

TURNING CLOCKWISE: USING UML in the REAL-TIME DOMAIN

"You mustn't forget one thing," the fairy godmother said.

*"Do not stay past midnight, for when the hour strikes,
everything will return to the way it was before.*

"I promise," said Cinderella.

BRAN SELIC

TIME IS A TYRANT. UNLIKE MOST OTHER PHYSICAL PHENOMENA, WE HAVE no control over it—we cannot stop it, stretch it or shrink it to fit our needs. The most we can do is measure its progress. This relentless quality of time is one of the primary reasons why designing real-time software is often such a difficult task. By definition, timeliness is paramount to real-time software—a late response is sometimes worse than no response at all.

We must not be misled, however, into thinking that achieving timeliness is the sole source of complexity in real-time software. All real-time systems ultimately involve interaction with the physical world. This world, as the apocryphal Murphy suggests, can seem to conspire against us: events will often occur at the wrong time or in the wrong order. The ultimate and inevitable source of complexity in real-time software, therefore, is the physical world itself. The sad fact is that no matter how hard we try and no matter which technology we use this fundamental complexity cannot be eliminated.

Compounding this is our persistent demand for

ever-greater functionality. This is fueled in part by competitive business pressures but also by the natural human propensity for increasingly sophisticated machinery. In telephony, for example, there are close to 500 different types of "features" defined. These are specialized services, such as call forwarding, voice mail, or mobile phone roaming, that we now take for granted. Unfortunately, many of these features were defined independently of each other and are often mutually incompatible. In spite of that, we expect the software inside a communication system to provide all of these various features and somehow to resolve the inconsistencies between them.

The level of complexity encountered in many real-time systems is already far in excess of what can be grasped by the unaided intellect. Mind-numbing software systems comprising 10–20 million lines of high-level language code are becoming common (as are the costly and often well-publicized failures of such systems). In a crude comparison, the most sophisticated modern jumbo jets involve approximately six million parts.

Modeling is one of the relatively few tools that we have for dealing with this rampant complexity.

Modeling is simply a form of abstraction that facilitates both problem understanding and problem resolution. A model captures only the significant features of a system and hides or ignores lower-level detail. The simplified yet relatively accurate view of reality presented by a model is normally much easier to comprehend than the actual system.

The Unified Modeling Language [1, 2, 9] is one of the most recent examples of a modeling tool. It was designed for use with object-oriented and object-based systems and applications. In 1997, the Object Management Group (OMG)—an international consortium of software vendors and users—adopted UML as a standard. Since then, interest in the UML has grown rapidly and it is becoming a lingua franca within the software community.

Real-Time Systems and the Object Paradigm

The designation “real-time” has been applied to a surprisingly broad variety of different software systems—ranging from embedded controllers with microsecond response times to more ponderous systems with response times of seconds or even

minutes. Nevertheless, in this rich cornucopia of diverse systems certain common characteristics do emerge: *a real-time system is a software system that maintains an ongoing and timely interaction with its environment.*

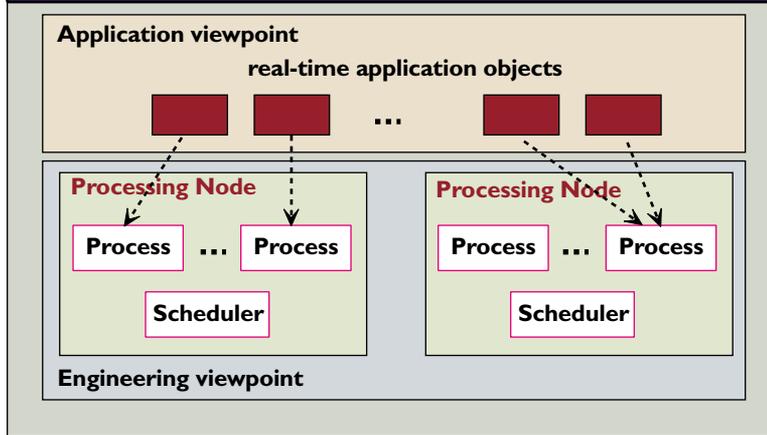
While timeliness is an obvious characteristic of any real-time system, what does it mean for a system to “maintain an ongoing interaction”? This relates to a quality that is often referred to as *reactivity* [8]. A reactive system persists across some interval of time (possibly indefinitely) during which

it responds to inputs as they occur. The interesting thing to note is that the dominant feature is the system itself, which is constant, and not the behavior, which comes and goes. This shifts the focus of design from the algorithmic to the structural—a reversal of the traditional procedural approach to software.

