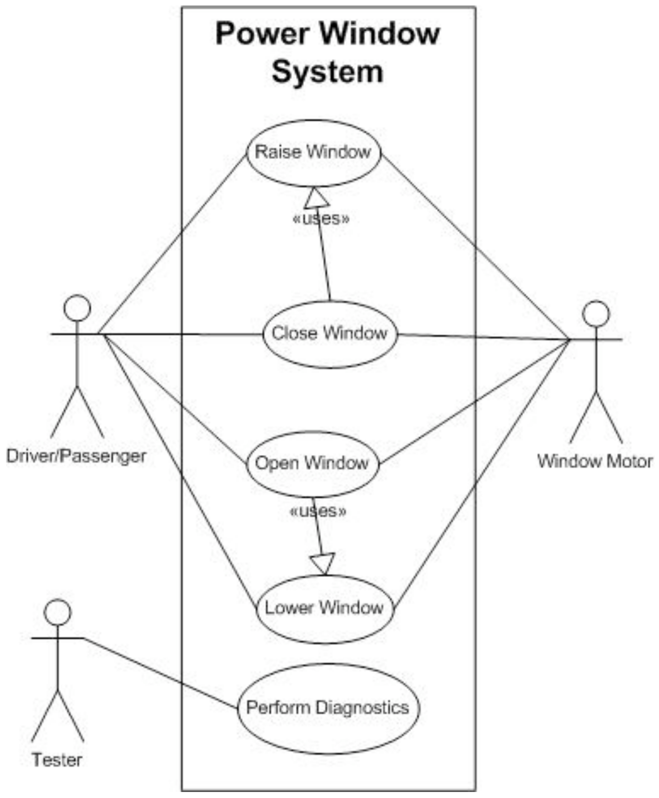

Figure 2. Use Case Diagram

Use Case requirements can be expressed in any suitable format. Traditionally this has been natural language, although such requirements can be incomplete, inconsistent and ambiguous. The availability of standard templates and requirements management tools has eased some of these problems. UML 2.0 also allows Use Cases to be expressed in terms of other UML models, most commonly Activity diagrams and Sequence diagrams. This hierarchical view of system requirements supports requirements traceability and automated consistency checking.

DESIGN VIEW

The design view describes the software elements and their interactions necessary to realize the functionality of the system. The design view provides the logical structure and behavior of the software without addressing how the software elements are distributed across the hardware platform. The UML diagrams that support the design view are

- Class diagrams
- Object Diagrams
- Activity Diagrams

- Composite Structure Diagrams
- Sequence Diagrams

DEPLOYMENT VIEW

The deployment view captures the details of how a system is configured and installed on physical hardware architectures. It can illustrate aspects of the system architecture such as network configuration and redundancy. The deployment view can be illustrated with the following UML diagrams

- Component diagrams
- Deployment diagrams
- Interaction Diagrams

IMPLEMENTATION VIEW

The implementation view is concerned with the configuration management of the system. It shows which source files implement which classes and the dependencies between system components. Implementation views can be expressed using the following diagrams

- Component diagrams
- Interaction diagrams
- Composite Structure diagrams
- Statechart diagrams
- Package diagrams

PROCESS VIEW

The process view illustrates the concurrency among the software elements and can be used to express performance and scalability requirements. The process view is expressed with the following UML diagrams

- Interaction diagrams
- Activity diagrams
- Timing diagrams

USING UML

As can be seen, individual UML diagrams can serve many purposes, supporting different views of a system. Many developers tend to use just a few UML diagrams to provide rough sketches of the system functionality. The Class diagram, Interaction diagram and Package diagrams tend to be the most commonly used models in this regard. To get the maximum benefit from UML, it needs to be regarded as more than just a diagramming notation. The semantics of UML 2.0 have been strengthened so that UML can now be considered a development language. New extensions to UML and tool support means that it is possible to create executable models of a system using UML. This allows for earlier verification of system functionality and support for model-based development at higher levels of abstraction.

In the latest version of UML, Each system requires a Use Case View that expresses the required system functionality. Once the requirements have been determined, the Design View is used to describe the fundamental components of the system and how they work together to realize the functionality of the system.

