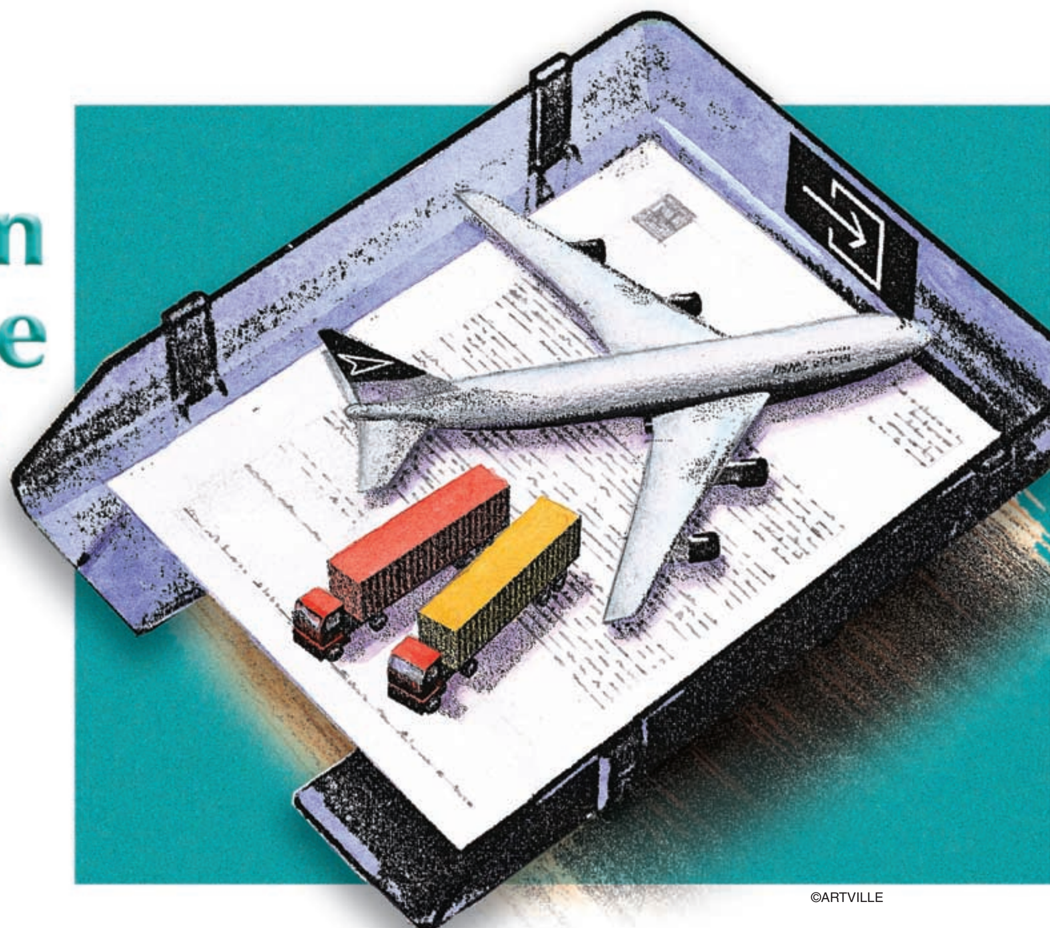
FEATURE

Using Models in Real-Time Software Design

Model-Driven Development Based on the Unified Modeling Language



©ARTVILLE

By Bran Selic and Leo Motus

Real-time software is rapidly gaining influence in the contemporary world; cars, phones, mobile phones, transport systems, air-conditioning systems, banking, supermarkets, and medical devices are prime examples. In most of these cases, the software is not visible; even the computers are not always visible. Nevertheless, in increasing numbers, the functionality of a service is, to a large extent, determined by real-time software.

Unfortunately, real-time software is particularly difficult to design. This is because, in addition to ever more complex functional requirements, real-time software has to satisfy a set of stringent *nonfunctional* requirements, such as maximum permissible response times, minimum throughputs, and deadlines. All too often, the inability of real-time software to meet its primary nonfunctional requirements becomes apparent only in the later stages of development. When this happens, the design may have to be heavily and hurriedly modified, even if all the functional requirements are satisfied, resulting in cost and schedule overruns as well as unreliable and unmaintainable code. This unhappy situation is primarily due to the common practice of postponing all consideration of so-called "platform issues" until the application logic of the software has been satisfactorily designed. Although "platform-independent design" is a good idea in principle, because it allows separation of application concerns and portability, it is often carried to the extreme. In particular, it is dangerous in situations where the physical characteristics of the platform can have a fundamental impact on the application logic itself. For example, a system with stringent fault tolerance requirements must incorporate control logic to deal with the failures that stem from the underlying platform. People with experience in designing high-availability systems know very well that fault tolerance is not something that can be retrofitted into an existing design.

Despite popular views to the contrary (see, for instance, [1] and [2]), software design in general, and real-time design in particular, is not very different in this regard from traditional engineering. It involves tradeoffs that balance desired func-

Selic (bselic@rational.com) is with Rational Software Canada, 770 Palladium Dr., Kanata, ON, K2V 1C8, Canada. Motus is with the Department of Computer Control, Tallinn Technical University, Tallinn, 19 086 Estonia.

0272-1708/03/$17.00©2003IEEE

tionality against various other nonfunctional concerns, such as the construction materials available (e.g., the processing and communication hardware), the anticipated workload, the type and capabilities of available tools, and so on. Both qualitative (functional) and quantitative (nonfunctional) issues and their interrelationships need to be considered.

The use of models as an effective means of understanding such tangled relationships is as old as engineering. A model helps because it removes or hides irrelevant details, allowing designers to focus more easily on the essentials. Good models not only facilitate the understanding of both problems and solutions, but they also serve to communicate design intent in an effective way. We often use them to predict the interesting properties of different design alternatives—and thereby minimize risk—before we go to the expense and trouble of building the actual system.

In contrast to models in most other engineering disciplines, software models have a unique and quite remarkable advantage: they can be directly translated into implementations using computer-based automation.

Although the use of models in software design is not new, it is still relatively uncommon and its effectiveness is often disputed. Indeed, experience has shown that models of software are typically difficult to construct and often unreliable. As our systems grow in complexity, however, traditional code-centric development methods are becoming intractable, and we have to resort to the kind of abstraction techniques that models provide. The semantic gap between the complex problem-domain abstractions of modern-day applications and the detail-level constructs of current implementation languages, such as C++ and Java, is just too great to be overcome by unaided human reasoning. For instance, a description of event-driven behavior expressed as a finite-state machine is far easier to define, understand, and explain than the equivalent C++ program. A key feature of models, therefore, is that they are specified using problem-domain concepts rather than computing-technology constructs.

