

UML's shortfalls in modeling complex real-time systems

A primary concern of all real-time systems is that they not only perform the required functions, but that they do so within specified time constraints.

Object orientation (OO) is playing an increasing role in the development of many real-time software systems. It's been recognized for some time that modeling is one of the important ingredients contributing to a successful development project. Within the last two years, the unified modeling language (UML) has become increasingly significant as a means of modeling software systems. But although UML provides a sound basis for OO modeling for many application areas, it's deficient when it comes to three key aspects of real-time systems—architecture, concurrency and schedulability. Modeling complex real-time systems requires the use of a suitable, repository-based tool.

UML is recognized as the principal OO modeling language by many developers and vendors. It's being widely applied to the practice of software development in a variety of industries and for differing applications. Its development is ongoing; under the auspices of the Object Management Group (OMG), it continues to evolve. For most real-time systems, the related issues of hardware architecture, concurrency and schedulability are paramount. A primary concern of all real-time systems is that they not only perform the required functions, but that they do so within specified time constraints.

Real-time issues

Most real-time systems also have an inherent need for concurrent processing and so need to identify and design the separate and independent processes (or tasks) that will handle the concurrent activities. One consideration is the hardware architecture. I/O devices will determine response and triggering requirements, processor and communications architecture will affect concurrency choices and execution speed. The way in which the software is partitioned into concurrent tasks, how these tasks are scheduled, and the architecture of the system will all affect the system's ability to meet its timing constraints.

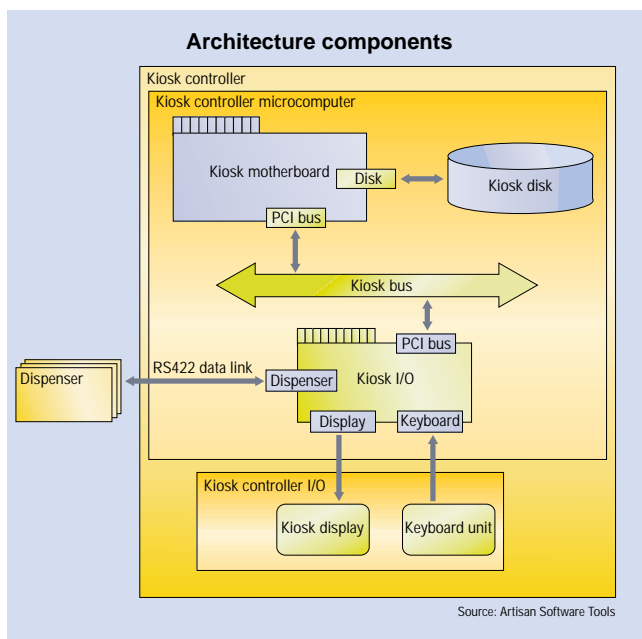
It's critical to be able to effectively model architecture, concurrency and schedulability if we're to be confident that the systems we construct will meet their functional and timing requirements.

System architecture is the physical structure of the hardware on which the software system sits and which it controls. This architecture will consist of a variety of components and the links between them. Modeling ranges from simplistic approaches—where each element in the model is a "one-off" occurrence and is unrelated to any other elements—to sophisticated models based on a richly connected, typed model, such as that supported by the UML metamodel. The former can be produced using a simple generic drawing tool,

*Alan Moore, vice president, product strategy; and
Tony Beckwith, senior methods engineer, Artisan Software Tools*

such as MS Paint or PowerPoint. The latter requires a purpose built, repository-based tool to manage the complex relationships between model elements. The benefits of modeling with such tools are considerable. They save development time and provide significant quality enhancements through element reuse and consistency checking. Real-time Studio from Artisan Software Tools (Capitola, CA) contains such facilities for modeling the system architecture.

The power of the system architecture model derives from the ability to define different types of system component, and to then create specific instances of these types. The architecture is organized into nodes (we'll use the UML term) and the connections between nodes. The basic set of node categories needs to be fairly rich, reflecting the need of system engineers to construct detailed models of their systems. Node categories



In this system architecture for part of a gas station control system, various architectural components, such as boards (kiosk I/O), buses (kiosk bus) and interface devices (keyboard unit), are displayed, as well as subsystems (dispenser, kiosk controller, kiosk controller microcomputer). Subsystems can contain other components, including other subsystems.