The developer can decide to use one or more of the other system views to highlight important requirements and design decisions. Non-functional requirements, for example performance constraints and software deployment, are sometimes expressed in natural language, but could be more clearly described using the Implementation, Deployment and Process view diagrams. With tool support it is possible to automatically verify these non-functional requirements when expressed as UML models.

UML FUNDAMENTALS

This section gives a brief overview of the main UML diagrams. The intention is to give the reader an idea of what is possible with UML. The reader is referred to the UML standard [ref] and some of the many UML books [ref] for a thorough treatment of the subject.

CLASS DIAGRAM

The Class diagram is the most fundamental of all the UML diagrams. A Class diagram describes the basic software elements that make up a system and the static relationships between them. A Class represents a set of things that have a common set of data (called attributes) and behavior (called methods). For example, in Figure 3 the class "Car" represents the general set of all cars. Its attributes might include details such as Make, Model, Model Year, Owner and Color. Its methods might include getColor(), getVIN() and changeColor(). Even though UML is fundamentally object-oriented, it can be used to model non-object-oriented (procedural) systems. The main thing to remember is that a Class can represent any important system concept, not just an object-oriented class in Java or C++. Classes can represent software components which encapsulate their own data (attributes) and behavior (methods). The Class diagram specifies the *interface* to the class and not its *implementation*. The class could be implemented in a procedural language such as C or even as a Simulink block. This flexibility in the interpretation of classes allows class diagrams to be used to describe both the system requirements in terms of domain concepts, as well as the subsequent realization of the requirements in terms of software elements. Developers distinguish between Analysis Class Diagrams, which contain only domain-level concepts, and Design Class Diagrams, which additionally contain software-specific classes to describe concepts such as data structures and data access protocols.

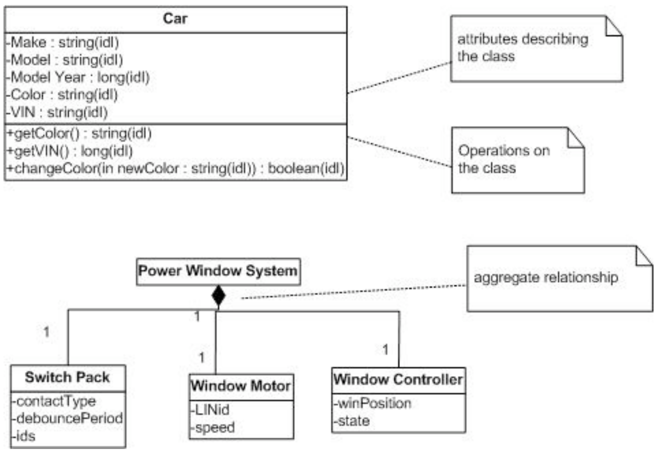

Figure 3. Class Diagram

Figure 3 illustrates the relationships between the class Car and some of the other system concepts. UML provides a shorthand notation for commonly used aggregate (assembly-subassembly) and inheritance (generalization-specialization) relationships.

During the requirements analysis phase the emphasis is on identifying the main system concepts/classes and the relationships between them. At the software design stage methods are assigned to each class to represent the responsibilities of the class in terms of the functionality that it provides. It is best to leave the assignment of methods to the design phase, since discovering a good, stable structure of system elements is the most important factor in architectural design. The assignment of responsibilities to the system elements tends to fall naturally out of a good system structure.

Class diagrams illustrate the general structure of the system in terms of similar software elements. They are static diagrams and so they do not show the run-time behavior of the software elements. At run time the classes are instantiated as individual objects which interact by calling each other's methods to execute system functionality.

SEQUENCE DIAGRAM

Sequence diagrams are used to show the order of method calls made by objects of each class. A particular class may be responsible for providing certain functionality, but sometimes it must delegate parts of that functionality to other classes in the system, in the same way that managers in an organization delegate certain tasks to subordinates. Objects of a class call the methods of other objects to invoke their behavior. Sequence diagrams chart chronologically, from top to bottom, the method calls required to implement a piece of functionality. Messages in a sequence diagram can be either synchronous (with solid arrow) or asynchronous (with open-ended arrow). The basic format of a Sequence diagram is shown in Figure 4.