This makes the object-oriented programming paradigm an excellent fit for many real-time systems. Under this paradigm, originally developed to simulate the physical world, software is specified at the highest level as a structure of interacting dynamic components. This style was used in many real-time systems even before the relatively recent advent of object technology.

Despite this clear and convenient conceptual match, there is still a wariness of object technology among many real-time practitioners. This comes from the perception that object technology necessarily entails unacceptable memory and performance overheads. While there is no doubt that abstraction mechanisms such as polymorphism and encapsulation can induce overhead, a careful analysis of the actual costs should be undertaken before any firm conclusions are reached. Many of the

Figure 1. The application and engineering viewpoints of a distributed system. This separation of software-based systems into different viewpoints is inspired by the ISO Reference Model for Open Distributed Processing [4].



problems of early generations of object technology have now been significantly mitigated or even eliminated. This, together with the ever-increasing proficiency of hardware, means that the level that makes object technology acceptable for real-time systems is steadily dropping. There are now many large-scale real-time systems that have been implemented fully or partially with object technology.

Even in real-time applications with very stringent performance requirements, it is typical for only a small proportion of the system to be performance sensitive (the common industry adage that 80% of the time is spent on 20% of the code). This means that the full power of the object paradigm could be exploited for the greater part of the software.

Requirements for Modeling Real-Time Systems

Real-time systems tend to be more constrained than other types of software. The need for timeliness is a prime example of these constraints. In so-called "hard" real-time systems, the violation of even a single deadline might mean the loss of human life. A crucial requirement, therefore, is to use models of such systems to derive early and accurate predictions of the key attributes of the actual system (response times, throughput, availability, and so forth).

Many of these attributes are quantitative, reflecting the physical nature and finite capacities of the underlying computing resources. Over the years, a number of very useful techniques have evolved in the real-time domain for predicting these attributes. These techniques include, notably, schedulability analysis techniques such as rate monotonic analysis [5], which determine whether a given system will meet all of its

deadlines, and performance modeling methods such as queuing theory [6], which compute delays and resource requirements as statistical quantities.

For models to be predictive, it is necessary to model not only the structure and behavior of the software, but also the logical and physical resources, or engineering infrastructure, on which that software relies. This requirement perhaps most clearly distinguishes modeling of real-time systems from other types of systems. In addition, there are unique requirements for modeling the idiosyncrasies of structure and behavior of real-time software. We now examine each of these sets of requirements in greater detail.

Infrastructure modeling requirements.

The engineering infrastructure consists of physical devices such as processors and networks as well as logical "devices" such as concurrent processes (tasks), locks, or queues (Figure 1). These can all be considered as specific types of resources. In general, a resource is characterized by physical attributes such as capacity and location. When these attributes are assigned numerical values in a model, the model can be used to make predictions about the actual system.

