Extending UML to Enable the Definition and Design of Real-Time Embedded Systems

Alan Moore
ARTiSAN Software Tools

The complexity and size of the average real-time system is increasing rapidly. The development of detailed system designs is vital if the system is to be well understood and to function correctly. In a bid to extract maximum reuse of system software components, and to overcome previous bad software development experiences, many senior system and software engineers are looking to Object Orientation (OO) as the design paradigm for new system development. This article discusses the suitability of Universal Markup Language, the popular OO modeling language, for defining and designing real-time and embedded systems.

Object Orientation (OO) has been successfully applied to the development of commercial software, but has the potential to be even more readily embraced by engineers in the real-time domain, where familiarity with the componentized nature of hardware leads naturally into the concept of objects.

The search is on for appropriate real-time OO modeling techniques, and not surprisingly the Unified Modeling Language (UML) is the notation most often proposed as the base for the techniques. The rise in UML's popularity has been rapid since its inception some 4 years ago. The fact that it is promoted by some of the biggest 'names' in the OO arena and supported by an increasing number of tool vendors goes some way to explain why.

Although it is a well-defined and flexible modeling language its origins in the world of commercial computing show up in the lack of certain modeling techniques for real-time systems and embedded systems. For most (if not all) real-time systems, the issues of timeliness and ability to schedule are paramount. By definition, real-time systems designers are concerned with time and ensuring that certain system activities can be carried out within defined time scales. This in turn requires consideration of the system's concurrency: The number and the characteristics of the concurrent tasks and the degree of task interaction between them can significantly affect system performance. Embedded systems designers are also concerned with the physical architecture of the embedded system and the hardware/software interface. None of these concerns are well met by UML at present.

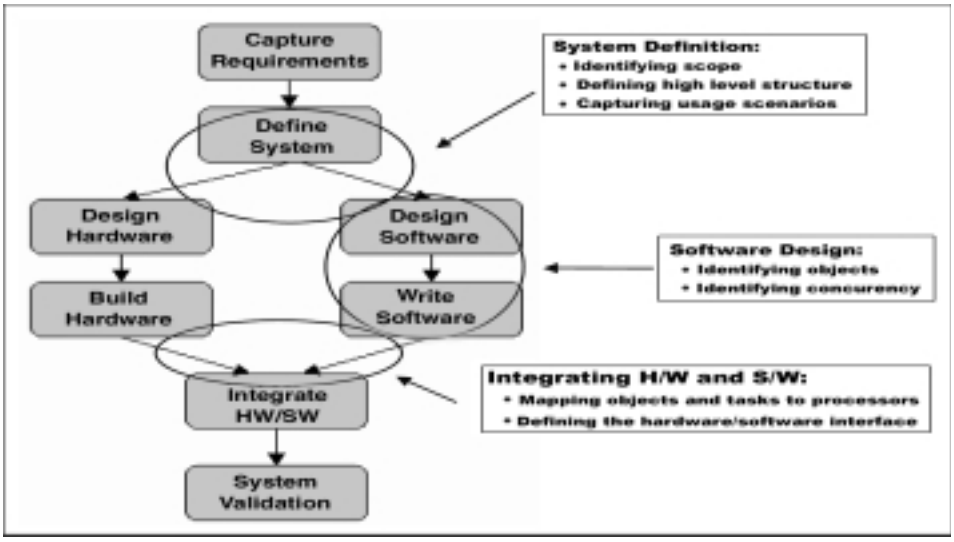
Recognizing this, the Object Management Group[1] has spawned a new initiative, the Real-time Analysis and Design Group (RTAD), specifically to recommend UML extensions in this area.

The System Development Process

The design of real-time embedded systems involves a multi-disciplined team of hardware, systems, and software engineers. System engineers design at a high level and influence both hardware and software composition of the system. They are concerned with the overall architecture and usually make trade-offs between implementing functionality in hardware, software, or both. Hardware engineers are concerned with the design of circuitry that will fulfill the system requirements as determined by the systems engineers. Software engineers usually have the largest design and implementation task, as the majority of the functionality of an embedded system lies typically within the software.

This (simplified) system development process illustrated in Figure 1, shows three critical areas of developing real-time/embedded systems: system definition, software design and hardware/software integration. Those areas are circled with some of the key activities outlined in the associated call-outs.

