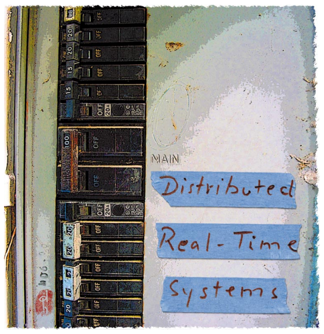
A Generic Framework for Modeling Resources with UML



For real-time systems, designers must consider physical and logical resources. Developers can use the OMG's Unified Modeling Language to model resources and thus predict crucial system properties before fully implementing a system.

Bran Selic
Rational Software

Philosophers, mathematicians, and software designers have a great fondness for the logical world—a pliant place unhampered by the laws of physics and the messy complexities of the real world. Current wisdom encourages designers to first focus on the logical aspects of their problem and to defer platform and technology issues until the concluding phases of development. This behavior is reasonable, considering that devising logically sound solutions is frequently the most difficult aspect of development. The solution that emerges from this approach is often portable to different target platforms and technologies.

Unfortunately—and this has been understated to date—there are many situations in which this approach is inappropriate. For example, numerous software design problems exist where the implementation technology's characteristics have a fundamental impact on the software solution. Real-time software design is one domain in which this situation is particularly obvious because the domain's requirements force software to interact with the physical world in some way.

A dominant characteristic of real-time software design is contending with the quantitative as well as qualitative limitations of underlying computing platforms. Here, the finite nature of the physical world is reflected in the generic notion of a *resource*. This term applies to physical devices—processors, memory, and other hardware facilities—as well as to buffers, queues, and process control blocks, which are not necessarily physical but which ultimately have physical underpinnings.

The ability to accurately model logical and physical resources is clearly an important facet of real-time design. It is also key to introducing into software development quantitative analysis techniques. Such techniques are used to predict crucial system properties—response times, queue sizes, and so on—before the system is fully constructed, an essential element of every classical engineering discipline.

Recently, a generic object-oriented framework has been proposed for modeling both physical and logical resources. Although the framework is generic, it is mainly used with the industry-standard unified modeling language (UML).[1] By providing a standard means for representing resources and their attributes, we can seamlessly transfer UML models of real-time systems between design and specialized analysis tools.

QUANTITATIVE MODELING

The majority of quantitative analysis techniques falls into one of the following two major categories:

- *Schedulability analysis* relies on a collection of primarily deterministic mathematical techniques for deciding whether a given real-time system will meet all its deadlines.[2,3] Schedulability analysis requires an accurate specification of the timing and processing load requirements for the job at hand. Because schedulability analysis produces deterministic results, it is particularly useful for hard real-time systems, where determining if even a single deadline may be violated is crucial. However, schedulability analysis applies primarily to systems with mainly static periodic loads.
- *Performance analysis* provides a set of analysis techniques based on queuing theory that can pre-

0018-9162/00/$10.00 © 2000 IEEE

dict response times, delays, and resource requirements for a variety of software systems.[4] Designers can apply these techniques to systems with dynamically changing loads, which are typically specified by probability distributions. Because the characterizations are probabilistic, performance analysis cannot be used to make definitive claims about deadlines, so it applies best to soft real-time systems.

Despite their limitations and the lack of qualified experts, designers use these and more specialized quantitative techniques extensively. We can safely assume that the variety, power, and use of quantitative methods for software design will increase.

UNIFIED MODELING LANGUAGE

Because most quantitative analysis techniques for real-time systems were developed before the advent of the object paradigm, does it make sense to focus this resource-modeling framework on UML, which is, after all, an object-oriented modeling language? Yes. UML is a good choice for several reasons.

- It is a widely adopted standard that is familiar to many software practitioners, widely taught in undergraduate courses, and supported by many books and training courses.
- Many tools from different vendors support UML.
- Most significantly, there is an excellent conceptual match between the object paradigm and real-time systems.

In contrast to procedural programming, which emphasizes algorithmic sequences, object-oriented programming uses a structure of collaborating parts or objects. Each part performs its specialized processing by reacting to inputs from its immediate neighbors. The superposition in time of these parts' localized behaviors, then, results in overall system behavior. In this model, structure is foremost because it literally provides a framework through which behavior flows.

