UML™ Profile for Schedulability, Performance, and Time Specification

This OMG document replaces the previous Final Adopted specification (ptc/02-03-02). It is an OMG Final Adopted Specification, which has been approved by the OMG board and technical plenaries, and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by December 6, 2002.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*.

The FTF Recommendation and Report for this specification will be published on February 10, 2003.

ptc/02-11-01
OMG Adopted Specification

UML™ Profile for Schedulability, Performance, and Time Specification

Final Adopted Specification
November 2002

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause atDFARS 252.227.7013 OMG® and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB, CORBA, CORBAfacilities, CORBAservices, and COSS are trademarks of the Object Management Group, Inc. X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents & Specifications, Report a Bug/Issue.

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

OMG Documents

The OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

The OMG documentation is organized as follows:

OMG Modeling Specifications

Includes the UML, MOF, XMI, and CWM specifications.

OMG Middleware Specifications

Includes CORBA/IIOP, IDL/Language Mappings, Specialized CORBA specifications, and CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

Includes CORBAservices, CORBAfacilities, OMG Domain specifications, OMG Embedded Intelligence specifications, and OMG Security specifications.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- ARTiSAN Software Tools, Inc.
- I-Logix Inc.
- Rational Software Corp.
- Telelogic AB
- TimeSys Corporation
- Tri-Pacific Software Inc.

The team has sought the opinion of many experts in the real-time domain. The authors of this document would like to acknowledge in particular the contributions of the following individuals[1]: Hamish Blair (Marconi), Prof. Alan Burns (University of York), Bruce Douglass (I-Logix, Inc.), Sebastien Gerard (CEA/LETI), Prof. Michael Gonzalez Harbour (Universidad de Cantabria), Luciano Lavagno (Cadence Design Systems, Inc.), Doug Locke (TimeSys Corp.), Jim McGee (Rational Software), Miguel de Miguel (Thales), Prof. Dorina Petriu (Carleton University), Prof. Jean-Paul Rigault (Esterel Technologies), James Rumbaugh (Rational Software), Therese Smith (Air Traffic Software Architecture, Inc.), Francois Terrier (CEA/LETI), Srini Vasan (TimeSys Corp.), Prof. Murray Woodside (Carleton University), and Ken Zink (HyPerformix, Inc.).

1. Note that the views expressed in this submission do not necessarily represent the opinions of the individuals cited in this acknowledgement.

Introduction 1

1.1 Background and Purpose of This Document

In March of 1999, the Analysis and Design Platform Task Force of the OMG issued a request for proposal (RFP) asking for a UML profile for “schedulability, performance and time”.

The RFP called for “proposals for a UML profile that defines standard paradigms of use for modeling of *time-, schedulability-, and performance-related aspects* of real-time systems” that would:

- Enable the construction of models that could be used to make quantitative predictions regarding these characteristics.
- Facilitate communication of design intent between developers in a standard way.
- Enable interoperability between various analysis and design tools.

This is just the first part of a larger initiative created by the Real-Time Analysis and Design Work Group to develop a comprehensive solution to the modeling of real-time systems. Subsequent parts are expected to deal with more general quality of service aspects (i.e., beyond time-related issues, including availability, reliability, etc.) and also with complex real-time systems.

In response to this RFP, a group of OMG member companies formed a working consortium for a joint response to this RFP. The group consists primarily of vendors of different kinds of real-time tools.

1.2 Proof of Concept

The essentials of the approach described in this document have been validated through several prototypes that involved automated interworking between analysis tools (from Tri-Pacific and TimeSys) on one hand, and UML modeling tools (from Artisan and Rational) on the other. The results were successful and were demonstrated at a number of real-time and embedded system trade shows and conferences.

In addition, the major elements of this profile have been validated by applying them to a variety of detailed and representative examples. The examples were defined and solutions to them developed by a team whose expertise covered fully the key technical areas encompassed by the document: the UML metamodel, real-time domain modeling, and schedulability analysis.

1.3 Compliance Statement

Compliance with this specification is defined as compliance with any of the following packages:

- The SAProfile sub-profile
- The PAProfile subprofile
- The RSAProfile subprofile

Compliance with a package implies complying with all prerequisite packages.

Rationale and General Principles 2

In this chapter we will look closer at the motivation behind the specification and some of its major goals.

Since the adoption of the UML standard, it has been used in a large number time-critical and resource-critical systems. (A significant number of these can be found in the various books, papers, and reports listed in the Bibliography at the end of this specification.) Based on this experience, a consensus has emerged that, while a useful tool, UML is lacking in some key areas that are of particular concern to real-time system designers and developers. In particular, it was noticed that the lack of a quantifiable notion of time and resources was an impediment to its broader use in the real-time and embedded domain. Therefore, as a general rule, we focused our attention on these areas.

Fortunately, and contrary to an often expressed opinion, in our work we discovered that UML had all the requisite mechanisms for addressing these issues, in particular through its extensibility faculties. This made our job much easier, since we did not find it necessary to add new fundamental modeling concepts to UML – so-called “heavyweight” extensions. Consequently, our job consisted of defining a standard way of using these capabilities to represent concepts and practices from the real-time domain.

2.1 Modeling Real-Time Characteristics in UML

Over time, a number of different modeling techniques and concepts have emerged within the real-time software community. Each technique has fostered its own diverse set of terminology and notations. One of the intentions behind this specification is to provide a common framework within UML that fully encompasses this diversity but still leaves enough flexibility for different specializations. In particular, we focused on properties that are related to modeling of time and time-related aspects such as the key characteristics of timeliness, performance, and schedulability.

The ability to predict these characteristics based on analyzing *models* of software—including notably models that are constructed prior to a line of code being written—is a fundamental objective of this specification. Accurate and trustworthy

predictions invariably involve formal quantitative analyses. Rather than relying exclusively on intuition and “feel”, an all too common and regrettable practice in software development, it is far better to rely on mathematically derived results stemming from accurate models. Problems detected early on during the development life cycle can be removed at a much lower cost, and with substantially less rework.

Therefore, we invested special effort in defining modeling capabilities that enable predictive quantitative analyses, such as the ability to determine the schedulability of a planned piece of software. A large part of this is based on the ability to model quality of service aspects, such as deadlines and priorities.

Note, in particular, that we are not inventing new model analysis techniques as part of this specification. Rather, the intention is to be able to annotate a UML model in such a way that various existing and future model analysis techniques will be able to take advantage of the provided features. There are, of course, additional benefits as well; the specification enables the communication of design intent between developers in a standard way, and further allows inter operability between tools utilizing the information (such as between various kinds of model analysis tools and design tools).

The concept of predictability associates parametric understanding of an implementation with functional requirements that are met by the system under consideration. Systems are typically designed and refined into elements exhibiting functional coherence. Each of these functions must then be scrutinized regarding its performance as implemented; this is typically done by executing test cases and measuring the results. In addition to after the fact measurements, model analysis and simulation are both used to provide indications or predictions about future performance. A combination of measurements, simulation, and model analysis is extremely meaningful to determine the quantitative relevance of a system implementation against its performance expectations.

Measurements provide insight and quantitative help. The ability by which measurements can be extrapolated into performance projections is not as straightforward as it seems. There is a continuum of understanding between measurements, statistical prediction, and guaranteed response.

Predictability refers to the quantitative assessment about an implementation of some expected functionality. These quantities refer to all aspects of a system - for example the amount of heat that it generates, the power that it consumes, etc.

One of the more predominant areas of discussion relating to predictability refers to timing and timing behavior of systems. Since we're dealing with real-time, the focus with which we discuss predictability centers on timeliness - e.g. execution prediction, timing, and expectations.

2.2 Guiding Principles

In putting together this specification, we adhered to a number of general principles to ensure a systematic and consistent response. These particular principles were chosen because they closely reflect the requirements of the RFP and because they embody the essential elements of a vision shared within the team regarding the future of real-time software development.

The main guiding principles are as follows:

- As much as possible, modelers should not be hindered in the way they use UML to represent their systems just to be able to do model analysis. That is, rather than enforcing a specific approach or modeling style for real-time systems, the profile should allow modelers to choose the style and modeling constructs that they feel are the best fit to their needs of the moment.
- It must be possible to construct UML models that can be used to analyze and predict the salient real-time properties of a system. In particular, it is important to be able to perform such analyses early in the development cycle.
- Modelers should be able to take advantage of different types of model analysis techniques without requiring a deep understanding of the inner workings of those techniques. (The steep learning curve behind many of the current model analysis methods has been one of the major impediments to their adoption.)
- The profile must support all the current mainstream real-time technologies, design paradigms, and model analysis techniques. However, it should also be fully open to new developments in all of these areas.
- It should be possible to automatically construct different analysis-specific models directly from a given UML model. Such tools should be able to read the model, process it, and feed the results back to the modeler in terms of the original UML model.

2.3 How This specification is to be Used

We assume a number of use cases for how we expect the profile to be used. In so doing, we rely on the following actors:

- The *modeler*: This is the category of system analysts, software designers and developers who construct UML models, and who would like to analyze these models in order to determine whether they will meet their performance and schedulability requirements.
- The *model analysis method provider*: These are the individuals and teams who are responsible for defining model analysis methods such as RMA or queuing theory, as well as tool vendors providing tools and processes for supporting particular model analysis methods.
- The *infrastructure provider*: These are the developers and vendors of run-time technologies such as Real-Time CORBA, real-time operating systems, real-time component libraries, etc.

The use cases that we will consider are summarized in Figure 2-1.

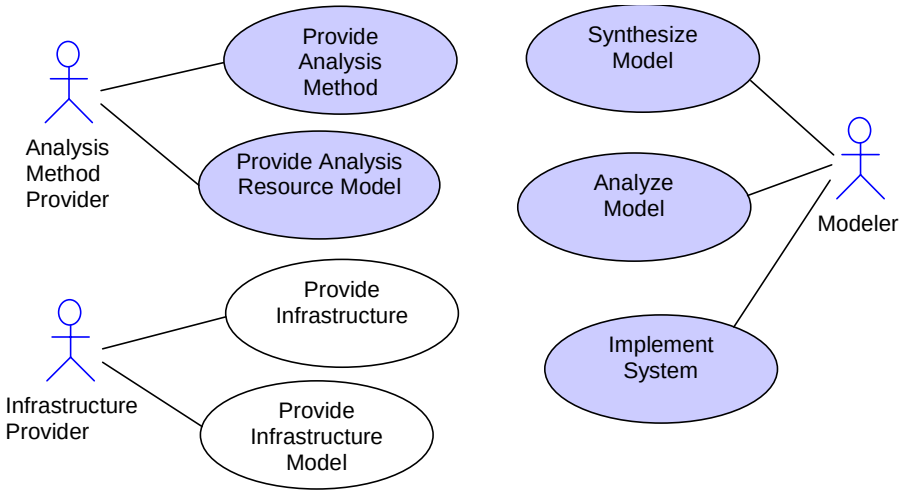


Figure 2-1 Some Use Cases Describing How to Use The Profile

Of course, neither the actors nor use cases described in this section represent an exclusive set for how this specification can be used, but rather reflect on some of the ways that we expect it to be used or (in most cases) expanded.

2.3.1 Provide Analysis Method

- Actor: analysis method provider
- Description: The provision of a new model analysis method includes a description of how it works and which attributes are essential. Furthermore, it needs to be tied in with the general resource model. It is advantageous if the model analysis method can be automated.
- Deliverables: The outcome of this use case is a model analysis method, which includes a process and a set of required techniques that applies to the model analysis method.

2.3.2 Provide Analysis Resource Model

- Actor. analysis method provider
- Description: It is the task of the analysis method provider to supply the relevant stereotypes and tagged values that are required to adequately model a specific model analysis method to make it analyzable. Note that this may provide several levels of fidelity, from very simple analysis to very detailed. Some base packages are provided as part of the specification, but it is expected that others will be defined.

- Deliverables: The outcome of this use case is a package containing the appropriate stereotypes to be able to annotate a UML model according to the given model analysis method. This package will import and likely provide a specialization of the basic resource model.

2.3.3 Synthesize Model

- Actor: modeler
- Description: A modeler synthesizes a model, iterating it through several stages, many of which may produce an analyzable model. Such a model uses the stereotypes and tagged values from the appropriate analysis method package, and possibly also from a chosen infrastructure package. Some basic property manipulation, such as aggregating up the actual work of composite actions may be automated by the modeling tool.
- Deliverables: The result of synthesizing the model is a system model, the eventual state of which is reached when the application is deemed sufficiently detailed to satisfy the project demands.

2.3.4 Analyze Model

- Actor: modeler
- Description: A model may undergo a number of different analyses, perhaps based on different model analysis methods, and certainly with varying levels of detail and completeness. In order for the model to be analyzed it needs to be well formed, i.e., model analysis must be possible based on the given information. An analysis tool provides results in a form that is not specified by the specification, and may suggest updates to the model based on those results.
- Deliverables: The primary result of analyzing a model is a set of discrepancies between offered and required resource attributes. Suggested changes to the model is a secondary result, and may offer either recommendations that will make the model correct or updates to the offered resource attributes.

2.3.5 Implement System

- Actor: modeler
- Description: At some point, the model has served its purpose, and it is necessary to create the actual running application. This may involve applying some infrastructure package. In order to verify that the model analysis is still valid, it may be necessary to provide a physical model of the system (including components and nodes) that can be checked against the earlier model analysis results.
- Deliverables. The result of implementing the system is the actual (real) system.

Approach and Structure 3

The guiding principles described in the previous chapter can be realized in a number of different ways. In this chapter, we describe the particular set of design strategies (approaches) that we have adopted in this specification as well as the overall structure of the profile itself.

3.1 Approach to Modeling Real-Time Applications

The term “real-time” is applied to a very broad spectrum of diverse categories of software systems including soft real-time systems and hard real-time systems, time-driven systems and event-driven systems, distributed systems and centralized systems, fault-tolerant systems and non-fault-tolerant systems, and so on. This has led to a large variety of different design styles and modeling approaches. Hence, there is no single canonical set of modeling and design concepts that will satisfy this tremendous diversity.

For example, in fault-tolerant real-time systems, it is often necessary to explicitly model major elements of the operating system infrastructure on which the real-time application is founded. This is because it may be necessary for the application to directly manipulate this infrastructure; for instance, to restart individual processes or processors, to dynamically load new software modules, or to relocate them on another site.

What is the “right” way to represent an operating system process in those circumstances?

The answer clearly depends on the level of detail that is required by the application. In some cases, the OS process may be represented by something as simple as an object, while in others it may be necessary to model it in greater detail including its heap, stack, and address space.

Therefore, the overall approach taken in this specification is *not* to pre-define a canonical library of “real-time” modeling concepts. Instead, we leave modelers the full power of UML to represent their real-time solutions in the way that is most suitable to their needs.

3.2 Approach to Modeling for Model Analysis

Unfortunately, the flexibility provided by the above approach makes it rather difficult to support the kinds of analyses that are required by the need for predictability. This is because most model analysis methods use a much simplified representation of a software system that focuses only on those aspects that are relevant to the model analysis. Thus, when an application model is presented for analysis, there is a need to reduce its inherent complexity to a small number of base abstractions.

One of the problems in achieving this is that the model analysis concepts rarely map one-for-one to application-level modeling concepts. That is, the same model analysis concept may be manifested in a variety of different ways within a single application model. For instance, in schedulability analysis, a key abstraction is the notion of a unit of scheduling, representing some work item that requires the services of a CPU. However, application models are rarely constructed to show the units of scheduling explicitly. Instead, they are implied by the presence of model elements such as active (concurrent) objects and asynchronous messages.

This is further exacerbated by the fact that different model analysis methods focus on different aspects of the model. In schedulability analysis, for example, thread priorities are of primary concern but are irrelevant for calculating program memory requirements. We will refer to these different perspectives as *model analysis views*. A model analysis view is a simplified version of the complete model and is extracted on the basis of a particular *model analysis* or *domain viewpoint* representative of a specific model analysis method (e.g., the “schedulability analysis viewpoint”).

Traditionally, domain viewpoints were the prerogative of model analysis method experts. With their deep understanding of a particular model analysis technique, they could examine the application model and extract the corresponding analysis model. However, this required highly specialized and usually scarce expertise. As a consequence, these highly useful methods were not used very often in practice, leading to many preventable software disasters.

To overcome this problem, we have provided a single unifying framework that captures the common elements of different real-time specific model analysis methods. This framework captures all the essential patterns used in deriving time-based analysis models from application models. The core of this framework is a common model of resources (of all kinds) and their quality of service (QoS) attributes. This core model is called the *general model of resource usage* or, *general resource model*. It is described in the following chapter.

How is this framework to be used?

The intent is for every different model analysis method to start with the concepts from the general resource model (and, thereby, gain the benefit of a proven approach) and then to specialize them according to the needs of the domain. The result of this is a *conceptual domain model*. Note that *these conceptual models are non-normative*. Their purpose is twofold:

- to define and explain the key concepts and relationships of the domain as well as their relationship to the general resource modeling framework and

- to serve as a guideline for defining the UML corresponding extensions (stereotypes, tagged values, and constraints)

The relationship of the various models and their components is illustrated in Figure 3-1. The conceptual models are on the left while the actual normative UML profile elements are shown on the right. The conceptual models are rendered in the form of UML package and class diagrams, with the classes in those diagrams representing domain concepts. The normative extensions packages, on the other hand, consist of UML stereotypes that map to the corresponding domain concepts.

Specific domain model elements, such as AnalysisResource[1] in Figure 3-1 capture analysis-specific notion of a resource, but it is presented as a specialization of the more general notion of Resource in the general resource model. In the corresponding profile (ADprofile), it is represented by the stereotype ADresource, which can be applied to either Class-type elements or Object-type model elements in a UML model. (The ability to apply the same stereotype to different kinds of model elements is a common technique used in this profile to achieve maximum flexibility in modeling real-time systems.)

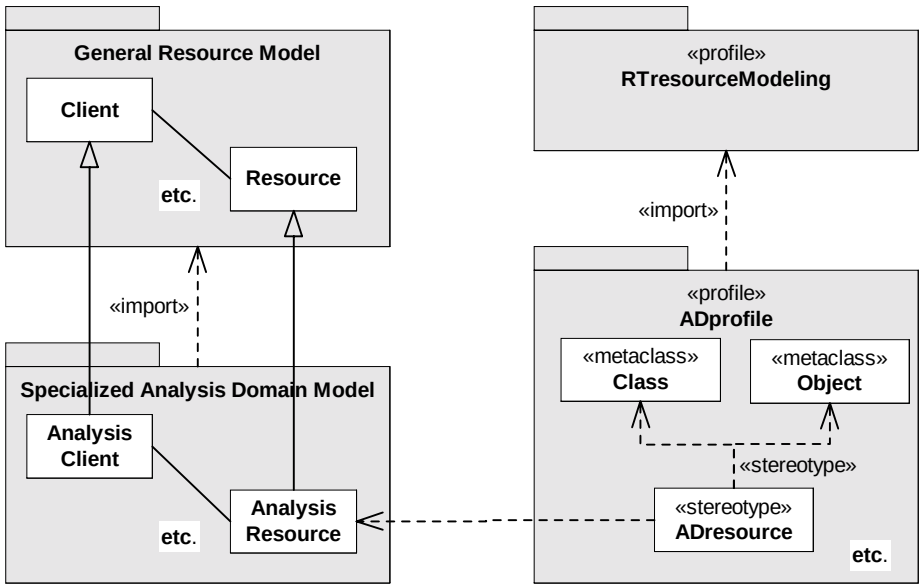


Figure 3-1 Domain models and corresponding profiles and the mappings between them

Note that not every domain concept will result directly in a UML stereotype or tagged value. This is because many domain concepts are abstract, representing generalizations that will not appear directly in any analysis model. For instance, the abstract notion of a “QoS characteristic” is very useful as an abstraction in our framework, but will only be manifested in its concrete forms (delay, throughput, capacity, etc.) in analysis models. While a corresponding stereotype could have been defined for this abstract concept, it

1. Note that this is just a hypothetical analysis domain defined purely for illustrative purposes.

would never be used in practice. Therefore, we have chosen to only define stereotypes for concepts that we envisage are actually going to be used in practical model analysis situations. This results in a simpler and more compact profile.

The intent is also to allow further specializations of the individual model analysis sub-profiles where this is appropriate. For example, in this specification, we have provided a specialization of the schedulability analysis sub-profile specifically for Real-Time CORBA (refer to the *Real-Time CORBA Applications* chapter).

The separation of the domain model from the UML equivalent is reflected in the structure of the chapters in this document that describe individual domains. Each such chapter is partitioned into two sections: a “domain viewpoint” and a “UML viewpoint.”

3.2.1 Modeling Resources

The modeling of resources is fundamental to this specification, and used as the basis for most other packages, including the other ones that are part of the common base. The general resource model specifies patterns that are present in many real-time model analysis methods, and defines a common terminology and conceptual framework that are intended to remove ambiguities arising for lack thereof.

The quantification of the constraints that apply to a real-time system is of great concern, and therefore QoS characteristics are an integral part of the resource model. However, such constraints are meaningless unless put in some context where there are resources and specific statements of intended loads (demands) on those resources. The model analysis problem is then reduced to comparing the demand (required QoS) against the offered QoS of the resources.

We distinguish between two ways of looking at the resource model. In the first, the so-called *peer interpretation*, a client and its used resource coexist at the same computational level. In this situation, quality of service is often compared using associations.

The *layered interpretation* is structurally very similar, but appears in a different context where a client (such as an application) is related to the resources that are used to implement it (such as the software and hardware environments used). Thus the client and the resource are not really co-existing, but rather two complementary perspectives of the same modeling constructs. We generically refer to these levels as the *logical model* and the *engineering model*, where the engineering model realizes the logical model. In this case, the client is an element in the logical model and the resource is a corresponding element in the engineering model.

The quantitative constraints that apply in the layered interpretation are similar in nature to the ones that appear in the peer interpretation, but are usually applied to realizes dependencies rather than to associations. Note that the engineering model may in turn be realized by yet another engineering model, in which case it would also be regarded as a logical model.

3.2.2 Modeling Time

Time is of course an ever-present aspect of real-time systems, and it has impact in several different areas. Only metric time is covered as part of this specification, as real-time systems are usually concerned only with the cardinality of time, such as the deadline for a response. In particular, we find it useful to distinguish between physical (continuous) time and simulated time where time does not necessarily increase monotonically.

Another case where time comes into play is in conjunction with services provided for example by a real-time operating system in the form of timers and clocks, and the specification provides a framework for modeling these facilities and their related characteristics.

Timing patterns of different kinds are essential in supporting schedulability and performance analysis, and include modeling whether something is periodic or not, and in the former case also modeling of period, distribution functions, and jitter.

3.2.3 Modeling Schedulability

Real-time systems comprise systems where the question of when a response to an event occurs is as important for the correct behavior of the system as its algorithmic behavior. A common distinction in this regard is between soft and hard real-time, the difference being that in the former a late reply is a good reply as long it is within some acceptable range, while in the latter case a late reply is useless, and sometimes totally unacceptable (fatal).

In the former category, statistical prediction is a way to model the relevant characteristics, which can usually be quantified as average performance and standard deviation. In the latter category, the problem is usually in making sure that there is an upper bound for response to a stimulus, i.e., there has to be a guaranteed response time. Actual systems often combine traits of both categories.

As part of the specification, we particularly focus on systems having hard timeliness requirements[1], and how to annotate the model in ways that allow a wide variety of schedulability techniques to be applied. The goal is of course to make it possible to determine whether or not a model is schedulable, i.e., if it will be able to meet its guaranteed response times.

Since there is a quite a lot of conflicting terminology in this domain (which occasionally also clashes with UML conventions) we have defined a single coherent terminology that is used throughout the specification.

Some of the model analysis methods that have provided valuable input to this specification are Rate Monotonic Analysis (RMA), Deadline Monotonic Analysis (DMA), and Earliest Deadline First (EDA). Note that the specification is not restricted to cover only these methods, but is flexible enough to handle a variety of other techniques.

1. By “hard” we mean timeliness requirements that must be satisfied without exception. Note that this is orthogonal to the notion of criticality, that is, whether the consequences of missing a hard deadline are unacceptable or not.

The set of annotations included in the specification is sufficient to perform basic schedulability analysis, but it is expected that individual schedulability tool vendors will provide specialized profiles to allow for more extensive analysis.

3.2.4 Modeling Performance

Performance analysis is primarily about determining the rate at which a system can perform its function given that it has finite resources with finite QoS characteristics Although not necessarily so, most performance analysis techniques are statistical, yielding statistical results. Hence, they are more appropriate for so-called soft real-time systems.

Note that, as in the case of schedulability analysis, the ability to model resources and their performance related characteristics, such as delay and throughput, is fundamental. In fact, there is much similarity between the concepts used in performance analysis and schedulability analysis. This commonality is reflected in the fact that the two share a common model – the general resource model.

The model that we have provided is fairly minimal, with the expectation that further specialized profiles will be defined by tool vendors or specialists by specializing the concepts defined in this profile. However, even without such specializations the concepts are sufficient for basic performance analysis of complex systems. We have tried to avoid any assumptions about whether the model analysis method will be based on queueing theory or simulation.

3.3 Approach to Model Processing (Analysis and Synthesis)

Since we want to support both model analysis *and* model synthesis, from here on we will use the general term *model processing* to encompass both. A schedulability analysis tool, for instance, which accepts a UML model and analyzes the schedulability of that model, is a kind of *model processor.* A different example is a synthesis tool that accepts a model with some characteristics unspecified and generates an appropriate optimal set of values for those characteristics. In this section, we describe a model-processing framework that is common to all forms of model analysis and synthesis.

The diagram in Figure 3-2 represents a conceptual model of the process envisaged for analyzing or synthesizing models based on the proposed profile. The modeler constructs a UML model that includes supplementary annotations required by the different model processors. The model is then passed to the model processor where it is analyzed and the results are fed back to the modeler. A key aspect of this process is that the exchange with

the model processor can be automated and that the details of model processing, including the specifics of its internal algorithms and data representations, are all hidden from the modeler. Details of this process are described in the *Model Processing* chapter.