Figure 1: *The System Development Process*



System Definition

The process of defining a system is often called system engineering. It involves the following:

- Identify the structure of the system, both in terms of hardware and software. For many real-time systems (e.g. mass produced embedded systems) major architectural decisions are often made early in the development life cycle. Almost all real-time developments will experience change in the system architecture during the development process. The nature and implications of the architecture, and any changes to it, need to be communicated among, and understood by, system, hardware and software engineers.
- Clarify communication between the system and the environment. Clarifying both the boundary of the system and

the types of messages exchanged is important both from the point of view of scoping the system and agreeing interfaces.

- Capture system usage and therein timing and other types of system performance (often called quality of service) associated with that use. System usage requirements often include specific timing information relevant to product performance. These timing constraints are captured and decomposed during the system definition stage in order to facilitate a system design that is fit for purpose, and to support system test. Real-time systems, by definition, are constrained by some aspect of time. The right answer received late is often wrong. The requirements driving the development of these systems often state specific timing considerations. It is critical to capture this timing information in the early system definition stage.

Software Design

As well as the standard need to model objects and their interaction, real-time system models also need to address concurrency. There is an inherent concurrency in most real-time, embedded systems as they control multiple input/output devices through a variety of interfaces simultaneously. Designing a solution for this type of problem often involves a multi-tasking, and possibly multi-processor architecture. The task and object design need to be integrated, so the ability to model both together is essential.

Hardware and Software Interfaces

Real-time embedded systems contain both hardware and software elements as part of the solution. Many of the worst and most time-consuming problems with developing real-time systems manifest themselves during system integration, often through poor interface specifications. Therefore, understanding the interfaces and interdependencies between the hardware and software is a necessary step in building a correct system. Here are examples of the types of data required:

- Software/hardware interface information, e.g. port addresses, memory maps.
- Hardware characteristics such as speeds, capacities, etc.

An Example

Throughout the remainder of this article an example of a real-time system will be used to present the basics of the UML, as well as to illustrate the extensions required to usefully define and design a solution to a typical problem.

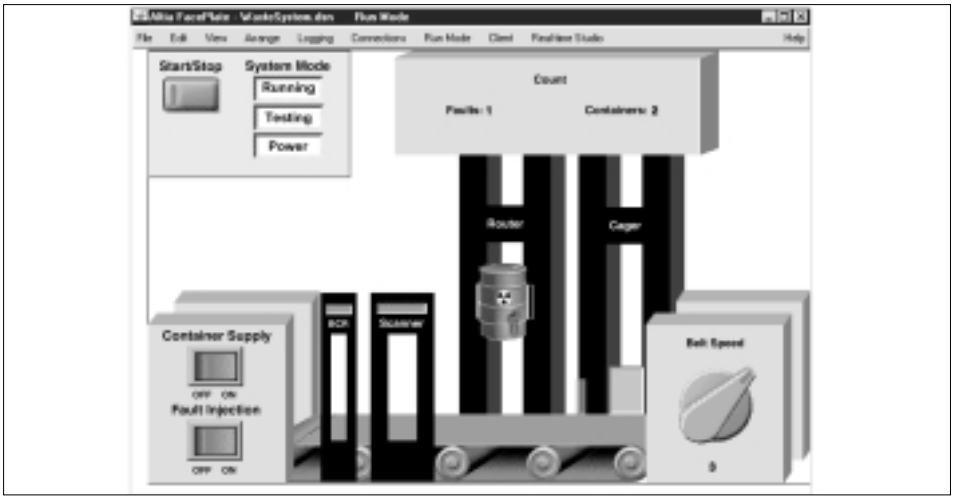


Figure 2: *Toxic Waste Processing Plant Simulation*

Figure 2 shows a simulation of a toxic waste processing plant. The plant handles containers of toxic waste, which are fed into a processing plant along a conveyor belt. Once detected and identified by an attached bar code, a container is scanned for faults before routing. If a container is badly damaged then it is routed off the belt by the first robot arm for special handling: If not, then it reaches the second robot arm where it is sent off for caging and normal processing. This simulation also provides diagnostic switches to supply containers and inject faults.

The Advantages and Deficiencies of UML

Here the three areas described in the system development process are revisited, highlighting how UML helps model these areas, and also where it is deficient. In those cases, additions or changes to UML are suggested that address those deficiencies.