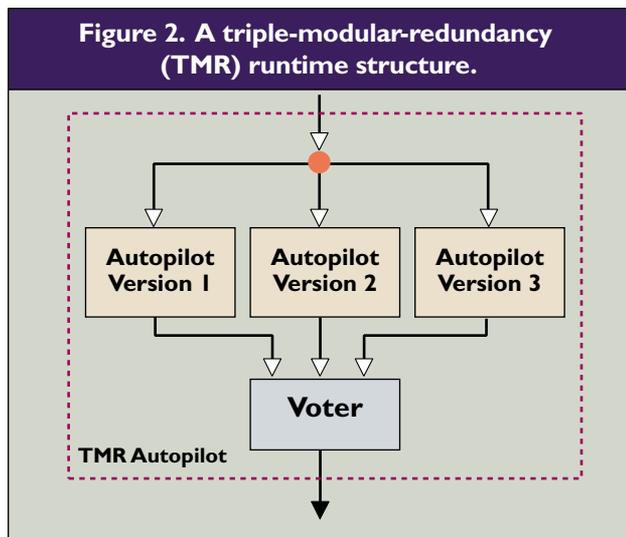
In real-time systems, relatively sophisticated models of resources are often required. For example, in some situations it may be necessary to have a very detailed rendering of the internal architecture of a physical computing environment comprising CPUs, caches, memories, and busses. This level of modeling resolution for resources may be needed for two reasons. First, it may be required to achieve the desired accuracy for predictions based on the models and second, it may be required for systems in which the application software itself manages the resources. In a fault-tolerant real-time system, for instance, the application may need to perform diagnostics on individual memory units and other hardware resources. In these cases the infrastructure model is part of the overall software model.

In addition to resources, the engineering infrastructure often includes basic operating system services, such as scheduling, timing, memory management, and so forth. These services may also have quantitative attributes such as response time and availability.

Last but not least is the ability to specify the binding relationship between the software and the resources (indicated by the dashed arrows in Figure 1). This includes the allocation of software elements to

specific resources and the demands that individual elements of the software place on resources. For performance modeling, for example, it might be necessary to know which objects are on which processors, as well as their individual memory and processing requirements.

Behavior modeling requirements. Because real-world events can occur unpredictably and concurrently with other events, the sequential paradigm of specifying behavior that characterizes procedural programming is often inappropriate for describing behavior of real-time systems. Instead, depending on the nature of the inputs to the system, two different styles have evolved for specifying behavior in real-time applications [7]. The event-driven style is best suited for modeling discrete behaviors in which concurrent events occur asynchronously. Processing occurs in discrete steps, triggered by the arrival of events. In con-



trast, the time-driven style is most suitable for continuous or periodic (sampled data) inputs. In this case, processing is not initiated by the arrival of events but by the regular beat of a real-time clock. Inputs are sampled and processed periodically. Any inputs that occur between the end of processing and the next clock tick are processed in the next period. Within a single system there may be multiple concurrent activities executing on different frequencies. The modeling of both styles needs to be supported.

Structure modeling requirements. Finally, there is the need to model the complex high-level runtime structures¹ of real-time systems. We noted earlier that structure is one of the dominant aspects of real-time systems. An example of a runtime structure often used

¹It is useful to distinguish runtime structures, which involve object instances, from design-time structures that represent static relationships, such as inheritance associations between classes.

in fault-tolerant real-time systems is the so-called triple-modular-redundancy structure shown in Figure 2. A key aspect of runtime structures is the topological relationship between the individual objects. These can be grouped into three basic categories:

- Peer relationships that occur between directly communicating objects. An example of this is the relationship between the servers and the voter in Figure 2.
- Containment relationships, such as the relationship between the “TMR Autopilot” and its components. This includes both composition in which the lifetime of components is dependent on the existence of the container, as well as aggregation² whereby the container merely encapsulates a component.
- Layering relationships. This is a special case of a client-server relationship in which the client uses a shared server as part of its implementation. Usually, an object encapsulates (hides) its implementation components. In this case, however, because the server is shared with other objects, it cannot be encapsulated. The effect of hiding is achieved by abstracting away such shared implementation services into a supporting “implementation layer” or “virtual machine.”

In addition to the ability to model the preceding three different structural forms, there is a need to model dynamic runtime structures. These are structures in which objects and links are constantly being created and destroyed as the load on the system changes. In a telephone switch, for instance, it is common practice to create a special “call” object when a new call is initiated and to destroy this object when that call is terminated.

Using UML for Modeling Real-Time Systems

UML is a general-purpose modeling language that provides an extensive conceptual base for a broad spectrum of application domains. Is this conceptual base broad enough to cover real-time systems, and, if so, how are these concepts to be used for dealing with the specific requirements identified in the previous section?