This approach fits well with many real-time systems, which have a heavy structural aspect rooted in the interfaces that connect them to the external world. The real-time system's basic task is to transform external inputs into appropriate timely outputs—for example, translating the dialed digits of a phone call into a destination line address. Many concurrent flows may pass through the system, but the system's internal structure remains more or less constant. This structure-dominant style is evident even in many real-time systems developed well before the emergence of the object paradigm and object-oriented programming languages.

Since the Object Management Group adopted UML as a standard in 1997, the language has rapidly become a *lingua franca* across different domains. Although conceived as a general-purpose modeling language, UML includes built-in facilities that allow customizations—or *profiles*—for a particular domain. A profile fully conforms to the semantics of general UML but specifies additional constraints on selected general concepts to capture domain-specific forms and abstractions.

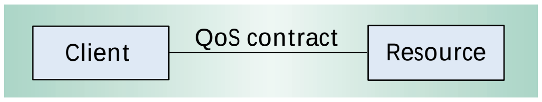


Figure 1. A QoS contract is a relationship between a resource and its client that incorporates explicit QoS specifications.

The OMG is currently working on a series of standard real-time profiles for modeling real-time applications. The first of these standards—expected to be adopted in 2001—will address the issue of modeling time and time-related facilities and services.[5] A primary requirement for this profile is the ability to construct UML models that can be analyzed using common quantitative techniques. At least two other standard real-time profiles are expected, one for modeling fault-tolerant systems and another for modeling the architectures of complex real-time systems.

The role of standards is crucial since they provide a common form for automatic exchange of models between design editing tools and specialized analysis tools. This automatic exchange eliminates the current error-prone and complex manual process of converting a design tool model into a model suitable for an analysis tool.

RESOURCE MODELING FRAMEWORK

Our resource modeling framework does not define any specific resources—such as semaphores, memory, and so on—but defines instead a generic approach for modeling such resources. The model therefore focuses on the notion of an *abstract resource*, which defines the common characteristics of resources regardless of their specific manifestation.

Quality-of-service characteristics

We use the term *resource* to denote any runtime entity for which the services can be qualified by one or more quality-of-service characteristics. We model a resource as a server with services that are characterized by both their functional and nonfunctional aspects, such as response time and availability. A QoS characteristic represents some aspect of the performance of a quantifiable service that generally reflects some underlying bounded quantities. Hence, QoS characteristics form the basis for true software systems engineering.

As Figure 1 shows, in the relationship between a client and a resource, there must be some correlation between the QoS the client requires—the *required QoS*—and the QoS the resource provides—the *offered QoS*. Ideally, the offered QoS should be at least as good as the required QoS, but in some applications clients may be willing to compromise and settle for something less. The relationship between a client and a resource that incorporates QoS characteristics is called a *QoS contract*.

QoS characteristics, whether required or offered, are typically more complex than simple numerical values. For example, they can be expressed as ranges or even probability distributions, and they can include the specification of policies to be applied when a QoS requirement can't be met.

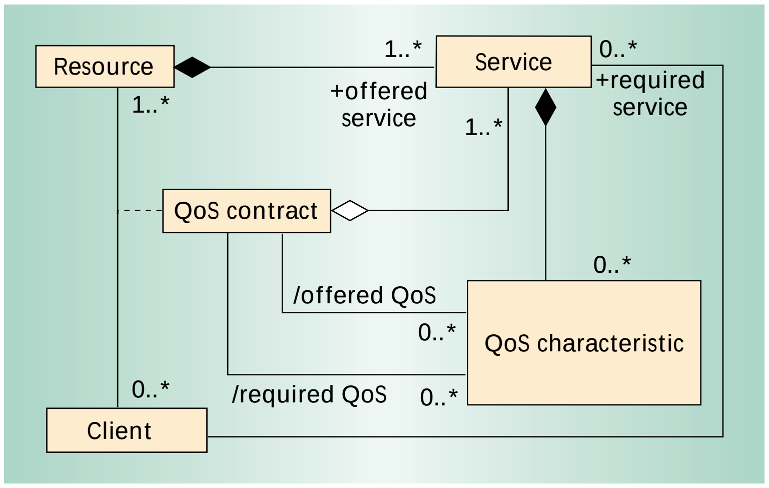


Figure 2. A generic QoS contract model. The resource provides one or more services that the client uses, and the system compares the required QoS service level with the corresponding QoS the resource offers to determine whether the QoS contract is feasible.