System Definition

System Context (or Scope)

One simple and often overlooked consideration when mapping out the parts of the solution is the external interface boundary. This concept is useful in determining what is within the context of the solution and what is outside this context. UML has no good notation for supporting this; engineers often use an object collaboration diagram, but this involves faking system-level devices as objects, which can cause confusion when *real* objects are introduced.

A System Scope Diagram, an extension to UML, can be used early in the problem analysis to help isolate the boundary of the analysis activity. It provides a graphical checklist of all of the external and interface boundaries in the solution. It is an easy way to make sure that the complete problem is being addressed. It shows external actors, hardware interface devices, and the system itself, as well as the messages (often called signals) that pass between them. One of its most important roles is to define a vocabulary for describing how the system interacts with the environment when it's being used. It is very similar to the context diagram used in traditional strutured techniques. (Sadly the UML Deployment Diagram does not allow actors or message flows and so it is hard to use for this purpose.)

The System Scope Diagram in Figure 3 (see page 6) shows how all the major components of the system (hardware and software) communicate. The plant control system itself is started and stopped by the front panel subsystem, and controls the plant subsystem. Each of the messages (or signals) has a rich description as exemplified by *frame*; frames occur only while a container passes through the scanner, hence they are episodic, and occur in bursts of four with a minimal interval of 75ms.

System Architecture

Having scoped the system, a major task is to identify the high-level architecture of the system, adding major architectural elements such as subsystems, buses and disks,

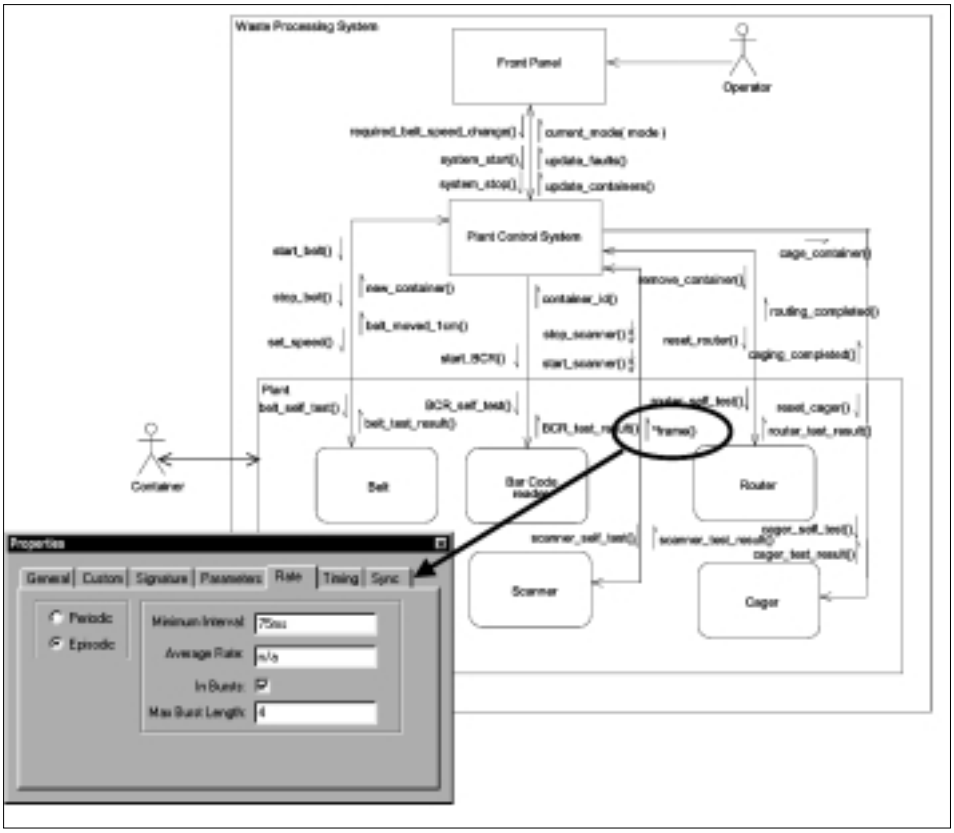


Figure 3: *The System Scope Diagram*

decomposing subsystems into boards and connecting them up appropriately. The UML has the Deployment Diagram for this, but it is woefully lacking in notation for buses, boards, etc., and so cannot be used to present an easily comprehensible picture to the engineering team developing an embedded system.

The System Architecture Diagram presented in Figure 4, another extension of the UML, is a form of Deployment Diagram, but with greatly enhanced symbolism[2], and an underlying type model allowing board, bus, disk etc., types to be defined once and subsequently reused.