In contrast to models in most other engineering disciplines, software models have a unique and quite remarkable advantage: they can be directly translated into implementations using computer-based automation [3]. Thus, the error-prone discontinuities encountered in other forms of engineering disciplines when switching from a model to actual construction can be avoided. This means that we are able to start with simplified models and then gradually evolve them into the final products. Models that can be evolved in this way have to be fully formal and, consequently, have the added major advantage that they are suitable for formal analysis. The potential behind software models and the maturation of automatic model translation techniques has increased the interest in model-oriented development methods. One notable example of this is the model-driven architecture (MDA) initiative originated by the Object Management Group (OMG) [4].

In this article, we review some major new developments in model-oriented software engineering in the real-time and embedded domains. The next section examines the general nature of software models and related techniques. Following that, the standard real-time Unified Modeling Language (UML) profile and associated developments are described. This is a prominent example of the model-oriented approach to software development. Finally, we present a short example illustrating the application of the profile to the common problem of determining the schedulability of a real-time system as well as a discussion on the timing analysis of interactions.

The Role of Models in Software Development

Whereas the elaboration of a software model can proceed smoothly in the same engineering medium and with the same tools, there is at least one transformation that may introduce discontinuities or misinterpretations. This transformation is related to capturing the initial information, constraints, and requirements from the user and the environment and compiling the first draft of the model. In the elaboration process of the draft model, various checks are applied to demonstrate that the captured information is consistent. However, the designer and/or user must always answer the question as to whether or not this information is sufficient to resolve the given task.

Pragmatically speaking, a software model is worthwhile if it fosters the efficiency and economic feasibility of the software process. Ideally, this can be done by demonstrating that the existing model contains consistent information and is coherent with the modeled object and by predicting properties of the product as early as possible, minimizing useless labor expenses for redoing erroneous and undoing unnecessary things. Assessment of consistency and prediction of properties is not always a straightforward process; quite often one has to build rather sophisticated (e.g., strictly formal) models and develop specific analysis methods.

Models used in software processes are often semiformal. Semiformal modeling methods smoothly transform the informal knowledge about the controlled object and goals of a software system into a model of software and thus represent pragmatic modeling. Semiformal modeling is comparatively

easy to learn for users and enables efficient manual, empirical checking of correspondence between the original informal knowledge of the modeled object and its model. Unfortunately, manual checking is too subjective for many dependable applications. Typical contemporary representatives of the group of pragmatic software models are built by using UML [3].

Usually, UML produces several interrelated models representing different views that capture the essential characteristics of the software that is to be developed, so as to provide good coverage of the complexity of the software and its multifaceted nature. Multiple models and views that describe different aspects of the same product have to be checked for cohesion and consistency. A *joint* formal analysis of information from several models and/or views is desirable to detect potential inconsistencies, to predict essential features of the future product, and to demonstrate cohesion between the required and predicted features. At present, such an analysis, especially at the early development stages, such as requirements specification, is usually done manually due to the semiformal nature of models and their interactions. Later in this article, an experimental formal model is described that is applicable at the early stages of software development.

This article departs from the experience obtained with informal use of UML and focuses on discussing possibilities for combining attractive features from semiformal and formal modeling techniques to increase the analyzing power of software models, especially for the case of quantitatively analyzable properties. For instance, it is important to perform quantitative timing analysis (not just qualitative ordering) during the requirements and system specification stages and scheduling analysis during the implementation stage of software development. In the context of real-time software, this means that, in addition to conventionally considered performance-related timing properties (e.g., response times and deadlines) and scheduling analysis (e.g., rate monotonic analysis), checks should be made of the cohesion and consistency of validity times for data, variable values, and events, as well as the time correctness of interactions.

The final goal of software modeling is automatic transformation from model to computer programs. This goal is a primary motivation for the MDA technology initiative. The central idea here is that models, created using modeling languages such as UML, should be the principal artifacts of software development instead of computer programs. Once a satisfactory model is constructed, automatic mechanistic processes can generate the corresponding computer programs. Thus, MDA and its objectives provide the new setting in which UML and its supporting OMG standards are evolving.

The generation of a computer program from a model is based on a series of transformations of the model, from the model of user requirements to that of a full-scale design, and then adjusting the design to a particular system software and hardware platform. To achieve user friendliness and wide usability of MDA technology, the models are often semiformal, and not all the transformations are necessarily formal (i.e., automatically truth preserving). It has been suggested that MDA technology allows separately developed and maintained formal models to be used for studying specific properties (typically related to quality of service, or "nonfunctional" requirements) of the software. For instance, schedulability analysis, performance analysis, and timing analysis of interactions should be performed on specific formal models that are based on information obtained from several UML diagrams [5]. Hence, the key problem is to ensure that the original UML description of the system includes all the necessary information for developing those specific models.

Although schedulability and performance analysis methods have been carefully studied for a long time, and well-established theory together with experience from practical application exists, timing analysis of interactions has not yet been widely accepted. The reason is that for a long time, the major goal of the software process has been to develop a functionally correct implementation of algorithms, assuming that algorithms capture the required transformation of input data correctly. Today, a rapidly increasing number of computer applications rely heavily on data-stream processing and on indefinitely ongoing computations (nonterminating programs) and are built from commercial off-the-shelf (COTS) components. An unexpected side effect of those new applications is "emergent behavior"—dynamically emerging behavior that cannot always be predicted from the static structure of the system.

Those new applications emphasize the importance of interalgorithm and intercomponent interactions in determining the computer system's behavior. First, theoretically strict attempts to handle the new type of computer applications were described in [6] and [7] by introducing an interaction-based paradigm of computations to extend the algorithm-based one. Two decades later it was demonstrated that UML provides an implementation of an interaction-based paradigm of computations [8].

Conventional engineering models do not normally demonstrate explicit emergent behavior—mostly because all the interactions between parts of a model are caused either by causal relations or by the static structure of material flows; both are persistent and defined by the laws of nature. In (real-time) software, one often has to deal with phenomena that can only be partly described with causal relations, either because one has incomplete information about the essence of such relations or because of insufficient computing power available for processing the complete knowledge. Typical cases leading to emergent behavior are illustrated by examples such as learning, decision making, commitments based on free will, and others that can lead to dynamic restructuring of interactions or radically change the previous functioning pattern.

At the same time, emergent behavior often has to meet strict predefined requirements and constraints, for in-

stance, behavior generated in real-time software by exception-handling subsystems. Conventional algorithm-based models of computations cannot explicitly study properties related to interactions and emergent behavior. Renewed attention and wider interest in interactions have come to light, partly in connection the with the wide practical acceptance of UML and the related MDA-based technologies. Interaction-centered models can be applied for deeper understanding of UML models [8]. Some details of the timing analysis of interactions are discussed in this article.

In the following sections, capabilities for automatic and semiautomatic analysis of software models are discussed. Automatic analysis becomes possible by using formal models that are not part of the UML environment but are derived directly from UML models based on the conventions defined in the standard real-time UML profile. Real-time software requires formal reasoning for many reasons, e.g., for certification purposes, for assessing fault tolerance, reliability, and other nonfunctional requirements, and, of course, for the timing analysis of interactions.