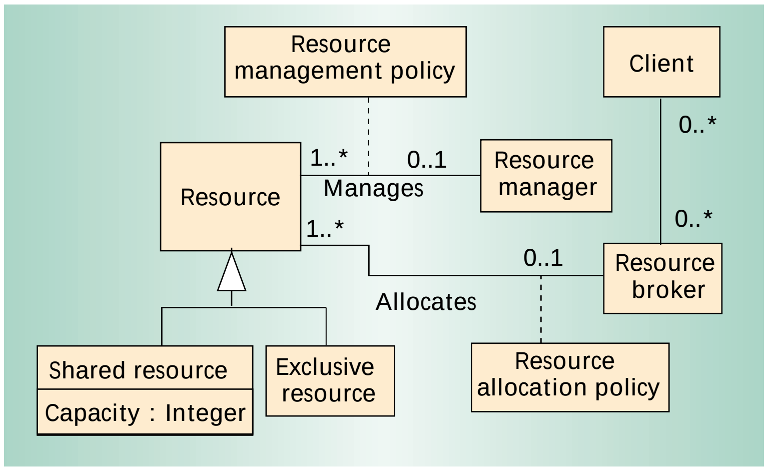


Figure 3. The management and allocation of resources. Clients obtain resources through a resource broker; a resource manager manages the resources according to specific policies.

The UML model in Figure 2 illustrates some of the finer details of the QoS contract concept. A QoS contract involves one or more services that a resource offers and that the client requires. The framework can define QoS characteristics for both the offered and required services. Since the two are related by the QoS contract, it may be possible to determine analytically whether the offered services can satisfy the client's QoS needs.

The UML diagram in Figure 3 depicts a second aspect of the generic resource model that describes how the system manages and allocates resources. In general, clients may need to access resources through *resource brokers*, entities responsible for allocating usage rights to resources according to some *resource allocation policy*. *Resource managers*, on the other hand, must manage resources, such as initialization and recovery, based on a *resource management policy*. Resource managers often act as resource brokers for the resources under their control.

A resource may itself be a client of other resources it needs for its functionality. This relationship means that a resource can have its own QoS requirements. In general, a resource can only guarantee its offered QoS to its clients if its resources meet its QoS requirements. Consequently, unless the resource does not require other resources, the offered QoS characteristics are always conditional.

Resources as engineering elements

Figure 1 shows a *peer interpretation* situation in which the client and the resource are distinct entities coexisting in the same environment. While the client will likely not function properly without the resource, its existence does not depend on the resource's existence, and vice versa. Figure 4, however, shows an alternative and equally useful view of the relationship between clients and resources—a *layered interpretation*. The two interpretations complement each other and can be used concurrently.

To understand the layered interpretation, it is useful to observe that we can model a software system from two different viewpoints:

- The *logical model* shows an object-oriented application as a mesh of collaborating objects. This model abstracts out the details of how a computer actually realizes the objects and their relationships.
- The *engineering model* describes how a particular technology implements logical model elements. This more implementation-oriented view of the application reveals a set of mechanisms such as memory, buses, and CPUs, which are the more concrete manifestations of the objects in the logical model.

The two viewpoints describe the same system but serve different purposes. We can use this dual view to separate our concerns about the logic of our software from its realization in a particular environment. The relationship between these two models is such that elements of the engineering model realize elements of the logical model. The mapping between a logical element and its corresponding realization, or engineering, elements is called a *realization mapping*.

In principle, we can change the engineering model without changing the logical model. This means that a logical model can adapt to changing technologies and different hardware configurations. However, for the two to be compatible, the new engineering model must meet all requirements of the logical model—notably the QoS requirements.

If we view the elements of the engineering model as resources and the elements of the logical model as their clients, we can apply the generic resource model shown in Figure 2. In this case, the realization mappings play the role of QoS contracts. We can use *layered interpretation* to achieve the objective of making the logical model independent of the engineering model while still accounting for the important quantitative aspects that influence the logical design. Thus, logical model elements must specify the target environment's QoS requirements, but don't need to specify the technology or internal structure of that environment.

The engineering model's resources are not necessarily hardware elements; they can also include software, as in buffers, operating system threads, and synchronization monitors. However, the actual hardware resources—such as memory, CPUs, and hardware communication channels—are also essentially abstractions. At one level of abstraction, we can view an engineering model as a logical model requiring its own engineering model—and so on, recursively, until we reach a level of "pure" hardware resources. Thus, no matter how complex the system or how numerous its resource layers, we can still apply the same generic model.