Embedded Software & Development Tools

supported by RTS are: boards, interface devices, disks, multidrop buses and sub-systems. Connection categories are: bus drops and point-to-point.

Within these categories, the metamodel supports both types and instances. It has the ability to define both types of nodes, like Intel's (Hillsboro, OR) AL 440 IX motherboard type, and instances of these types like the Kiosk motherboard.

The figure on the next page shows the system architecture for part of a gas station control system. Note the various architectural components, such as boards (kiosk IO), buses (kiosk bus) and interface devices (keyboard unit). Also shown are subsystems (dispenser, kiosk controller, kiosk controller micro-computer); subsystems can contain other components, including other subsystems.

These are physical instances, which will be built into the final system, but they've been drawn from a library of subsystem, board, bus and interface device types, which already exist in the RTS repository.

As an organization models more systems, engineers tend to build up these libraries of system component types. To define a board type, for example, you'd specify its purpose (e.g., motherboard) and its content (e.g., processor, memory, I/O devices). Then for each processor type, memory type, or I/O device type present on the board, you could specify particular properties (e.g., speed, memory mapping, port addresses, etc.). This provides a template that can be used by creating specific instances of this board type, wherever such a board is required within a system. You can also easily specify new board types from existing subtypes (processor, etc.), perhaps adding some new subtypes where required. Creating new types doesn't necessarily mean starting from scratch. It may simply require a minor alteration to an existing type.

Each type will have a set of properties that can specify the details of speeds, memory mapping, port addresses, etc. that apply to all instances of that type. In addition, each instance may specify further properties that relate to that specific instance.

Typing provides a powerful mechanism for constructing an architectural model that facilitates ease of understanding and refinement, but without information overload. An important

goal underpinning the wide range of node categories is improving the necessary communication among the system, hardware and software engineers, making it quicker and safer. In particular, the ability to specify the features of a particular board type, such as processor, memory addresses, I/O device ports and register maps, let software engineers derive the right level of data without being swamped with circuit diagrams.

Deployment diagrams

The UML approach to architecture modeling uses deployment diagrams to illustrate how objects and components (concrete groupings of objects and classes, usually offering a coherent service) are deployed onto nodes. As

Multitasking places a requirement on developers to ensure that the structure and scheduling of tasks allows the system to meet its time constraint requirements.

with all UML elements, nodes can be stereotyped to indicate a broad category of node type. Components and object instances may be contained within nodes, thus illustrating how the software is deployed onto the hardware. Components may have interfaces attached that can then be used by other components.

But the UML deployment diagram is somewhat restricted when it comes to architecture modeling. For example, there are no predefined node stereotypes to help improve standardization, and although components may have interfaces, nodes (arguably the most obvious candidates for interfaces) have none. This leaves the deployment diagram lacking the type of architecture components required for real-time systems. The lack of a set of predefined stereotypes also means that there's no capability to capture the depth of information required to fully describe operational properties of the system and its devices (speeds, memory maps, port addresses, interrupt handling, etc.). The

notations available in RTS system architecture diagrams are one suggested set.

Concurrency and schedulability

The issues of concurrency and schedulability are strongly related. Concurrency involves the identification of tasks and task inter-communication. Schedulability analysis tries to determine whether a system composed of many tasks can meet all of its deadlines.

A task here is defined as a single, sequential thread of control within the program. The nature of many real-time systems is such that certain processing activities need to be carried out simultaneously with other processing activities. A common example is when alarm monitoring needs to take place continuously as a background activity to other system functions. Thus, you need a task that will carry out the alarm monitoring activities and one or more other tasks to carry out the other system functions. Multitasking, although it's a logical (and sometimes necessary) solution to the problem of concurrent activities, brings with it a new set of problems.

The first of these is how to implement the required concurrency. One solution is to use multiple processors, one processor per task. This is often far too expensive a solution, and may involve an unacceptable degree of hardware complexity. The alternative is to use a real-time operating system (RTOS), which will allocate processor time slices to each task on a task priority basis—the higher priority tasks preempting the execution of lower priority tasks.

It's important to note that the RTOS itself becomes a task that requires processor time. On a single processor, this solution isn't true concurrency in that the processor is only executing one instruction from one task at any one instant. True concurrency is still achievable, but only by distributing the tasks over two or more processors.