Although there have been a number of calls to extend UML for the real-time domain by adding new concepts to its base, experience has proven that this is not necessary. Instead, real-time modeling can be fully

²The use of the terms “composition” and “aggregation” to distinguish the different forms of containment are based on UML terminology.

accommodated by specializing appropriate base concepts. This means providing additional semantics that supplement but do not violate the general semantics of the concepts from which the semantics are derived. For example, rather than introduce a new base concept to UML to capture the notion of a real-time clock, we can model it simply as an instance of a UML class that has the usual operations for reading and setting the time. Since such a concept is useful in most real-time systems, we may choose to define it as a special “real-time clock” stereotype of the class concept. The stereotype may have additional constraints attached that fully capture the semantics of real-time clocks. In effect, this creates a reusable domain-specific concept.

The use of stereotypes avoids the infamous “language bloat” syndrome, something that plagued many earlier language unification schemes. The net result is a language that remains compact yet, because the refinements fully respect the general semantics of their parent concepts, retains its “universal” quality. That is, any practitioner with a general UML background will be able to understand a model based on these specializations, even without knowledge of the added semantics. (Of course, this understanding will not extend all the way down to the detailed semantic level.)

Modeling the Engineering Infrastructure

There are several aspects to engineering infrastructure modeling, as described and detailed in the following paragraphs.

Modeling resources and system services. Most resources can be modeled by appropriate stereotypes of the general UML concepts of Classifier or Class. Processors and other hardware devices may be modeled as stereotypes of the Node concept (which is a specialization of Classifier). This allows flexible modeling of different resources, including resources with complex semantics and complex internal structures.

Each resource provides some type of specialized functionality required by the logical elements that it supports. In general, it is convenient to view resources as servers whose services can be accessed through interfaces. The operations (features) accessed through these interfaces typically have quality of service attributes, such as access time or response time, that can be assigned numerical values. One of the primary quality of service attributes of is capacity. For example, a semaphore may be modeled as an object that has “get-semaphore” and “release-semaphore” operations and a capacity attribute. This attribute identifies the maximum number of rights that the semaphore can dispense at any given time.

The same approach can be used to represent oper-

ating system services. Depending on the modeling resolution required, the entire operating system substrate can be modeled as a single object with the appropriate service call interfaces (and quality of service attributes) or as a set of discrete specialized objects.

A resource is said to support one or more elements of a logical model in the sense that it provides implementation support for those elements. Conversely, a model element may require the support of multiple resources. The “supports” relationships between logical elements and the corresponding resources are represented by stereotyped usage dependencies (indicated by the dashed arrows in Figure 1) that include the requisite demand attributes.

Element-to-resource mapping. It is common for a real-time system to be designed so that it can run on a variety of different hardware configurations. A particular system may be designed to operate on two physically distinct processors, but it could also execute successfully on a single processor with sufficient processing power. This means that different element-to-resource mappings may be required for the same logical model. It is convenient to combine a mutually consistent set of element-to-resource mappings of a given logical model into a single package. This set of mappings, called a resource-mapping package, relates a logical model to a particular hardware configuration. A different resource-mapping package can be used to relate the same model either to a different implementation machine or to a different set of element-to-resource relationships.

There can be various degrees of flexibility in assigning a resource-mapping package. In some cases, systems may be hardwired to a single implementation machine as part of the basic system specification; in other cases, the actual assignment may be finalized at load time or even, dynamically, at runtime.

Resource-to-value mapping. A different mapping related to resources is the association of resource attribute values (processor speeds, queue sizes, and so forth) to the resources of an implementation machine. This is required for any type of quantitative analysis of a model.

As in the case of element-to-resource mappings, it is also useful to allow multiple different value mappings for a given implementation machine. For the same reasons, it is convenient to combine mutually consistent sets of value mappings into UML packages, called resource value packages.

Modeling time and timing facilities. Since time plays a central role in real-time systems, it is useful to describe how the resource model applies here. In UML, time itself is a semantic variation point. No assumptions are made about whether time is global or

Figure 3. Functional behavior: a communication protocol handler (alternating bit).