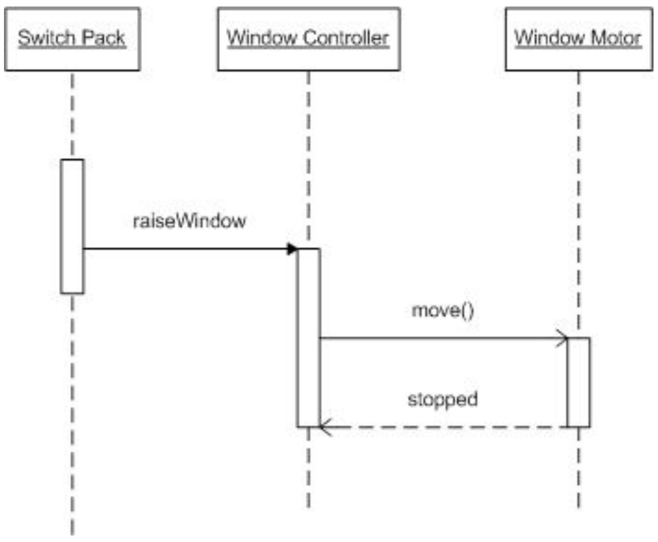

Figure 4. Sequence Diagram

STATECHART DIAGRAM

Statechart or State Transition diagrams capture the internal state transitions of UML elements such as a Class, subsystem or the whole system. They are very useful in describing the operation of event-driven real-time embedded systems. The Statechart notation has been enhanced in UML 2.0 to allow for easier expression of hierarchical state machines, reusable states, composite states for parallel processing and high-level transitions. In addition there is some new notation for emphasizing transitions and associated input/output signals that would be very useful in ECU modeling. Figure 5 provides an example of a Statechart showing some of the new features.

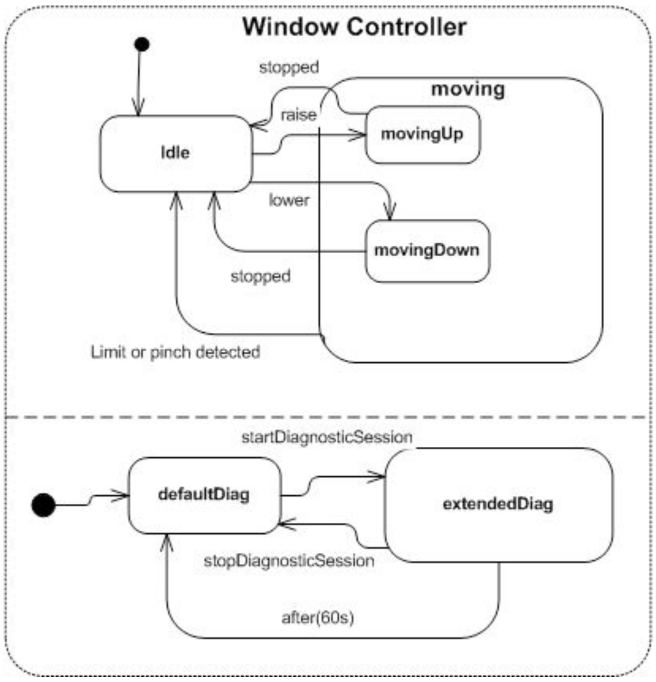

Figure 5. Statechart Diagram

COMPOSITE STRUCTURE DIAGRAM

Modern vehicle control systems are usually implemented by a combination of cooperating hardware and software components. Each component provides a well-defined interface and functionality to client components. Components are a fundamental method of reuse, whether they are hardware components, software modules or Java/C++ classes. UML now has a Composite Structure diagram that provides both black-box and white-box views of a component and its interfaces. The white-box view allows the implementation of a component to be specified in terms of other basic components, providing a hierarchical resolution of a system or component.

Components can have both *provided interfaces* and *required interfaces*. Provided interfaces represent the functionality implemented by the component in question. Required interfaces represent functionality that the component expects other components in turn to supply to it so that it can carry out its activities. UML provides the concepts of ports and connectors to enable components to be interconnected at interfaces. Figure 6 is an example of a black-box component showing both provided and required interfaces.