The original UML definition did not provide a standard means for exploring the quantitative aspects of many software systems.

Using UML for Modeling Real-Time Systems

The UML [5] was adopted by the OMG as a standard facility for constructing models of object-oriented software. Since its introduction in 1997, UML has been adopted quite rapidly and is now widely used by both industry and academia. However, although it has proven suitable for modeling the *qualitative* aspects of many software systems, the original definition did not provide a standard means for expressing the *quantitative* aspects of those systems. For example, in the real-time domain, it is often necessary to specify temporal constraints associated with specific elements of a model, such as the maximum acceptable duration of certain actions, necessary and available communication throughput rates, and time-out values. As a result, many different and mutually inconsistent methods were defined to include this information in a UML model (e.g., [9]-[15]). This type of diversity clearly diminishes some of the principal benefits of using a standard.

To remedy this, the OMG sought to supplement the original UML specification by asking for a "profile" that would define standardized ways for specifying temporal properties in a UML model [16]. (A profile is the UML term for a tightened semantic interpretation of the relatively general concepts in the standard, intended to meet the requirements of a particular domain. For example, a profile might specialize the general UML concept of a "class" to represent a domain-specific notion such as a clock.) This profile goes under the rather unwieldy name of the "UML Profile for Schedulability, Performance, and Time" [17] that, for convenience, we shall refer to simply as the "real-time profile." Note that, in addition to standardizing the modeling of time and timing mechanisms, the profile is intended to support formal analysis of UML models for certain time-related properties such as schedulability and various performance measures. Formal timing analysis, especially during the early stages of user requirements specification, was left out of the initial version of the standard, primarily due to the absence of a widely accepted theoretical basis and insufficient experience. Fortunately, it can be added later when more experience is obtained by inserting minimal modifications into the time model used by OMG [18].

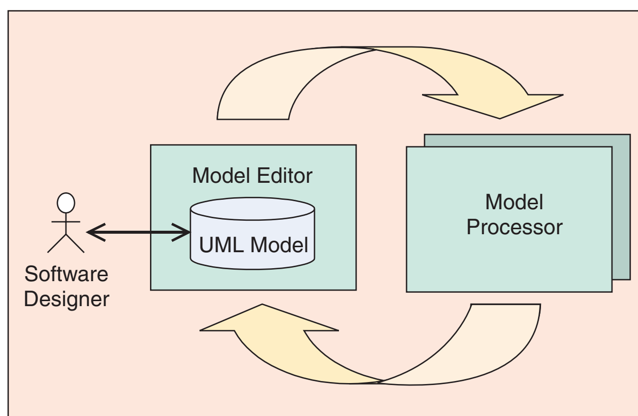


Figure 1. *The model processing paradigm.*

Profile Requirements

A principal requirement for the real-time UML profile was to allow the construction of predictive UML models; that is, models that can be formally analyzed to determine key quantitative characteristics of the proposed system, such as response times or system throughput. Based on such predictions, it is possible to consider and evaluate several alternative design approaches early and at relatively low cost. This, of course, is precisely how classical engineering uses models. In software engineering, however, the use of quantitative analysis methods, except for crude estimates mostly based on intuition, is still relatively rare. These methods tend to be quite sophisticated, and it is difficult to find experts with sufficient training and experience to apply them successfully.

A practical way around this problem is to automate the analysis process so that highly specialized computer-based

tools take on the role of the expert (see Figure 1). Software developers begin by constructing models of their designs using UML. Various model processors can then analyze these models, which capture the application logic as well as the associated quantitative characteristics. Typically, the model processor will first transform the model into a form suitable for analysis (e.g., a queuing network model), perform the analysis, and present the results. To be properly understood by application designers, the results should be presented in the context of the original model rather than the specialized analysis model.

This approach also enables an iterative design process wherein a candidate design is repeatedly refined and analyzed until the desired system characteristics are attained.

Note that a standardized modeling language is crucial to this scheme since it provides a common interchange format between various specialized tools that might be developed by different vendors.

Structure of the Profile

The main components and general structure of the real-time profile are depicted in the UML package diagram in Figure 2. It comprises several specialized profiles ("subprofiles") that can be used either together or separately.

The general resource modeling framework package contains several specialized frameworks that cover aspects common to practically all time- and resource-sensitive systems. At the core is an abstract conceptual model of resources (the RTresourceModeling subprofile). This introduces the fundamental notion of quality of service (QoS), which is the basis for specifying all the quantitative information used in quantitative analyses. In addition, there is a general facility for modeling time and time-related mechanisms (RTtimeModeling) and a facility for modeling concurrency (RTconcurrencyModeling).

This core set of frameworks is used to define specialized analysis subprofiles. The latter defines analysis-specific extensions, which are used to add QoS-related information to UML models so that they can be formally analyzed. The current standard defines three such specialized subprofiles, but it is open to additional ones.

- *SAprofile* supports a variety of established schedulability analysis techniques that determine whether or not a given system will meet all of its deadlines.
- *RSAprofile* is a specialization of the SAprofile for schedulability analysis of systems that are based on the OMG Real-Time CORBA standard [19].
- *PAprofile* supports various performance analysis techniques based on queuing network theory.

In addition to subprofiles, the standard also provides a model library that contains a prefabricated model of real-time CORBA. Models of applications based on this technology can reuse this library to save time and effort. This library is also intended as an illustration of the kind of information that will likely be considered mandatory when the software parts industry reaches maturity. In a direct analogy with the hardware components industry, where vendors provide VHDL models of their components, the intent is that vendors of commodity software (operating systems, code and class libraries, etc.) will supply UML models of their software.

The Conceptual Foundations: Resources and Quality of Service

As noted earlier, at least when it comes to real-time and embedded systems, software engineering is very similar to other forms of engineering; that is, in addition to functional correctness, it is also necessary to consider the quantitative correctness of the software. These are all, directly or indirectly, a function of the underlying hardware. (In a sense, the hardware represents the construction material out of which our software is made.)