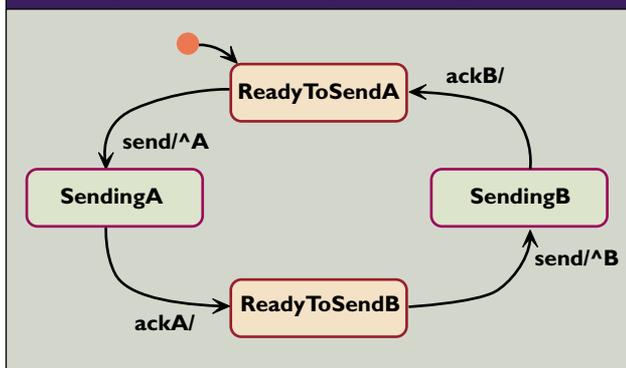
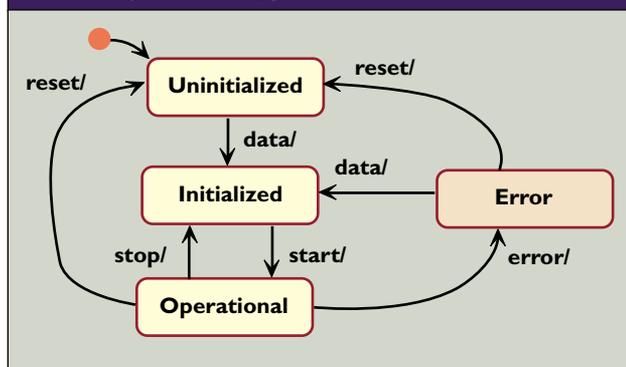


Figure 4. A typical control behavior.



distributed. A base type, Time, is defined in the standard, but its semantics are not specified. This leaves applications free to define different models and representations of time.

All timing facilities, such as clocks, calendars, and timing services can be modeled directly as resources. They will have service interfaces for reading current time values and for requesting timing services and control interfaces through which a clock may be adjusted or reset. A timing service can also provide special dedicated timers for individual timings. At the expiration of a specified interval or upon reaching a specified clock value, the timer can generate an event. This allows modeling of time-driven behavior even in an event-driven environment.

Modeling Real-Time Behavior

The fundamental notion of concurrency is modeled using the “active” object concept of UML. Note that the standard intentionally leaves plenty of leeway here for modeling the different flavors of concurrency that might arise. For instance, it does not specify the delivery order of inter-process messages or the policy used to schedule concurrent processes. Clearly, these are factors that may be resolved very differently in different situations, and

no single approach can satisfy all possible real-time applications.

Event-driven (reactive) behavior modeling.

Event-driven behavior seems to be most conveniently described using some type of transition system formalism [8]. The underlying model is of an entity that performs a transition from one quiescent state to another in reaction to the arrival of an event. The net result is that any protracted activity is sliced into a sequence of event-response pairs that can be easily interrupted if another more critical concurrent activity occurs. This is important for real-time applications where concurrency is the norm and the ability to quickly respond to high-priority events is crucial.

The state machines of UML—derived from David Harel’s statechart formalism [3]—are an example of a transition system that is particularly suitable for highly concurrent real-time systems. The power of this technique is demonstrated here by showing how this formalism can be used to resolve the important real-time problem of conflicting control and function behaviors.

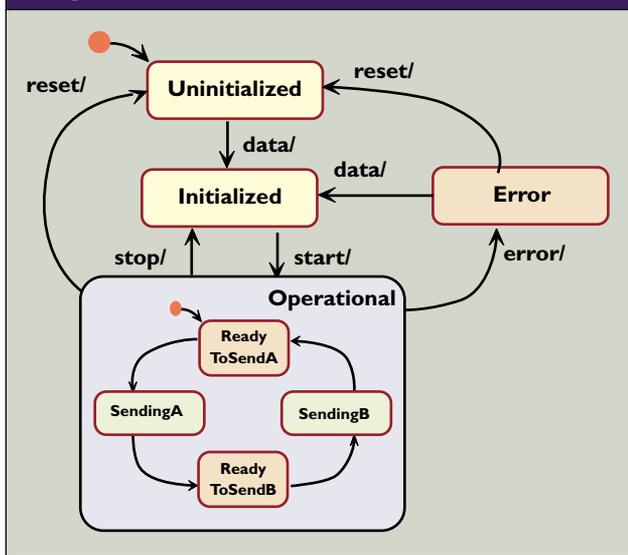
The processing of services that are the primary responsibility of an object will be referred to here as its functional behavior. This behavior is typically specified by a state machine, such as the one shown in Figure 3 (the example shows the state machine of an alternating-bit protocol handler).