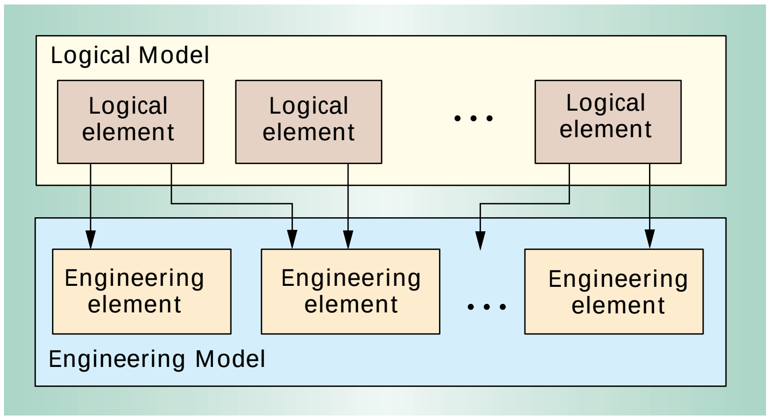


Figure 4. Two different viewpoints of a software system. The logical model captures the logic of the software of a particular technology independently, whereas the engineering model captures the realization of the logical model on a given technology.

Validity analyses and aggregation

QoS analysis compares the required QoS with the offered QoS, independent of whether you use peer interpretation or layered interpretation. In most cases, the offered QoS should equal or exceed the required QoS. What we deem equal or better depends greatly on

- the type of service under consideration—for example, availability, response time, or security; and
- the service's format—for example, simple number or probability distribution.

While this analysis may look relatively straightforward, it is greatly complicated when we consider *aggregated* QoS characteristics, a composite's overall QoS characteristics given the components' individual QoS characteristics. Several issues exacerbate this situation.

- *Different QoS characteristics combine in different ways*. For example, to determine the availability characteristics of a composite, we must not only know each component's availability characteristics but also whether the components are combined in series or in parallel.
- *Resources are often shared*. Sharing resources leads to contention and complex interference patterns between clients. For example, we might need to determine whether a given composite system can satisfy the maximum acceptable response requirements for a specific use case. If the use case involves a chain of interactions, or operation invocations, between individual components, we must determine the end-to-end delay from the initiating event to the last operation's completion. We need to know each component's duration. how the components map to resources (processing sites), and which other use cases (and their processing times and frequency characteristics) may be sharing the resources concurrently.

QoS aggregation presents a complex problem with no general solution. Certain techniques for specific QoS characteristics, performance modeling, and schedulability analysis apply only in special cases.

UML MAPPING

Although our model's general nature makes it independent of a specific modeling language, for our purposes we want to apply it to UML.

In general, any type of UML modeling element that directly or indirectly represents a runtime entity capa-

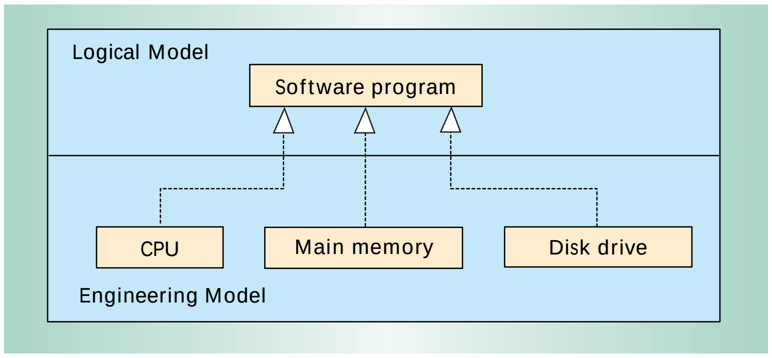


Figure 5. Realization mapping using UML. The logical model element software program is realized by three engineering model elements: a CPU, main memory, and a disk drive.

ble of performing services can model a resource. This category includes classes, classifiers, association classes, and their instances. These elements are clearly distinguishable as resources only because one or more of their features have offered QoS characteristics.

Offered QoS characteristics can be specified by constraints or tagged values attached to model elements that specify the behavior of a classifier: state machine transitions, operations, interactions, and so on. They can also be specified as attributes.

Because QoS characteristics form the basis for any type of quantitative analysis, we must specify the required QoS characteristics for elements of the logical model. In general, we can specify required QoS characteristics as constraints to model elements that specify behavior at runtime, including use cases, interactions, operations, state machine transitions, activities, and individual actions. We can also specify them for model elements with implicit services, such as CPUs. A CPU is a resource that, among other things, provides a processing service. Because its processing service is not accessed through explicit operations, the QoS characteristics are attached to the CPU as a whole.