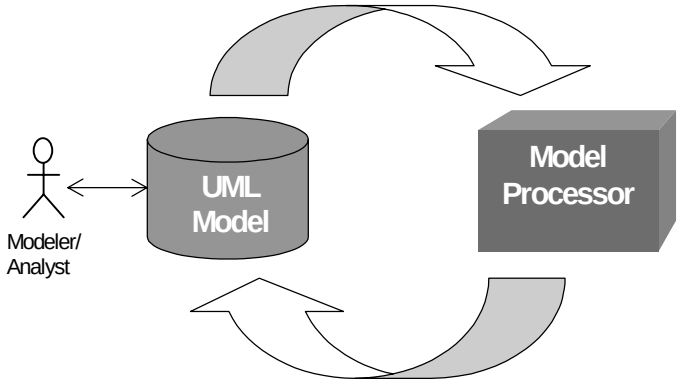


Figure 3-2 The model processing paradigm

To protect the modeler from the specifics of individual tools, all information entered and viewed by the modeler is included as part of the model. This means that the information exchanged between the model processor and the model editing tools consists of complete or partial UML models. This has the additional advantage that the standard XMI-based UML interchange format is sufficient.

In practice, of course, some knowledge of the model processor and its techniques is both necessary and useful. This is no different than explicit control of the options on a compiler. It allows the user to optimally utilize the capabilities of model processing tools, but without being overwhelmed by detail. The mechanisms that allow this type of control are also defined in this specification (see the *Model Processing* chapter).

3.4 The Structure of the Profile

Figure 3-3, describes the overall structure of the profile specified in this specification. The structure is modularized to allow users to only use those elements of the profile that they need. This structure is also well suited to future extensions.

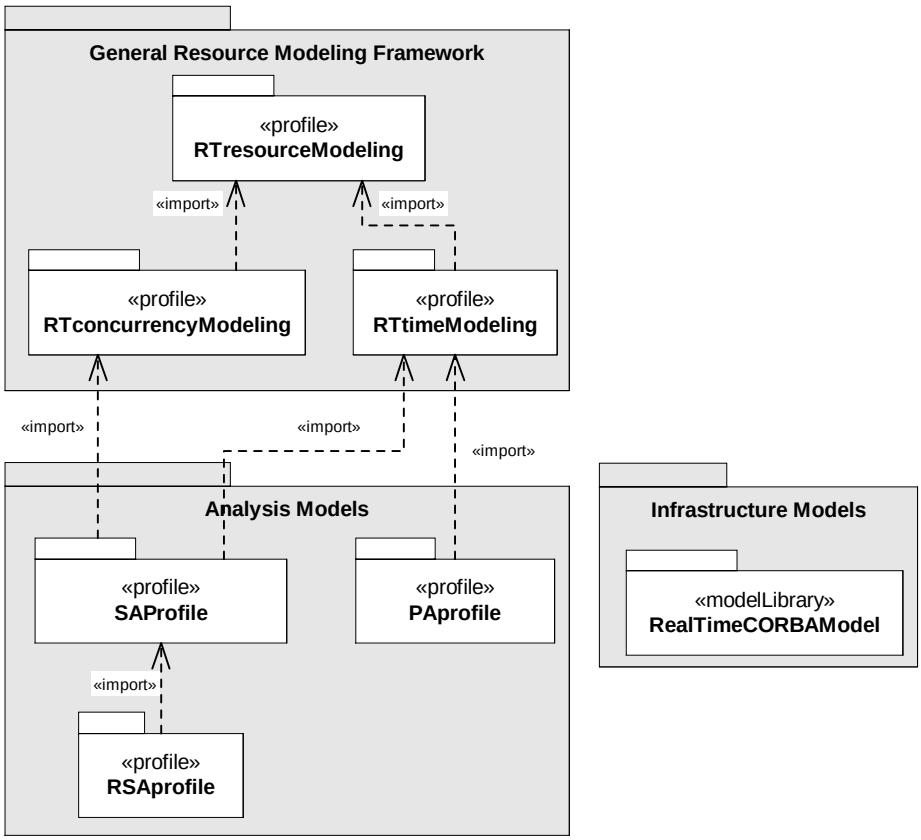


Figure 3-3 The structure of the profile

The profile for schedulability, performance and time is partitioned into a number of *sub-profiles*, profile packages dedicated to specific aspects and model analysis techniques. At the core of the profile is the set of sub-profiles that represent the general resource modeling framework. These provide a common base for all the analysis sub-profiles in this specification. However, it is anticipated that future profiles dealing with other types of QoS (e.g., availability, fault tolerance, security) may need to reuse only a portion of this core. Hence, the general resource model is itself partitioned into three separate parts.

The innermost part is the resource modeling sub-profile, which introduces the basic concepts of resources and QoS. These are general enough and independent of any concurrency and time-specific concepts. Since concurrency and time are at the core of the requirements behind this specification, they each have a separate sub-profile.

The three different model analysis profiles defined in this specification are all based on the general resource modeling framework. One sub-profile is dedicated to performance analysis and another is dedicated to schedulability analysis. The latter is then further specialized to deal with schedulability analysis of Real-Time CORBA applications.

In addition, we have included in this specification a model library that contains a high-level UML model of Real-Time CORBA. The expectation is that this can be used as a basis for complex models in which it is important not only to model the application but also the infrastructure that supports it – in this case, the Real-Time CORBA infrastructure.

The modular structure shown in Figure 3-3 allows users to use only the subset of the profile that they need. This means choosing the particular profile package and the transitive closure of any profiles that it imports. For example, a user interested in performance analysis, would need the PAprofile, RTtimeModeling, and RTresourceModeling packages.

3.4.1 Extension Specification Format Conventions

Note that this profile is based on the extension mechanisms defined in version 1.4 of the UML specification [37]. Both stereotypes and tag specifications are defined using the tabular form as shown in the following example:.

Stereotype	Base Class	Tags
«RTstimulus»	Stimulus	RTstart RTend
	ActionExecution	
	Action	
	ActionSequence	
	Method	

This table defines a stereotype, «RTstimulus», which can be applied to any of the five UML modeling concepts listed (Stimulus, ActionExecution, Action, ActionSequence, and Method) or to their respective subclasses. This stereotype has two associated tagged values, RTstart and RTend, which are defined by a separate table::

Tag	Type	Multiplicity	Domain Attribute Name
RTstart	RTtimeValue	[0..1]	TimedStimulus::start
RTend	RTtimeValue	[0..1]	TimedStimulus::end

This second table defines the type of each tag (in this example, both tagged values are instances of the RTtimeValue data type). Each tag also has a multiplicity indicating how many individual values can be assigned to each tag. A lower bound of zero implies that the tagged value is optional. Finally, the “Domain Attribute Name” column identifies the

attribute (or association end) in the domain model to which the tag corresponds. For example, TimedStimulus::start indicates that the RTstart tag corresponds to the start attribute of the TimedStimulus concept in the domain model (see Figure 5-4).

General Resource Modeling 4

This chapter describes the essential framework for modeling real-time systems using UML. At the core of this framework is the notion of *quality of service (QoS)*, which provides a uniform basis for attaching quantitative information to UML models. Specifically, QoS information represents, either directly or indirectly, the physical properties of the hardware and software environments of the application represented by the model. We refer to this framework as the *general resource modeling framework*, or, in abbreviated form, as the *GRM*.

The GRM is envisaged as the foundation required for any quantitative analysis of UML models. It comprises two closely related viewpoints. One is a *domain viewpoint* that captures, in a generic way, the common structural and behavioral concepts and patterns that characterize: (1) real-time systems and (2) real-time system analyses methods. To ensure that this model accurately reflects the real-time domain and is not influenced unduly by metamodelling concerns, this part of the GRM is to a great extent defined independently of the UML metamodel.

The second viewpoint in the GRM is the *UML viewpoint*—a specification of how the elements of the domain model are realized in UML. This consists of a set of UML extensions (stereotypes, tagged values, constraints) and is supplemented by specifications of the mappings of the domain concepts to those extensions. Not all concepts in the GRM are reified as concrete extensions. This is because, the GRM provides mostly abstract concepts that are not applied directly to elements of a UML model. Their purpose, instead, is to provide a basis for more concrete refinements in the various specialized parts of profile.

For example, the GRM defines a generic notion of a “QoS characteristic”, but there is no corresponding “QoS characteristic” stereotype. Instead, for practical reasons, different QoS characteristics are represented in different ways (either as tagged values or as stereotypes). Nonetheless, the abstract concept provides a guideline in particular cases that helps to identify what needs to be defined and how it should fit in with other related concepts.

This dual viewpoint structure is also used for all the other parts of the real-time profile. The nature of the relationship between these two viewpoints and their elements is shown in Figure 4-1 (Note that this is merely an informal diagram defined for illustrative purposes and does not imply a specific package structure or mapping.).

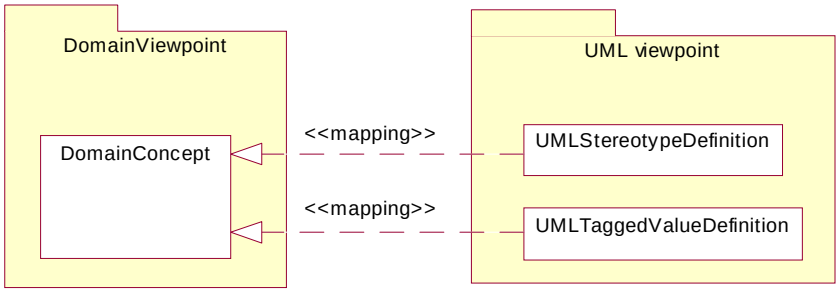


Figure 4-1 The domain and UML viewpoints and the relationships between their elements

4.1 Domain Viewpoint

In this and subsequent sections we will use UML itself to describe the domain viewpoint. This is because we expect readers of this document to be familiar with UML and also because UML class diagrams are a particularly convenient means for describing conceptual models. However, *care must be taken not to confuse elements of the domain viewpoint with elements of the UML metamodel*. As noted above, the domain viewpoint is defined independently of the UML metamodel[1].

Note that, although the intent of this domain model is to be precise, it is not fully formal since its purpose is primarily to describe the concepts and relationships of the domain. Thus, in some cases strict formality was sacrificed for reasons of clarity.

1. This is a bit of an oversimplification. In cases where there was a modeling choice between a modeling approach that was easier to map to the UML metamodel and one that is not, we naturally chose the former provided that it did not result in reduced modeling accuracy. An example of this can be found in the Section4.1.2, “ The Causality Model Package,” on page 4-6.

The GRM has many facets that are grouped in individual packages. The overall package structure is shown in Figure 4-2 The purpose and contents of each package are described in subsequent sections.

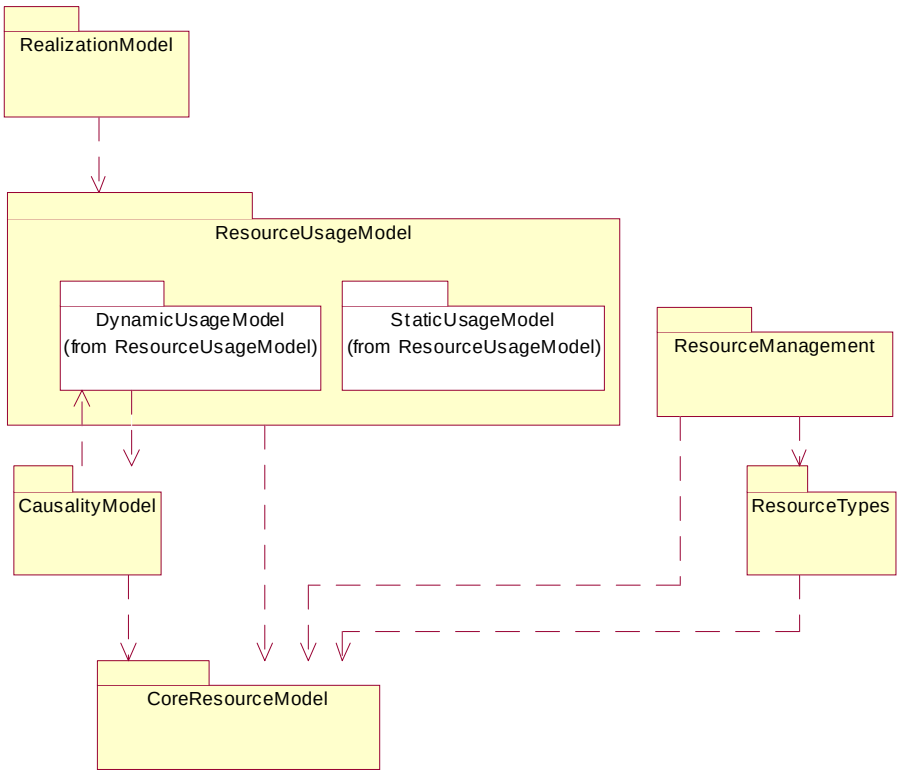


Figure 4-2 Structure of the general resource modeling framework

4.1.1 The Core Resource Model Package

This package defines the essential concepts of resources and quality of service (QoS), upon which most of the concepts in the rest of the specification are based. Although we have defined these concepts in the context of time-related QoS issues, it is anticipated that the domain concepts in this package may serve as a foundation for other types of QoS domains such as reliability or availability.

For our purposes, it is fundamental to distinguish between design-time *descriptor* elements, such as classes and types, and *run-time instance elements* that are created on the basis of those descriptors. This is because practically all time-based analysis methods operate on models that describe specific instances and their linkages. Thus, it may not be meaningful to apply such analyses on a model described purely by a UML class diagram since this model really shows only the relationships between descriptors.

However, it is often useful to be able to associate values of QoS characteristics with descriptors. This simply means that such values apply to all instances created on the basis of those descriptors and not that the descriptor itself has that value. (This requires special care, however, and may not always be appropriate. In case of interface specifications, for example, there could be many realizations of the same interface, each with different service characteristics.)

This basic partitioning into descriptors and instances is reflected in the top two elements of the class diagram that depicts the core resource model (Figure 4-3). Any number of instances can be created from a given instance descriptor. The descriptor is referred to as the type of the instance. Notice that an instance may have multiple types (which can be used either to represent different viewpoints or multiple inheritance).

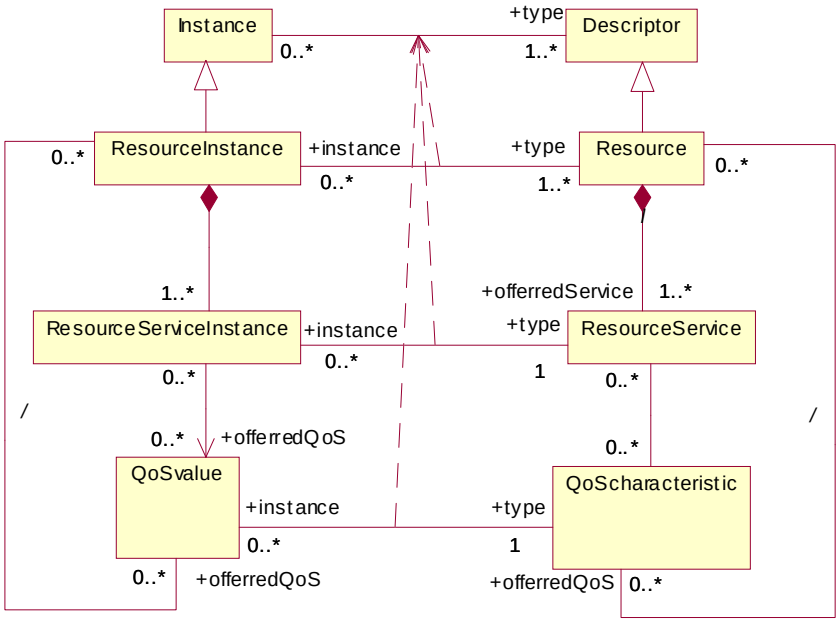


Figure 4-3 The core resource model – the instance and descriptor aspects

All the elements on the left of Figure 4-3 represent instance-based concepts whereas the elements on the right represent their corresponding descriptors[1]. The dashed-line dependencies between the top-level association connecting Instance and Descriptor and the lower associations are meant to convey that the latter are specializations of the former.

Because real-time analyses are instance based, in the rest of this model we will not define explicitly any additional descriptor concepts. Instead, we will assume that for each instance-based concept that is introduced a corresponding descriptor also exists.

1. Although it is not shown explicitly (in order not to clutter the diagram), both ResourceService and QoS characteristic are subclasses of Descriptor. Similarly, ResourceServiceInstance and QoSvalue are subclasses of Instance.

The central concept of the GRM is the notion of a resource instance. A *resource instance* represents a run-time entity that offers one or more *services* for which we need to express a measure of effectiveness or quality of service (QoS). For instance, this measure might be the execution time of the service or its precision. In all cases, the fact that it is necessary to specify the quality of service is simply a reflection of the finite nature (limited capacity, limited speed, etc.) of the resource's physical underpinnings. The concept is not useful if one assumes infinitely fast execution times or unlimited capacities.

A *resource service instance* is a specific incarnation of a resource service description that is provided by a specific resource instance. Note that a given description could be realized in different implementation environments leading to different QoS values for the same descriptor. The actual QoS values of a service instance are often referred to as the *offered QoS* (values). Depending on the nature of the service and the characteristic, QoS values can vary in complexity from simple numbers to complex structured values (e.g., probability distributions).

In some cases it is more convenient to talk about the offered QoS of a resource rather than its services. This is useful when the resource offers a single service so that it is not necessary to explicitly define the service in the model. For instance, a physical processor offers a "processing service" which may include a throughput characteristic of some kind. Rather than talk about the QoS of the processing service, it is much more common to talk about the throughput of the processor resource rather than its processing service. For this reason, we provide the capability to associate QoS characteristics (values) directly with resources in the GRM. However, this should be recognized for what it is: a convenient shortcut and not an alternative conceptual model.

Although we have represented the notion of a QoS characteristic as an explicit concept in this core model, in most cases to follow specific QoS characteristics are represented as attributes of other concepts. (On instances, of course, they are represented as attribute values.) For example, the execution time characteristic of an action might be represented as an attribute of the action rather than as a subclass of the QoS characteristic concept that is associated with the action concept. Although this is not quite formally correct, it results in a much simpler domain model that is easier to understand.

4.1.2 The Causality Model Package

This is an important model that is used as a basis for any dynamic modeling associated with the profile. It captures the essentials of the cause-effect chains in the behavior of run-time instances. The model is based on the dynamic semantics of UML 1.4, but is more detailed and more precise in certain aspects.

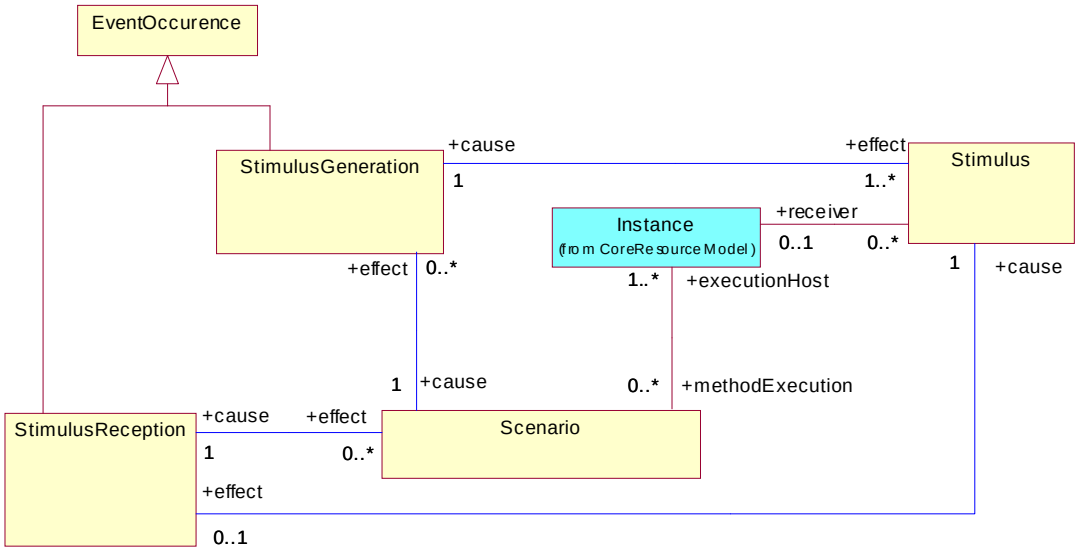


Figure 4-4 The basic causal loop model

A fundamental concept in the causality model is the notion of an *event occurrence*. This corresponds to an instance of the UML event notion (which is a descriptor that specifies a kind of change of state). There are many different kinds of event occurrences, but the most interesting ones for our purposes are the *stimulus generation* and *stimulus reception* events. A *stimulus*, which fully corresponds to the eponymous UML 1.4 concept, is an instance of a communication in transit between a calling object and a called object. A stimulus generation event occurs when an object executes an action that invokes an operation on another object (the *receiver*) or sends a signal to it. The effect of the stimulus generation event is the creation and dispatching of a stimulus that identifies the parameters of the communication (the operation invoked, the values of the parameters, etc.). The stimulus will eventually result in a stimulus reception event. This event occurs when an object executes some kind of reception operation. The occurrence of this event will either trigger a transition in the receiver or result in the execution of a method. (Details of how the event is received, scheduled, and dispatched are outside the scope of the GRM and depend on the specific situation.)

This, in turn, leads to a scenario execution (or simply, *scenario*). A scenario execution may result in the execution of an ordered set of actions (see Section4.1.5, “ The Dynamic Usage Model Package,” on page 4-9 for details), some of which may generate further stimuli, and so on.

In addition to the stimulus generation and stimulus reception events, in a number of analyses it is also useful to consider the events that occur when a scenario starts and ends its execution (*scenario start event* and *scenario end event* respectively). This is depicted in Figure 4-5.

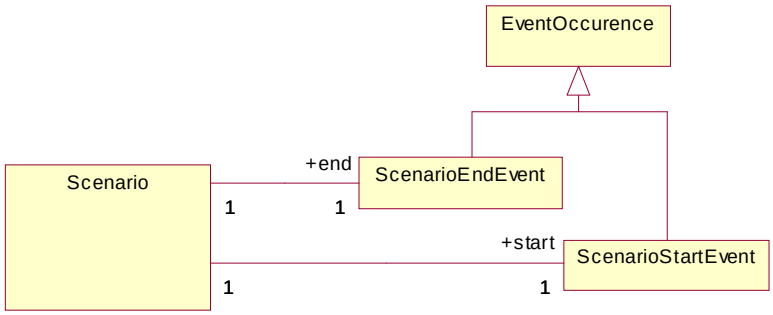


Figure 4-5 Scenario start and end event occurrences

4.1.3 The Resource Usage Model Package

Complementing the core resource model is the notion of a resource usage. Whereas the core resource model represents the server side of the GRM, the resource usage model represents its client side. A *resource usage* represents a pattern that describes how a set of clients uses a set of resources and their services. It closely corresponds to a use-case instance. A resource usage can be either *static* or *dynamic*, depending on the needs of the model analysis. In some cases it is necessary to describe the process of resource usage in detail (order, timing, etc.). This is supported by various dynamic usage models (see Section 4.1.5, “The Dynamic Usage Model Package,” on page 4-9). In other cases, it is sufficient to simply show the static linkages between clients and resources without delving into the details of how and when a resource is used by a client (see Section4.1.4, “The Static Usage Model Package,” on page 4-8).

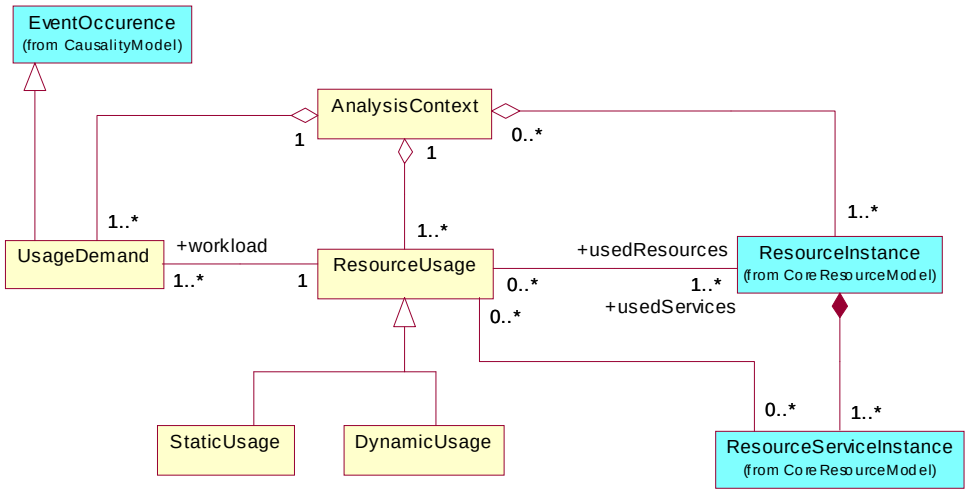


Figure 4-6 The resource usage framework

Note that in the model in Figure 4-6 we do not show the client as an explicit concept. This is because in some types of analyses a client may be implicit, represented indirectly by the load that it imposes on the resource.

In most analyses, it is useful to distinguish the usage from the event that induced it. This allows the usage pattern, which occurs within a system, to be specified independently of the external factors that lead to it. The event occurrence that causes the resource usage is called the *usage demand* because it represents the externally imposed load on the system. In a sense, QoS characteristics associated with the usage demand represent the required QoS values of the system for that specific usage.

To assist model analysis tools in determining what part of a model is to be analyzed, we introduce the concept of an *analysis context*. It consists of a set of resource usages with corresponding densities and the set of resource instances that are used by the resource usages. The main purpose of an analysis context is to define a starting point for model analysis. Starting with the analysis context and its elements, a tool can follow the links of the model to extract the information that it needs to perform the model analysis.