The functional behavior is distinct from the control behavior of the object. The control behavior describes the actions performed by an object as a controlled entity within the overall system. It encompasses control activities such as initialization and termination, error handling and recovery, and so on. It is standard practice for the control behavior of different types of objects to be the same regardless of the type of service they provide. This strategy provides a common model of control resulting in simpler and more controllable systems. An example of such a function-independent control automaton for objects is shown in Figure 4.

The problem is that we have two different state machines that describe different aspects of the behavior of the same object. These two have to be combined in some way. With simple non-hierarchical state machines, the only course of action is to combine the machines into a single automaton. This type of merge is prone to errors and often results in complex and confusing state machines. The control and functional behaviors become intertwined and are difficult to change independently.

One solution would be to compose the two state machines by combining them in two peer concurrent states. However, this approach has two major difficulties. The first is that the semantics of orthogonal state

Figure 5. The combined UML state machine.



machines can be so complex that system response time may be jeopardized due to high overhead. The second difficulty is that the resulting behavior is not really the behavior that is desired. For example, there is no point in performing the protocol handling function until the protocol handler object is initialized with the operational data it needs to perform that function. Functional behavior is only valid when the object is in the operational state.

A much simpler and more effective solution is to use hierarchical states and transitions as shown in Figure 5. This yields a nice separation of the two behaviors, allowing each to be modified independently of the other. Furthermore, we can take advantage of inheritance and define the control automaton just once in an abstract class. All other classes that have the same control behavior can be defined as subclasses of that abstract class and would automatically inherit the control behavior.

Time-driven behavior modeling. Since time-driven behavior is based on periodic activities, it may seem that it is not suitable for modeling using the object paradigm. However, this view ignores the previously discussed question of control. Before the periodic activities can be initiated, the system has to reach an operational state. Similarly, when a failure occurs, the system has to go through a fault handling and recovery procedure before it can resume its functions. Since control activities are inherently event-driven (the occurrence of failures, the arrival of operational data, and so forth), we conclude that even in case of time-driven behavior, it is appropriate to use state machines at the highest level of description of an object.

Once an operational state is reached, periodic behavior is activated by a periodic timeout event.

When a timeout occurs, an internal transition associated with the operational state is triggered. The action associated with that transition is the periodic activity. In fact, the same object may perform multiple periodic activities at different rates. Each separate activity would be handled by a separate transition triggered by its own timer. This has the further advantage of packaging all the different activities associated with an object in one place.

Object interaction modeling. While state machine diagrams are useful to describe the reactive behavior of individual objects, it is useful to specify and view the emergent behavior of a set of collaborating objects. The UML provides a number of facilities for this purpose: sequence diagrams, interactions (within collaboration diagrams), and activity diagrams. Of these, by far the most widely used in real-time development is the sequence diagram.

From a real-time perspective, a particularly useful feature is the ability to specify timing marks in UML sequence diagrams. Timing marks denote specific time instants, such as the reception of a message by an object. The precise semantics and resolution of timing marks are left to the application. For instance, it may be important to differentiate the instant when a message is queued at the receiver from the instant when it is scheduled for processing. Timing marks are used to graphically specify timing constraints and time interval values.

Another useful way of modeling inter-object behavior in real-time systems is through protocols [10]. A protocol is a special type of UML collaboration that specifies the set of valid message exchange sequences between two or more active objects. Protocols are defined independently of any specific objects or classes. Instead, they are defined in terms of protocol roles. A protocol role (modeled by a UML Classifier Role) specifies the set of valid incoming and outgoing messages relative to that role as well as the set of valid message orderings. Because protocols are defined independently of any specific context, they are reusable. This is quite handy since it often happens that the same protocol applies in a number of different situations within a real-time system. (For instance, in the TMR system in Figure 2, the voter-server protocol is repeated three times.) Another useful feature of protocols is that they can be specialized using inheritance. This means that variations of a protocol can be produced starting from a common abstract protocol.