The plant control system is a rack-mounted PC with three boards: the motherboard, a network card, and an I/O board. The network card is there to facilitate communication with the factory bus. The I/O Board will be off-the-shelf but will need enough ports to handle the various devices attached to it, including the Front Panel devices (not shown).

Use Cases

Use case modeling is the primary UML technique for specifying functional requirements, and one of the most widely accepted and used. A Use Case Diagram illustrates the ways that elements external to the system (actors) interact with the system under development. Actors may be people, for example end users, operators, maintenance engineers, etc., or they may be other systems or items of equipment. All actors are represented diagrammatically by the stick person symbol. Lines connect the actors to one or more use cases (the ellipses). A use case represents a system service provided to one or more of the actors linked to it. Associated with each use case is a textual description of the activity involved in the delivery of that service.

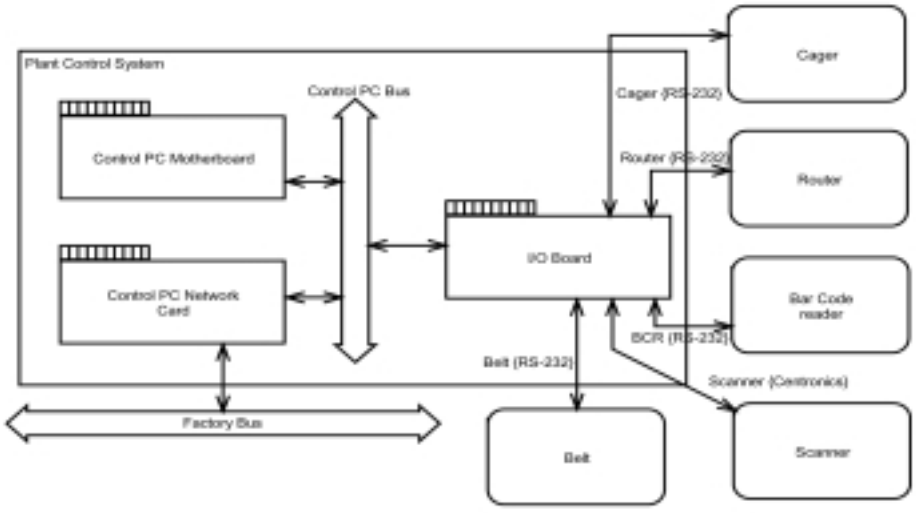
Figure 5 shows a Use Case Diagram, identifying the major functional requirements of the system. The major use case, process container, makes use of two lesser use cases, scan container and detect container, hence the <<include>> relationship. There are two exceptional cases highlighted by the <<extends>> relationship, handle defective container and emergency stop and restart which describe deviations from the standard behaviour of process container.

Use case descriptions provide a specification of functional requirements in a form that is understandable to, and therefore can be validated with, the system sponsor. However the main drawbacks of using natural language for specification (ambiguity, misconception, duplication, etc.) may be present, although not necessarily visible. UML provides a modeling technique that helps in overcoming these drawbacks: the Sequence Diagram.

Sequence Diagrams

The Sequence Diagram presents a usage scenario in a useful and intuitive way, as an ordered sequence of message exchanges (or interactions) between system elements. Although UML provides the capability for interaction modeling with Sequence Diagrams, it does not make it easy to distinguish between the different types of elements involved. The implicit assumption in UML is that this diagram exists to illustrate message passing between the actors and software objects. This is jumping the gun during system definition. It is certainly the case that, during the design stage, we will need to model object interaction.

Figure 4: *The System Architecture*



But at the requirements stage we ideally want to express the details of use case activity by clearly indicating the interaction between the actors, hardware devices, and the entire software system; these are likely to be far more familiar to the project sponsor.

The Scanning a Container diagram in Figure 6 is an extension to UML. As can been seen in the diagram the plant control system is informed of belt movements (asynchronously, indicated by the half-arrow). In this scenario it determines that a container is approaching the scanner, whereupon it starts the scanner, receives four frames of scan data from it, and then stops the scanner again. The system waits for the scanner to both start and stop before proceeding (indicated by the cross on the arrow shaft). The frame takes 50ms to capture and 25ms to transmit from the scanner. This gives a total scan time of 375ms. Note that all of the elements shown here are derived from the earlier System Scope Diagram, Figure 3.