4.1.4 The Static Usage Model Package

This model (Figure 4-7) is used in cases where the relationship between the clients and resources can be viewed as static. This does not necessarily mean that it is static, but simply that the dynamics of usage are not relevant to the model analysis on hand. The domain model in this case includes an explicit *client*, which is also a kind of instance. In this case, however, the notion of resource services is not required.

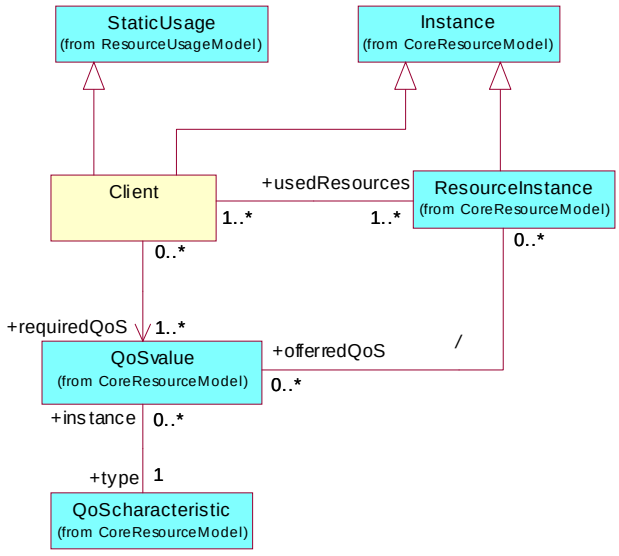


Figure 4-7 The static resource usage model

Associated with a client is a set of required QoS values. These can be matched against the offered QoS values of the resource to determine if the requirements can be met, or conversely, to determine the necessary resource characteristics to support the clients in the usage.

The most obvious interpretation of the static model is that the client and the resource instances that it uses are peers in the sense that they are collaborating instances at the same level of abstraction. For example, the resource may be a semaphore that protects some shared data and the client would be a software task that uses the semaphore to access the shared data. We will refer to this as the *peer interpretation* of the client–resource relationship.

However, the same static model can be applied to the case where the clients represent elements of a software model and the resource instances represent elements of the hardware/software environment on which the software elements are deployed. This view of resource usage is called the *layered interpretation*. An expanded description of this is provided in Section 4.1.8, "The Realization Model Package (Deployment Modeling)," on page 4-12.

4.1.5 The Dynamic Usage Model Package

The dynamic usage model (Figure 4-8) applies in situations where the order and time of occurrence of the loads on resources is relevant to the model analysis. In this case, the usage is represented by a *scenario* instance—an ordered series of *steps* called *action executions*. The ordering of steps follows a predecessor-successor pattern, with the possibility of multiple concurrent successors and predecessors, stemming from concurrent thread joins and forks respectively.

The granularity of a step is often a modeling choice that depends on the level of detail that is being considered. Hence, a step at one level of abstraction may be decomposed further into a set of finer-grained steps (action executions). For this reason, we model an action execution as a kind of scenario. The recursion ends when a scenario is represented by a single indivisible (atomic) step.

Each scenario (and action execution) may use one or more resources and resource operations. In those cases, it may specify explicitly the *required QoS* that it needs in order to meet its own obligations. The essential problem of any kind of dynamic real-time analysis is to compare these QoS requirements against the offered QoS characteristics of the resources and services used by the scenario.

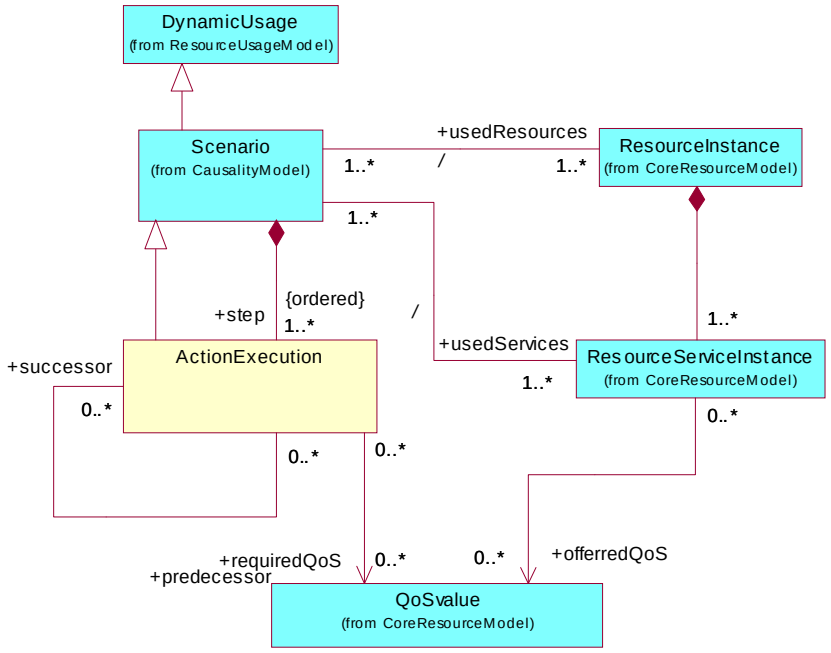


Figure 4-8 The dynamic resource usage model

4.1.6 The Resource Types Package

This part of the GRM (Figure 4-9 on page 4-11) deals with the different categorizations of resources that are useful for modeling real-time systems. A given resource instance may belong to more than one type, although it can be classified by at most one type from each category.

Based on *purpose*, resources are classified into:

- *processor resources*, which represent either virtual or physical processing devices capable of storing and executing program code
- *communication resources*, whose primary purpose is to enable communications between other resources and
- *devices*, which represents other types of resources that are neither processors nor communication resources (e.g., disks, sensors, motors, etc.)

Based on *activeness*, resources are categorized as:

- *active resources*, which are capable of generating stimuli concurrently or pseudo-concurrently without being prompted by an explicit stimulus instance (i.e., devices that appear capable of “spontaneous” unprompted behavior such as hardware, operating system processes and threads, etc.)
- *passive resources*, which cannot generate their own behavior, but only react when prompted by a stimulus

Based on *protection*, resources can be one of:

- *protected resources*, which are resources that offer one or more *exclusive service instances*; these are services to which concurrent access is restricted according to some *access control policy*
- *unprotected resources*, whose services are not subject to any access protection.

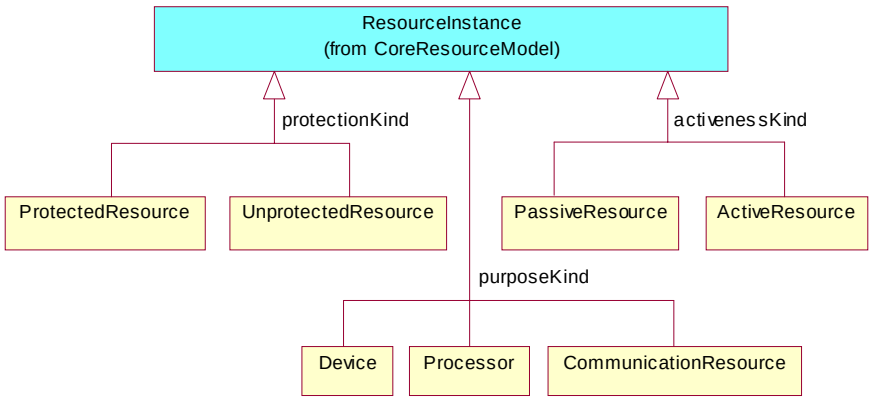


Figure 4-9 .Resource types model

To use the exclusive services of a protected resource, it is necessary to first execute an appropriate *acquire service action* (Figure 4-10). If the action is defined as a blocking action, then it will block the caller until the exclusive service is made available. This is determined by the associated access control policy. If it is a non-blocking action the service will either be made available immediately or a failure result will be returned, depending on the access control policy and the state of the resources at the time. The action that releases an appropriated service is the *release service* action. This action is non-blocking and can be invoked at any time, whether the service has previously been acquired or not.

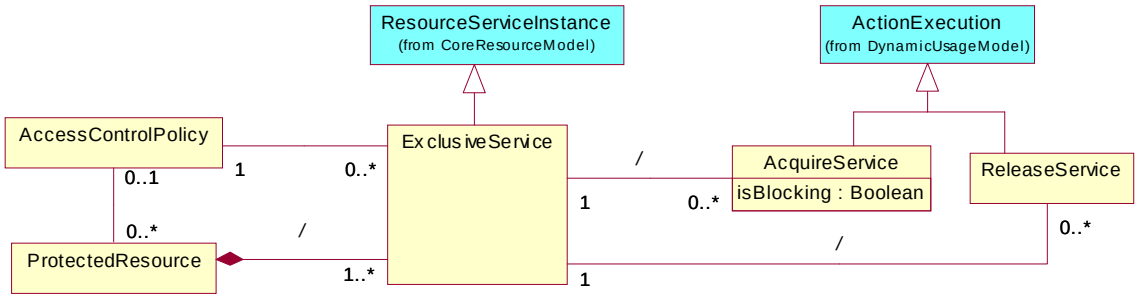


Figure 4-10 Exclusive service instances and corresponding actions

Note that in some cases, it is more convenient to associate the access control policy directly with the protected resource. This is merely a conceptual shortcut for resources that have only one kind of service or multiple services that are administered using a single access control policy.

4.1.7 The Resource Management Package

This is a utility package for modeling various resource management services, such as those found in most operating systems. In general, we distinguish between a number of different roles here:

- The *resource broker*, is an entity that is responsible for allocation and deallocation of a set of resource instances (or their services) to clients according to a specific *access control policy*. For example, a memory manager will allocate memory from a heap upon request from a client and also return it back into the heap once the client no longer needs it. The access control policy determines the amount of memory provided to individual clients, the prioritization of competing requests, etc.
- The *resource manager*, on the other hand is responsible for creating and maintaining resources according to a *resource control policy*. For example, a buffer pool manager is responsible for creating a set of buffers from one or more chunks of heap memory. Once created and initialized, the resources are typically handed over to a resource broker. In most practical cases, the resource manager and the resource broker are the same entity.. However, since this is not always true the two concepts are modeled separately (they can be easily combined by designating the same entity as serving both purposes).

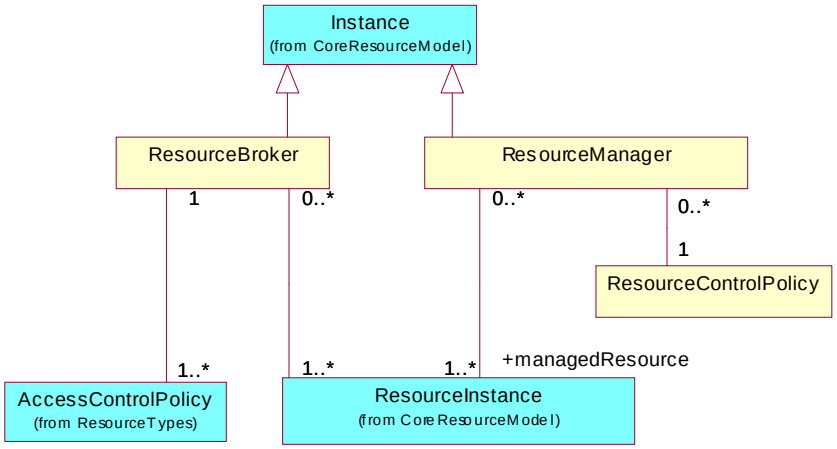


Figure 4-11 Resource management model

4.1.8 The Realization Model Package (Deployment Modeling)

Layering is one of the most commonly used techniques in software. It is a particular form of the general “divide-and-conquer” strategy for dealing with complexity. A layer is an environment in which all the constituents are specified at the same level of detail relative to a given viewpoint. Note that viewpoints play a key role in the definition of layers, although this is often overlooked or left unstated. A viewpoint represents a set of related concerns. For instance, the well-known seven-layer OSI model decomposes a system from a communications perspective. Layers are differentiated based on the degree of technological specificity. Thus, the deeper one descends through the layer hierarchy, the communication mechanisms become more and more specific until actual hardware is reached.

It turns out that layers can be pure abstractions, which means that they only exist as a mental tool to help us understand a complex system by partitioning the problem in some graduated fashion. However, when it comes to software systems in particular, layers may actually be concrete things with an independent existence. For example, an operating system is a layer that exists independently of any application software that runs on top of it. The concrete and abstract forms of layering are often confused with each other. To reduce the likelihood of such confusion, we examine each in a bit more detail in the following two subsections.

4.1.8.1 Refinement

In this case, we have on hand two separate models of the same system such that one is a more detailed (i.e., more refined) rendering of the other. By convention, the more abstract model is considered as the “upper” layer in the abstraction hierarchy and the more refined model is the “lower” layer. For obvious reasons, we will refer to this form of layering as *refinement layering*.

For instance, in Figure 4-12 below, the upper UML layer depicts a highly-abstract view of a specific fragment of some program. The lower layer shows a more detailed rendering of the same fragment with explicit details of the C++ implementation.

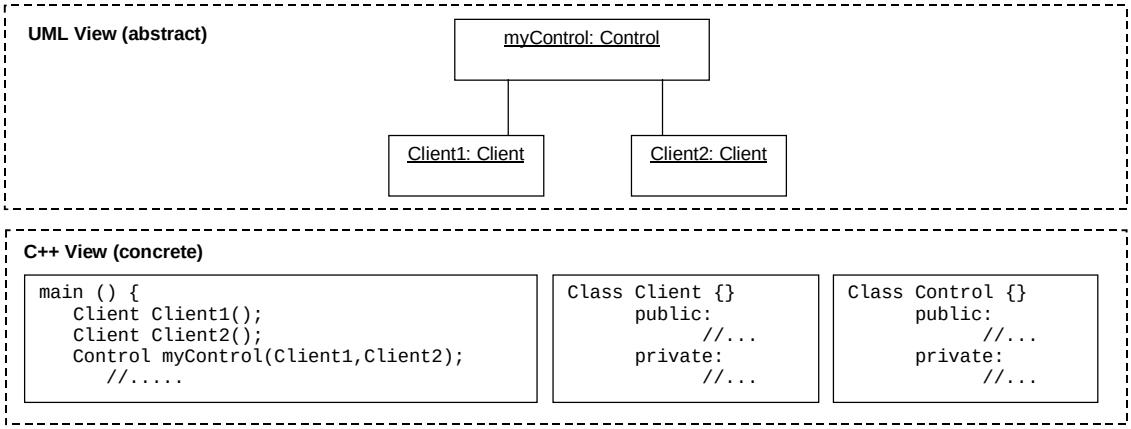


Figure 4-12 Example of refinement-type layering

A very well-known example of this type of layering is the ISO reference model for open distributed processing (RM-ODP). In that case, the abstraction hierarchy is based on the degree of technology specificity. The topmost layer, the Enterprise Viewpoint, completely abstracts out technology and represents a system from the perspective of a business enterprise. In this viewpoint, the need for some type of information processing is identified, but without any reference to a specific technology that might realize that need. At the next level down is the Information Viewpoint, which starts to bridge the gap towards computing technology by expressing the system in terms of abstractions that are familiar in the computer world: entities and relationships. This continues on until the bottom level is reached, the Technology Viewpoint, in which the actual hardware and software implementation technologies are depicted.

4.1.8.2 Realization

It is common practice, particularly at higher levels of abstraction, to depict software that is executing on some real or virtual machine as a separate *layer* placed over that machine, which is itself shown as a layer. This relationship may extend to multiple levels as shown in Figure 4-13 Here, the bottom layer is the computing hardware. The next level up is the operating system, shown as a layer running on top of the hardware. Finally, there is an application layer, which is executing on top of the "virtual machine" realized by the operating system.

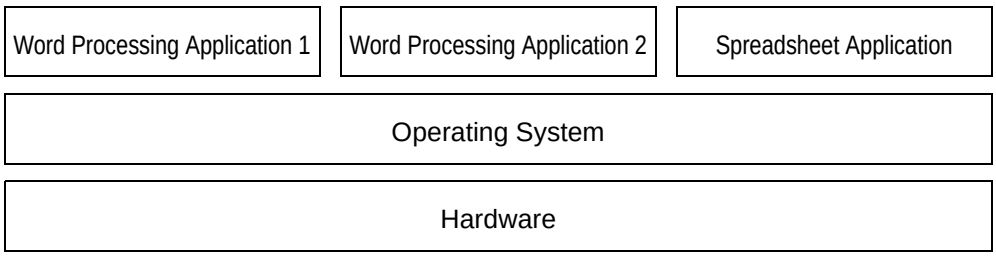


Figure 4-13 An example of realization layering

In contrast to refinement, each level in this hierarchy defines a distinct part of the system. The information in each layer is unique and the full system is only defined by the aggregate of all the layers. The OSI 7-layer model is perhaps the most well known example of this form of layering.

Since the lower layer is required to execute the program specified in the upper layer, we refer to this form of layering as *realization layering*. That is, the lower layer realizes the upper layer (i.e., in the same sense that computer hardware realizes a software program).

The following are the distinguishing characteristics of realization layering:

- Each layer contains distinct information about the system; this information exists only in that layer (in contrast to refinement layering where the same information exists in all layers)
- Elements in the upper layer cannot function properly unless its supporting lower layers are fully functional[1].
- The lower layer is independent of the upper layer, that is, it can exist and function (offer its services) independently of the upper layer.

To more easily distinguish realization layering relationships from refinement relationships, we will refer to elements of the upper layer as *logical model elements* and to elements of the lower layer as *engineering model elements*. This was based on the view that upper layer elements are generally more abstract ("logical"), whereas the lower-layer elements are more technology-specific (i.e., engineering oriented) since they

1. In fact, it may not even be able to exist independently of the lower layer; e.g., a running software program has no existence if the hardware is not operational.

are responsible for the realization of logical elements. Note that this is a relative designation, a layer may be an engineering model for the layers above but is a logical model for the layers below. An exception is the bottom layer (usually hardware).

4.1.8.3 Deployment

Realization is synonymous with the notion of “deployment” in UML: the mapping of elements of a software model (the upper layer) to elements of an environment model (the lower layer). Note that, as shown in Figure 4-13, the environment need not necessarily represent hardware, but can also go between software and software. For example, an active object may be mapped to (i.e., realized by) an operating system task.

The characteristics of realization layering are fully analogous to the characteristics of the relationship between clients and resources instances in the GRM static usage model (see Section 4.1.4, “The Static Usage Model Package,” on page 4-8)[1]. That is, the logical layer in a layering relationship is a client of the engineering layer. The engineering layer, on the other hand, can be viewed as a resource (e.g., processor) that provides basic resource services to the logical layer. This means that deployment can also be represented by the GRM domain model. The lower layer defines a set of resources and resource services with offered QoS values (e.g., processor throughput, memory capacity, communication bandwidth), which can be compared to the required QoS values of the elements in the upper layer. The domain model corresponding to this interpretation is shown in Figure 4-14

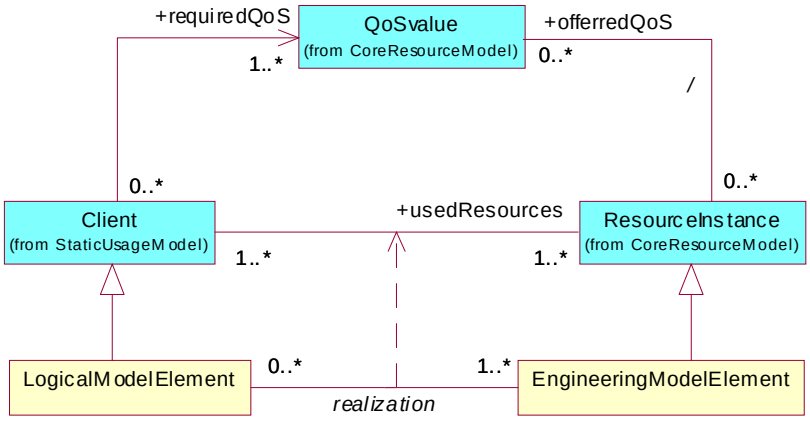


Figure 4-14 The model of realization (deployment) represented as a static resource usage

1. In principle, there is no reason why a dynamic usage relationship cannot be used for the layered interpretation. However, the simpler static model is sufficient for the purposes of this specification so we will restrict ourselves to that.

Note that realization/deployment mappings can be represented at different levels of abstraction. For example, the deployment of the spreadsheet application in Figure 4-13 can be specified either in terms of its deployment to the operating system environment or directly to the hardware. In the latter case, we are bypassing, or "abstracting out", the operating system layer. The choice abstraction level depends on the needs of the moment.

4.1.9 Domain Concepts Details

In this section we provide a precise specification of all the concepts in the GRM.

4.1.9.1 AccessControlPolicy

This is an instance of the policy by which access is controlled to an instance of an exclusive service. The responsibility for administering the policy rests with a resource broker.

Associations

exclusiveService	the exclusive service that is controlled by this instance of the access control policy
resourceBroker	the resource broker that administers this policy
protectedResource	the protected resource instance that is accessed using this policy (note that this is a conceptual shortcut for the cases where a resource uses only a single control policy for all of its services or it has only one exclusive service)

4.1.9.2 AcquireService

An instance of an action execution that is used to gain access to an instance of an exclusive service on some protected resource. An exclusive service of a protected resource cannot be accessed unless this action is executed successfully. The operation may fail, in which case attempts to use the protected resource will also fail. Once the resource is no longer needed, it can be released with a "releaseService" action. Until the latter action is executed, the resource remains in "possession" of the entity that executed the acquire action.

Associations

exclusiveService	the instance of a service of some protected resource that is acquired by this action

Attributes

isBlocking	a Boolean that indicates whether the acquire action is blocking or not; if it is blocking, then the scenario executing the action will not proceed to the successor steps, until the appropriate resource broker gives permission (according to the access control policy), at which point the operation has succeeded; if the action is non-blocking, then, if the exclusive service is accessible when the action is executed, it will be immediately successful, otherwise it will fail immediately

4.1.9.3 ActionExecution

This represents a single execution of some action specification. Each execution of the same action will result in a different action execution (i.e., each action execution has a distinct identity). Action execution is a subclass of scenario and inherits its associations. This allows action executions to be decomposed into finer-grained steps, if so required.

Associations

scenario	the scenario instance of which this action execution is a part.
successor	the action executions, which are part of the same scenario, that immediately follow this action execution; if there are no successors, then this is the last action in the scenario; if there is more than one successor action execution, then it is assumed that they represent mutually concurrent actions (concurrent fork).
predecessor	the action executions, which are part of the same scenario, that immediately precede this action execution; if there are no predecessors, then this is the first (top) action in the scenario; multiple predecessors imply that this action execution represents a point where concurrent execution threads join into a single thread - in that case, the action execution will not proceed until all the predecessor actions have completed.
requiredQoS	an optional set of QoS values that are associated with this action; each value represents the minimal QoS that must be provided by the resources and/or resource services used by the action execution.
usedResources	(inherited from Scenario) the set of resources that the action execution accesses during its execution.
usedServices	(inherited from Scenario) the set of resource services that the action execution accesses during its execution.
executionHost	(inherited from Scenario) a resource instance that is executing this action execution.

4.1.9.4 ActiveResource

This is a resource that is capable of generating its own stimuli concurrently (i.e., asynchronously of other activities) without necessarily being prompted by an explicit scenario. This is a subclass of Resource Instance (see Section 4.1.9.25, “ResourceInstance,” on page 4-23for association details).

4.1.9.5 AnalysisContext

This is a context that is the root for an analysis of a model (there can be more than one context for a given model).

Associations

usageDemand	the event occurrences whose required QoS characteristics define the external workloads in the analysis context.
resourceUsage	the set of resource usage instances that are part of this context; there

must be at least one resource usage instance in a context (it may be either static or dynamic).

resourceInstance — the set of resources involved in this context; there must be at least one resource instance in every analysis context (e.g., a processor).

4.1.9.6 Client

This is an explicit run-time instance that uses resources. It is used in static model analysis schemes. In dynamic analyses, it is implied by a scenario.

Associations

usedResources — (inherited from ResourceUsage) the set of resources that are directly used by this client; these resources may have offered QoS values specified.

requiredQoS — the set of values that characterize the quality of service that this client requires of the resources that it uses (where appropriate, these values may be used instead of the workload specification).

workload — (inherited from ResourceUsage) the set of workloads imposed by this client (where appropriate, these may be used instead of the required QoS values).

4.1.9.7 CommunicationResource

This is a resource whose primary purpose is to connect two or more other types of devices in order to enable them to communicate. This is used to model various kinds of networks, channels, etc. This is a subclass of Resource Instance (see Section 4.1.9.25, “ResourceInstance,” on page 4-23 for association details).

4.1.9.8 Descriptor [abstract]

An abstract concept representing some kind of design-time specification. This concept includes all kinds of descriptors such as classifiers, collaborations, data types, etc. It is generally assumed that every instance element in the domain model may have an implicit or explicit descriptor.

Associations

instance — the set of run-time instances that are incarnated based on this descriptor.

4.1.9.9 Device

This is a resource instance that is neither a processor kind of device nor a communication device. In real-time systems, it is used for modeling various kinds of specialized devices such as sensors, effectors, secondary storage devices, etc. This is a subclass of Resource Instance (see Section 4.1.9.25, “ResourceInstance,” on page 4-23 for association details).

4.1.9.10 DynamicUsage

A kind of resource usage instance that involves some kind of dynamic scenario with an ordered usage of resources involved in the scenario (see Section 4.1.9.30, “Scenario,” on page 4-25). This is typically used for more precise types of analyses.

Associations

workload	(inherited from Resource Usage) the event that caused this resource usage and which defines the external demand for that usage.
usedResources	(inherited from Resource Usage) the set of resources that are directly used by this dynamic usage.
usedServices	(inherited from ResourceUsage) the set of services that are directly used in this dynamic usage.

4.1.9.11 EngineeringModelElement

An instance of a model element that realizes one or more logical model elements in the layered interpretation of resource usage (see Section 4.1.8, “The Realization Model Package (Deployment Modeling),” on page 4-12). In this role, the model element acts as a resource instance with specified offered QoS values.

Associations

logicalModelElement	the set of logical model elements that are realized by this engineering model element; this association represents the “realizes” relationship.