An inherent property of the physical world (at least from the pragmatic perspective of engineering) is that it is finite

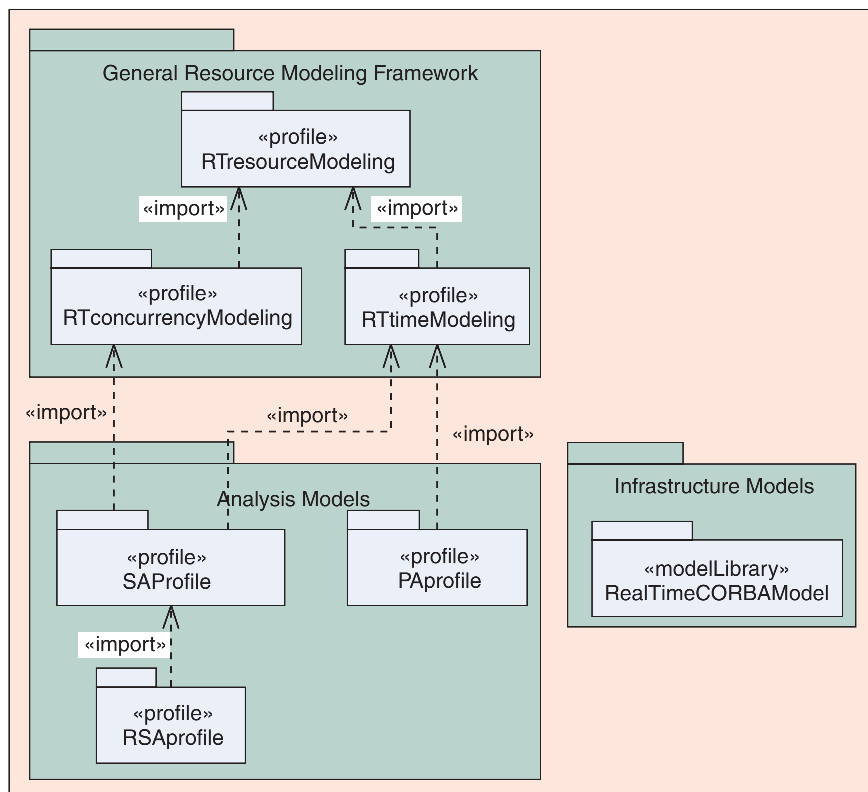


Figure 2. *The structure of the real-time UML profile.*

and, hence, restrictive in some way. For instance, even at computer speeds, real programs are not infinitely fast, bandwidth limitations constrain how much information can be transferred in a given time interval, the amount of memory and CPU speed at our disposal is limited, and so on. Things that are not in infinite supply are called resources in the standard. The notion of resource is the common basis for modeling all quantitative aspects of software systems. Note that resources do not necessarily have to be physical devices (e.g., message buffers, virtual circuits), but there is ultimately a physical underpinning in all cases.

In the real-time profile, a resource is modeled as a server that provides one or more services to its clients. The physical limitations of a resource are represented through its QoS attributes. In general, these attributes characterize either how well a resource service can be performed or how well it needs to be performed. We distinguish between offered QoS on the resource side and required QoS on the client side. Note that most quantitative analyses reduce to a comparison of these two sets of values, i.e., determining whether supply (offered QoS) meets demand (required QoS). Of course, even though the question is simple, its solution is not. In software systems there are many complex and dynamic interference patterns between clients competing for the same resources, which greatly complicate analysis.

The UML class diagram in Figure 3, as defined in the real-time profile, represents the conceptual model behind these fundamental notions of resources, services, and QoS. (Note that UML is being used here as a convenient means of representing abstract concepts, such as resources and services, and their relationships. Readers should be careful to distinguish such conceptual "domain" diagrams from the actual UML extensions (stereotypes, etc.) defined in the profile.)

Note that this model makes a fundamental distinction between descriptors, which are specifications of things that may exist at runtime, and instances, which are the actual runtime things. The relationship between these general concepts is exemplified by the relationship between a blueprint for a building (a descriptor) and an actual building constructed from that blueprint (an instance). Although this distinction is elementary and well understood, people often confuse the two concepts in practical situations. In general, UML concepts can be classified according to this distinction. For example, the concepts of Class, Association, and Action are all descriptor concepts in UML, whereas Object, Link, and ActionExecution are their corresponding instance concepts. In Figure 3, each of the elements on the right-hand side of the diagram is a kind of descriptor (to reduce visual clutter, the generalization relationship is only shown for Resource, although each right-hand element in the figure is a subclass of Descriptor). To the left of each of these is its corresponding instance type.

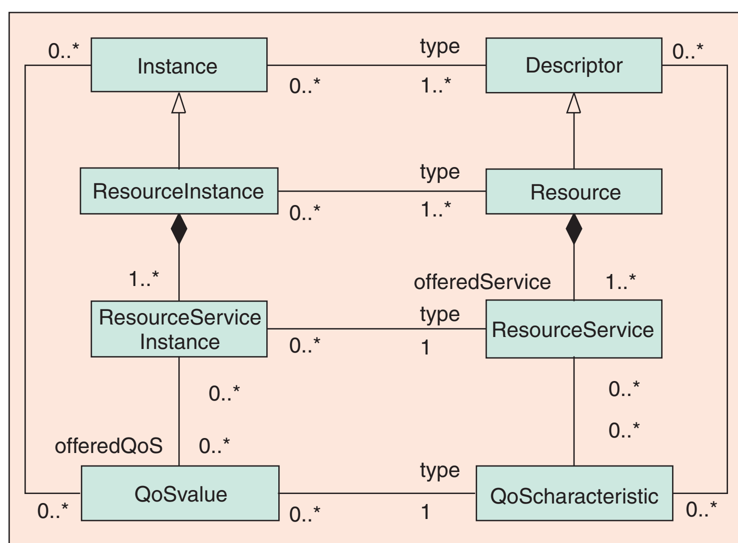


Figure 3. *The essential concepts of the general resource model.*

Understanding this descriptor-instance dichotomy is crucial, since most useful analysis methods are instance based; that is, they operate on models that describe situations involving instances rather than descriptors. For example, in performance analysis, it is necessary to know precisely how many instances are involved, what their individual service rates are, the precise interconnection topology between them, and similar detail. Note that two instances of the same class may perform very differently due to differences in their underlying processors, relative scheduling priorities, etc. Since UML class diagrams abstract out this type of information, they are generally inadequate as a basis for such analyses.

A specific situation in which a collection of resource instances is used in a particular way at a given intensity level is referred to as an analysis context. This represents a generic formulation of the standard problem that is addressed by the analysis techniques covered by the profile. The conceptual model of an analysis context is shown in Figure 4. Note that each specific analysis subprofile refines this abstract model to suit its needs.