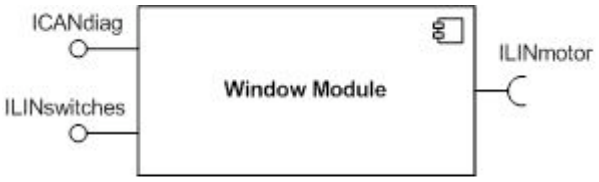

Figure 6. Black-Box Component View

The white-box view of a component can be used to show the classes that work together to provide the functionality at the component interfaces. Ports can be connected to internal classes to show which classes provide the functionality at component interfaces. In this way the end-to-end flows through system components can be documented. An example of a white-box component realization is shown in Figure 7.

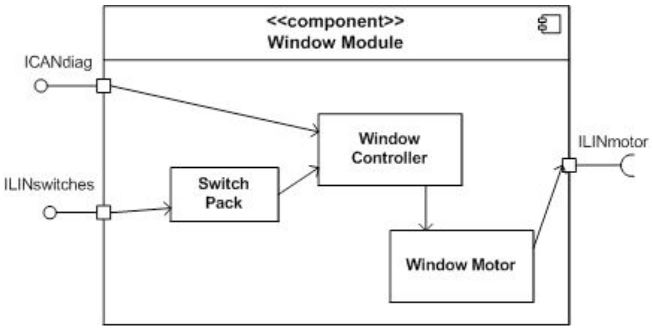

Figure 7. White-Box Component View

Classes can also be broken down in the same way to show how internal functionality is achieved using other delegate classes. These Composite Classes are new to UML 2.0 and provide both a hierarchical structuring of classes and an illustration of the end-to-end flows within a class.

ACTIVITY DIAGRAM

Sometimes the best way of describing a piece of system functionality is by using a procedural approach; a sequence of steps, much like traditional flowcharts. UML now has an Activity diagram that uses a mixture of control and data flow notation to describe system behavior. There is support for decision-making, concurrency and synchronization modeling using Activity diagrams. Figure 8 is an example of an Activity diagram.

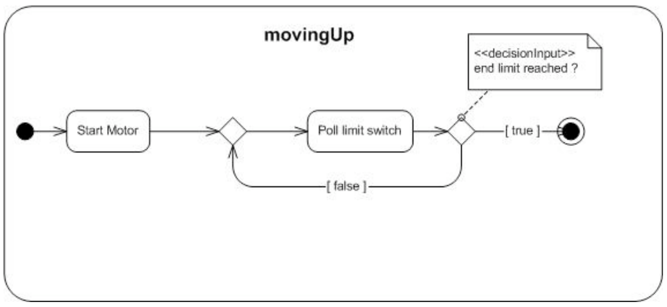

Figure 8. Activity Diagram

REAL-TIME SYSTEM MODELING

Real-time system development has to address the following concerns as part of the analysis and design phases.

- System deployment to distributed hardware and software components
- Concurrency
- Constraints on system resources
- Performance requirements

The enhancements found in UML 2.0 provide good support for the above concerns, making UML more suitable than ever for real-time system modeling. The gap between systems engineering and software engineering has been bridged with UML 2.0, providing a single modeling technique that can be used to model system architecture and subsequently refine this to software implementation models of each system component.

FUNCTION DEPLOYMENT

Composite Structure diagrams can be used to break down the system into a set of cooperating hardware and software components with clearly defined interfaces. Deployment diagrams can be used to show the allocation of software components to networked Electronic Control Units (ECUs) together with the network topology details. An example of a Deployment diagram is shown in Figure 9.