4.1.9.12 EventOccurrence [abstract]

An instance of the occurrence of an event (change of state) of some type. Event occurrences are assumed to be instantaneous.

4.1.9.13 ExclusiveService

A resource service instance associated with a protected resource that can only be accessed if the appropriate resource broker approves. The resource broker makes these decisions on the basis of an access control policy set for each exclusive service. This service can only be used after an acquire service action has been executed successfully (see Section 4.1.9.2, “AcquireService,” on page 4-16) and up to the time that the release service action has been executed (see Section 4.1.9.21, “ReleaseService,” on page 4-22).

Associations

protectedResource	the instance of a protected resource that owns this instance of the exclusive service (the resource may also serve as the resource broker in some cases).
accessControlPolicy	the access control policy instance that is used to control access to this service instance.

acquireService the set of acquire service executions that accessed this service instance.

releaseService the set of release service action executions that are associated with this service instance.

4.1.9.14 Instance

An abstract concept representing some kind of run-time instance that is created based on one or more type specifications (descriptors). This concept includes all kinds of instances, including objects, data values, etc.

Associations

type the set of design-time descriptors that are used to specify all the aspects necessary to run this instance.

stimulus the set of stimuli that are received by this instance; the actual handling of the stimulus depends on the kind of instance.

method execution the set of scenario executions that are executed by this instance as a result of receiving stimuli; in effect, this is the response of an object to the reception of a stimulus.

4.1.9.15 LogicalModelElement

An instance of a model element that is realized by one or more engineering model elements in the layered interpretation of resource usage (see Section 4.1.8, “The Realization Model Package (Deployment Modeling),” on page 4-12). In this role, the model element acts as a client with specified required QoS values. In general, a logical model element may be realized by a set of engineering model elements.

Associations

engineeringModelElement the set of engineering model elements that realize this logical model element; this association represents the “realizes” relationship.

4.1.9.16 QoScharacteristic [abstract]

This concept represents the descriptor of some kind of resource service characteristic that specifies either how well the service needs to be or can be performed. It is an abstract concept that is used to describe the overall GRM framework. In case of specific domain concepts, however, QoS characteristics are usually modeled by attributes, since this is both simpler and easier to understand.

Associations

resourceService the set of different kinds of resource services to which such a characteristic may be attached.

instance the set of QoS values used to represent specific instances of this characteristic.

resource the set of resources for which this is a QoS characteristic (this is

used in cases where the modeling of resource services is superfluous).

4.1.9.17 QoSvalue [abstract]

The value of a QoS characteristic for a specific resource or resource service instance. The value can range from a single number to a complex structured value such as a probability distribution.

Associations

type the descriptor that specifies the characteristic to which this value corresponds.

resourceServiceInstance the instance of a resource service to which this value applies.

resourceInstance the specific instance of a resource to which this value applies (this is used in cases where the modeling of resource services is superfluous).

4.1.9.18 PassiveResource

This is a simple resource that is incapable of generating stimuli unless it is prompted by a scenario. This is a subclass of Resource Instance (see Section 4.1.9.25, “ResourceInstance,” on page 4-23 for association details).

4.1.9.19 Processor

This is a resource instance that is capable of storing and executing a computer program and its associated data. It is used for modeling physical CPUs, computers, etc. as well as virtual machines (“logical” processors). This is a subclass of Resource Instance (see Section 4.1.9.25, “ResourceInstance,” on page 4-23 for association details).

4.1.9.20 ProtectedResource

This is an instance of a resource that provides one or more exclusive services—services that can only be accessed according to an access control policy administered by an associated resource broker. (The resource broker is a role that may be played by the resource itself.) The semantics of those services are described in detail in Section 4.1.9.13, “ExclusiveService,” on page 4-19.

Associations

exclusiveService the set of exclusive services offered by this resource; there has to be at least one such service for a protected resource (although it may be implicit).

accessControlPolicy the access control policy that is used by this resource instance (note that this is a conceptual shortcut for the cases where a resource uses only a single control policy for all of its services or it has only one exclusive service).

4.1.9.21 ReleaseService

This is an instance of an action that is used to release an instance of an exclusive service that was acquired by a previously executed acquire service action (see Section 4.1.9.2, “AcquireService,” on page 4-16). This action execution is not bound by access control restrictions and can be executed at any time, even if the resource service has not been acquired before (in which case it has no effect).

Associations

exclusiveService	the instance of an exclusive service that is being released by this action execution.

4.1.9.22 Resource

A design-time descriptor that specifies a kind of resource. A resource is an element that has resource services whose effectiveness is represented by one or more QoS characteristics.

Associations

instance	the set of resource instances created on the basis of this descriptor; this association is a specialization of the general instance-descriptor association (see Section 4.1.9.8, “Descriptor [abstract],” on page 4-18).
offeredService	the set of specifications (descriptors) that define the services that are offered by this resource; there has to be at least one service (although it may not be explicitly specified in a model).
qoScharacteristic	the set of QoS characteristic descriptors of the quality of service of the services offered by instances of this resource; this form is only used when explicit specification of resource services is superfluous.

4.1.9.23 ResourceBroker

An instance of a run-time entity responsible for controlling access to the exclusive services of a resource. Note that this role may be played by the resource itself. The broker processes acquisition requests from clients of the service (see Section 4.1.9.2, “AcquireService,” on page 4-16) and, based on the appropriate access control policy for that service, it dispenses access to the service. If a service instance is busy, then the reply may remain pending until access is possible.

Associations

accessControlPolicy	the set of access control policies administered by this resource broker; a single broker may administer multiple control policies for different resource service instances.
resourceInstance	the set of protected resource instances for which this broker acts as an access controller.

4.1.9.24 ResourceControlPolicy

An instance of a policy that is used to manage one or more resource instances. This policy is used by a resource manager. Note that the role of a resource manager may be played by the resource itself.

Associations

resourceManager	the set of resource managers which use this instance of the control policy; usually there is just one such policy per resource manager instance.

4.1.9.25 ResourceInstance

An instance of some resource. A resource is a run-time entity whose services can be characterized by QoS values. This is a generic superclass that can be specialized in a number of ways (see Section 4.1.6, “The Resource Types Package,” on page 4-10).

Associations

type	the set of descriptors that specify the structure and behavior of this instance; there may be multiple descriptors for the same type representing either multiple viewpoints of multiple inheritance; this association is a specialization of the general association between Descriptor and Instance (see Section 4.1.9.8, “Descriptor [abstract],” on page 4-18).
resourceServiceInstance	the set of resource service instances that are provided by this resource instance.
qoSvalue	the set of values used to define the QoS characteristics of this resource instance; this form is used in cases where the specification of an explicit resource service instance is superfluous.
stimulus	(inherited from Instance) the set of stimuli for which this resource instance is a receiver.
methodExecution	(inherited from Instance) the set of scenario executions (methods) performed by this resource instance in response to stimuli that it received.
resourceUsage	the set of resource usages in which this resource instance appears.
analysisContext	the set of model analysis contexts in which this resource instance appears.

4.1.9.26 ResourceManager

The run-time instance responsible for managing one or more resource instances. This is primarily a role that can be played even by resource instances. The intent behind this concept is to allow modeling of resource managers that typically appear in run-time system infrastructures (memory managers, device controllers, etc.). In general, management includes the creation, maintenance, and possible destruction of resources according to some resource control policy. A resource manager may also play the role of a resource broker (see Section 4.1.9.23, “ResourceBroker,” on page 4-22).

Associations

resourceControlPolicy an instance of a control policy for managing resources.

resourceInstance the set of resource instances managed by this manager instance.

4.1.9.27 ResourceService

A design-time descriptor of a service offered by some resource. This service may have QoS characteristics that are used to describe how well instances of the service perform their functional responsibilities.

Associations

resource the resource descriptor that owns this resource service descriptor.

instance the set of run-time instances of this resource service; this is a specialization of the general association between Instance and Descriptor (see Section 4.1.9.8, “Descriptor [abstract],” on page 4-18).

qoScharacteristic the set of descriptors of the QoS characteristics associated with this service; in most cases, these characteristics are specified as attributes of concepts rather than by separate concepts.

4.1.9.28 ResourceServiceInstance

This is a specific instance of a resource service attached to a specific resource instance. While it is somewhat unusual to talk about “instances of a service,” it makes sense in cases where the service may have QoS characteristic values. Thus, even though two service instances may have the same descriptor, the individual instances may have different QoS values.

Associations

resourceInstance the resource instance to which this service instance belongs.

type the descriptor used to define this resource instance.

offerredQos the set of QoS values that characterize the service instance; these values are based on the QoS characteristics of the corresponding resource service descriptor; in most cases, these values are associated with the attributes.

resourceUsage the set of resource usage instances that invoke this resource service instance.

4.1.9.29 ResourceUsage [abstract]

An abstraction of an instance of a pattern of usage of one or more resources. A resource usage instance may be static (Section 4.1.9.33, “StaticUsage,” on page 4-26) or dynamic (Section 4.1.9.10, “DynamicUsage,” on page 4-19). The pattern itself is kept separate from a specification of the external demand which induces the usage. Although this represents a specific instance, in case of periodically occurring usages, a single instance is used to represent all repeated incarnations.

Associations

workload	the usage demand that causes this resource usage instance (NB: this association is an abstraction of part of the causal loop); multiple workloads may be used to identify alternatives or to identify complex workload specifications (e.g., minimal, maximal, and average).
analysisContext	the analysis context in which this usage instance appears.
usedResources	the set of resources used by this usage instance; there must be at least one resource instance for a usage instance.
usedServices	the set of resource service instances used by this usage instance.

4.1.9.30 Scenario

An instance of a dynamic usage of a set of resources and resource services that consists of an ordered collection of individual steps (action executions). This concept is used for modeling complex dynamic situations for more refined types of analyses. The concept is defined recursively to allow the modeling of complex hierarchies of scenarios. Thus, the components of a scenario may themselves be scenarios and so on. The model of scenario instances supports both concurrent and sequential scenarios.

Associations

step	the ordered collection of individual action executions that comprise a scenario execution; the order may be a partial order that allows the modeling of concurrent forks and joins (see Section4.1.9.3, “ActionExecution,” on page 4-17).
executionHost	the set of run-time instances that acts as the hosts for the scenario execution; these run-time instances must be instances of active resources (i.e., processors of some kind); a single scenario may be hosted on a number of active resources.
start	the event occurrence that is the start of the execution of the scenario.
end	the event occurrence that is the completion of the execution of the scenario.

4.1.9.31 ScenarioEndEvent

This event occurrence takes place at the instant when a scenario fully completes execution.

Associations

scenario	the scenario execution that corresponds to this event occurrence.

4.1.9.32 ScenarioStartEvent

This event occurrence takes place at the instant when a scenario commences execution.

Associations

scenario	the scenario execution that corresponds to this event occurrence.

4.1.9.33 StaticUsage

A kind of resource usage instance that only identifies a set of clients and resources. This is typically used for very high-level types of analyses

Associations

workload	(inherited from ResourceUsage) the event occurrence that results in this resource usage.
usedResources	(inherited from Resource Usage) the set of resources that are directly used by this static usage.

4.1.9.34 Stimulus

A concept imported directly form the UML dynamic semantics model. It represents an instance of a communication that is in transit between a sender instance and a receiver instance (or a set of receiver instances). A stimulus is generated by the occurrence of a stimulus generation event, which is, in turn, induced by the execution of an action that generates an interaction between run-time instances. This concept is primarily used to model various dynamic situations.

Associations

cause	the stimulus generation event occurrence that directly results in the generation of this stimulus instance.
effect	the set of stimulus reception event occurrences that are directly caused by this stimulus instance; note that the method by which the stimulus reception is induced is not specified (i.e., it can be done in a variety of ways).
receiver	the run-time instance that is the direct receiver of the stimulus; the receiver may represent either an object or some resource (e.g., a processor), depending on the level of abstraction being used in the model.

4.1.9.35 StimulusGeneration

An occurrence of an event that results in the generation of a stimulus (see above). This occurs as a direct consequence of a scenario that includes an action execution step (such as a call to an operation or a signal send).

Associations

effect	the set of stimuli generated in this event occurrence.
cause	the scenario execution instance that resulted in the generation of this event occurrence; note that the scenario could be either a single action execution or a complex ordered set of action executions -- the choice depends on the needs of the model on hand.

4.1.9.36 StimulusReception

An occurrence of an event that represents the acceptance of a stimulus by a receiver instance. This is normally the result of some instance executing an implicit or explicit receive action. The details of this are outside of the scope of the GRM and may be modeled in detail in specific cases where that is important for model analysis.

Associations

cause	the stimulus that resulted in this event occurrence.
effect	the set of scenarios that are executed as a direct consequence of the occurrence of this event; this could be the trigger of a transition in a state machine or the invocation of a method; the details of how the event occurrence results in scenario executions is outside the scope of the GRM; note that a single stimulus reception occurrence may result in multiple scenarios being executed (e.g., in a concurrent state machine).

4.1.9.37 UnprotectedResource

An instance of a resource which does not offer exclusive services and whose access, therefore, is not protected by an access control policy.

4.1.9.38 UsageDemand

A kind of event occurrence that leads to a usage demand. set of values (possibly complex) that characterize the external load intensity imposed by a resource usage in a given analysis context.

Associations

resourceUsage	the resource usage to which this set of values applies.
analysisContext	the context to which this demand belongs.

4.2 The UML Viewpoint

The usual method of mapping a domain concept into a UML extension is to define a corresponding stereotype. Such a stereotype can then be applied to elements of a UML model so that the domain concept can be readily recognized by domain experts or model analysis tools. In the case of GRM, however, most of the domain concepts are abstract and are intended to be refined further in specialized packages and sub-profiles before they are turned into concrete stereotypes. Hence, there are very few stereotypes defined for the elements of the GRM domain model.

4.2.1 Modeling Realization Relationships

In this section we propose a method for modeling realization layering with UML. We will not consider refinement further in this document, but we will remark that there is a lot of homomorphism between the two that probably leads to common solutions.

In UML, realization is modeled as a stereotype of the more general Abstraction relationship. Specifically, realization is defined as follows:

> *"[A relationship that relates] a specification model element or elements (the supplier) and a model element or elements that implement it (the client)."*

Every kind of Abstraction, including realizations, have an associated *mapping* that specifies the relationship between the supplier and the client. The precise nature of the mapping is left completely open and is, therefore, a convenient point to specialize in a profile.

4.2.1.1 Realization Mappings

The definition of abstraction allows for both clients and suppliers to be multi-element sets. In case of realization, this means that one or more supplier elements may be realized by one or more client elements, as illustrated in Figure 4-15.

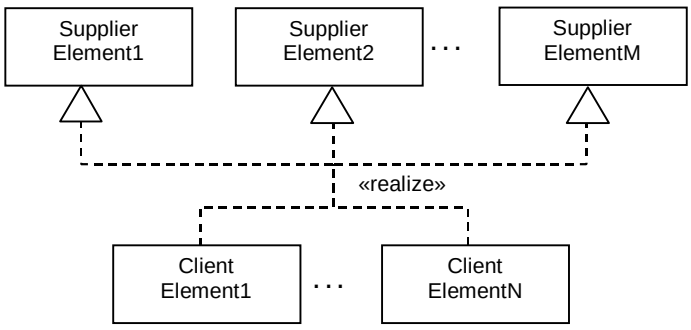


Figure 4-15 A many-to-many realization mapping

The mapping indicated by a simple realizes relationship in this case is not detailed enough for us to understand precisely how the client elements realize the supplier elements. For certain kinds of analyses much more detail may be required.

To specify such detailed mapping information in practical cases where the number of elements that need to be mapped may be very large, it is often more convenient to render the realization relationship using a tabular form such as:

Logical Elements	***Engineering Elements***	***Mapping Details***
SupplierElement1	ClientElement1 ClientElement2	…
SupplierElement2	ClientElement1 ClientElement3	…
SupplierElement3	ClientElement6 ClientElement9	…

In this table, the “mapping details” column may include further semantic details, such as constraints or transformation rules, that apply to individual mappings. We look into further details of this in the following subsections.

The mapping details may include a nested mapping table with more refined mapping information. This is a way of defining nested realization specifications.

Mapping Semantics

The semantics of the mapping, of course, depend very much on the type of elements that are being mapped and the desired effect. Figure 4-16 is an example of some additional kinds of realization mappings.

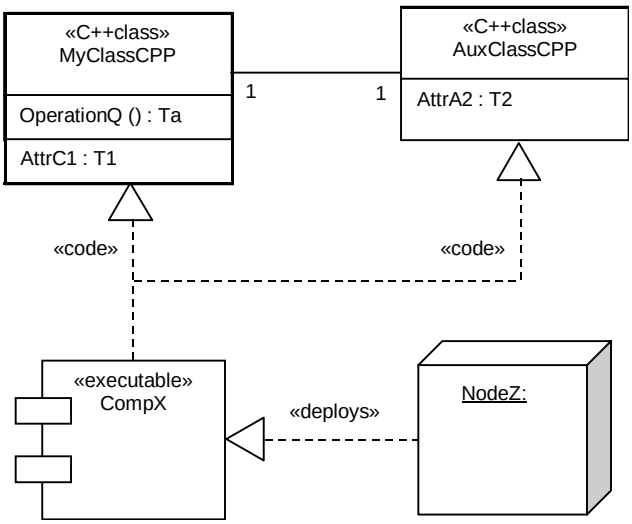


Figure 4-16 Different kinds of realization mappings

The mapping from the two classes in Figure 4-16 to the executable component (artifact) is an example of a *code* mapping, that is, the packaging of program specifications for the two classes into a unit of deployable and executable code. The mapping from the component to the node is an example of a *deployment* mapping, such as is usually represented in UML deployment diagrams.

Code and deployment realization mappings are specializations of the realizes relationship. Others specializations are also possible. For the purposes of the real-time profile, we have identified the following kinds:

«code»	A mapping from a model element whose corresponding run-time behavior is specified by a program of some type (e.g., a classifier, state machine, operation, method) to either a UML component (usually an artifact) or to a node. This mapping is used to denote that the client physically contains the program code for the supplier.
«deploys»	A general mapping that indicates that instances of the supplier are located on the client. This relationship is often defined implicitly by graphically placing the deployed artifact in the context of the deploying entity. For example, in the diagram above, instead of an

explicit deployment dependency from the node to the executable component, the component could have been placed directly inside the node symbol to indicate that it is deployed on that specific node.

«requires»	A specialization of the «deploys» mapping which is used to indicate that the client elements represent a generic specification of the minimal acceptable deployment environment required by the supplier. In effect, it is saying that if the actual deployment environment cannot satisfy the minima specified by the client of this mapping, it is not possible to guarantee that supplier will provide its full functionality or its offered QoS. We describe this mapping in more detail later in the document.

Regardless of the type of realization mapping, the following semantics seem to apply to all of them (except for the *requires* mapping):

- If the supplier fails, the client will also likely fail in some way, unless there is a one to many realization mapping between a client and a supplier (see below)
- The run-time information/data (state) of the client is collocated with the supplier and failures of that supplier may mean loss of information
- The QoS achievable by the client is constrained by the QoS characteristics of the supplier. The precise semantics of this constraint depend on the type of QoS and type of element.

When a supplier element is mapped to two or more client elements, this can mean one of the following (note that if the cardinality of a supplier element is greater than one, then this is considered to represent multiple suppliers):

- *Inclusive* – means that the functionality of the supplier is somehow realized by the collection of all of the client elements.
- *Exclusive static* –means that the client element is realized by exactly one of the supplier elements; however, once the choice has been made it will not change during the lifetime of the client element.
- *Exclusive dynamic* – means that the client element is realized by exactly one of the supplier elements and that the chosen element may vary over the lifetime of the client.

For example, if an active object is to be realized by any thread in a threadpool, it would use the exclusive static form.

The Deploys Mapping

The nature of this mapping depends on the combination of source and client types. The following combinations are defined:

- *synchronous* – This type of deployment exists when both the client and supplier are software elements and the interaction between them is in the form of procedure calls. The direction of the call is usually from the supplier to the client (*downcalls*) but it may also be in the opposite direction (*upcalls*). The effect of this mapping is additive in the sense that the functionalities of the two elements are combined into a single complex function.

- *asynchronous* – This type of deployment exists when both the client and the supplier are software elements and the interaction between them is in the form of asynchronous interactions (message sends). Other than that, the semantics of this deployment form are the same as in the synchronous case.
- *replacement* – This type of deployment exists when the client represents hardware and the supplier represents software[1]. In this case, the effect is not additive. Instead, the behavior of the supplier becomes the behavior of the client. This is syntactically similar to refinement layering, only differentiated by the fact that the software and the hardware that runs it are still different entities.

The semantics of the synchronous and asynchronous types of deployment are such that the realization relationship can be substituted in a model by a corresponding link (or association, if the model elements represent descriptors rather than instances) as shown in Figure 4-17. This represents a "collapsed" view of the layered model, which is often useful in analyses.

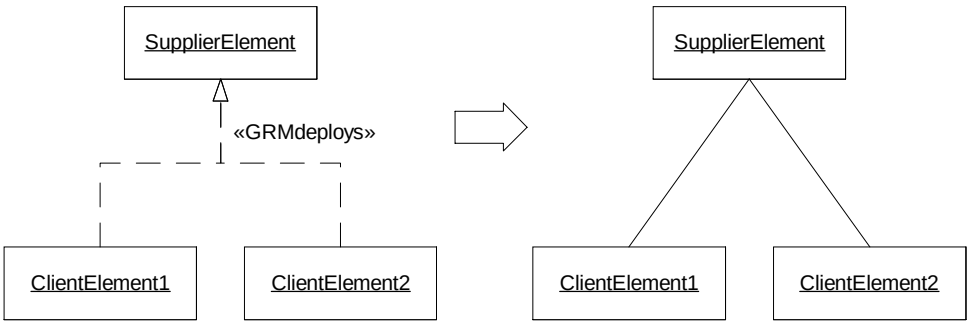


Figure 4-17 Substitution semantics of synchronous and asynchronous forms of deployment

In case of replacement deployment, the supplier element is simply replaced by its client elements, whose behavioral definition is augmented with the behavior of the supplier element.

The Requires Mapping

In real-time systems, software is often designed with specific technological capabilities in mind. For instance, a real-time software component[2] may be designed to work in an environment that supports a minimal real-time clock resolution or a minimal CPU speed. Such a component may not function properly or not function at all in an environment that does not support the requisite minima. If we want to do formal validation of such software systems, it is clearly necessary to have a means by which the *required environment of a piece of software can be formally specified.* This is simply a formal statement of the minimal QoS characteristics that must be met for the software to function according to its specifications.

---

1. The remaining case of hardware-to-hardware mappings represents refinement layering rather than deployment.
2. Note that we are using the term "component" here in the generic sense of the term rather than in its UML-specific interpretation.

The specifications of this required environment can be quite complex and may include not only basic physical properties such as memory size, CPU power, communication throughput, but also more sophisticated characteristics such as levels of availability, specific forms of concurrency, and so on. Because these environments can be arbitrarily complex, the most general way to define these characteristics is to build a model of a complete supporting execution environment whose offered QoS values represent the desired minima and then to map the software model to this environment model using standard realization mappings[1]. The underlying required environment model could be as simple as a single node but may also represent a complex distributed network of nodes and software services.

Consider, for instance, the case of some real-time system in which the external temperature is sampled by a dedicated cyclical task at the rate of 50 times a second and the current value deposited in a well-known memory location. Concurrently with temperature sampling task are two separate tasks one running at 50 times a second and the other at 25 times a second. Each of these two tasks needs to consult the current value of the external temperature as it executes its run.

Because these tasks access a shared memory location, they need to share a common address space and, therefore, must all be allocated to the same operating system process (we assume here that a “process” is an operating system entity that defines a virtual address space). However, since the three tasks are concurrent, each one has to be mapped to a separate lightweight thread within the process that provides the shared address space. Let us assume that there is a requirement that the context switching time of these threads cannot exceed 10 microseconds. Furthermore, the process that contains these threads must not have a context switching time that is greater than 80 microseconds.

The required environment in this case might be described by a model that looks as shown in Figure 4-18.

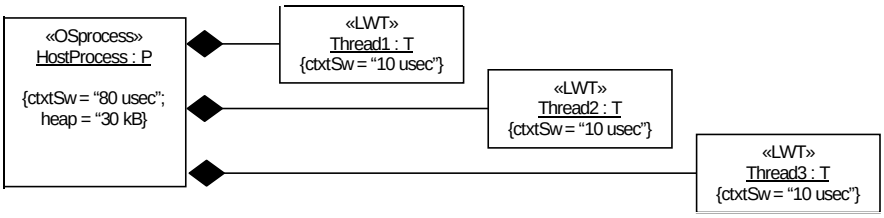


Figure 4-18 Example of a required environment specification

1. Of course, this is not and should not be the only way. In less complex cases, it may be quite sufficient to attach the required environment characteristics directly to the model elements without forcing modelers to define a complete environment model.

The complete system specification, including the required environment part would then be described by the composite model depicted in Figure 4-19.