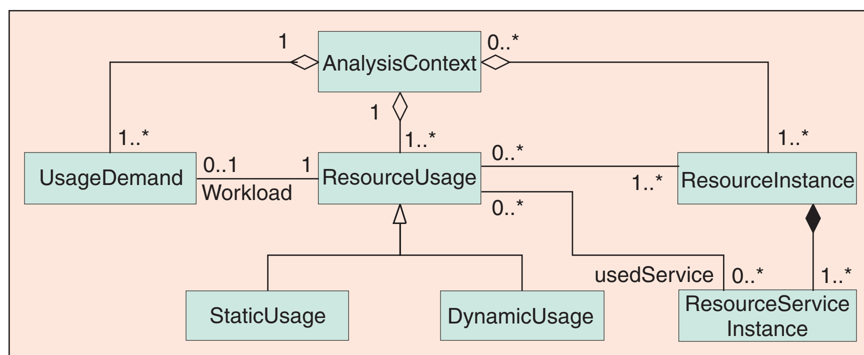


Figure 4. *The generic model of analysis situations.*

Each resource instance participating in an analysis context offers one or more resource service instances. As indicated in Figure 3, these may be characterized by their offered QoS values (e.g., response time). The clients of these resources and the details of how they use the resources are captured in the concept of a resource usage. This can be expressed in a number of different ways depending on the type of analysis desired. The simplest usage models are static, typically consisting of a list of clients matched against the resources they use. More sophisticated usage models may describe the dynamics of resource usage, such as the order in which the clients use resources, the type of access, and the holding time. The final element of this generic model is the concept of usage demand, which specifies the intensity with which a usage is applied to the set of resources (e.g., arrival frequency). The task of analysis is to determine if a given concrete model (involving specific resource instances, usages, demands, and explicit required and offered QoS values) is internally consistent or not. If not, this is an indication that demand exceeds supply and that a different model is required.

More details of the general model of dynamic resource usage are shown in Figure 5. A dynamic usage comprises one or more concrete scenarios—sequences of actions that involve accessing the resources and their services. These accesses will include specifications of the required QoS values (e.g., timing deadlines, maximum delays, throughput rates), which can be matched against the offered QoS values of the resources.

Note that an action execution is modeled as a kind of scenario. This allows an action at one level of abstraction to be refined into a full-fledged scenario in its own right, consisting of an ordered set of finer-grained action executions. The choice of abstraction level and level of decomposition is determined by users based on the level of accuracy desired from the analysis.

The real-time profile allows the modeling of many different kinds of resources in a variety of ways. The general taxonomy of resources supported is shown in Figure 6.

Resources are classified in different ways. Based on their primary functionality, they may be processors (devices capable of executing code), communication resources (for transfer of information), or general devices (e.g., specialized sensors, actuators). Depending on whether or not they require controlled access, they may be either protected or unprotected resources. Finally, depending on whether or not they are capable of initiating activity, they may be active or passive. The different categorizations can be combined for the same model element. For example, an element representing a physical CPU might be designated simultaneously as a processor resource that is both active and protected.

Pragmatically speaking, a software model is worthwhile if it fosters the efficiency and economic feasibility of the software process.

Modeling Time

The general UML standard does not impose any restrictions on the modeling of time. It neither assumes that time is discrete or continuous nor that there is a single source of time values in a system. This semantic flexibility allows different representations of time. This flexibility is retained in the real-time profile, although certain key concepts related to time are defined more precisely. The time model supported in the profile is shown in Figure 7.

In an abstract sense, physical time can be thought of as a relationship that imposes a partial order on events. Physical time is viewed as a continuous and unbounded progression of physical time instant*s*, as perceived by some observer, such that

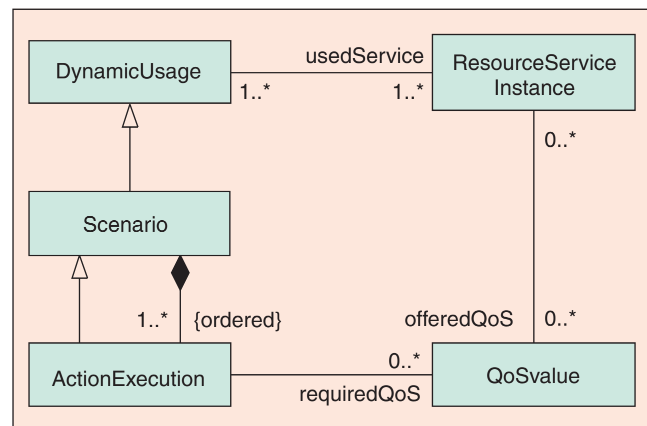


Figure 5. *The dynamic usage model.*

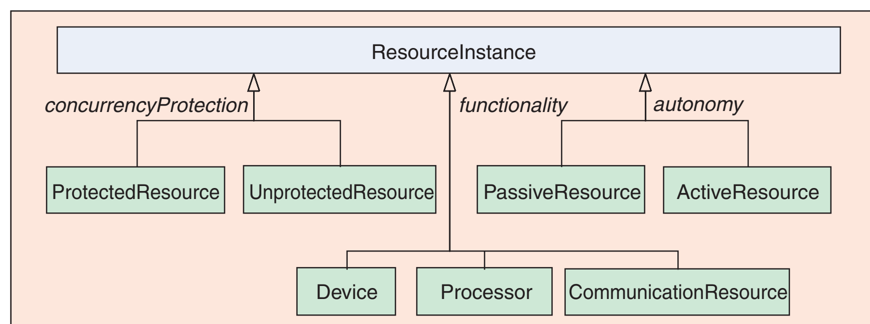


Figure 6. *The taxonomy of supported resource types.*

- it is a fully ordered set, which means that, for any two distinct elements of the set, p and q, either p precedes q or q precedes p
- it is a dense set, which is to say that there is always at least one instant between any pair of instants.

A distinction is made between dense time, corresponding to the continuous model of physical time, and discrete time, which represents time that is broken up into quanta. Dense time can be represented by the set of real numbers, whereas discrete time corresponds to the set of integers.

Physical time is assumed to progress monotonically (with respect to any particular observer) and only in the forward direction. Note that these restrictions apply to our model of physical time but not necessarily to other models of time that may be useful in modeling. For example, there exist "simulated" time models in which time does not necessarily progress monotonically or "virtual time" that may even regress under certain circumstances.

Since physical time is incorporeal, its progress is typically measured by counting the number of expired cycles of some strictly periodic reference clock starting from some origin. This way of measuring time necessarily results in a discretization effect in which distinct but temporally close physical instants are associated with the same count. However, this granularity is merely a consequence of the measurement method and not an inherent property of physical time (at least not in our conceptual model). In other words, any desired time resolution can be obtained simply by choosing a sufficiently short cycle time (resolution) for our reference clock.

The count associated with a particular instant is called its measurement. In the conceptual model, a time measurement is represented by a special value called time value. Time values can be represented by simple integers (discrete time values) or by real numbers (dense time values), as well as by more sophisticated structured data types, such as dates.

Duration is the expired time between two instants. Since this too is represented by time values, it is useful to be able to distinguish it from a time value that represents a specific instant. Hence, the profile introduces the semantic notion of a time interval.

Figure 7. *The conceptual model for representing time.*

In addition to the model of time, the real-time profile permits the modeling of two different kinds of timing mechanisms: clocks and timers. A clock is an active resource that periodically generates clock interrupts in synchrony with the progress of physical time. A timer is an active resource that detects the expiration of a specific duration after which it generates a time-out signal. The model of timing mechanisms is shown in Figure 8.