Realization relationship

In UML, the general abstraction dependency "relates two elements or sets of elements that represent the same concept at different levels of abstraction or from different viewpoints."[1] The realization relationship is a specialized form, or stereotype, of abstraction that relates "a specification model element with the elements that implement it."[1] This is precisely the relationship between logical model elements and engineering model elements bound by realization mappings.

The semantics of realization relationships can be quite complex and highly dependent on the specific nature of the mapped elements. For example, mapping a logical object to a physical processor implies that the system must load that object's program component into the same processor, that the object is created and executes within that processor, and that its processing requirements combine in a very intricate way with the processing requirements of all other objects mapped to that processor. As Figure 5 shows, UML uses a special notation for the realization relationship.

In its most basic form, a realization mapping is merely a syntactical declaration that a particular resource supports a particular logical element in some unspecified way. The modeler must determine the realization's semantics and validity. More formal and more sophisticated forms—defined as standard stereotypes of the basic realization relationship—involve semantic knowledge of the nature of the logical and engineering model elements being bound. Specialized tools with that knowledge built in can compare the corresponding offered and required QoS characteristics to make statements about the model's validity.

We can use formal and semantically precise realization mappings to generate software that adapts a generic logical model element to a specific resource type. For example, either a CORBA or a COM channel can realize a communication link between two objects in a logical model. A code generator with knowledge of the two target resource types' semantics could automatically generate the appropriate implementation code for the two cases. This approach frees developers from getting involved with the complex implementation details of the various technologies and allows truly generic design.

Realization packaging

Frequently, logical elements can be bound to a set of resources in different ways. For example, we may want to experiment with several methods for partitioning a set of objects across a collection of physical processors to determine the allocation that optimizes performance. Doing so implies that a single logical model element may have multiple mutually exclusive realization associations. Further, two or more logical elements and their realization mappings may have mutual dependencies. For example, queue lengths may be a function of the specified CPU speed.

Clearly, we need to package together the set of realization mappings that represents a consistent set—mappings that are mutually compatible and nonexclusive. This *realization package* is conveniently modeled as a UML package. A given logical model can have any number of realization packages, each of which represents one distinct mapping of the logical model to exactly one engineering model.

The stringent demands typically placed on real-time systems have led to the development of quantitative techniques, such as performance modeling and schedulability analysis. This development has enabled software engineers to model systems and predict some of their salient characteristics before actually constructing them. As software becomes more instrumental to the everyday functioning of society, we can expect that the use and diversity of such techniques will increase. These encouraging signs indicate that software may be moving gradually toward becoming a bona fide engineering discipline.

UML, as an industry standard with widespread acceptance throughout the software community, provides an excellent opportunity to advance this highly desirable trend. Incorporating a generic QoS framework into UML gives us a standard way of producing quantifiable and, hence, precisely analyzable software models. In addition to bolstering the use of current quantitative techniques—for example, by allowing specialized analysis tools to import and exchange models—this capability will also provide a unified base on which to develop new techniques and methods. ✷

Acknowledgments

I thank my colleagues on the real-time profile submission team—Morgan Bjorkander, Bruce Douglass, Mark Gerhardt, Alan Moore, Jim Rumbaugh, Srini Vasan, and Ben Watson; as well as Murray Woodside of Carleton University, who provided useful feedback on the work described here. I also thank the anonymous referees whose excellent suggestions significantly improved this article.

References

1. Object Management Group, *The Unified Modeling Language Specification*, Nov. 1999, http://www.omg.org.
2. A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, 2nd ed., Addison Wesley Longman, Reading, Mass., 1997.
3. M. Klein et al., *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer Academic, Boston, 1993.
4. R. Cooper, *Introduction to Queuing Theory*, Macmillan, Indianapolis, Ind., 1972.
5. Object Management Group, *RFP for Scheduling, Performance, and Time*, OMG Document No. ad/99-03-13, Mar. 1999.

Bran Selic is a principal engineer at Rational Software Inc. in Kanata, Canada, and an adjunct professor at Carleton University in Ottawa. His experience covers real-time software, fault-tolerant distributed systems, and object-oriented modeling techniques. He holds a BS in electrical engineering and an MS in systems theory from the University of Belgrade. Selic is a member of the IFAC Technical Committee on Real-Time Software Engineering. Contact him at bselic@rational.com.