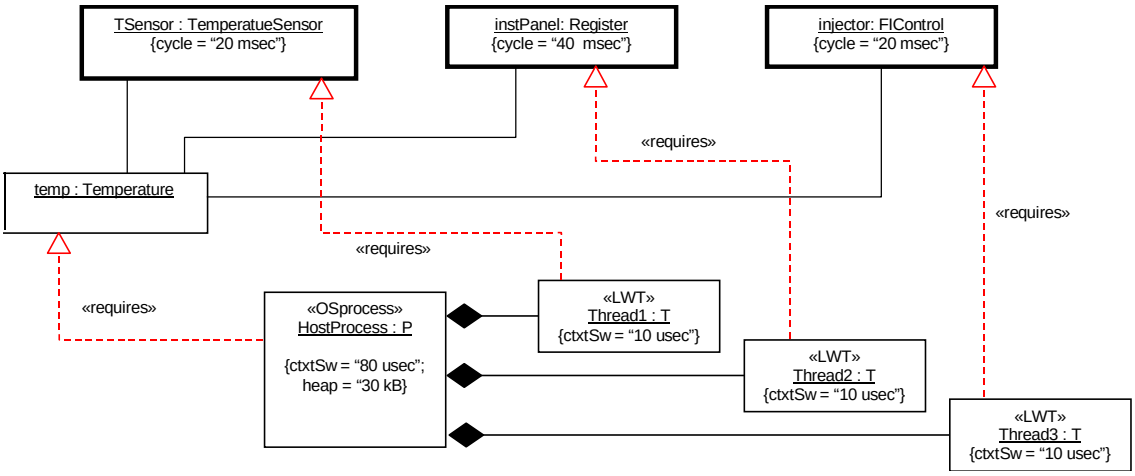


Figure 4-19 Complete system specification for the example

4.2.1.2 Mapping to UML

The three different kinds of realization mappings are all specializations of the basic UML «realize» stereotype (which is itself a specialization of the Abstraction dependency). In other words, they map onto appropriate subclasses of the Realization stereotype: «GRMcode», «GRMdeploys», and «GRMrequires».

(There is a case to be made that the “requires” mapping is not quite the same as the other two and should perhaps be modeled differently. This is because, in contrast to the code and deploys variants, the clients of that relationship do not define concrete run-time instances but merely represent a convenient means of specifying required QoS characteristics.

One possibility is to not use a relationship at all, but corresponding stereotypes of various concepts, such as Node and Classifier that would identify these as required environments rather than as actual run-time entities. However, this would require a repeated definition of the entire mapping mechanisms described above but in a different form, for each such stereotype.)

Further refinements of mapping types is defined by the following tagged values:

- *mode* – whose values can be any one of the values from the set: {“inclusive”, “exclusive static”, “exclusive dynamic”}
- *linkage* – which indicates the interaction mode between the source and client elements and can be any of the values from the enumeration set: {“sync”, “async”, “replace”}.

In tabular form, this is shown as follows (using the example from Figure 4-19):

Logical Element	*Engineering Element*	*Mode*	*Linkage*	*Additional Constraints*
tsensor	Thread1	Inclusive	replace	--
temp	HostProcess	Inclusive	replace	--
instPanel	Thread2	Inclusive	replace	--
injector	Thread3	Inclusive	replace	--

4.2.2 UML Extensions

4.2.2.1 Naming Conventions

To minimize the possibility of confusion and conflict with other profiles, we prefix all extension names pertaining to this portion of the real-time profile with the “GRM” prefix.

4.2.2.2 Profile Package

All the extensions defined in this section are part of the RTresourceModeling package.

4.2.2.3 Stereotypes and Associated Tags

The set of stereotypes and tagged values used for general resource modeling are defined in this section and are listed in alphabetical order. The semantic descriptions of the concepts that correspond to these stereotypes are provided in the sections Section 4.2, “The UML Viewpoint,” on page 4-27.

«GRMacquire»

This is a general stereotype that represents the execution of an operation that acquires a resource as defined in Section 4.1.9.2, “AcquireService,” on page 4-16.

Stereotype	Base Class	Tags
«GRMacquire»	Stimulus	GRMblocking GRMexclServ
	Message	
	ActionExecution	
	Action	
	Operation	
	Reception	
	Method	
	ActionState	
	Transition	
	SubactivityState	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
GRMblocking	Boolean	[0..1]	AcquireService::isBlocking
GRMexclServ	Reference to an Action, ActionExecution, Operation, Method, ActionState, or SubactivityState	[0..1]	AcquireService::exclusiveService
	If the stereotyped model element is a Stimulus or Message then this value can be deduced from the model element that plays the “receiver” role of the model element		

«GRMcode»

This is a specialization of the realizes stereotype that identifies the component instance that incorporates the executable code for the logical element.

Stereotype	Base Class	Parent	Tags
«GRMcode»	Abstraction	«GRMrealize»	GRMmapping

The tag definitions are the same as for “«GRMrealize»” on page 4-36.

The following constraint is defined for this stereotype:

- The client (engineering element) of this relationship must be a kind of UML Artifact that represents a binary code file
- The supplier (logical element) of this relationship must be a kind of object, link, association role, classifier, classifier role, or association.

«GRMdeploys»

This is a specialization of the realizes stereotype that identifies the actual deployment of logical elements to engineering elements.

Stereotype	Base Class	Parent	Tags
«GRMdeploys»	Abstraction	«GRMrealize»	GRMmapping

The tag definitions are the same as for “«GRMrealize»” on page 4-36.

The following constants are defined for this stereotype:

- The “replace” linkage value in the mapping expression can only be used if the client of the dependency is a kind of Node and the supplier is a kind of Classifier or Instance.
- The “sync” and “async” linkage values can only be used if both the client and the supplier are kinds of Classifier or Instance.

«GRMrealize»

This is a general stereotype that represents the realization mapping as defined in Section 4.2.1.1, “Realization Mappings,” on page 4-28). It is a subclass of the UML “realize” concept, which is itself a stereotype of the UML Abstraction concept.

Stereotype	Base Class	Parent	Tags
«GRMrealize»	Abstraction	«realize»	GRMmapping

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
GRMmapping	GRMmappingString	[0..1]	Realization Mapping::mapping table (see Section 4.2.1.1, “Realization Mappings,” on page4-28).

The “mapping” tagged value should only be used if the mapping details are not fully specified by the realization relationship itself and the details are relevant to the model analysis on hand.

«GRMrelease»

This is a general stereotype that represents the execution of an operation that releases a resource as defined in Section 4.1.9.21, “ReleaseService,” on page 4-22.

Stereotype	Base Class	Tags
«GRMacquire»	Stimulus	GRMexclServ
	Message	
	ActionExecution	
	Action	
	Operation	
	Method	
	Method	
	ActionState	
	Transition	
	SubactivityState	

The tag GRMexclServ has the same interpretation as defined in Section 4.1.9.2, “AcquireService,” on page 4-16.

«GRMrequires»

This is a specialization of the realizes stereotype that identifies the required environment for a given set of logical elements (see “The Requires Mapping” on page 4-31).

Stereotype	Base Class	Parent	Tags
«GRMrequires»	Abstraction	«GRMrealize»	GRMmapping

The tag definitions are the same as for “«GRMrealize»” on page 4-36.

4.2.2.4 Tagged Value Types

The following types of tag value strings are defined for use with the stereotypes above. We have used TVL to describe these often complex strings. They are all instances of the TVL list type. The elements of the list are typically mixtures of strings, numeric literals, TVL variable names, and TVL expressions.In representing the syntax of these types, we use the following standard BNF notational conventions:

- A string between double quotes (“) represents a literal.
- A token in angular brackets (<element>) is a non-terminal.

- A token enclosed in square brackets ([<element>]) implies an optional element of an expression.
- A token followed by an asterisk (<element>*) implies an open-ended number of repetitions of that element.
- A vertical bar indicates a choice of substitutions.

Note that TVL uses parentheses to identify arrays, commas to separate elements of arrays, and single quotes for string literals.

GRMmappingString

This string provides a string representation of the mapping table in a realization relationship. The general format for this string is given by the TVL array:

"("<TableEntry> ["," <TableEntry>]* ")"

The format for a single table entry is:

<TableEntry> ::= "(" <LogicalElementName>* ","
<EngineeringElementName>* "," <MapType> ","
<MapLinkage> "," <Constraints> ")"

where:

<LogicalElementName> and <EngineeringElementName> are string representing the full path name of the appropriate model element

<MapType> ::= 'inclusive' | 'exclusive static' | 'exclusive dynamic'

<MapLinkage> ::= 'sync' | 'async' | 'replace'

<Constraints> is either an empty string or a string representing an expression in some constraint language such as OCL[1].

4.2.3 Modeling Guidelines and Examples

For examples of the specialization of the abstract concepts of the GRM, refer to the other profile packages in this specification. Examples of the use of the stereotypes can be found in Section 4.2.1.1, "Realization Mappings," on page 4-28.

4.2.4 Required UML Metamodel Changes

The concept of an action execution (see Section 4.1.9.3, "ActionExecution," on page 4-17) figures prominently in the GRM and the various kinds of analyses that it supports. This represents the run-time execution of some action. Unfortunately, the

1. This specification does not define the semantics or syntax of such constraint expressions; this may be useful to define in future versions of the profile.

current UML 1.4 metamodel does not provide such a concept[1]. The UML notation definition document finesses over this by mapping an activation to the action whose execution is represented. Unfortunately, this is not adequate for the needs of real-time analyses, which are generally instance based. It is necessary to differentiate between the *descriptor* of an action, which may include its required QoS values, from an instance of the *execution* of that action. The latter may be characterized by some of the same QoS characteristics. What is crucial is that different executions of the same action specification could have very different individual QoS measures.

Therefore, what is required is an extension to the UML metamodel Common Behavior package. The new concept, called *ActionExecution*, is integrated into the current metamodel as indicated in Figure 4-20.

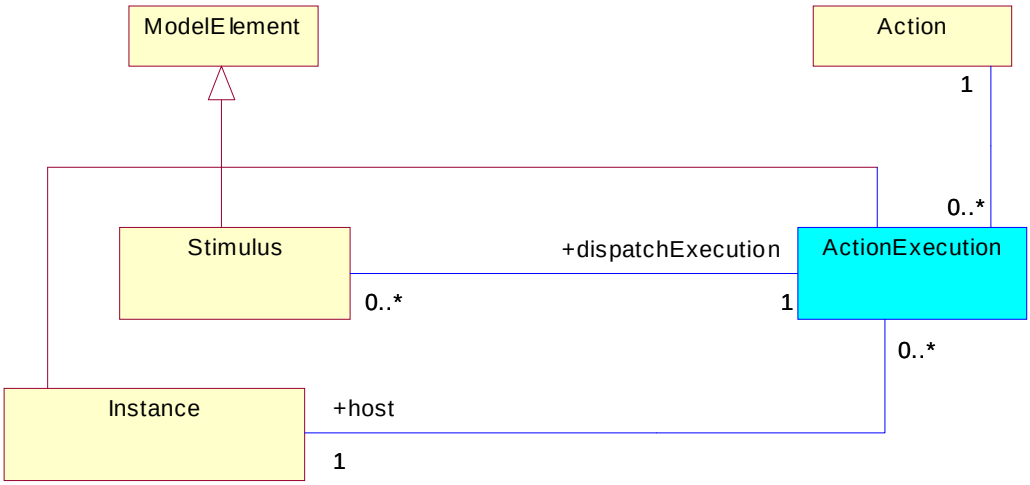


Figure 4-20 Proposed changes to the UML metamodel (ActionExecution)

The main part of the proposed metamodel change consists of the following:

- The addition of a new metamodel element called ActionExecution (defined below).
- The replacement of the association between Action and Stimulus by a semantically similar association between Stimulus and ActionExecution. This change is not strictly necessary, but it more directly relates two closely-coupled run-time concepts (Stimulus and ActionExecution). The link back from Stimulus to Action can be effected through the link between ActionExecution and Action.
- The addition of a new association between Action and ActionExecution. This takes place of the original association between Stimulus and Action.
- The addition of a new association between ActionExecution and Instance. This is also not strictly necessary, but allows the direct association between an entity and the actions that it executes. (This will also simplify the notation mapping for focus of control).

1. Strangely enough, it does have a notational equivalent, which is called an “activation,” and which is represented by so-called “focus of control” blocks in collaboration diagrams.

This has to be supported by the following additional changes to the UML Semantics document:

- The well-formedness rules for Stimulus should be amended as follows:

[1]
self.dispatchExecution.action.actualArgument->size = self.argument->size

[2]
self.dispatchExecution.oclIsKindOf (SendAction) **or**
self.dispatchExecution.oclIsKindOf (CallAction) **or**
self.dispatchExecution.oclIsKindOf (CreateAction) **or**
self.dispatchExecution.oclIsKindOf (DestroyAction)

- The Action specification should include a description of the association with ActionExecution as follows:

actionExecution the set of run-time executions of this action

- The specification of Stimulus should add descriptions for the association to ActionExecution instead of action (*dispatchAction)* as follows:

actionExecution the set of run-time action executions performed by this instance

- The definition of ActionExecution needs to be provided in the CommonBehavior section as follows:

ActionExecution

An action execution reifies the execution of an action. There can be many executions of the same action specification. An action execution is performed by an instance whose behavioral features realization includes the corresponding action. It may result in one or more stimuli to other instances (or to itself).

Associations

action	the action that is a specification of this execution
stimulus	the set of stimuli created as a result of this execution
host	the object which is executing this action

4.2.5 Proposed Notational Extensions

The notational extension required to support the GRM is the tabular form used to denote the mapping table of the various kinds of realization relationships. See Section 4.2.1.1, “Realization Mappings,” on page 4-28 for an example of such a table.

General Time Modeling 5

In this section, we describe a general framework for representing time and time-related mechanisms that are appropriate for modeling real-time software systems. These serve as a base for the standard modeling elements defined in subsequent sections.

Since real-time systems are specifically concerned with the cardinality of time (e.g., delay, duration, clock time), we shall only consider *metric* time. Thus, we do not cover in this specification so-called “logical” time models,

5.1 Domain Viewpoint

The time domain model identifies the set of time-related concepts and semantics that are supported, directly or indirectly, by this profile. Hence, any concepts or semantics not covered in this domain model are considered to be outside of the scope of this profile and must be modeled by some other means. The model is quite general, but a given application need only use the subset of the concepts and semantics that it needs.

The time domain model is partitioned into the following separate but related groups of concepts:

- Concepts for modeling time and time values.
- Concepts for modeling events in time and time-related stimuli.
- Concepts for modeling timing mechanisms (clocks, timers).
- Concepts for modeling timing services, such as those found in real-time operating systems.

The concepts are grouped into a set packages as shown below.

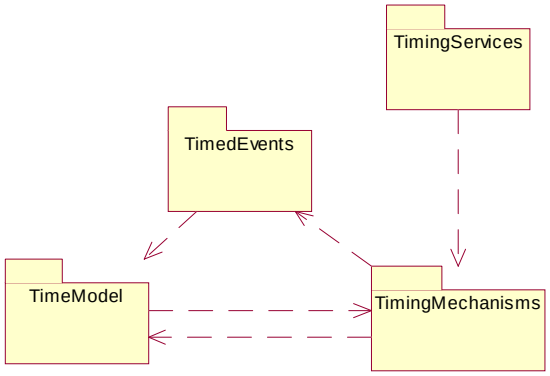


Figure 5-1 The modules of the time domain model

Note the mutual dependency between the time modeling package and the time mechanisms package. This is because it is impossible to be specific about time without referring to clocks.

All of these packages are based on the general resource model. In the following section, we provide a high-level description of the concepts in these packages. Detailed specifications of each of the concepts and their semantics are provided in the following sections.

5.1.1 The Time Model

The conceptual model for representing time and time values that is supported by this specification is shown by the UML diagram below.

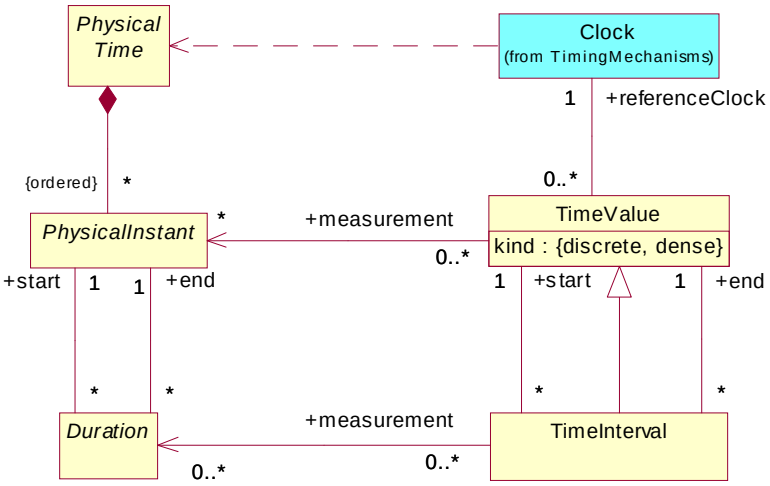


Figure 5-2 Basic time modeling concepts

In an abstract sense, physical time can be thought of as a relationship that imposes a partial order on events. We consider physical time as a continuous and unbounded progression of *physical time instants*, as perceived by some observer, such that

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.

The latter property implies that our model of physical time is continuous. However, we note that since computers only work with finite precision numbers, it is not always possible to represent physical time accurately. For this reason, we distinguish 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.

We assume that physical time progresses 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 do not necessarily apply to other models of time that may be useful in modeling. For example, we may have simulated time in which time does not necessarily progress monotonically or “virtual time” that may even regress under certain circumstances.

Since physical time is incorporeal, we typically measure its progress by counting the number of expired cycles of some strictly periodic *reference clock*[1] 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, we can obtain whatever time resolution we need 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 our domain 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 a time value, it is useful to be able to distinguish it from a time value that represents a specific instant. Hence, we introduce the semantic notion of a *time interval*.

1. For simplicity, we assume that the reference process is co-located with the observer (negligible observation delay).

5.1.2 Timing Mechanisms

In modeling the support infrastructure of real-time systems it is often necessary to explicitly identify those aspects of a model that represent timing mechanisms, that is mechanisms that measure time. There are two basic types of timing mechanisms in our domain model: *timers* and *clocks*.

Timers are mechanisms that may generate a *timeout event* when a specified time instant occurs. This may either be the instant when some clock reaches a pre-defined value or when a pre-specified time interval has expired relative to a given instant (usually the instant when the timer is started). Clocks are mechanisms that periodically cause a *clock tick event* to occur. A clock tick may in turn cause a stimulus called a *clock interrupt*.

These timing mechanisms are kinds of resources as defined in the generic resource model. The model fragment shown in Figure 5-3 depicts the timing mechanisms and related concepts.

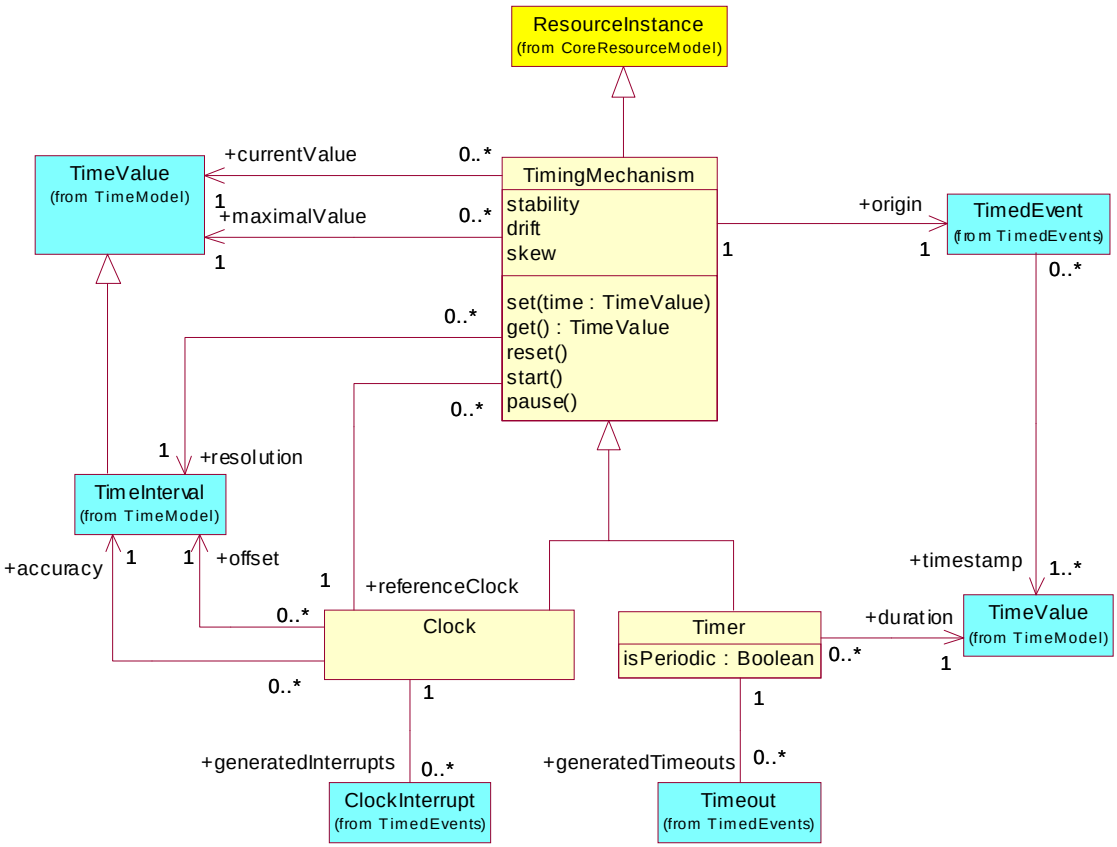


Figure 5-3 Timing mechanisms concepts

In general, a timing mechanism is any mechanism that measures the progress of time and that generates events as a result. In our domain model, we view all such mechanisms as resources, which means that they offer services that may have offered QoS values specified.

Regardless of the type of mechanism, certain properties characterize all timing devices[1]:

- a *current value* that identifies how far in time it has progressed.
- a *reference clock* to which it is somehow related.
- an *origin*, which is some clearly identified timed event from which it proceeds to measure time.
- a *maximal time value* which the current value cannot exceed (an offered QoS characteristic).
- a *resolution*, which is the minimal time interval that can be recognized by the mechanism (an offered QoS characteristic).
- a *stability* characteristic, which is the ability of the mechanism to measure the progress of time at a consistent rate (an offered QoS characteristic).
- a *skew* characteristic that identifies how well the mechanism tracks the reference clock (an offered QoS characteristic).
- a *drift* characteristic which is the rate of change of the skew (an offered QoS characteristic).

In addition, in our model we assume that timing mechanisms may provide the following standard services:

- a *set time* service for setting the appropriate time value of the mechanism (the interpretation of this is mechanisms specific).
- a *get time* service, for accessing the current value of the mechanism (the interpretation of this value is specific to each mechanism).
- a *reset* service for setting the mechanism to its initial state (which is also specific to each mechanism).
- a *pause* service, which is used to suspend the measurement of time; which means that the current value of the mechanism stop progressing.
- a *start* service, which is used to resume a paused mechanism.

If a clock has not been paused, then its *current time value* represents the total amount of time that the clock has been running (i.e., generating clock ticks) since the event of its origin, or its most recent reset, or since it last reached its maximal value (for clocks that automatically roll over when that value is reached). If it had been paused and resumed one or more times, then its current value will be reduced by the cumulative duration of the intervals during which it was paused.

The reference clock of a timing mechanism is typically some kind of near-to-ideal clock, such as a clock maintained by an international standards organization. It is generally considered desirable to keep a clock as closely synchronized with its reference clock as

1. A more precise definition of each of these characteristics can be found in [32]

possible. The *offset* characteristic of a clock at some instant is the absolute time difference between the time of its clock tick and the corresponding tick of its reference clock. The *accuracy* of a clock represents the maximum offset of a clock over time. *Drift* represents the maximum absolute difference in the relative frequencies of the clock and its reference clock between two successive ticks.

A timer is a mechanism that generates timeout events. The *current value* of a timer represents the duration of time that must expire before the timeout event occurs. A timer is always associated with a particular clock and, therefore, assumes the offered QoS characteristics of that clock.

Timers will generate a single timeout event when their duration expires, unless they are periodic. Periodic timers are timers that act like clocks (except that they generate timeout events instead of clock ticks), whose resolution is equal to the value of the timer.

5.1.3 Timed Events Model

In our domain model, as in UML, an event is assumed to occur instantaneously. That is, it takes place at a particular time instant and has no duration. An event occurrence can be associated with a time value relative to some clock to identify the time when it occurred. Of course, different observers in different inertial frames of reference or observers using different clocks may associate different time values with a given event. In our foundational model, we do not assume or preclude an absolute time reference. The choice depends on the modeling needs of the application.

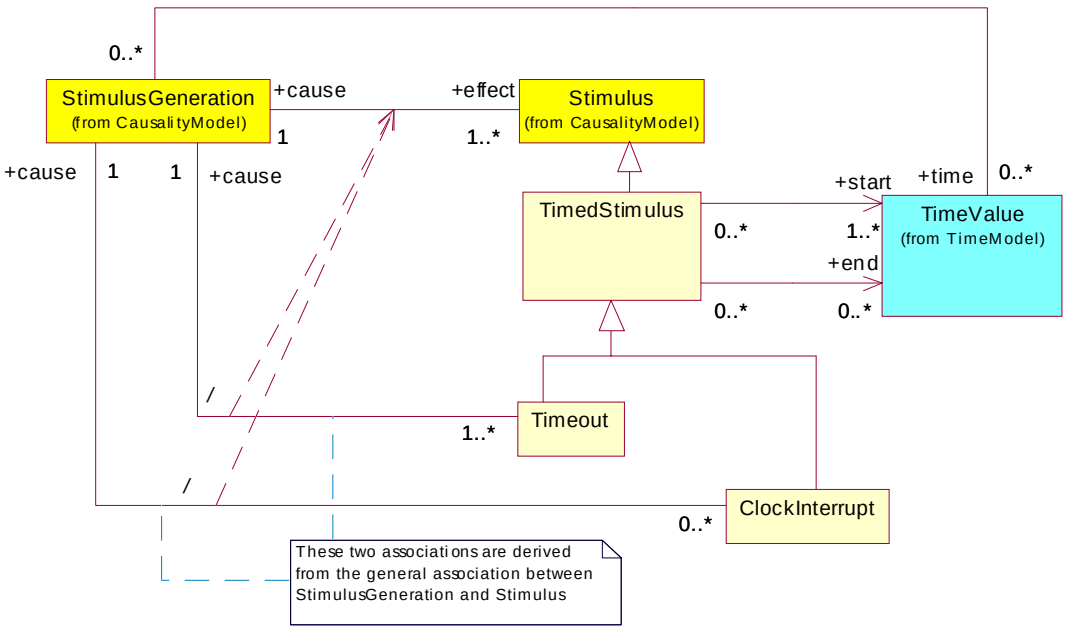


Figure 5-4 Timed stimulus concepts

As specified by the general resource model, when an event occurs, it may cause a number of stimuli to be generated. To allow modeling of stimuli that have an associated timestamp, we introduce the notion of a *timed stimulus* into the model (the same event may have multiple timestamps corresponding to different clocks). This is a stimulus that has at least one associated time value (timestamp).