Modeling Real-Time Structure

Designers often underestimate the importance of structure. In essence, runtime structures represent the architectural core of a real-time system. Needless

to say, architecture is of paramount importance in complex systems. A well-designed architecture will allow a system to easily accommodate evolutionary growth and prolong its useful lifetime.

For this reason, much attention was devoted in UML to providing concepts suitable for the precise modeling of structures. The basic tool for this purpose is the collaboration diagram. It captures topological arrangements, including all three basic runtime structural forms, in a generic way: instead of using specific classes, collaborations are defined in terms of “roles,” the equivalent of a job qualification description. This is particularly handy for modeling real-time structures since different objects can be

moved into and out of specific roles, as needed.

Architectural modeling. The use of UML collaboration diagrams for specifying complex real-time architectures has been the focus of some recent work [11]. The central idea is to capture architectural specifications in a formal way. This has a number of important advantages. First, it means that the architectural models can be formally analyzed for consistency and completeness. It also means that the models are executable and allow early and precise assessment of the validity of different architectural approaches. Finally, using collaboration diagrams allows implementations to be derived directly from architectural specifications using automatic code generation. This

OMG'S Real-Time Analysis and Design Initiative

In 1998, the OMG appointed a special working group, the Real-Time Analysis and Design Work Group, to examine the issue of applying UML in the real-time domain. This group formulates requests for proposal (RFPs) for suitable technologies and also recommends for adoption any submissions that it deems appropriate.

The objective of the working group is to define common ways of modeling fundamental real-time concepts in a way that will facilitate model interchange between different tools. Particularly useful is the interchange between a modeling tool and complementary engineering tools. This eliminates the need to produce manually independent specialized models for the different kinds of analyses that need to be performed. For example, a UML model that includes a model of the engineering-level resources and their attribute values can be passed automatically from a design tool directly to a performance modeling tool where it can be analyzed and the results can be fed back to the design tool.

In its initial analysis, the working group collected requirements from experts in industry and academia as well as from vendors of real-time software. These requirements were

classified into three groups. The first contains modeling requirements related to time, specifically, issues of time modeling, modeling of operating system timing services, schedulers and scheduling techniques, and performance modeling. The second group covers general quality of service issues other than time (such as availability and safety) and modeling of common fault-tolerance techniques. The third group covers complex system issues such as the modeling of dynamic real-time structures.

The working group decided to issue a number of RFPs based on the collected requirements. The first of these RFPs was issued in March 1999 with a submission deadline set for November 1999 and an anticipated adoption date of July 2000. The initial RFP covers the issues belonging to the first group of requirements; specifically, timeliness, schedulability, and performance.¹ Subsequent RFPs are expected to be issued over the next 12 months.

The issued RFP asks for a UML “profile.” A profile is simply a way of packaging UML specializations. The original standard already includes two such profiles, one for modeling

software development processes and one for business modeling.

Profiles are based on the so-called “extensibility mechanisms.”² These mechanisms include:

- stereotypes, used to define variations of base UML concepts,
- constraints, which formally specify restrictions on some more general set of semantics, and
- required tags, named items associated with a base concept or stereotype used to store some domain- or application-specific information.

In addition to these, a profile might include an optional notation for the specialized concepts as well as additional informal descriptions.

Profiles always work within the semantic variation points defined in the UML standard. These are placeholders within the general UML framework for injecting domain-specific or application-specific specializations. These specializations add more refined semantics to certain base concepts, but do not contradict the general semantics of those concepts. **□**

¹Object Management Group, *UML Profile for Scheduling, Performance, and Time (Request for Proposal)*. OMG document ad/99-03-13, March 1999.

²The term “extensibility” is somewhat misleading, since the purpose of these mechanisms is not to add new root concepts to UML, but to specialize existing ones.

means that architectural decisions are automatically enforced avoiding the phenomenon known as architectural decay—the gradual corruption of an architecture during implementation and subsequent evolution.