Altough the UML allows annotations to be added to any diagram, it is more useful if timing notes, in particular, are mapped directly onto the appropriate model elements (as tagged values), making timing information more than just notes on the Sequence Diagram. There are two main categories of timing information that can be modeled, latency and duration. The amount of time it takes for two entities to collaborate (i.e., communicate) can be described as the message latency. The amount of time required for an entity to perform its task once it has received a signal is described as the duration. Through a combination of latency and duration specifications, we identify general timing constraints over a sequence of steps in a usage scenario (as an end-to-end time). These constraints map back onto specific, use cases and the original requirements.

Software Design

Class Diagrams

The UML Class Diagram is perhaps the best-known aspect of UML. Class Diagrams form the basis for the object architecture of the system. They document the static nature or architecture of the objects in the system. The Class Diagram is concerned with static structure and designing good partitioning (encapsulation) and abstraction. It works well as is for real-time systems design.

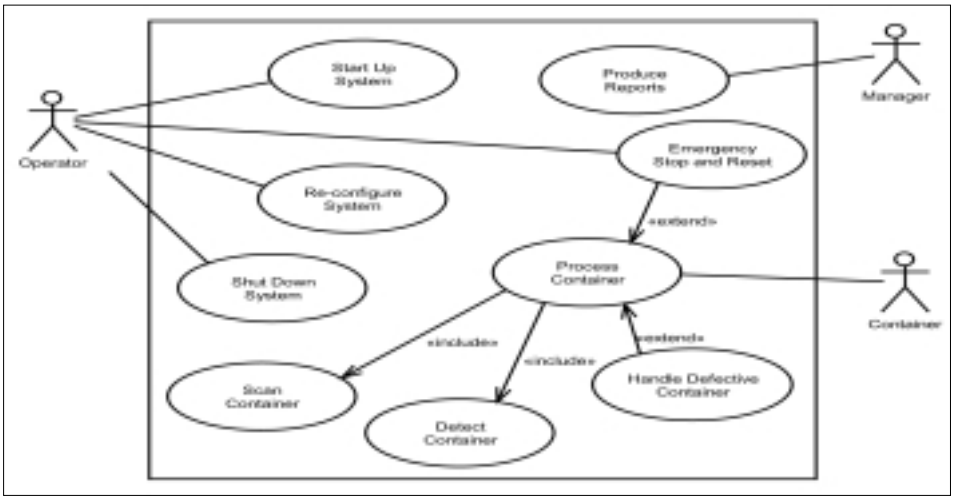


Figure 5: *Use Case Diagram*

Collaboration Diagrams

Where the Class Diagram defines a static relationship structure between the classes, the Collaboration Diagram defines a communication structure between the objects of those classes. Collaboration Diagrams can be used to detail specific scenarios (referred to as realization in UML) or as a means of synthesizing the overall collaboration of objects.

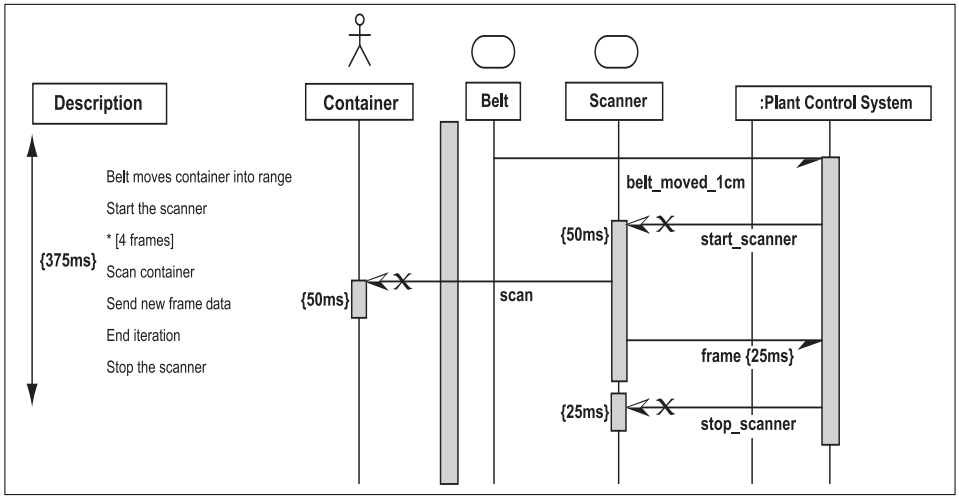
The UML Collaboration Diagram in Figure 7 (see page 8) shows the collaborating objects and the sequence of operation calls (via the sequence numbers) to scan a container. The object *theBelt* tracks all containers so is well placed to inform *theScanner* when one approaches. theScanner communicates through *theScannerDriver* to control the *Scanner* device and to receive frame data. Once all frames are processed the status of *currentContainer* is updated with the result. The messages between the drivers and interface devices provide continuity back to the earlier Sequence Diagram (see Figure 6).