There are many different kinds of event occurrences (e.g., the sending and receiving of signals, the invocation of an operation) and their corresponding stimuli. However, for the purposes of time modeling, we identify two specializations of time-related stimuli that are of particular interest:

- A *clock interrupt* represents an asynchronous signal sent by a clock mechanism, and
- a *timeout* is the generation of an asynchronous timeout signal by some timer.

A common requirement is to be able to discuss the time of occurrence of an event. For this purpose, we provide the concept of a timed event as shown in Figure 5-5. A timed event is simply an event with an associated timestamp.

It is also very useful to have a common abstraction for an action that takes time to complete: a *timed action*. This provides a generic facility for modeling behavior that has a definitive start time and a definitive end time. A special kind of timed action is a deliberate *delay* action, which delays execution for some time interval.

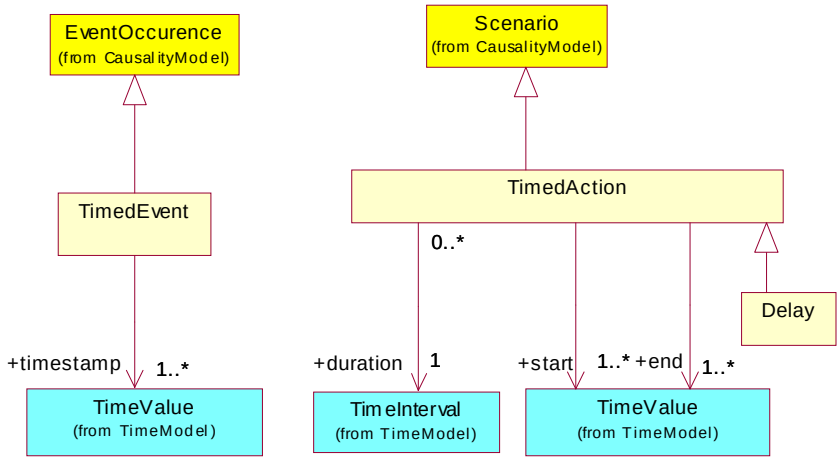


Figure 5-5 Timed action and timed event concepts

5.1.4 Modeling Timing Services

The last part of the domain model deals with concepts required to model timing services, such as those found in real-time operating systems. The domain model here is relatively straightforward and simple to allow maximum flexibility in dealing with the idiosyncrasies of individual realizations of timing services.

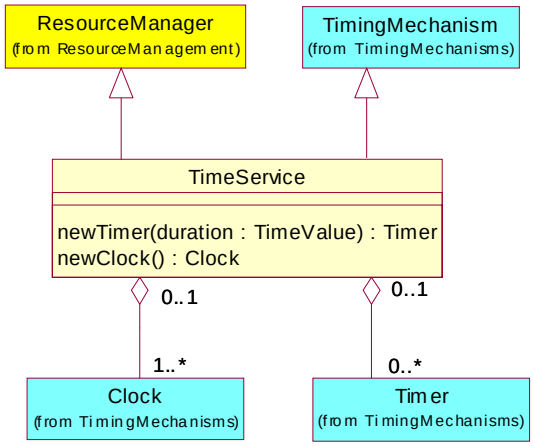


Figure 5-6 Timing service concepts

The timing service could simply be part of an operating system interface. It offers time reading and setting services directly, which is a common feature of many operating systems. Hence it too is a type of timing mechanism.

A timing service acts as a clock and timers factory – a resource manager, in effect. This means that it will create and provide either of these mechanisms on demand. Once such a mechanism is created on request from a client, the mechanism may be passed on to the client or may remain the ownership of the timing service itself.

5.1.5 Domain Concept Details and Usage

In this section we provide a more detailed explanation of each of the concepts in the general time model. Note that these are not specifications of the actual UML stereotypes, but are used as a basis for deriving such stereotypes. For each domain concept we provide a description of the semantics of each feature and association. We distinguish between concrete and abstract concepts. Concrete concepts are the ones used directly by the modeler, whereas abstract concepts are used to define common features of two or more related concepts. Abstract concepts are clearly identified as such below.

5.1.5.1 PhysicalTime [abstract]

This is the notion of physical time. A crisp definition of physical time has forever eluded scientists and philosophers alike, so we are not going to be presumptuous and attempt to do so here. Our fundamental model of physical time is that it is a continuous and unbounded progression of instants. Formally, time is viewed merely as a means of imposing a partial order on events by associating events with time instants.

We make no assumptions about whether time is global, which permits the modeling of relative time[1].

It is an abstract concept since we do not anticipate that real-time system modelers will need to represent physical time itself in their models. Instead, it is assumed that they will represent it indirectly through time values, time-measuring mechanisms, etc.

Associations

physicalInstant	the ordered set of physical instants that constitute physical time

5.1.5.2 PhysicalInstant [abstract]

This is the concept of a physical time instant – a point in time. Like geometrical points, instants do not have any extent (duration). Given any two different physical time instants, one will always precede the other.

As with physical time, we do not anticipate that system modelers will need to represent physical time instants directly, but only through associated concepts such as time measurements or events.

Associations

duration (2)	the time instant may be the start or end of any number of time intervals.
measurement	the set of time values that correspond to (i.e., measure) this instant; since a given time instant may be measured by any of a number of observers using clocks in the same or different inertial frames of reference, there can be more than one measurement corresponding to a given instant .
physicalTime	an instant is just a precisely positioned point in physical time.

1. However, since we do not provide any framework for identifying frames of temporal reference the capability to model relative time will require additions to the domain model.We consider that to be outside the scope of this profile.

5.1.5.3 Duration [abstract]

This represents the interval between two physical time instants. As with physical time, we do not anticipate that system modelers will need to represent physical durations directly, but only through associated concepts that can measure it. Hence, we do not provide direct support for modeling physical durations, but only for modeling durations as measured by timing mechanisms.

Associations

start	the instant at which the interval starts.
end	the instant at which the interval ends.
measurement	the set of time values that measure this interval; since the measurement may be done by any number of observers using clocks in the same or different inertial frames of reference, there can be more than one measurement corresponding to a given duration.

5.1.5.4 TimeValue

A value that corresponds to a particular physical instant in time as measured by some reference clock in some inertial frame of reference.

Attributes

kind	identifies whether the value represents a dense (continuous) time or discrete time type.

Associations

clockTick	the clock tick event occurrence of the reference clock which corresponds to this time value; this information is generally not expected to be used in modeling but we include it here for completeness of the domain model.
physicalInstant	the physical instants measured by this time value; note that the same value may represent multiple instance either due to clock roll-over effects or due to measurements taken in multiple inertial frames of reference.
timeInterval (2)	the time interval in which this time value is either the start or end instant.
referenceClock	the clock associated with this time value; note that a reference clock always exists, but it may be implicit.

5.1.5.5 TimeInterval

A kind of time value corresponding to a duration. There are two kinds of durations: absolute and relative. The former start and end at specific points in time whereas the latter are not rooted to any particular time.

Associations

duration	the set of physical time durations corresponding to this time

	interval.
end	the time value that represents the end instant of the measured duration.
start	the time value that represents the initial instant of the measured duration.

5.1.5.6 *TimingMechanism [abstract]*

An abstract concept that captures the common features of resources that specialize in performing time measurement and timing-related functions. The operations of the timing mechanisms represent its offered resource services and its attributes represent the corresponding QoS characteristics. Also included in the model are certain common operations that might be useful in certain types of more detailed analyses. For example, in analyzing a complex scenario, it may be important to determine that a participating timing mechanism has been reset or stopped.

Of course, since not all clocks will have available all of the QoS characteristics listed here, only those that are relevant to the case on hand need be used in modeling.

Attributes

stability	the ability of a timing mechanism to report consistent intervals in time; this is usually measured by a small number of derivatives of the clock tick rate.
drift	the maximum absolute difference between the frequency of the timing mechanism relative to the frequency of its reference clock.
skew	the rate of change of the offset between the timing mechanism and its reference clock.

Operations

set (time:TimeValue)	an operation that sets the current value of the timing mechanism to the argument.
get()	an operation that reads the value of the timing mechanism.
reset()	an operation that stops the timing mechanism and sets it back into its initial state.
start()	an operation that start the timing mechanism; if the mechanism had been stopped before, this operation resumes its function, otherwise it has no effect.
pause()	an operation that suspends the timing mechanism; if it is running, it stops all time measurement until the start operation is invoked; otherwise, it has no effect.

Associations

origin	the event occurrence relative to which the timing mechanism measures time; the current time value of the timing mechanism represents the duration between the time of this event and the present (modulo any pauses, resets, and maximum value rollovers).
currentValue	the current value of the timing mechanism.

generatedEvents	an ordered set of timed events generated by this mechanism (this is only included for completeness of the domain model and is not expected to be modeled explicitly in user models).
maximalValue	the maximum value that the current value of the timing mechanism can take.
referenceClock	the reference clock whose QoS characteristics serve as a reference for specifying the QoS characteristics of this timing mechanism.
resolution	an offered QoS attribute that identifies the minimal duration that can be distinguished by the timing mechanism.

5.1.5.7 Clock

This is a kind of timing mechanism that generates a *clock interrupt* periodically. This concept inherits most of its features from TimingMechanism.

Associations

generatedInterrupts	the ordered set of clock interrupt stimuli generated by this clock (this is only included for completeness of the domain model and is not expected to be modeled explicitly in user models).
offset	the intended time shift between the current value of the clock and its reference clock (e.g., time zone adjustment).
resolution	an offered QoS attribute that identifies the minimal duration between two successive ticks of the clock.
accuracy	the difference in the current value between the clock and its reference clock.
timingMechanism	the set of timing mechanisms for which this clock acts as a reference clock.

5.1.5.8 Timer

This is a kind of timing mechanism that generates one or more timeout signals. The timeout is generated at the instant when the duration of the timer has expired relative to the time the timer was started or to the instant when the preceding timeout was generated (for periodic timers) plus any time spent while the timer was paused. It inherits most of its features from TimingMechanism.

Attributes

isPeriodic	a QoS characteristic that identifies if the timer is periodic (when true) or not (when false); periodic timers keep generating timeout events until they are paused or destroyed.

Associations

duration	the time interval measured by the timer.
generatedTimeouts	the set of timeout events generated by this timer (this is only included for completeness of the domain model and is not expected to be modeled explicitly in user models).

5.1.5.9 TimedAction

This is a generic concept for modeling activities that either have known start and end times or that have a known duration. Note that a timed action can also be expressed in terms of its start and end events.

Associations

duration	the time interval during which the action is occurring.
start	the time of the event occurrence when the action started.
end	the time of the event occurrence when the action was completed.

5.1.5.10 TimedEvent

This is a generic concept of an event that has an associated timestamp. It can be used for all kinds of events, such as action start and end events, stimulus generation events, etc.

Associations

timestamp	the set of time values that represent the instant when the event occurred; a single event may have multiple timestamps based on different reference clocks.

5.1.5.11 TimedStimulus

This represents any stimulus that has an associated timestamp (a time value). The timestamp represents the instant of occurrence of the event that generated the stimulus. Although there are two special kinds of concrete timed stimuli related to timing mechanisms (clock interrupts and timeouts), this concept is more generally applicable and can be applied to any kind of stimulus that has a known time of occurrence, not just those that are generated by timing mechanisms.

Associations

start	the set of time values that represents the instant when the occurrence that generated the time event happened; a single event may have multiple such timestamps based on different reference clocks.
end	the set of time values that represents the instant when the reception of the stimulus occurred[1]; a single event may have multiple such timestamps based on different reference clocks.

5.1.5.12 ClockInterrupt

A kind of timed stimulus generated by the regular operation of a running clock.

Associations

1. Reception is defined as the instant of arrival of a stimulus at the receiver, but does not necessarily coincide with the start of processing of the event.

clock	the clock that generates this stimulus.
cause	the event occurrence that resulted in this clock interrupt (i.e., the “tick” of a clock).

5.1.5.13 Timeout

A kind of timed stimulus generated by a running timer.

Associations

timer	the timer that generates this stimulus.
cause	the event occurrence that resulted in this timeout (i.e., the expire of the timer’s duration).

5.1.5.14 Delay

A kind of timed action execution that represents a null operation for a pre-specified time interval. This action has no side-effects except to delay the action execution that follows it.

5.1.5.15 TimeService

This is a model of a time service (or time server). The assumption is that clients can either approach the service itself to get basic timing service capabilities or they can ask the service to provide them with a mechanism that will perform the desired service. The service is viewed as a manufacturer of timing mechanisms but not necessarily their owner, since in some systems, the ownership of timing mechanisms may be passed to the clients.

Associations

timer	the set of timers created by this service.
clock	the set of clocks created by this service.

5.2 UML Viewpoint

In this section we describe how the domain concepts can be represented in UML. Because of the flexibility built into the generic resource model, there are typically multiple different ways of representing a given domain concept in UML. First we discuss the mappings for individual domain concepts, and then introduce the actual UML extensions defined for this purpose.

5.2.1 Mapping Timing Domain Concepts into UML Equivalents

Based on the general resource model, all the domain concepts represent instances of some kind. However, in most cases it is possible to apply the domain concepts to the descriptors (classifiers, types, etc.) that define these instances. This is merely a

convention used to define default QoS attribute values that are inherited by all instances based on such descriptors. However, these default values are automatically overridden by the values specified directly on the instances themselves.

5.2.1.1 *PhysicalTime, PhysicalInstant, and Duration*

As noted earlier, it is assumed in this profile that there is no need to support the modeling of physical time directly. What is supported is the modeling of time-measuring devices and their measurements.

5.2.1.2 *TimeValue*

There are two ways to specify time values using this profile. The first is to use the «RTtime» stereotype to identify model elements that represent time values. The second is to use instances of the TVL data type RTtimeValue (or its subclasses), which is defined in this profile. This second approach is used exclusively in situations where it is required to specify the value part of a tagged value that represents time. For instance, the following tagged value specifies the resolution of a timing mechanism to be 1 microsecond:

{RTresolution = (1, ‘usec’)}

The «RTtime» stereotype approach can be used on model elements that represent data values with time semantics. For example, the initial value of an attribute of a calendar might be specified as follows:

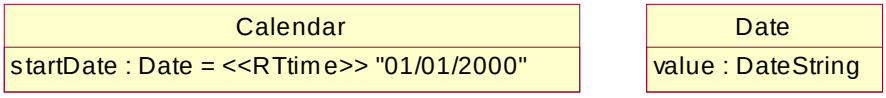


Figure 5-7 Example of the use of the «RTtime» stereotype for labelling data values

This could be quite cumbersome, since the stereotype labels will typically take up a lot of space in a diagram, possibly obscuring the actual literal value[1]. It is usually much more convenient to stereotype the appropriate data type instead. The semantics of stereotyping a classifier with the «RTtime» stereotype is that all instances of that classifier will automatically assume time semantics:

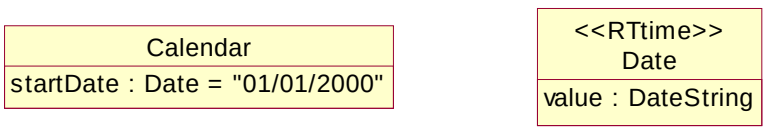


Figure 5-8 Example of the use of the «RTtime» stereotype for labelling classifiers

1. Of course, a tool could hide this information.

The reference clock of a time value is optional. However, if required, it can be specified as described in the section that describes the modeling of clocks below.

The kind of time (discrete or dense) can be specified with an optional tag RTkind, which is an enumeration consisting of two elements: ‘dense’ and ‘discrete’. The default value, which is assumed if the tagged value is not specified is ‘discrete’.

5.2.1.3 *TimeInterval*

Since time intervals are subclasses of time values, modeling of time intervals is quite similar to the modeling of time values. The «RTinterval» stereotype is used to identify instance-based concepts that represent time intervals (data values, instances, objects). This stereotype can also be applied to the descriptors of those instance concepts (data types and classifiers) as a means of specifying default values.

A time interval can be represented either as an absolute interval, in which case it has both a start and end time specified (tags RTintStart and RTintEnd respectively, or as a duration (RTintDuration tag), in which case it is a relative time interval. The values of these tags are instances of the TVL RTtimeValue data type.

5.2.1.4 *TimingMechanism*

The «RTtimingMechanism» stereotype is defined as an abstract stereotype that captures the common characteristics of timers and clocks. It is not intended to be used by modelers, who are expected to use either «RTtimer» or «RTclock» for their modeling.

The common QoS characteristics of timing mechanisms – stability, drift, skew, maximal value, origin, resolution, offset, accuracy) as well as its current value[1] – are modelled by the appropriate tagged values associated with the stereotype (RTstability, RTdrift, RTskew, RTmaxValue, RTorigin, RTresolution, RToffset, RTaccuracy, RTcurrentVal respectively). QoS attributes that are not required for a given case can be left out.

Invocations of timing mechanism operations are modelled by appropriate stereotypes of action executions or any model element that implies an action execution. These are «RTset», «RTreset», «RTstart», and «RTpause»[2]. Note that the «RTset» stereotype includes a tag, RTtimePar, an instance of the TVL RTtimeValue type, which identifies the value to which the timing mechanism will be set. Once these operations have been identified, a model analysis tool has the capability to detect their usage in a model.

The reference clock of a timing mechanism is specified as follows:

Identifying Specific Clocks

Specific clock instances can be identified in a number of ways:

---

1. Note that the current value only makes sense for instance-based concepts. Hence, it should not be defined for a descriptor type element.
2. Note that we do not define the get operation, since it does not seem useful for analyses.

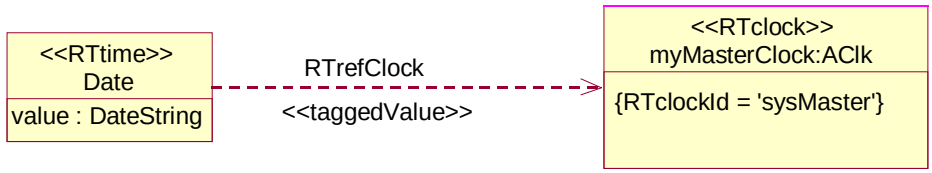
- If the model element is a kind of instance element (e.g., an instance of a UML Instance) and it is stereotyped as an «RTclock», then it may have a (unique) string name assigned using the RTclockId tag.
- If it is a standard clock, then it is not necessary to have an explicit clock model element. Instead, such a clock is identified implicitly by its well-known standard string name (UTC, TAI, etc.) as defined in Section , “RTtimeValue,” on page 5-31.

Referencing Specific Clocks

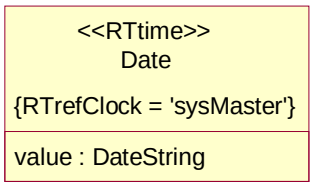
The relative nature of time sometimes requires that the source of a time value is identified, i.e., its reference clock. This profile provides the following mechanisms for identifying which clock is being used:

- By using the string name of a standard reference clock.
- By specifying a reference value for the RTrefClock tag, as shown in the example in Figure 5-9(a).
- By explicitly naming the clock using the string identified as the value field of an RTclockId tag of the appropriate model element (which requires that it had to be stereotyped as an «RTclock»), as shown in the two examples in Figure 5-9(b) and Figure 5-9(c).

a)



b)



c)

```
{RToffset = (60, 'min', ('refClk:', 'sysMaster'))}
```

Figure 5-9 Three different ways of denoting references to a specific clock

5.2.1.5 Clock

Clocks are modelled by applying the «RTclock» stereotype to instance-based concepts (instances, objects, data values, or classifier roles). If they are applied to the descriptors of those entities, then they imply that all instances of that stereotyped element are clocks and they all inherit the QoS values specified by the descriptor (unless overridden by the instance).

An instance of a clock can be identified using the RTclockId tag as described above (see Section 5.2.1.4, “TimingMechanism,” on page 5-16).

5.2.1.6 Timer

Timers are modelled by applying the «RTtimer» stereotype to instance-based concepts (instances, objects, data values, or classifier roles). If they are applied to the descriptors of those entities, then they imply that all instances of that stereotyped element are timers and they all inherit the QoS values specified by the descriptor (unless overridden by the instance).

5.2.1.7 TimedAction

This very general and very useful concept is modelled by applying the «RTaction» stereotype to any model element that specifies an action execution or its specification (which is a way of defining defaults for instances of those specifications). This includes action executions, methods, actions (including entry and exit actions of state machines), action states, subactivity states, states, and transitions. It can also be applied to stimuli (and their descriptors) to model stimuli that take time to arrive at their destination.

The start and end times of the action are specified by appropriate tagged values (RTstart and RTend respectively). Alternatively, they may be tagged with the RTduration tag. The two forms are mutually exclusive. In some cases, it may be more convenient to use a timed event designation (see below).

5.2.1.8 TimedEvent

This very general and very useful concept is modelled by applying the «RTevent» stereotype to any model element that implies an event occurrence. Since event occurrences do not show up explicitly in UML models, this stereotype can be applied to any item that implies an event, such as action executions, methods, actions (including entry and exit actions of state machines), action states, subactivity states, states, and transitions. It can also be applied to stimuli (and their descriptors) to model stimuli that take time to arrive at their destination. In all these cases, it specifies the time of the start of the associated behavior. Thus, it can be used as an alternative to timed actions where the duration or end of the action are not significant.

5.2.1.9 TimedStimulus

This concept is useful for modeling any stimulus that has an associated timestamp. This includes invocations of operations, the sending of signals, etc. as well as their descriptors. The stereotype used for this purpose is the «RTstimulus» stereotype which can be attached to stimuli or action executions of actions that generate stimuli (such as call action, send action, method, etc.) as well as their descriptors (messages, actions). In the latter case, the stereotypes are used to define default values that apply to all instances (signals and action executions) unless they are overridden with explicit tagged values associated with the instance.

The start and end times of the stimulus, corresponding to the stimulus generation and stimulus reception event occurrences are defined by the RTstart and RTend tagged values.

5.2.1.10 ClockInterrupt

This is a special type of timed stimulus that is generated by a clock. The stereotype is called «RTclkInterrupt» and it can be applied either to stimuli or messages. The start time (RTstart) represents the time of the interrupt.

5.2.1.11 Timeout

Timeouts are modelled by stimuli or messages that are stereotyped as «RTtimeout». The start time (RTstart) represents the time of the timeout.

5.2.1.12 Delay

This is modelled by a model element that is stereotyped as «RTdelay». It can only have an RTduration tag associated with it. Delays can be placed on the same model elements as timed actions (see above).

5.2.1.13 TimeService

This is represented by any instance (object, instance, classifier role) or its descriptor (classifier) that is stereotyped as «RTtimeService». The associations between the service and the clocks and timers that it creates and owns are not modelled explicitly, since this information is not germane for the type of time-based analyses that are the subject of this profile[1].

Invocations of the operations of the time service are identified by corresponding stereotypes of ActionExecution or any model element that implies an action execution: «RTnewTimer» and «RTnewClock». The stereotype «RTnewTimer» has a tag, RTtimerPar, an instance of the RTtimeValue TVL type, which can be used to specify the value of a requested timer.

5.2.2 UML Extensions

In this section we give formal definitions of the UML extensions that pertain specifically to modeling time and timing mechanisms.

5.2.2.1 Naming Conventions

To minimize the possibility of confusion and conflict with other profiles, we will prefix all extension element names pertaining to this portion of the real-time profile with the “RT” prefix.

---

1. One possibility is to define a common stereotype for dependency and composition, which can be used to associate a model element with its owner.

5.2.2.2 Profile Package

The stereotypes for modeling time and timing mechanisms are all defined in the RTtimeModeling package.

5.2.2.3 Stereotypes and Associated Tags

The set of stereotypes and tagged values used for time domain modeling are defined in this section. They are listed in alphabetical order. The semantic descriptions corresponding to these stereotypes and tagged values are provided in Section 5.2.1, “Mapping Timing Domain Concepts into UML Equivalents,” on page 5-14.

«RTaction»

This models any action that takes time (see Section 5.1.5.9, “TimedAction,” on page 5-13).

Stereotype	Base Class	Tags
«RTaction»	ActionExecution	RTstart RTend RTduration
	Stimulus	
	Action	
	Message	
	Method	
	ActionSequence	
	ActionState	
	SubactivityState	
	Transition	
	State	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
RTstart	RTtimeValue	[0..1]	TimedAction::start
RTend	RTtimeValue	[0..1]	TimedAction::end
RTduration	RTtimeValue	[0..1]	TimedAction::duration

The following constraints are defined for this stereotype:

- In a given model element stereotyped as RTaction, the tag RTduration is mutually exclusive with either the RTstart or RTend tag.
- For a given reference clock, the start value must be lesser than or equal to the end value.
- The difference between the end value and the start value, for a given clock, must be equal to the duration.

«RTclkInterrupt»

This models a clock interrupt (see Section 5.1.5.12, “ClockInterrupt,” on page 5-13).

Stereotype	Base Class	Parent	Tags
«RTclkInterrupt»	Stimulus	«RTstimulus»	RTstart (inherited) RTend (inherited)
	Message		

«RTclock»

This models a clock mechanism (see Section 5.1.5.7, “Clock,” on page 5-12).

Stereotype	Base Class	Parent	Tags
«RTclock»	DataValue	«RTtimingMechanism»	RTclockId (see parent for others)
	Instance		
	Object		
	ClassifierRole		
	Classifier		
	DataType		

Tag definitions:

Tag	Type	Multiplicity
RTclockId	String	[0..1]

«RTdelay»

This models a pure delay action (see Section 5.1.5.14, “Delay,” on page 5-14).