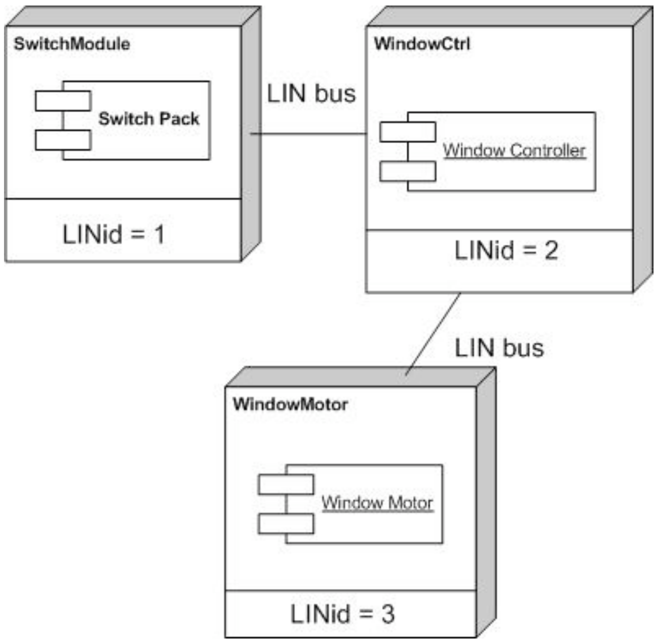

Figure 9. Deployment Diagram

CONCURRENCY

In addition to the concurrency implied in the Deployment diagrams, UML can represent parallel activities using composite states on Statecharts as shown in Figure 5.

An enhancement to Sequence diagrams allows for sets of method calls to be executed in parallel as shown in Figure 10.

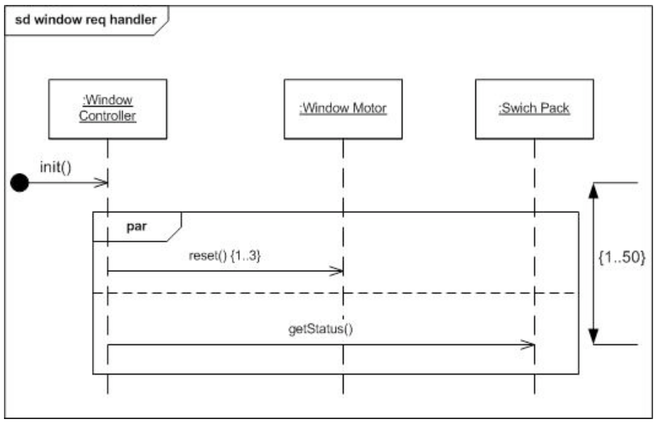

Figure 10. Enhanced Sequence Diagram

UML 2.0 allows portions of a timeline to be broken down, providing a hierarchical organization of sequence diagrams. This solves the problem with previous UML sequence diagrams which did not scale very well for use in large complex systems. UML now supports *Interaction Occurrences* on Sequence diagrams which allow common sequences of messages to be reused in many different Sequence diagrams.

CONSTRAINT MODELING

The Object Constraint Language is a declarative language that can be used to express constraints or invariants on any aspect of a UML model. OCL has its own set of keywords and operators, just like any other language. A developer can use OCL to express constraints on such things as the allowed range of values for a class attribute, a pre-condition for executing an action on a Statechart or allowed network bit rates on a deployment diagram. There are considerable benefits from using OCL to specify non-functional requirements as part of a Use Case. For a start, OCL expressions are unambiguous and can also be automatically checked for consistency with the help of some tools.

PERFORMANCE MODELING

There are a number of ways in which the performance aspects of a system can be specified in UML models. The Statechart notation includes the transition keyword *after,* which specifies the maximum allowed time before a transition will occur. This is useful for describing timeout conditions in state machines. An example of this is shown in Figure 5. The Sequence diagram can also be annotated to show the maximum allowable execution times for a method, as well as the time allowed for the method call itself. Examples of these notations are given in Figure 11.

UML has a Timing diagram that is used to show the effects of method calls on objects as time progresses. The Timing diagram can be used to show timing constraints on method calls and the sequencing of method calls. Figure 11 is an example of a Timing diagram, showing that the window should begin moving up within 10 milliseconds of receiving a raise command.