By analyzing several Sequence Diagrams together as a whole and representing the overall connectivity of the objects involved in those scenarios, it is possible via a Collaboration Diagram to visualize the communication architecture of the proposed design. The Collaboration Diagram also gives some insight into the dynamic relationships between the objects. This allows us to create a more complete view of the overall communication architecture as opposed to any one specific scenario.

Concurrency Diagrams

What the Collaboration Diagram cannot describe is whether the executing objects are concurrent or not, or how they use a real-time operating system (RTOS) to communicate. Concurrency has been a concern in the real-time domain for many years and there are a variety of recognized strategies for identifying tasks and opti-

Figure 6: *Scanning a Container*



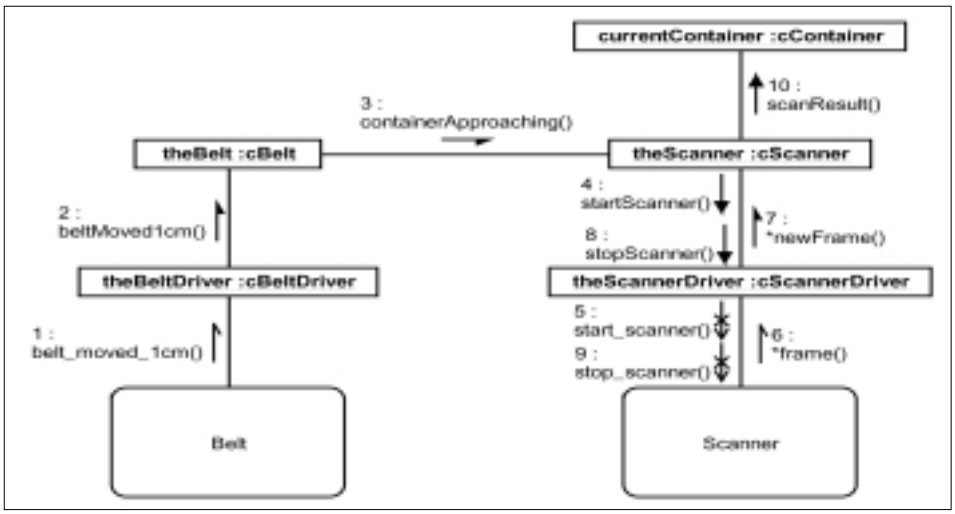


Figure 7: *The UML Collaboration Diagram*

mizing task design. In such a complex area of design, modeling can play a vital role in verifying the feasibility and correctness of a concurrency design. There is no current UML notation clearly representing tasks, or any of the RTOS mechanisms that can be used to protect shared resources or facilitate inter-task communication.

A diagram that shows the tasks, shared resources, mutexes, queues, event flags, mailboxes, etc., together with the way in which these various components interact, is required. The diagram will need to illustrate message passing in a clear and unambiguous manner so that it is clear exactly where and how the RTOS is being used. It is possible to produce a UML diagram that can represent the information shown in using standard notation Figure 8, but the UML does not have the richness of symbolism that makes this particular diagram so clear. This is not an unimportant point. Clarity is an aid to understanding; models that can be easily misinterpreted can, in some cases, be worse than no models at all.

We can now see how *cScanner* actually handles frames by looking at its internals. The identifiers of approaching containers (provided by the call to *containerApproaching*) get queued for processing by the task to scan containers. The arrival of new frame data (indicated via the call to *newFrame*) signals an event flag upon which a task (handle frame) is waiting. As frames are received (via some new calls to *theScannerDriver*), they are stored and their identities (frame ids) are passed to another task (process image) that processes the image for faults.