Stereotype	Base Class	Parent	Tags
«RTdelay»	ActionExecution	«RTaction»	RTstart (inherited) RTend (inherited) RTduration (inherited)
	Stimulus		
	Action		
	Message		
	Method		
	ActionSequence		
	ActionState		
	SubactivityState		
	Transition		
	State		

«RTevent»

This models any event that occurs at a known time instant (see Section 5.1.5.10, “TimedEvent,” on page 5-13).

Stereotype	Base Class	Tags
«RTevent»	ActionExecution	RTat
	Stimulus	
	Action	
	Message	
	Method	
	ActionSequence	
	ActionState	
	SubactivityState	
	Transition	
	State	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
RTat	RTtimeValue	[0..*]	TimedEvent::timestamp

Note that an event may have multiple timestamps, for different clocks.

«RTinterval»

This models a time interval (see Section 5.1.5.5, “TimeInterval,” on page 5-10).

Stereotype	Base Class	Tags
«RTinterval»	DataValue	RTintStart RTintEnd RTintDuration
	Instance	
	Object	
	DataType	
	Classifier	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
RTintStart	RTtimeValue	[0..1]	TimeInterval::start
RTintEnd	RTtimeValue	[0..1]	TimeInterval::end
RTintDuration	RTtimeValue	[0..1]	TimeInterval::duration

The following constraints are defined for this stereotype:

- In a given model element stereotyped as RTinterval, the tag RTintDuration is mutually exclusive with either the RTintStart or RTintEnd tag.
- The difference between the end value and the start value, for a given clock, must be equal to the duration.

«RTnewClock»

This models an invocation of the operation of a timing service that acquires a new clock mechanism (see Section 5.1.5.15, “TimeService,” on page 5-14).

Stereotype	Base Class
«RTnewClock»	ActionExecution
	Stimulus
	Action
	Message
	Method
	ActionSequence
	ActionState
	SubactivityState
	Transition
	State

«RTnewTimer»

This models an invocation of an operation of a time service that returns a new timer (see Section 5.1.5.15, “TimeService,” on page 5-14).

Stereotype	Base Class	Tags
«RTnewTimer»	ActionExecution	RTtimerPar
	Stimulus	
	Action	
	Message	
	Method	
	ActionSequence	
	ActionState	
	SubactivityState	
	Transition	
	State	

Tag definitions:

Tag	Type	Multiplicity	Description
RTtimerPar	RTtimeValue	[0..1]	TimeService::timer

«RTpause»

This models the invocation of a pause operation on a timing mechanism (see Section 5.2.1.4, “TimingMechanism,” on page 5-16).

Stereotype	Base Class
«RTpause»	ActionExecution
	Stimulus
	Action
	Message
	Method
	ActionSequence
	ActionState
	SubactivityState
	Transition
	State

«RTreset»

This models the invocation of an operation that resets a timing mechanism (see Section 5.2.1.4, “TimingMechanism,” on page 5-16).

Stereotype	Base Class
«RTreset»	ActionExecution
	Stimulus
	Action
	Message
	Method
	ActionSequence
	ActionState
	SubactivityState
	Transition
	State

«RTset»

This models the invocation of an operation that sets the current value of the timing mechanism (see Section 5.2.1.4, “TimingMechanism,” on page 5-16).

Stereotype	Base Class	Tags
«RTset»	ActionExecution	RTtimePar
	Stimulus	
	Action	
	Message	
	Method	
	ActionSequence	
	ActionState	
	SubactivityState	
	Transition	
	State	

Tag definitions:

Tag	Type	Multiplicity	Description
RTtimePar	RTtimeValue	[0..1]	TimingMechanism:: set(time:TimeValue)

«RTstart»

This models the invocation of a start operation that can be invoked on a timing mechanism (see Section 5.2.1.4, “TimingMechanism,” on page 5-16).

Stereotype	Base Class
«RTstart»	ActionExecution
	Stimulus
	Action
	Message
	Method
	ActionSequence
	ActionState
	SubactivityState
	Transition
	State

«RTstimulus»

This models a timed stimulus (see Section 5.1.5.11, “TimedStimulus,” on page 5-13).

Stereotype	Base Class	Tags
«RTstimulus»	Stimulus	RTstart RTend
	ActionExecution	
	Action	
	ActionSequence	
	Method	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
RTstart	RTtimeValue	[0..1]	TimedStimulus::start
RTend	RTtimeValue	[0..1]	TimedStimulus::end

The following constraint is defined for this stereotype:

- For a given reference clock, the start value must be lesser than or equal to the end value.
- The difference between the end value and the start value, for a given clock, must be equal to the duration.

«RTtime»

This models a time value or object (see Section 5.1.5.4, “TimeValue,” on page 5-10).

Stereotype	Base Class	Tags
«RTtime»	DataValue	RTkind RTrefClk
	Instance	
	Object	
	DataType	
	Classifier	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
RTkind	Enumeration of: {‘dense’, ‘discrete’}	[0..1]	TimeValue::kind
RTrefClk	Reference to a model element stereotyped as «RTclock»	[0..1]	TimeValue::referenceClock
	String (value of an RtclkId tag)		
	String (name of a clock standard)		

The following constraints are defined for this stereotype:

- The reference value of the RTrefClk tagged value must point to a model element that is stereotyped as an «RTclock».
- The string value of the RTrefClk tagged value must indicate a model element that is stereotyped as an «RTclock».

«RTtimeout»

This models a timeout signal or action (see Section 5.1.5.13, “Timeout,” on page 5-14).

Stereotype	Base Class	Parent	Tags
«RTimeout»	Stimulus	«RTstimulus»	RTstart (inherited) RTend (inherited)
	ActionExecution		
	Action		
	ActionSequence		
	Method		

«RTtimer»

This models a timer mechanism (see Section 5.1.5.8, “Timer,” on page 5-12).

Stereotype	Base Class	Parent	Tags
«RTtimer»	DataValue	RTtimingMechanism	RTduration RTperiodic See also parent list
	Instance		
	Object		
	ClassifierRole		
	Classifier		
	DataType		

Tag definitions (see also the list for “«RTtimingMechanism»” on page 5-30):

Tag	Type	Multiplicity	Domain Attribute Name
RTduration	RTtimeValue	[0..1]	Timer::duration
RTperiodic	Boolean	[0..1]	Timer::isPeriodic

«RTtimeService»

This models a time service (see Section 5.1.5.15, “TimeService,” on page 5-14).

Stereotype	Base Class
«RTtimeService»	Instance
	Object
	ClassifierRole
	Class

«RTtimingMechanism»

This is an abstract stereotype that provides a common base for specialized stereotypes representing specific timing mechanisms. It is not intended to be used directly in modeling (see Section 5.2.1.4, “TimingMechanism,” on page 5-16).

Stereotype	Base Class	Tags
«RTtimingMechanism»	DataValue	RTstability RTdrift RTskew RTmaxValue RTorigin RTresolution RToffset RTaccuracy RTcurrentVal RTrefClk
	Instance	
	Object	
	ClassifierRole	
	Classifier	
	DataType	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
RTstability	Real	[0..1]	TimingMechanism::stability
RTdrift	Real	[0..1]	TimingMechanism::drift
RTskew	Real	[0..1]	TimingMechanism::skew
RTmaxValue	RTtimeValue	[0..1]	TimingMechanism::maximalValue
RTorigin	String	[0..1]	TimingMechanism::origin
RTresolution	RTtimeValue	[0..1]	TimingMechanism::resolution
RToffset	RTtimeValue	[0..1]	TimingMechanism::offset

RTaccuracy	RTtimeValue	[0..1]	TimingMechanism::accuracy
RTcurrentVal	RTtimeValue	[0..1]	TimingMechanism:currentValue
RTrefClk	Reference to a model element stereotyped as «RTclock»	[0..1]	TimeValue::referenceClock
	String (value of an RtclkId tag)		
	String (name of a clock standard)		

5.2.2.4 *Tagged Value Types*

The following types of tag value strings are defined for use with the stereotypes above. We have used TVL to describe these often complex strings (see Appendix A - The Tag Value Language). They are all instances of the TVL list type. The elements of the list are typically mixtures of strings, numeric literals, TVL variable names, and TVL expressions.In representing the syntax of these types, we use the following standard BNF notational conventions:

- A string between double quotes (“) represents a literal.
- A token in angular brackets (<element>) is a non-terminal.
- A token enclosed in square brackets ([<element>]) implies an optional element of an expression.
- A token followed by an asterisk (<element>*) implies an open-ended number of repetitions of that element.
- A vertical bar indicates a choice of substitutions.

Note that TVL uses parentheses to identify arrays, commas to separate elements of arrays, and single quotes for string literals.

RTtimeValue

The general format for expressing time value expressions is described by the following extended BNF (NB: the production names are assumed to be self-explanatory):

<timeValStr> ::= (<timeStr> | <dateStr> | <dayStr> | < metricTimeStr>)
["," <clock-id>]
<timeStr> ::= <hr> [":" <min> [":" <sec> [":" <centisec>]]]
<hr> ::= "00".."23"
<min> ::= "00".."59"
<sec> ::= "00".."59"
<centisec> ::= "00".."99"
<dateStr> ::= <year> "/" <mon> "/" <dayOfMon>
<year> ::= "0000".."9999"
<mon> ::= "01".."12"
<dayOfMon> ::= "01".."31"
<dayStr> ::= "Mon" | "Tue" | "Wed" | "Thr" | "Fri" | "Sat" | "Sun"
<metricTimeStr> ::= "(" [<number> | <PDFstring>] "," <timeUnitStr>")"
<number> ::= <Integer> | <Real>
<timeUnitStr> ::= " 'ns' " | " 'us' " | " 'ms' " | " 's' " | " 'hr' " | " 'days' " | " 'wks' " | "
'mos' " | " 'yrs'

<clock-id> ::= 'TAI' | 'UT0' | 'UT1' | 'UTC' | 'TT' | 'TDB' | 'TCG' | 'TCB' | 'Sidereal' |
'Local' | <clock-string-name>

where the interpretation of the above strings is defined as follows::

TAI =	International Atomic Time
UT0 =	diurnal day
UT1 =	diurnal day + polar wander
UTC =	TAI + leap seconds
TT =	terrestrial time
TDB =	Barycentric Dynamical Time
TCG =	Geocentric Coordinate Time
TCB =	Barycentric Coordinate Time
Sidereal =	hour angle of vernal equinox
Local =	UTC + time zone

<clock-string-name> = a string name of the clock as defined in the model (see Section 5.2.1.4, "TimingMechanism," on page 5-16), or some other name, which, however, cannot be the same as any of the strings listed above.

The standard probability distribution function values are described by the following extended BNF:

<PDFstring> ::= "(<bernoulliPDF> | <binomialPDF> | <exponentialPDF> |
<gammaPDF> | <geometricPDF> | <histogramPDF> |
<normalPDF> | <poissonPDF> | <uniformPDF> "," <unitsStr>")"

where <unitsStr> is a string that identifies the metric units of the sample space (e.g., microseconds, seconds). For time-based distributions, this is specified by <timeUnitStr>.

- The *Bernoulli* distribution has one parameter, a *probability* (a real value no greater than 1):

<bernoulliPDF> ::= “ ‘bernoulli’ ,” <Real>

- The *binomial* distribution has two parameters: a *probability* and the *number of trials* (a positive integer):

<binomialPDF> ::= “ ‘binomial’ ,” <Integer>

- The *exponential* distribution has one parameter, the *mean* value:

<exponentialPDF> ::= “ ‘exponential’ ,” <Real>

- The *gamma* distribution $[(x^{k-1} \cdot e^{-(x/a)}) / (a^k \cdot (k-1)!)]$ has two parameters (“*k*” a positive integer and “*a*” the mean):

<gammaPDF> ::= “ ‘gamma’ , “ <Integer> “,” <Real>

- The *histogram* distribution has an ordered collection of one or more pairs which identify the start of an interval and the probability that applies within that interval (starting from the leftmost interval) and one end-interval value for the upper boundary of the last interval:

<histogramPDF> ::= “ ‘histogram’ , ” { <Real> “, “ <Real>}* “ , “ <Real>

- The *normal* (Gauss) distribution has a mean value and a standard deviation value (greater than 0):

<normalPDF> ::= “ ‘normal’ ,” <Real> “ , “ <Real>

- The *Poisson* distribution has a mean value:

<poissonPDF> ::= “ ‘poisson’ , ” <Real>

- The *uniform* distribution has two parameters designating the start and end of the sampling interval:

<uniformPDF> ::= “ ‘uniform’ , ” <Real> “ , “ <Real>

RTarrivalPattern

This string is used to specify concrete values of arrival patterns and has the following general format.

<bounded-string> | <bursty-string> | <irregular-string> | <periodic-string> | <unbounded-string>

Where:

<bounded-string> ::= “ ‘bounded’ ,” <time-value> “,” <time-value>

describes a bounded interarrival pattern, where the left time value is the minimal interval between successive arrivals and the one on the right is the maximum; both values are expressed using the RTtimeValue type.

<bursty-string> ::= “ ‘bursty’ ,” <time-value> “,” <integer>

describes a bursty interarrival pattern, where the time value is the burst interval expressed using the RTtimeValue type and the integer identifies the maximum number of events that can occur during that interval.

<irregular-string> ::= “ ‘irregular’ ,” <time-value> [‘,” <time-value>]*

describes an irregular interarrival pattern, where the ordered list of time values (expressed using the RTtimeValue type) represent successive interarrival times.

<periodic-string> ::= “ ‘periodic’ ,” <time-value> [“,” <time-value>]

describes periodic interarrival patterns, where the left time value defines the period and the optional second time value represents the maximal deviation; both values are expressed using the RTtimeValue type.

<unbounded-string> ::= “ ‘unbounded’ ,” <PDF-string>

describes a pattern specified by a probability distribution function defined in “RTtimeValue” on page 5-31.

5.2.3 Required UML Metamodel Changes

This package assumes that the UML metamodel is modified to support the action execution concept. This is defined in Section 4.2.4, “Required UML Metamodel Changes,” on page 4-38. In addition to these, this part of the profile requires the following new concepts.

5.2.3.1 Action Execution Timing Marks

UML 1.4 only models two specific kinds of event occurrences: stimulus generation and stimulus reception. Although they are not modeled as explicit metamodel elements, they are represented by so-called “timing marks”, *sendTime()* and *receiveTime ()* respectively. These are two operations that are defined on both messages and stimuli.

Note that a stimulus that was received may not necessarily be processed immediately since it may be queued at an active object (see, for instance, the handling of the “helloMsg” stimulus in Figure 5-10 below). For this reason, we introduce a new type of expression that can be applied to actions, called *“startTime()”*. This represents the event instant when an action starts executing. Similarly, an *“endTime()”* expression is also useful to designate the instant when an action completes execution. Thus, the duration of an action is identified by the difference between the endTime value and the startTime value.

This allows us to write constraint expressions that involve these event occurrences, such as:

{(helloHandler.endTime( ) - helloMsg.receiveTime( )) < 20}

which means that the interval between the time the “helloMsg” stimulus was received and the “helloHandler” execution completed must be less than 20 time units.

5.2.4 Proposed Notational Extensions

For real-time systems, it is often important to be able to clearly identify when certain critical events occur. Four types of event occurrences are important:

- When a stimulus (message) was generated (corresponding to the “sendTime ( )” timing mark).
- When it was received (corresponding to the “receiveTime ( )” timing mark).
- When an action execution was started (corresponding to the “startTime ( )” timing mark).
- When it was completed (corresponding to the “endTime ( )” timing mark).

These times can be shown in a diagram using the «RTstimulus» and «RTaction» stereotypes (or their subclasses) as shown in Figure 5-10.

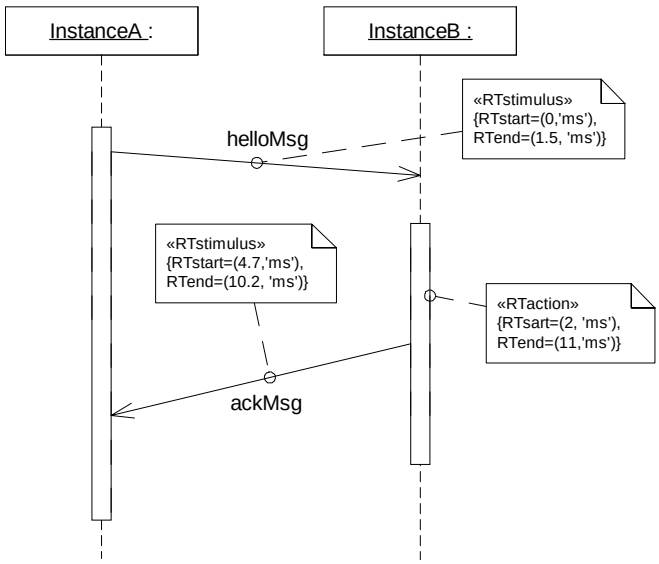


Figure 5-10 Time annotations in sequence diagrams

Since this notation clearly leads to visual clutter, we propose a shorthand representation of these in sequence diagrams, using so-called *anonymous timing marks*. These consist of only the corresponding time values of the related events. The interpretation of such annotations is the following:

- Anonymous timing marks attached to the point where the stimulus (or message) originates, represent the time of the stimulus origination event (start time of the stimulus), and
- Anonymous timing marks attached to the point where the stimulus (or message) joins an object lifeline represents the occurrence of the stimulus reception event (end time of the stimulus).

Analogous rules apply to action executions (or actions).

The value of an anonymous timing mark may either represent an absolute or a relative time value. If the values are relative, then they are relative to the event whose timing mark value is zero. If no timing marks have a value of zero, the interpretation of the values is application specific (i.e., they may be relative or absolute depending on convention).

The notational convention for timing marks associated with action executions is similar to the convention for messages. An anonymous timing mark attached to the start of a double solid line that represents an action execution represents a start time mark, while one attached to the end of that line represents an end time mark. If the receive time and send time coincide, then one mark is sufficient.

The proposed notation for anonymous timing marks is illustrated below.

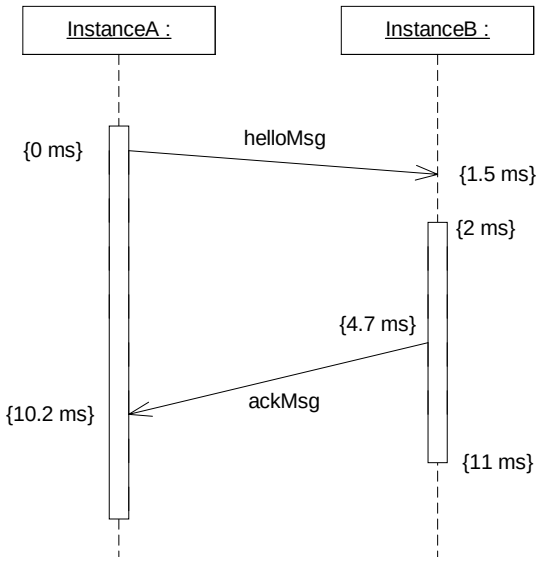


Figure 5-11 Anonymous timing marks

This diagram is interpreted as follows: at some point in its execution, the instance identified as “InstanceA” generates a “helloMsg” signal, which is received 1.5 microseconds later at the instance identified by “InstanceB”. However, when this stimulus arrives, it is not processed immediately (e.g., due to a scheduling delay), but half a millisecond later (i.e., its start time is 2 milliseconds after the “helloMsg” signal was sent). The action execution caused by the response to “helloMsg” ends 11 milliseconds from the start, which means that the entire action executed in 9.5

milliseconds. While it was executing, the action sent an “ackMsg” signal to “InstanceA”. This signal took 5.5 milliseconds (10.2 - 4.7) to arrive at its destination, 10.2 milliseconds after the original “helloMsg” signal was sent.

To further reduce visual clutter, all the timing marks can be shown on one or the other side of the diagram connected to their corresponding events using horizontal dashed lines (Figure 5-12).

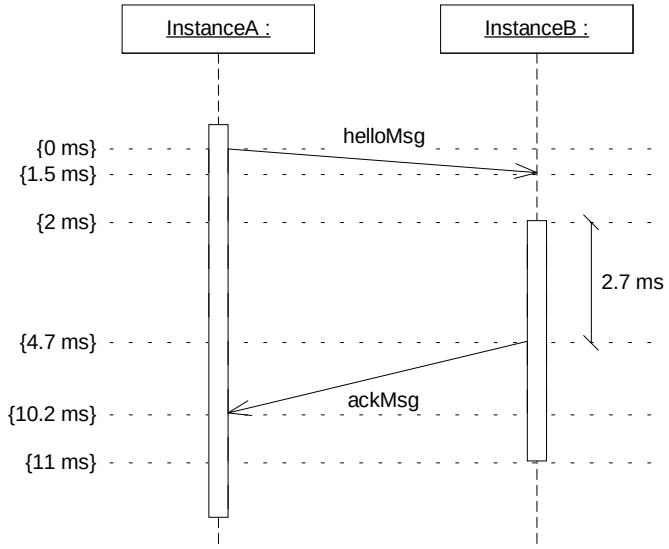


Figure 5-12 Alternative notation for Figure 5-11

General Concurrency Modeling 6

The general concurrency model serves two primary purposes:

- It enables modelers to describe a rich enough domain model of concurrently executing and communicating objects that can serve as a base for more concrete analysis models.
- It enables providers of real-time system infrastructures (e.g., operating systems) to describe the concurrency and communication mechanisms of their system.

6.1 Domain Viewpoint

The purpose of the domain viewpoint is to define and describe the basic concepts of the concurrency domain and their relationships, as supported in this specification.

6.1.1 Concurrency Domain Model

The domain model showing the general concurrency concepts is shown in Figure 6-1.

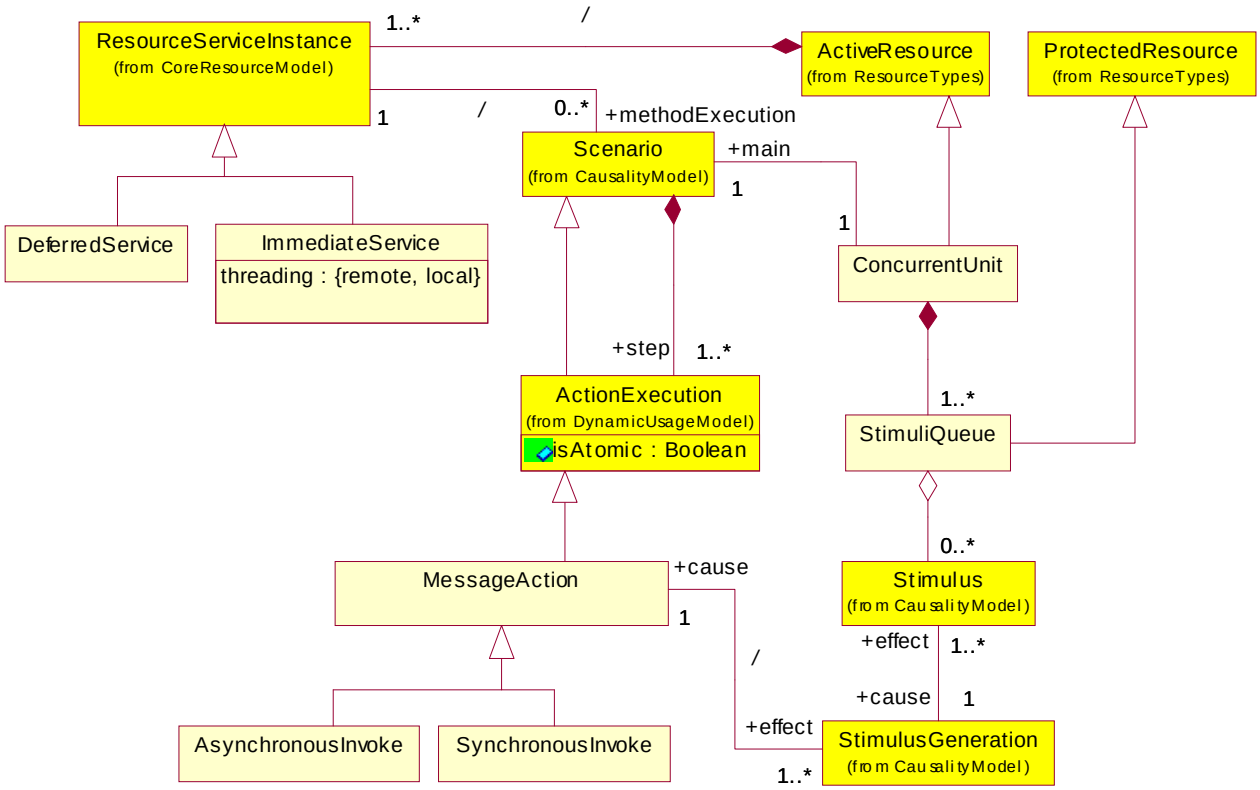


Figure 6-1 General concurrency modeling concepts

The general concurrency model is based on the causality model of the GRM (see Section 4.1.2, “The Causality Model Package,” on page 4-6). As actions (which are parts of scenarios) execute, they generate stimuli. In the concurrency model we specifically identify so-called *message actions*. These are action executions that generate one or more stimuli. Following the standard causal loop, a stimulus targets a particular service instances of a specific object instance. This causes the execution of the scenario corresponding to the method associated with the resource service instance. This leads to further action executions, and so on.

For the concurrency model, of particular interest is the notion of a *concurrent unit*[1], an active resource instance that executes concurrently with other concurrent units. Ultimately, all behavior in the system is a consequence of actions executed by concurrent units. Following creation[2], each concurrent unit commences to execute one *main* scenario

1. Concurrent units are also known as “active” units, but we will avoid this term to reduce the possibility of confusion with other types of active resources.
2. The direct causal link between object creation and commencement of execution is not defined in this model.

(method execution). This scenario executes until the concurrent unit is terminated. During its execution, the main method execution (a scenario) may perform explicit receive actions in order to accept any stimuli sent to it. A receive action by a concurrent unit leads directly to the activation of the appropriate service instance and its service method. During the execution of the service method the main method may either be blocked (the so-called “run-to-completion” paradigm), or it may proceed executing concurrently.