The second problem that multitasking brings is inter-task communication. There are really three distinct issues here: synchronization, mutual exclusion and data transfer. Two tasks need to synchronize whenever the processing taking place in one task needs to know that certain events or activities within another task have occurred. Synchronization normally implies one of the tasks sus-

Embedded Software & Development Tools

pending until the other task reaches the synchronization point. Both tasks need to signal to the RTOS when they reach the synchronization point; it's the RTOS' responsibility to suspend the first task to reach the synchronization point, and to wake it up when the second task signals.

Mutual exclusion is required whenever two (or more) tasks need to access the same resource. A mechanism (normally a semaphore) is required to ensure that when one task is using a shared resource, the other tasks are excluded.

Finally, the requirement to transfer data between tasks may require synchronization. Synchronized data transfer is essentially as described above but the sending task adds the data to the signal it sends to the RTOS, and the receiving task receives the data from the RTOS in response to it sending the synchronization signal.

Asynchronous data transfer involves the sending task transmitting the data to the RTOS and carrying on with its processing. The receiving task requests the data from the RTOS and will be suspended if the required data isn't available.

All inter-task communication is carried out via the RTOS. It's effected by means of RTOS primitives, where different types of primitives are used for synchronous and asynchronous communication of events and data. The RTOS, therefore, plays a key role in multitasking, both in terms of scheduling tasks and handling communication between them.

Time is of the essence

All real-time systems, by their very nature, have a requirement to meet certain time constraints. Multitasking places a requirement on developers to ensure that the structure and scheduling of tasks allows the system to meet its time constraint requirements. The key is to have a concurrency model that permits worst-case timing scenarios to be determined.

The concurrency model needs to show the tasks and their priorities, as well as their communication and resource sharing. It also needs to provide the types of primitives required

UML recognizes the concept of an 'active' object in object diagrams, but it's very vague about what that is. One definition is that it's an object that includes at least one thread.

supporting mutual exclusion and inter-task communication. Concurrency allows engineers to reason about such phenomena as task blocking and deadlocks. The model will also need to provide different types of inter-task communication identifying different types of blocking behavior. Engineers also need to be able to provide estimates (or actual times, if available) for task activities, task switching and task inter-communication. These are stored against elements in the concurrency model.

UML recognizes the concept of an 'active' object in object diagrams, but



Tony Beckwith, senior methods engineer at Artisan Software Tools is responsible for evolving and refining Artisan's Real-time Perspective incremental development process and producing case study material for training, seminars and supporting Artisan's Real-time Studio toolset. His previous experience includes

software development, project management, consulting and lecturing in a variety of application areas.



Alan Moore, vice president of product strategy at Artisan Software Tools has 15 years of experience in the development of real-time and, object-oriented methodologies, and its application in a variety of problem domains. He's been actively involved in product development, training and consulting

related to OOAD and structured development tools.

Alan co-authored a book on GUI design, published several papers and has lectured on a variety of analysis and design issues. He's 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 (RTAD) of the Object Management Group (OMG).


it's very vague about what that is. One definition is that it's an object that includes at least one thread (or task).

When it comes to reasoning about schedulability and such issues as deadlock, it's important to recognize explicitly there are threads and resources they share. In addition, there's a very sparse set of useful stereotypes to convey vital information about the use of operating system primitives.

Such primitives, including tasks, are objects, if they're appropriately wrapped in class wrappers (most O/S providers do this). They're also something else—tasks and resources. Communicating this clearly and storing the appropriate data about them is the first step to ensuring a successful approach to dealing with multitasking issues.

Real-time systems are often complex systems. Developing software for real-time systems is a complex process. To deal with this complexity, you need as much help as you can get. Modeling provides a mechanism that helps you understand the inherent complexity of real-time systems and to design relevant and successful solutions to real-time problems.

Three key areas of real-time systems where the complexity is particularly obvious are architecture, concurrency and schedulability. The requirement of real-time systems to meet time constraints implies that you must thoroughly understand the inter-relationship between these areas if you are to design and build systems that meet all of their requirements.

These areas in particular require strong, detailed modeling support. While UML is now seen as the most important OO modeling language, in its current format, it's deficient in these three areas. 

FOR MORE INFORMATION

Artisan Software Tools
Capitola, CA
(831) 475-5554