The central architectural entity here is the capsule. A capsule is an active object that may have an internal structure described by a collaboration diagram. The nodes of the internal structure are also capsules, which may have an internal structure of their own and so on. This simple recursion allows the modeling of arbitrarily complex structures. Since the specification of a capsule is provided by a class, its internal structural pattern can be created repeatedly and reliably simply by creating a new instance of the class.

Capsules may have one or more special attributes called ports. Ports perform a two-way interface function on behalf of their capsule. All incoming and outgoing communications with a capsule takes place through a port. As a result, a capsule never references any external entities directly. This makes it a self-contained architectural component that is fully reusable in different contexts.

Support for this form of UML-based architectural modeling is now available in a commercial tool from Rational Software (Rose RealTime). This tool allows the construction of executable UML models and has the capability of automatic generation of complete systems from such models.

Assessment

Using UML stereotypes to capture real-time-specific concepts has enabled smooth integration of already proven techniques developed in previous generations of real-time methods (such as ROOM [10] or OCTOPUS [1]) directly into the UML mainstream. This has created an opportunity for the accelerated penetration of modern software design practices that can deal more effectively with the extreme complexity of real-time software. As has been shown here, this includes the use of architectural-level modeling, state machine behavior specifications, and automatic code generation techniques.

In addition, by standardizing on these modeling techniques—an effort that is now under way within the OMG (see the sidebar)—it is possible to apply traditional engineering techniques, such as performance modeling and schedulability analysis, directly to UML models.

Conclusion

Cinderella didn't keep her promise to her fairy godmother. The glamour of the royal ball proved too distracting and an important timing constraint was

violated. This tale mirrors the predicament of most real-time software designers. Despite their best intentions, the complexity of the task can be so overwhelming that it is difficult to keep track of timeliness requirements, let alone guarantee them.

Designing real-time software is immensely challenging not only because of timeliness constraints but also because of the need to contend with the daunting complexity of the physical world. In these circumstances, modeling is an important tool and, as has been shown in this article, UML is particularly well suited to this purpose. This is because UML incorporates most of the basic modeling abstractions that are used in the real-time domain, such as state machines and collaborations, and also because the flexibility of its stereotype mechanism allows these general abstractions to be specialized to the desired degree of accuracy. Furthermore, as a standard, it is the basis for tool interchange allowing different specialized tools to work off the same UML model. This eliminates the learning barrier and supplementary modeling overhead that have previously impeded many real-time developers from exploiting the many useful verification techniques that are currently available.

The UML has been around for only a short time, but we are already seeing its significant penetration in the real-time domain. All of the techniques described in this article have been used in large-scale industrial projects and many are directly supported by commercial software tools. ■

REFERENCES

1. Awad, M., Kuusela, J., and Ziegler, J. *Object-Oriented Technology for Real-Time Systems*. Prentice Hall, NJ, 1996.
2. Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, MA, 1999.
3. Harel, D. Statecharts: A visual formalism for complex systems. *Science of Comp. Prog.* 8 (July 1987), 231–274.
4. International Standards Organization (ISO). *ODP Reference Model Part 1: Overview*. ISO/IEC 10746-1, 1995.
5. Klein, M. et al. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.
6. Kleinrock, L. *Queueing Systems, Vols. 1 and 2*. Wiley, NY, 1976.
7. Kopetz, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
8. Manna, Z. and Pnueli, A. *The Temporal Logic of Reactive and Concurrent System's Specification*. Springer-Verlag, 1992.
9. Object Management Group. *OMG Unified Modeling Language Specification, Version 1.3*. June 1999.
10. Selic, B., Gullekson, G., and Ward, P. *Real-Time Object-Oriented Modeling*. Wiley, NY 1994.
11. Selic, B. and Rumbaugh, J. *Using UML for Modeling Complex Real-Time Systems*. ObjecTime Limited/Rational Software whitepaper, 1998; (www.objecttime.com/).

BRAN SELIC (bran@objecttime.com) is Vice President of Advanced Technology with ObjecTime Limited in Ontario, Canada.