Figure 8. *The timing mechanisms model.*

Note that both clocks and timers are derived from a more general concept of an abstract timing mechanism. This captures the common characteristics of all timing mechanisms, such as resolution, origin, and maximal value.

Mapping to UML

So far, we have only discussed the conceptual domain model, but we have not indicated how this model is mapped to existing UML modeling concepts such as Class, Association, etc. This is achieved using the extensibility mechanisms of UML: stereotypes, tagged values, and constraints. A stereotype is used to define a specialization of a generic UML modeling concept. For example, imagine that in our application domain we need to model objects that represent real-time clocks. In addition to the standard properties of all objects, these real-time clock objects have the added feature that their value changes spontaneously with time. Hence, we need a modeling concept that adds these semantics to the basic UML Object concept. We can achieve this in UML by defining a "real-time clock" stereotype of the basic Object concept and attaching the domain-specific semantics to the stereotype. The stereotype may also include additional constraints and tagged values. The latter are used for representing additional attributes of the domain concept, such as the resolution of a clock.

An example of the use of stereotypes in practice is shown in Figure 9. Figure 9(a) simply shows an object diagram with two linked objects. Each is an instance of the general UML Object concept. However, we cannot determine from this diagram if any of these objects represents a clock (we cannot rely on class or object names, since users have the freedom to name and rename their classes as they see fit). To do that, we need to use the "RTclock" stereotype as shown in Figure 9(b). Note the use of the tagged value "RTresolution" to specify the resolution of this particular clock—this is, in fact, an offered QoS value. It is one of the additional characteristics of clocks defined for timing mechanisms in Figure 8.

The real-time profile consists of a collection of such stereotype definitions, each one mapping a domain concept to appropriate UML modeling concepts.

Model Analysis with the Real-Time Profile

Three different kinds of analysis subprofiles were mentioned previously. Due to space limitations, we will focus on just the schedulability subprofile, although some preliminary work for another, not yet developed, subprofile (timing analysis of interactions) is also briefly discussed in the following section. However, as noted in the discussion of analysis contexts, there is significant semantic similarity between all three subprofiles.

Schedulability Analysis

Schedulability analysis refers to the set of techniques used to determine whether a given configuration of software and hardware is "schedulable"; that is, whether or not it will meet its specified deadlines. This field is quite active, as a number of useful schedulability analysis techniques have been defined to date (see [20]-[23]) and new ones are constantly emerging (e.g., [24]).

The schedulability analysis subprofile defines a specialization of the general analysis context shown in Figure 4, with refinements of the basic concept. It introduces, among others, the following key stereotypes:

- «SASituation», a specialization of the "analysis context" domain concept for the purpose of schedulability analysis (this is a stereotype of the UML Collaboration and CollaborationInstanceSet concepts)
- «SASchedulable», a specialization of the "resource instance" domain concept for representing schedulable entities such as operating system threads (a stereotype of the UML Object and ClassifierRole concepts)
- «SAEngine», a specialization of the "resource instance" domain concept that represents either a virtual or physical processor capable of executing multiple concurrent schedulable entities/threads (UML Object, ClassifierRole)
- «SAResource», a different specialization of the "resource instance" concept for representing passive protected devices (UML Object, ClassifierRole)
- «SAAction», a specialization of the "action" domain concept defined in the dynamic usage model in Figure 5 (UML ActionExecution, Message, Stimulus)
- «SATrigger», a further specialization of the "action" concept that identifies the action that initiates a scenario (UML ActionExecution, Message, Stimulus)
- «SAResponse», a specialization of the "scenario" domain concept that identifies the sequence of actions initiated by a trigger; this stereotype has the tagged value "isSchedulable," which will be "true" if the scenario can be scheduled and "false" otherwise (UML ActionExecution, Message, Stimulus, Interaction).

A given real-time situation typically consists of multiple types of triggers and responses utilizing a shared set of resources and execution engines. One example of a fully annotated real-time situation is shown in Figure 10.

This example illustrates some of the key capabilities of the real-time profile. Consider, for instance, the message C.1:displayData () going from the real-time clock (TGClock) to the telemetry displayer. This element of the UML model is stereotyped in two ways: as a trigger («SATrigger») that initiates a complete response and as the response itself («SAResponse»). The latter choice may seem unusual, since

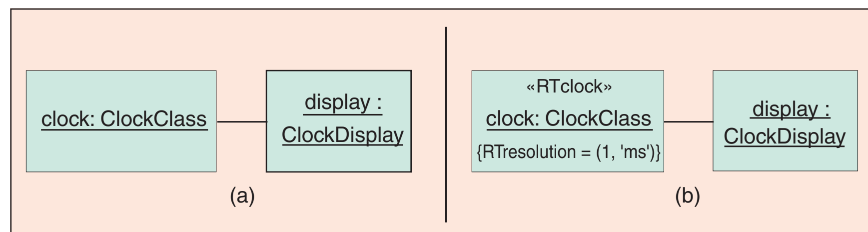


Figure 9. *The use of stereotypes to attach domain-specific semantics.*

the actual response consists of a sequence (scenario) of several messages (C.1, C.1.1, and C.1.1.1). However, since in this model no single element captures the full response, the triggering element is also used to represent the entire sequence.

The profile provides a sophisticated means for specifying time values. For example, the tag RTat for message C.1 has a value that specifies that the associated event is periodic with a period of 60 ms. It is also possible to define stochastic values that are represented by common probability distributions. If required, the time value can specify whether it represents a measured, estimated, or predicted value. It is even possible to define the value as a formula involving one or more variables.

Note also that the «SASchedulable» tag value for this trigger is defined as a variable ($R2). By convention, variables whose values are not specified represent the outputs that are to be calculated by the analysis. Hence, the appropriate schedulability model processor will insert the appropriate values (“true” or “false”) in the results model that it returns. The model processor recognizes which parts of the overall UML model it is expected to analyze by looking for appropriate contexts (in this case, the collaboration stereotyped as «SASituation».)