Hardware and Software Interfaces

Deployment Diagram Limitations

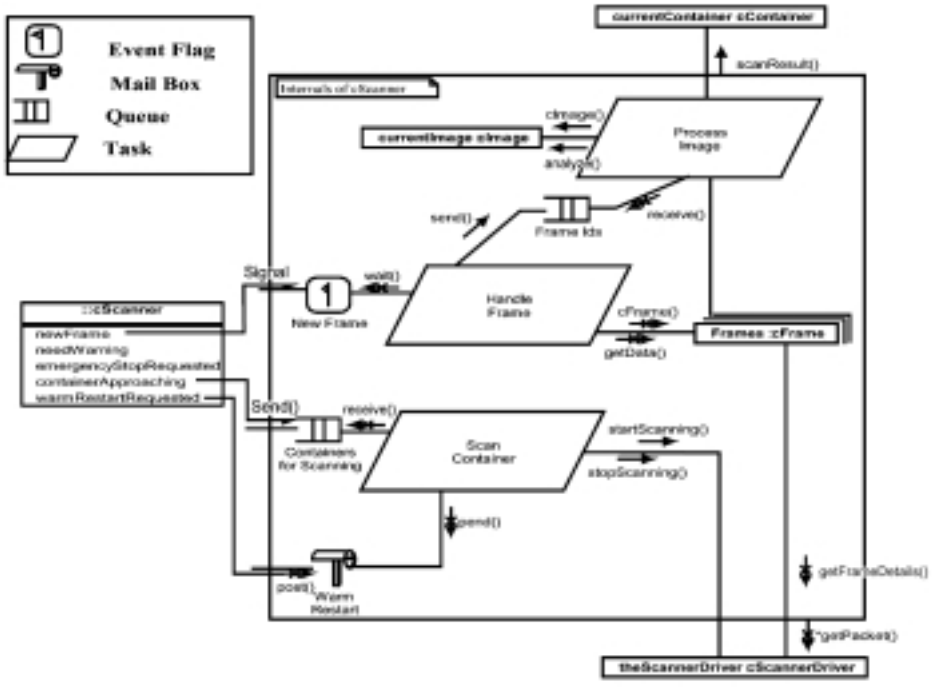
A thorough understanding and representation of the software/hardware interfaces and the allocation of software to hardware is important to construct an application that not only meets functional requirements but also can be deployed into the proper environment.

The UML provides a Deployment Diagram for capturing this kind of representation. It allows engineers to place objects (software) on a set of connected nodes (hardware). However, its description of the processing architecture is very simple, and it allows no account of the placement of tasks at all. It offers a very limited set of symbols that make it difficult to adequately represent the range and characteristics of the physical entities inherent in many real-time systems. Nor are Development Diagrams able to easily model interface information such as the memory map and interrupt request vectors.

In order to support this type of description, a much more detailed view of the board and processor architecture is needed with specific model elements to describe them. This type of information neatly fills the gap between traditional UML and the detailed hardware schema available from Electronic Data Access (EDA) tools. Hardware related information like this enables software engineers to undertake detailed device driver development, without the need to refer to hardware schematics generated by the hardware team. Figure 9 shows an Architecture Diagram as an extension to the UML Deployment Diagram. This diagram allows both hardware and software engineers to access the same information, significantly improving communication, and thus reducing the number of errors in integration testing.

Figure 9 shows the connections at the board level. Each of the board's I/O devices is further described to the detail required by the software, as is shown in the overlay. We can see the memory map as well as any IRQs or I/O port addresses.

Figure 8: *The Internals of cScanner*



Progress in the OMG

Recognizing the need for real-time extensions to UML, two Object Management Group (OMG) task forces, the Analysis and Design Task Force and the Real-time Task Force jointly spawned a specific group to look at extending UML to support real-time systems.

The Real-time Analysis and Design Group

The goal of the Real-Time Analysis and Design (RTAD) initiative is to issue a comprehensive set of Requests for Proposals (RFPs) leading to OMG standards that will support the use of object-oriented approaches in the analysis, design, and development of real-time computing systems.

There have been three RFP's proposed by the RTAD:

- UML™ Profile for Scheduling, Performance, and Time;
- UML™ Profile for Quality of Service (QoS) other than timeliness;
- UML™ Profile for large-scale, distributed systems.

The first of these has been formally issued by the OMG; its schedule shows the following:

- Initial submissions received in September 2000.
- Revised submissions due in July 2001.
- Approval by the Architecture Board of the OMG by October 2001.
- Specification adoption by January 2002.