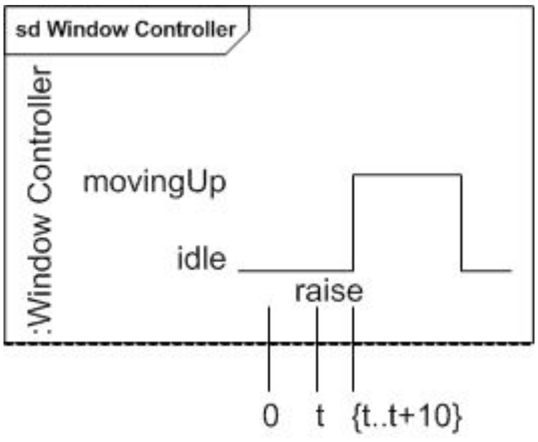

Figure 11. Timing Diagram

SYSTEM INTEGRATION SUPPORT

The effects of poor system requirements definition become apparent at the system test and integration phase. In the global automotive industry, where many cooperating system components are developed by geographically dispersed teams, the effects of poor interface definition are exacerbated. With UML 2.0 there is now a widely-used modeling notation that is comprehensive enough to span both the system modeling and software modeling domains. The use of a common notation by both system developers and software designers allows models from both domains to be synchronized earlier in the development process to avoid any major integration problems later on. The well-defined semantics of UML 2.0 together with the use of OCL provide better tool support for activities such as architecture validation and test case generation.

EXECUTABLE CODE SUPPORT

UML 2.0 is an extensible modeling environment. UML *profiles* can be used to extend the notation to support domain-specific concepts and implementation platforms. For example, an OSEK profile can be defined which is essentially a Class diagram describing OSEK concepts and the relationships between them. The profile defines a set of *stereotypes* which are the names of the OSEK concepts or classes and *tagged values*, which are used at design time to influence the configuration of stereotypes. When a developer is describing an automotive application to be implemented on OSEK the application classes can be further labeled with stereotypes (roles) such as <<Task>>, <<Alarm>> or <<Message>> instead of the generic label *class*. The purpose of the stereotypes is to give additional information to readers about the intended use of a class. However, tools such as code generators can also use the stereotypes to guide the generation of code and configuration details for the software element. A class with a <<OSEKtask>> stereotype is shown in Figure 12.

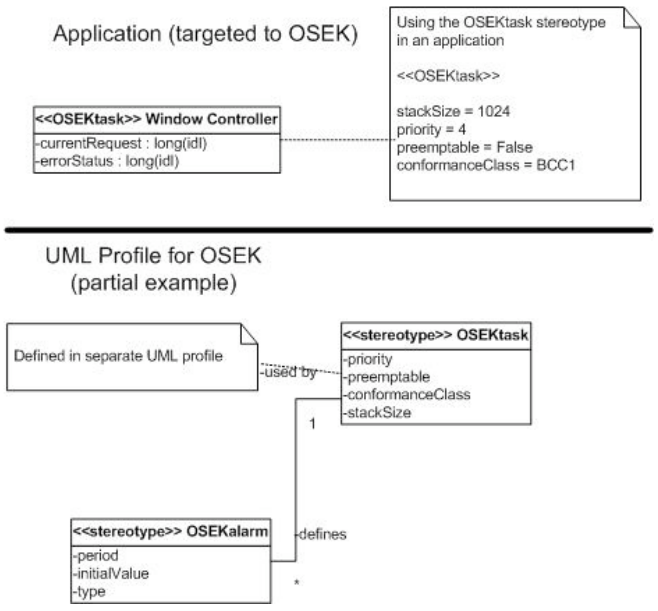

Figure 12 OSEK Task Stereotype

By adorning the UML models with appropriate stereotypes to indicate the intended implementation and defining suitable implementation profiles it is possible to automatically generate executable code from the UML models.

EXAMPLE – NETWORK MANAGEMENT SPECIFICATION

The Gateway module which is normally the electronic cluster has the basic task of transferring normal messages and generating pseudo gateway messages. The normal messages are transferred from one network to the other. The pseudo messages are generated based on internal information within the electronic cluster.

There are a number of network dependencies associated with gateways such as:

- management of the network when one bus is asleep and another is awake, therefore deciding what messages are transferred;

- Management of the network rings when both buses are asleep and how the rings wake up and are established once again. In all the above situations the gateway has to ensure there is no negative impact on both buses in terms of invalid faults such as missing messages and entering false limp home modes.