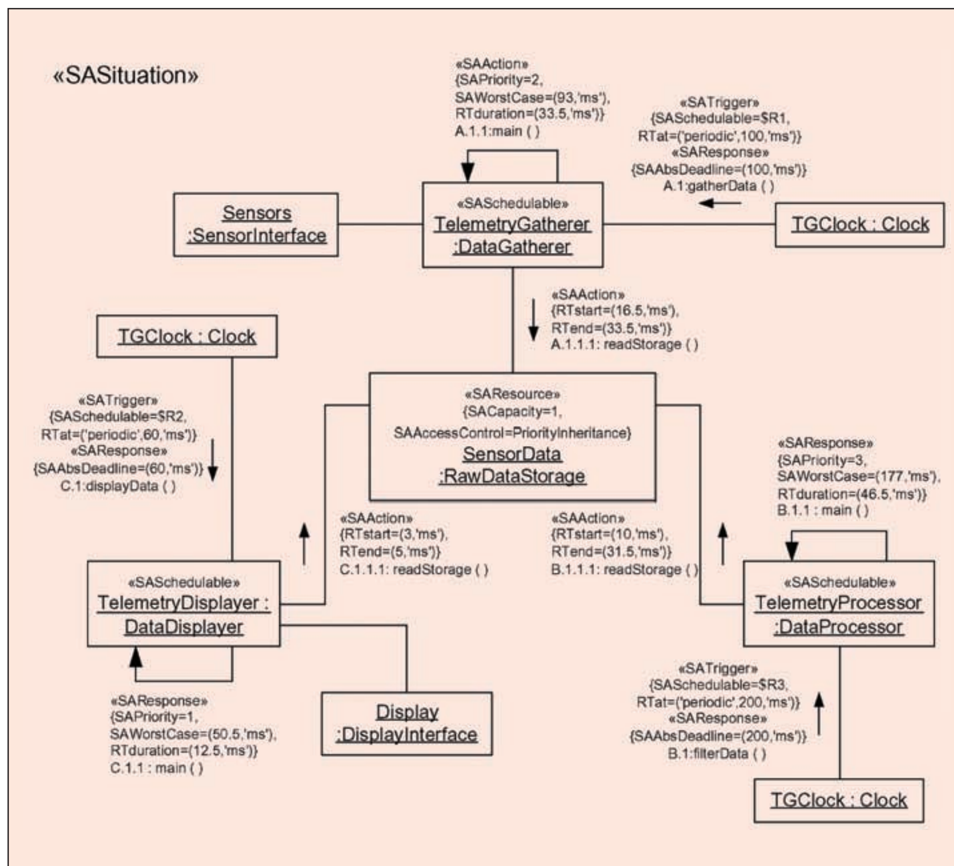


Figure 10. *Example of a real-time situation for schedulability analysis.*

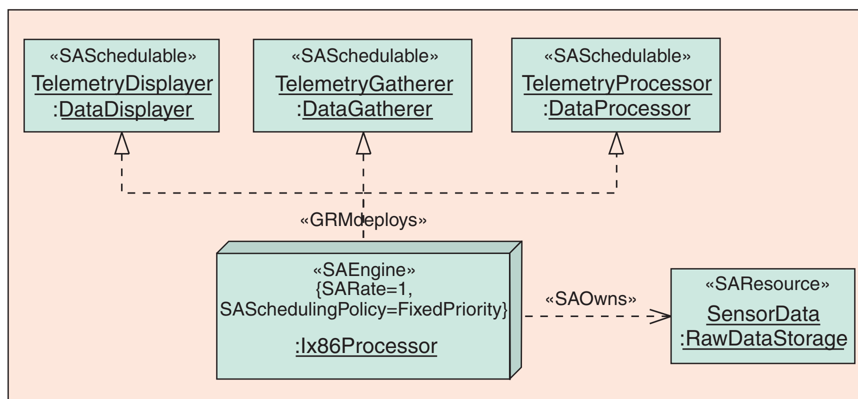


Figure 11. *A platform model and deployment specification for the system in Figure 10.*

When this situation model is combined with the model that shows how the software elements of the situation are mapped to the underlying platform (Figure 11), there is sufficient information for a schedulability analysis tool to do its job. (Note that the mapping of software elements to platform elements is indicated by special «GRMdeploys» stereotypes of the UML realization relationship.)

Timing Analysis of Interactions

The correct operation of real-time software assumes, in addition to correct algorithmic functioning of programs, the satisfaction of several QoS-related requirements. The majority of those QoS requirements qualify under the common denominator of timing. Timing analysis can be partitioned into three separate steps:

- performance analysis that is already part of the real-time profile
- scheduling and schedulability analysis, which was discussed in the previous section and which is also part of the real-time profile
- timing analysis of interactions that focuses on the timeliness of intercomponent interactions, the timeliness of interactions between the computing system and its environment, as well as the appropriate age of data and events in the system.

Untimely interaction and/or inappropriate age of data and events may cause violation of QoS requirements even if scheduling and performance requirements are satisfied. Just imagine a set of time-wise inconsistent values of state variables displayed to a human operator for decision making.

The timing analysis of interactions is essentially based on the use of interaction-centered models of computation. Despite the remarkable practical success of interaction-centered paradigms, for instance, the wide acceptance of UML, methods for the strict mathematical analysis of time behavior of interactions still need research and testing. Some of the problems in this area are discussed in [25]-[28]. A prototype for a UML model processor (see Figure 1) for timing analysis of interactions, based on the Q-methodology, has been developed and tested [27]. Since the testing was done in an OMT environment that does not include any timing requirements, the transformation from class model to Q-model was manual. The required time parameters, requirements, and constraints were inserted manually during the transformation. The time model in the real-time profile enables specification of the time parameters and requirements in UML models and may lead to straightforward (semi)automatic transformation to and from the Q-model. An interesting by-product of the Q-methodology is its ability to automatically generate a system's prototype for animation of the system's behavior in time with a choice of built-in or manually added scenarios at the early stages of system development.

Each computing agent in the Q-model is equipped with an individual system of times comprising a discrete strictly increasing time, a fully reversible relative time in each granule of the strictly increasing time, a relative time with moving origin for each in-agent and interagent interaction, and (maybe) a single universal time for a neutral observer [28]. It is also assumed that all the parameter and/or constraint values are given as interval estimates. A set of such individual time systems is a necessary precondition for reasoning about interactions and can easily be built based on the time model adopted by the OMG (and previously discussed in this article).

Typically, reasoning with interaction-centered models of computation implies the use of higher-order predicate calculus, in this case, a weak second-order predicate calculus [25]. The applicable analysis methods focus on demonstrating that previously proven theorems regarding certain universal properties of the application are retained. This minimizes the need to retrain software designers, keeps the user interface simple, and provides good scaling properties of the tool. Usually, the objective of analysis is to demonstrate that data and events used in the interaction are of the required age and that the interaction itself does not violate time constraints imposed on the consumer and the producer.

Some examples of properties that can be handled with the existing analysis are

- the age of data and delays caused by the interaction for cases where the producer and consumer agents are executed synchronously, semi-synchronously, or asynchronously
- estimation of the maximum potential number of parallel copies of an agent that are to be generated to satisfy time-related QoS requirements
- detection of informational deadlocks caused by the absence of data with appropriate age
- estimation of the maximum non-transport-related delays of messages due to the interaction of asynchronously executed computing agents.

The final goal of software modeling is automatic transformation from model to computer programs.