The Submission Team

There looks likely to be only one team offering a submission for the initial RFP. The team consists of modeling tool vendors ARTiSAN Software Tools, I-Logix, ObjecTime (with Rational) and Telelogic, and schedulability of tool vendors Tri-Pacific Software and TimeSys.

Conclusion

Given the popularity and notational robustness of the current version of UML, real-time developers can reasonably begin exploiting OO technology in their efforts. However, it is important to realize that certain characteristics of real-time systems may be difficult, if not impossible, to capture in most modeling tools limited to standard UML notation and semantics. Real-time development demands extensions to UML provided only by specialized tools. This article proposes some feasible extensions to UML shown in Table 1. ◆

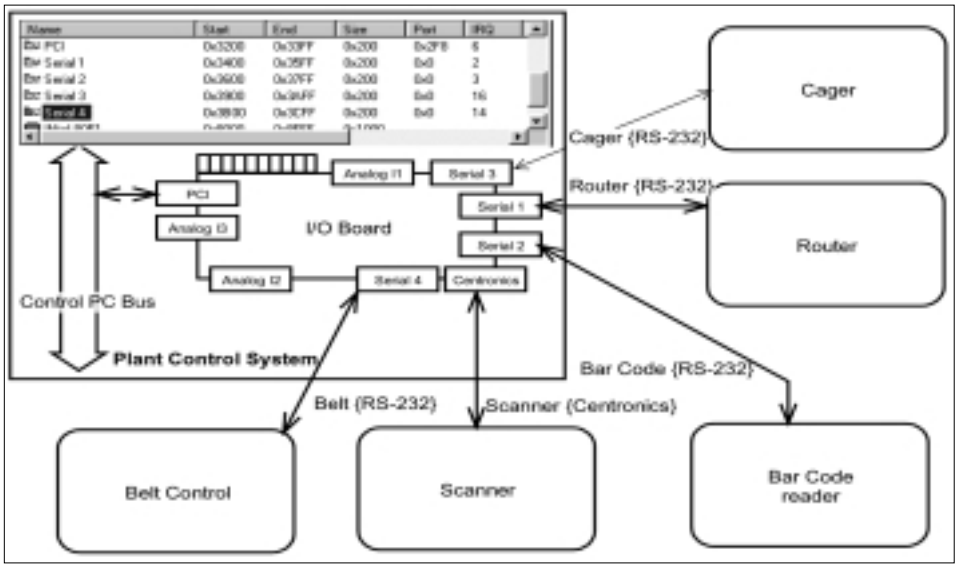


Figure 9: *An Architecture Diagram Showing Board-Level Connections*

Notes

1. The Object management Group (OMG), www.omg.org, is a consortium of companies and other interested parties responsible for the specification and standardization of the Unified Modeling Language. OMG ratified the inital version of UML, UML 1.1, in 1996 and has more recently specified and ratified the current version, UML 1.3 (June 1999).
2. UML provides three formal extension mechanisms, namely constraints, stereotypes, and tagged values. The latter two mechanisms can be used to provide additional symbolic elements such as busses, boards, and disks as extensions to the standard notational elements called classifiers.

Table 1: *UML Collaboration Diagram*

Standard UML Diagrams Referred to in this article	Extensions to UML for Real-time development
Class Diagram Use Case Diagram Sequence Diagram Collaboration Diagram Deployment Diagram	System Scope Diagram Architecture Diagram (Extended) Sequence Diagram Concurrency Diagram

About the Author

Alan Moore has 15 years experience in the development of real-time and object-oriented methodologies, and their applications in a variety of problem domains. He has been actively involved in product development, training, and consulting related to Object Oriented Analysis and Design and structured development tools. Moore has co-authored a book on graphical user interface design and published several papers, and has lectured on a wide variety of analysis and design issues. Moore is responsible for the specification, planning and management of the ARTiSAN product strategy. He is the author of ARTiSAN Real-time Perspective, a pragmatic approach to the development of real-time systems and is an active participant in the Real-time Analysis and Design Group of the Object Management Group .

Diagrams showing the extensions discussed in this article were prepared using Real-time Studio from ARTiSAN Software Tools (www.artisansw.com), a UML-based modeling tool used by real-time and embedded systems developers.

Vice President of Strategy
ARTiSAN Software Tools
Stamford House, Regent St.,
Cheltenham, Glos., UK GL501HN
Voice: +44-1242-229-300
Fax: +44-1242-229-301
E-mail: alanm@artisansw.com
www.artisansw.com