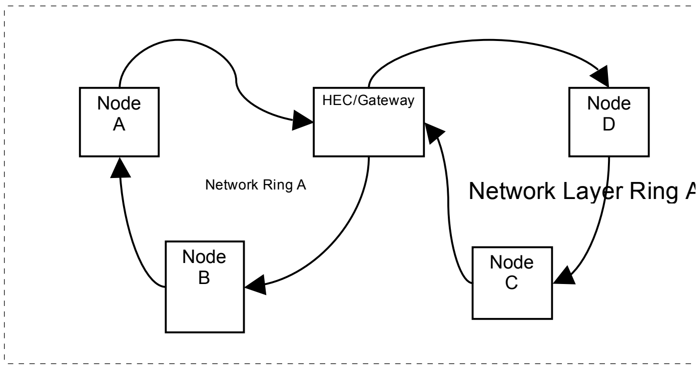

Figure 12. Example Logical Ring Architecture for Two Buses

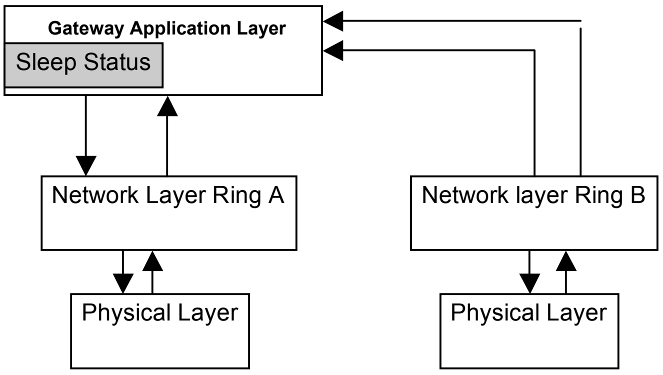

Figure 13. A Gateway module trying to match two networks into sleep mode

The problem of matching both network into a particular network state can best described using UML especially since the OSEK documentation is too detailed and cumbersome for the ordinary system engineer to understand properly.

The OSEK/VDX network management concept and application programming interface documentation does not show how to link two buses properly so that when one goes to sleep the other will do the same via sleep indication bits within network management messages. Using UML Statecharts and Sequence Diagrams to describe these scenarios would allow network management component developers and integrators to better understand and correctly implement this functionality.

CONCLUSION

This paper presented an overview of UML notation with an emphasis on new UML 2.0 features to support the modeling of real-time embedded systems such as those found in automotive applications. UML has developed into a comprehensive system and software modeling language that is semantically rigorous enough to provide tool support for development activities such as architecture validation, model-driven development and automatic code generation. A significant development has been the expansion of UML notation to support better integration of system modeling and software modeling.

REFERENCES

1. www.omg.org
2. UML 2.0 in a Nutshell. Dan Pilone. O'Reilly Media, 2005.
3. Applying UML and Patterns. Craig Larman. Prentice Hall PTR, 1998.

CONTACT

Brendan Jackman B.Sc. M.Tech.

Brendan is the founder and Leader of the Automotive Control Group at Waterford Institute of Technology, where he supervises postgraduate students working on automotive software development, diagnostics and vehicle networking research.. Brendan also lectures in Automotive Software Development to undergraduates on the B.Sc. in Applied Computing Degree at Waterford Institute of Technology. Brendan has extensive experience in the implementation of real-time control systems, having worked previously with Digital Equipment Corporation, Ireland and Logica BV in The Netherlands.

Email: bjackman@wit.ie

Website: http://www.wit.ie/automotive

Shepherd Sanyanga (BEng BSc MSc PhD CEng EurIng MIEE)

Shepherd has worked for United Nations in Africa in infrastructure development programs, worked for some years in the Aerospace Industry developing military electronic systems and then worked for a number of Tier One Automotive Suppliers (Lucas Electronics (UK), Sagem (France) TRW (UK). He is currently involved in a joint project between Ford Motor Company (Europe), Johnson Controls (France) and Takosan (Turkey). In his spare time he is also an external examiner on a masters degree automotive programme in a university in Ireland.

Email: ssanyan2@ford.com