Conclusion

An idealized but widely held view is that software design should be an unadulterated exercise in applied mathematical logic [1], [2]. However, although logical correctness is clearly crucial in any system, it is not always sufficient—a system must also satisfy its nonfunctional requirements. Furthermore, as many examples of time-sensitive concurrent and distributed software systems indicate, there is often a critical interdependency between system functionality and the quantitative characteristics of the underlying platform. Quantity can indeed affect quality. For instance, bandwidth and transmission limitations may mean that status information required for decision making may be out of date, something that will have a fundamental impact on the design of control algorithms. These quantitative aspects are almost invariably a reflection of physical phenomena, leading us to the certain conclusion that the laws of physics must be understood and respected even in software design. Logic alone is not sufficient.

Models can play a crucial role in this process; early semiformal or formal analogs of the desired software product can be analyzed to predict the key system characteristics and thereby minimize the risks of software design. Numerous practical analysis methods are available, such as the schedulability, performance, and timing analysis methods discussed in this article.

This benefit of modeling is well known and is the reason it is the keystone of any true engineering discipline. However, when it comes to software, models offer an additional exceptional opportunity: the ability to move directly from model to end product by growing and elaborating the

model without error-prone discontinuities in the design medium, tools, or method. Such model-based methods of software development are becoming feasible, thanks to various advances in computing technologies and the emergence of software standards. A salient example is OMG's MDA initiative and the UML, along with its standard real-time profile.

These and similar developments will provide the basis for a mature and dependable software engineering discipline, despite several open research and technological issues.

References

[1] W-L. Wang, "Beware the engineering metaphor," *Commun. ACM*, vol. 45, no. 5, pp. 27-29, May 2002.

[2] E. W. Dijkstra, "Under the spell of Leibniz's dream," manuscript EWD-1298, Aug. 2000 [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1298.PDF

[3] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Reading, MA: Addison Wesley, 1999.

[4] Object Management Group, "Model driven architecture (MDA)," OMG document ormsc/2001-07-01, July 2001.

[5] Object Management Group, "Unified modeling language specification (version 1.4)," OMG document formal/2001-09-67, Sept. 2001.

[6] W.J. Quirk and R. Gilbert, "The formal specification of the requirements of complex real-time systems," AERE, Harwell, U.K., Tech. Rep. 8602, 1977.

[7] R.A. Milner, *A Calculus of Communicating Systems*. Berlin, Germany: Springer Verlag, 1980.

[8] D. Goldin, D. Keil, and P. Wegner, "An interactive viewpoint on the role of UML," in *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, K. Siau and T. Halpin, Eds., Hershey, PA: Idea Group Publishing, 2001, pp. 250-264.

[9] B. Douglass, *Doing Hard Time, Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Reading, MA: Addison-Wesley, 1999.

[10] H. Gomaa, *Designing Concurrent, Real-Time and Distributed Applications with UML*. Reading, MA: Addison-Wesley, 2000.

[11] L. Kabous and W. Neber, "Modeling hard real-time systems with UML: The OOHARTS approach," in *Proc. 2nd Int. Conf. Unified Modeling Language, UML'99* (Lecture Notes in Control Systems, vol. 1723). Berlin, Germany: Springer Verlag, 1999, pp. 339-355.

[12] P. Kähkipuro, "UML based performance modeling framework for object-oriented distributed systems," in *Proc. 2nd Int. Conf. Unified Modeling Language,UML'99* (Lecture Notes in Control Systems, vol. 1723). Berlin, Germany: Springer Verlag, 1999, pp. 356-371.

[13] A. Lanusse, S. Gerard, and F. Terrier, "Real-time modeling with UML: The ACCORD approach," in *Proc. 1st Int. Conf. Unified Modeling Language, UML'98* (Lecture Notes in Control Systems, vol. 1618). Berlin, Germany: Springer Verlag, 1998, pp. 319-335.

[14] B. Selic, "Turning clockwise: Using UML in the real-time domain," *Commun. ACM*, vol. 42, no. 10, pp. 46-54, Oct. 1999.

[15] B. Selic, "A generic framework for modeling resources with UML," *IEEE Computer*, vol. 33, pp. 64-69, June 2000.

[16] Object Management Group, "RFP for scheduling, performance, and time," OMG document ad/99-03-13, Mar. 1999.

[17] Object Management Group, "UML profile for schedulability, performance, and time specification," OMG document ptc/2002-03-02, Mar. 2002.

[18] Object Management Group, "Enhanced view of time (initial submission)," OMG document orbos/99-07-15, Aug. 1999.

[19] Object Management Group, "Real-time CORBA 2.0: Dynamic scheduling specification," OMG document ptc/2001-08-34, Sept. 2001.

[20] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, 2nd ed. Reading, MA: Addison-Wesley, 1997.

[21] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzalez Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston, MA: Kluwer, 1993.

[22] D. Niehaus, J. Stankovic, and K. Ramamritham, "A real-time system description language," in *Proc. 1995 IEEE Real-Time Technology and Applications Symp.*, May 1995, pp. 104-115.

[23] J. W. S. Liu, *Real-Time Systems*. Upper Saddle River, NJ: Prentice-Hall, 2000.

[24] J. Drake, M. Gonzalez Harbour, J. Guiterrez, and J. Palencia, "Modeling and analysis suite for real-time applications (MAST)," Grupo de Computadores y Tiempo Real, Universidad de Cantabria, Int. Rep., 2000.

[25] L. Motus and M.G. Rodd, *Timing Analysis of Real-Time Software*. Oxford UK: Pergamon/Elsevier, 1994.

[26] P. Lorents, L. Motus, and J. Tekko, "A language and a calculus for distributed computer control systems description and analysis," in *Proc. Software for Computer Control*, Oxford UK: Pergamon/Elsevier, 1986, pp. 159-166.

[27] L. Motus and T. Naks "Formal timing analysis of OMT designs using LIMITS," *Comput. Syst. Sci. Eng.*, vol. 13, no. 3, pp. 161-170, 1998.

[28] L. Motus and M. Meriste "Time modeling for requirements and specification analysis," in *Proc. 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming*, Poland, May 2003.

Bran Selic is a principal engineer at Rational Software Canada and an adjunct professor of computer science at Carleton University. He has more than 30 years of experience in industry in the design and development of large real-time systems. He is the principal author of a popular book that pioneered the application of object technology and model-driven development methods in real-time applications. Since 1996, he has participated in the definition of the UML standard and its standard real-time UML profile. He received his bachelor's degree in electrical engineering and his master's degree in systems theory from the University of Belgrade in 1972 and 1974, respectively. He has been living and working in Canada since 1977.

Leo Motus is professor of real-time systems at Tallinn Technical University, Estonia. His research is focused on the timing of interactions in software systems. He has published one book and over 100 articles and conference papers; his CASE tool LIMITS is used for practical studies of timing. He was editor-in-chief and is now the consulting editor of the *Journal on Engineering Applications of Artificial Intelligence*. He is on the editorial board of the *Journal of Real-Time Systems* and the *Journal for Integrated Computer-Aided Engineering*. He is a Member of the IEEE and a Fellow of the IEE.