Of course, a stimulus may arrive before the targeted concurrent object is ready to receive it. In such situations it may be necessary to defer the response to the stimulus until the corresponding receive action is executed (as we shall see later, whether the decision to defer depends on the service that is invoked). For this reason, a concurrent unit needs one or more *queues* for holding deferred stimuli. (Multiple queues may be used to differentiate between stimuli of different priorities or sources.)

There are two choices at either end of the communication, which affect the detailed causality between concurrent threads of control.

At the server end, the service request may either be handled *immediately*, or *deferred*. In the immediate case, a further property describes whether the receiving instance creates its own concurrent execution thread to handle the service request (the so-called *local* option), or assumes that there is an existing thread available (the *remote* option).

At the receiver end, the message action may either represent an *asynchronous* or *synchronous invocation* of the service. If the request is asynchronous, then execution proceeds immediately; if the request is synchronous then the client is blocked until a response is received from the receiver.

Instances that are not concurrent do not have a main method and, hence, have no direct choice in controlling how a service request is handled.

6.1.2 Domain Concepts (Detailed)

In this section we provide a detailed explanation of each of the concepts in the general concurrency model. Note that these are not specifications of the actual UML stereotypes, but are used as a basis for deriving those stereotypes. The actual stereotypes are specified in the section Section 9.2.2, “UML Extensions,” on page 9-7.

Model associations, where they cannot unambiguously be derived from existing UML relationships, will be defined as stereotypes of the standard UML Usage dependency. (In the UML metamodel, a usage dependency means that the client requires the supplier.)

6.1.2.1 ActionExecution (extended)

This is a direct extension of the action execution concept from the GRM (see Section 4.1.9.3, “ActionExecution,” on page 4-17) that explicitly adds the notion of pre-emption.

Attributes

isAtomic	a Boolean that indicates whether the action is atomic, or can be pre-empted during its execution.

6.1.2.2 ConcurrentUnit

A kind of resource instance that is capable of executing a single scenario concurrently with other concurrent units. This scenario corresponds to the so-called “main” method associated with the resource type. It starts execution after the unit is created and continues until the object is terminated. In the course of execution of this scenario, message actions may be executed as well as actions to explicitly receive stimuli from other concurrent units. An explicit receive action may result in the execution of a method corresponding to the service invoked by the received stimulus.

Associations

main	the unique scenario execution performed by this concurrent unit.
stimuliQueue	a set of queues used to hold stimuli whose handling has been deferred until they are accepted by an explicit receive action.
resourceServiceInstance	(inherited from ResourceInstance) the set of service instances supported by this concurrent unit.

6.1.2.3 MessageAction

An action execution that results in the occurrence of a stimulus generation, which leads to the creation and dispatching of a stimulus. This is an abstract concept, inspired by the work on UML action semantics.

Associations

invoke	a service on a deployable unit.
effect	the set of stimulus generation occurrences that will lead to stimuli generated as a result of this action.

6.1.2.4 StimuliQueue

A protected resource that is used to store stimuli whose processing has been deferred by the concurrent unit. This can be used for explicit modeling of queues required in certain types of analyses.

Associations

stimulus	the set of stimuli instances that are queued on this concurrent unit.
concurrentUnit	the instance of the concurrent unit that owns this queue.

6.1.2.5 SynchronousInvoke

A kind of action execution that results in the invoking scenario being blocked until the associated service scenario is completed.

6.1.2.6 AsynchronousInvoke

A kind of action execution in which the invoking scenario continues with its execution after the stimulus is generated.

6.1.2.7 DeferredService

A kind of service instance that is deferred until the receiving object is willing to receive it explicitly.

6.1.2.8 ImmediateService

A kind of service instance that is handled immediately (subject to exclusivity).

Attributes

threading	indicates whether a new scenario execution is created to handle this reception (the “local” option), or whether the invoking scenario execution is used (the “remote” option).

6.2 UML Viewpoint

In this section we describe how the domain concepts are realized in UML including the use of extensions defined in this profile.

6.2.1 Mapping Concurrency Domain Concepts into UML Equivalents

6.2.1.1 Representing Concurrency

A concurrently executing entity is indicated by stereotyping any resource (classifier or instance) with the «CRconcurrent» stereotype. The main thread of the active instance is represented by a tagged value on a classifier, referencing a method; it is always derived for a concurrent instance, from its classifier.

6.2.1.2 Representing Causality

Causality within procedures is dealt with largely using standard UML concepts, such as Action Sequence and Action. However, methods need to be linked to actions, which is represented by a stereotype («CRcontains») of the standard usage relationship.

Causality between procedures is dealt with by stereotyping both actions (in this case Call and Send), and Operations and Receptions (or Messages and Stimuli as convenient). Calls and Sends may be stereotyped as either «CRasynch» or «CRsynch» to indicate whether they wait. Operations and Receptions may be stereotyped either «CRimmediate», or «CRdeferred» to indicate how messages are dealt with. In the case of immediate handling, the indication of threading is dealt with by a Tag Value.

The linkage between message action and handling procedure is achieved using standard UML associations.

6.2.1.3 Atomicity

Atomicity of actions is dealt with by applying a stereotype («CRaction») and setting the ‘CRatomic’ Boolean tagged value.

6.2.2 UML Extensions

In this section, we provide formal definitions of the UML extensions that are used to denote concurrency domain concepts.

6.2.2.1 Naming Conventions

To minimize the possibility of conflict with other profiles, we will prefix all extensions related to this part of the real-time profile with the “CR” prefix.

6.2.2.2 Profile Package

The stereotypes for modeling concurrency concepts are all defined in the RTconcurrencyModeling package.

6.2.2.3 Stereotypes

«CRaction»

Represents an action execution in the domain model (see Section 6.1.2.1, “ActionExecution (extended),” on page 6-3).

Stereotype	Base Class	Tags
«CRaction»	Action	CRatomic
	ActionExecution	
	Message	
	Stimulus	
	Method	
	ActionState	
	SubactivityState	
	Transition	
	State	

the tag is defined by:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
CRatomic	Boolean	[0..1]	ActionExecution::isAtomic

«CRasynch»

Represents the concept of an asynchronous invocation (see Section 6.1.2.6, “AsynchronousInvoke,” on page 6-5).

Stereotype	Base Class
«CRasynch»	Action
	ActionExecution

«CRconcurrent»

Represents a concurrent unit concept (see Section 6.1.2.2, “ConcurrentUnit,” on page 6-4).

Stereotype	Base Class	Tags
«CRconcurrent»	Node	CRmain
	Component	
	Artifact	
	Class	
	Instance	

the tag is defined by:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
CRmain	A reference to a Method model element	[0..1]	ConcurrentUnit::main
	A String that contains the full path name of a method		

The tag value must be constrained to point to a method.

«CRcontains»

Represents a stereotype of the standard UML Usage dependency and is used to model various domain relationships that may not be present or easily discernible within a model.

Stereotype	Base Class
«CRcontains»	Usage

«CRdeferred»

Represents the concept of deferred receive (see Section 6.1.2.7, “DeferredService,” on page 6-5).

Stereotype	Base Class
«CRdeferred»	Operation
	Reception
	Message
	Stimulus

«CRimmediate»

Represents the concept of an immediate service instance (see Section 6.1.2.8, “ImmediateService,” on page 6-5).

Stereotype	Base Class	Tags
«CRimmediate»	Operation	CRthreading
	Reception	
	Message	
	Stimulus	

the tag is defined by:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
CRthreading	Enumeration: {‘remote’, ‘local’}	[0..1]	ImmediateService::threading

«CRmsgQ»

Represents the concept of a stimuli queue (see Section 6.1.2.4, “StimuliQueue,” on page 6-4).

Stereotype	Base Class
«CRmsgQ»	Instance
	Object
	Class
	ClassifierRole

The model element stereotyped with this stereotype must be in a composition relationship (as the owned element) with a model element that is stereotyped as «CRconcurrent».

«CRsynch

Represents the concept of a synchronous invoke (see Section 6.1.2.5, “SynchronousInvoke,” on page 6-4).

Stereotype	Base Class
«CRsynch»	Action
	ActionExecution

6.2.3 Required UML Metamodel Changes

This package assumes that the UML metamodel is modified to support the action execution concept. This is defined in Section 4.1.9.3, “ActionExecution,” on page 4-17. No other metamodel changes are assumed or required.

6.2.3.1 Relationship to Action Semantics

The notion of procedure is taken from Action Semantics, and the two types of service invocation, but as yet nothing else. A more expressive version of this model would need to lean heavily on the Action Semantics profile. As yet, although the concepts are taken from the Action Semantics work, the profile does not assume it.

6.2.4 Modeling Guidelines and Examples

Figure 6-2 shows a sequence diagram where the various model elements have been stereotyped from the concurrency model.

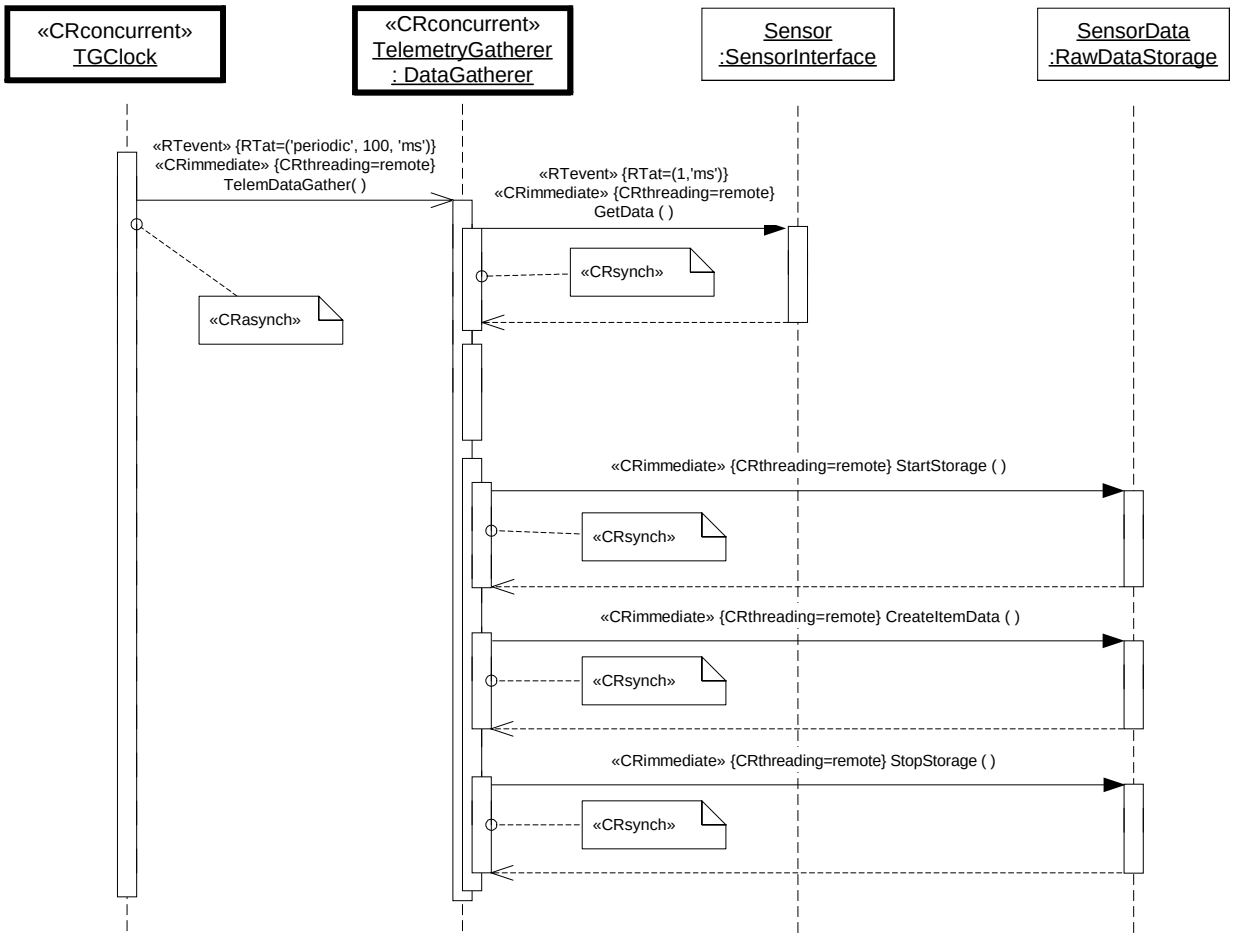


Figure 6-2 Use of Concurrency Model - Synchronous Case

A 100ms clock, which is an active instance, triggers the gathering of telemetry data (in its own thread). During the gathering process, data is obtained from the sensor and then placed in to storage so that later processing elements can access it.

Notes:

- This is an instance collaboration and so the model elements are instance elements: Class Instance (Object); Action Execution, and Stimulus.
- The stereotype indications and property strings are shown in two forms (in conformance with UML notational conventions):
  - in notes attached to the symbol that maps to the model element being stereotyped;
  - in the body of the symbol;

- It is assumed that the focus-of-control bars can be used to represent action executions as well as actions.
- Often the properties of the action executions may be derived from the actions (descriptors), for example whether or not an invocation is synchronous or asynchronous.

Figure 6-3 shows how data is displayed in the Telemetry System; this features some asynchronous communication.

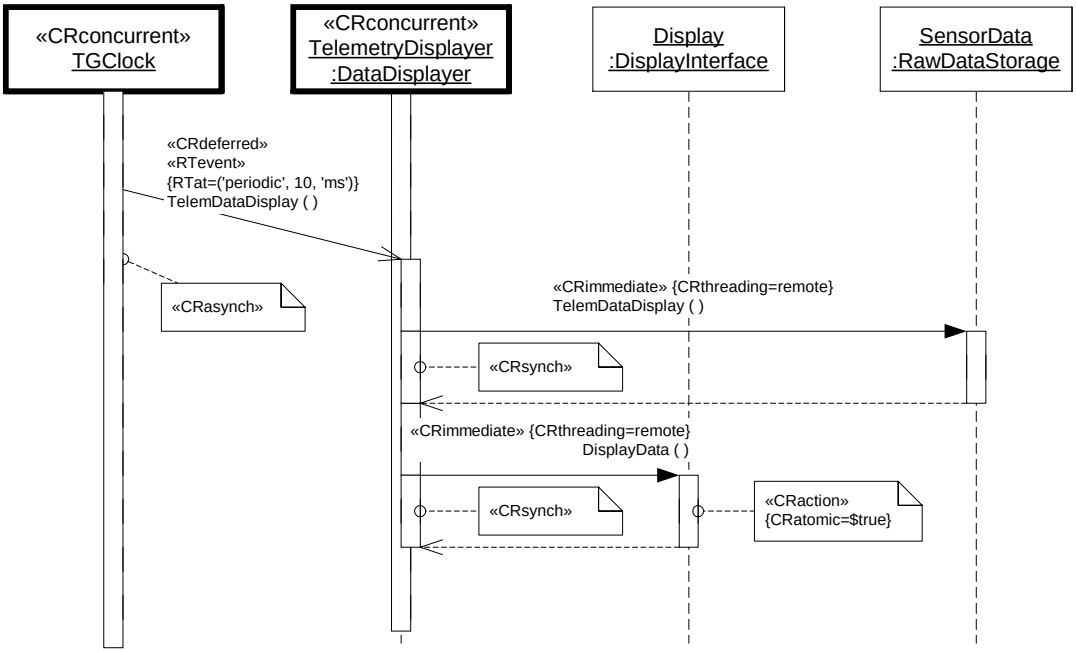


Figure 6-3 Concurrency Model - asynchronous case

In this case a 10ms clock calls TelemetryDisplayer asynchronously and then continues. Meanwhile, Telemetry Displayer handles the service in a deferred fashion, although in this case it is already waiting for it. It then gets the data from the sensors in one operation, and then displays the data. The DisplayData operation needs to be atomic (i.e., not be preempted) because the display needs to be updated in one chunk.

6.2.5 *Proposed Notational Extensions*

No additional notational extensions are provided for modeling concurrency.

Schedulability Modeling 7

Note – The team intended this model analysis sub-profile as a general base for supporting a wide variety of known and future schedulability analysis methods. However, given that (a) the major tools available in the market place are all based on rate monotonic analysis and (b) the members of the team had experience and expertise predominantly with rate-monotonic analysis, it is possible that this specification may be unintentionally biased towards that method.

In this chapter, we describe a component of the proposed profile that is intended specifically for schedulability analysis. Typical tools for this class of model analysis provide two important functions. The first is to calculate the schedulability of the system; that is, the ability of the system to meet all of the deadlines defined for the individual scheduling jobs. Such tools typically indicate which entities are schedulable and which are not. The second function is assistance with determining how the system can be improved. That may mean suggestions for making an entity schedulable or it may mean epitomizing system usage for a more balanced system. A system designer will typically want to analyze the system under several scenarios using different parameter values for each scenario while maintaining the same overall system structure.

This chapter describes a minimal set of common scheduling annotations. This minimal set furnishes enough information to perform very basic schedulability analysis. Each vendor is encouraged to supply a profile that extends this set in order to perform model analysis that is more extensive.

The structure of this section follows the convention adopted throughout this document: First, a domain viewpoint is described which identifies the basic abstractions used in schedulability analyses. The semantics of these abstractions and their relationships are explained with the aid of a UML model. The second part of the chapter describes how these abstractions are expressed in the UML metamodel. This is done through a series of UML extensions (stereotypes, constraints, and tag definitions). Supplementing this description is a set of illustrative examples showing common ways of applying this part of the profile.

7.1 Domain Viewpoint

7.1.1 Background

Scheduling jobs are the granular parts of an application that contend for use of the execution resource that executes the schedule – the processor (called more generally an execution engine).

By a *schedule*, we mean an assignment of all the scheduling jobs in the system on available execution engines, produced by the scheduler. For systems of execution engines, the schedule is the union of the schedules on the individual execution engines.

Scheduling jobs are scheduled and any required resources are allocated according to a chosen set of scheduling algorithms and resource arbitration policies. The component that implements these algorithms is called the *scheduler.*

Specifically, schedulers assign execution engines to scheduling jobs, or equivalently, assign scheduling jobs to execution engines. We say that a scheduling job is scheduled in a time interval on an execution engine if the execution engine is assigned to the scheduling job, and hence the scheduling job executes on the execution engine, in the interval. The total amount of execution engine time assigned to a scheduling job according to a schedule is the total length of all the time intervals during which the scheduling job is scheduled on some execution engine.

A *scheduling policy* is defined to be the methodology used to establish the order of scheduling job (e.g. process, or thread) execution. It is a combination of optimality criteria and algorithm.

A scheduling algorithm is chosen to provide schedule optimality according to desired scheduling optimization criterion. Example real-time scheduling optimization criteria include: meet all hard deadlines; minimize the number of missed deadlines; minimize the mean tardiness. (The most common soft-real-time scheduling optimization criterion is to maximize flow a.k.a. throughput.). Examples of *scheduling algorithms* include rate monotonic, earliest deadline first, minimum laxity first; maximize accrued utility, and minimum slack time. In general, a given scheduling optimization criterion can be optimized by more than one scheduling algorithm. For example, the Earliest Deadline First (EDF) policy is optimal for different optimization criteria, including meeting all hard deadlines, and a more relaxed one of minimizing the maximum lateness.

A *scheduling mechanism* defines an implementation technique used by a scheduler to make decisions about the order to choose threads for execution. Examples of scheduling mechanisms include fixed priority schedulers, and earliest deadline schedulers. The algorithms chosen sometimes have the same name as the scheduling mechanism. The term rate monotonic scheduler is often used but can be more accurately characterized as a rate monotonic scheduling policy and a fixed priority scheduling mechanism.

To analyze the schedulability of a system, three things must be understood:

- A *scheduling policy* or algorithm by which to assign scheduling order.
- A *resource arbitration scheme* (allocation control policy) by which allocation decisions about resources can be made.

- A *model analysis method* by which parameters which affect scheduling and resource usage can be assigned.

7.1.2 Types of Model Analysis Methods

Two major categories of scheduling policies, and therefore two types of analysis, are available. One category is *static* in nature - i.e., parametric decisions about scheduling importance are all made "up-front" and the entire collection of execution possibilities and contexts is known beforehand. The other category involves *dynamic* scheduling - i.e., scheduling decisions are made at runtime using information available within the dynamic context of execution. It is the intention of this specification to support both categories.

Depending on the policy, parameters like scheduling priority may be statically determined by the analyst, with or without the aid of model analysis tools, or dynamically by portions of the system that continuously analyze context and adjust internal parameters like priority. Earliest Deadline First scheduling is an example of such a dynamic activity. Deadlines- the amount of time remaining in which the defined work of a thread must be done – changes continually when that thread is not running. This means that the earliest deadline is a dynamically changing value. Rate Monotonic Analysis, on the other hand, is determined from the complete static set of schedulable threads, their resources, and rates of invocation.

7.1.2.1 Static scheduling and related model analysis

Rate Monotonic Analysis

Rate Monotonic Analysis assigns scheduling priority to periodic scheduling jobs by ordering scheduling priority according to the frequency of repetition of execution, i.e. the rate by which a periodic scheduling job needs to be scheduled to execute. The name Rate Monotonic means that the priority ordering is a monotonic function of the rate of execution. This model analysis technique can be extended to include both periodic and sporadic scheduling jobs. Detailed discussion of these topics can be found readily in the literature.

Deadline Monotonic Analysis

Rate Monotonic Analysis is used for analysis of periodic scheduling jobs where the deadline coincides with the next required execution to start – i.e. the period and the deadline are the same. Sometimes this is not the case. A slight variation of RMA is deadline monotonic analysis where the deadline for a periodic scheduling job need not be the same as its period. Detailed discussion of deadline monotonic analysis can be found readily in the literature.

Dynamic Scheduling - value or utility based scheduling

Dynamic Scheduling deals with the condition where the values used to order the scheduling of the CPU are a changing function over time. Therefore, dynamic scheduling uses a scheduler that makes decisions based on importance of each scheduling job, but the importance is continuously reexamined within the dynamic

context of execution of the system containing the scheduler.This class of scheduling policy is often called value based or utility based scheduling; it uses a supplied function (which may be but doesn't have to be a function of time, v(t)) to obtain a value for scheduling importance.

Earliest deadline first is a simple concrete example of a specific value function; it is a widely used scheduling policy implemented in a dynamic scheduling manner in many domains, including the telecommunications community. Although earliest deadline is a popular value function the notion can be generalized to any value function that makes sense for a specific domain.

Value based scheduling is currently receiving significant attention.

7.1.3 Domain Concepts Details

It should be noted that schedulability analysis is inherently instance-based. That is, it is generally not meaningful to analyze generic (descriptor-based) specifications consisting of classes and associations, since they abstract away much of the specific quantitative information that is necessary to determine schedulability. Instead, model analysis can only be performed on specific instantiations of those generic specifications. However, most UML models are typically a mixture of both generic and instance type specifications. As a result:

- Stereotypes apply to both instance concepts as well as generic descriptor concepts.
- The tag values of descriptor-type elements should be viewed as defaults for derived instances, which can override the defaults.
- The only way that a descriptor-based specification can be analyzed is in the special case where there is precisely one instance created from every descriptor.

There is a one-to-one mapping between the domain concepts and the stereotypes defined in the next section. Thus the process for adding schedulability domain concepts to a UML model is to add the appropriate stereotype to an existing model element, or in the case of the various conceptual associations to add an appropriate UML dependency and then to stereotype it with the corresponding stereotype.

Figure 7-1 depicts a general schedulability model that identifies the basic abstractions and relationships used in schedulability analysis. This model is defined as a specialization of the core models, as shown in Figure 7-2. This allows us to inherit the built-in QoS framework.

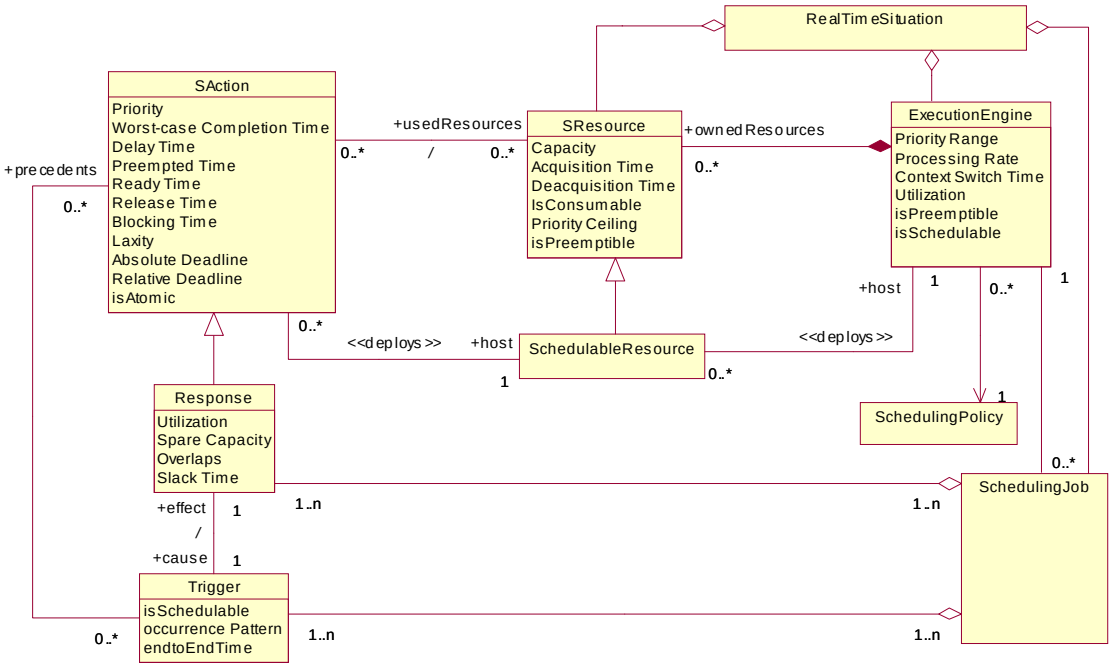


Figure 7-1 The core schedulability model

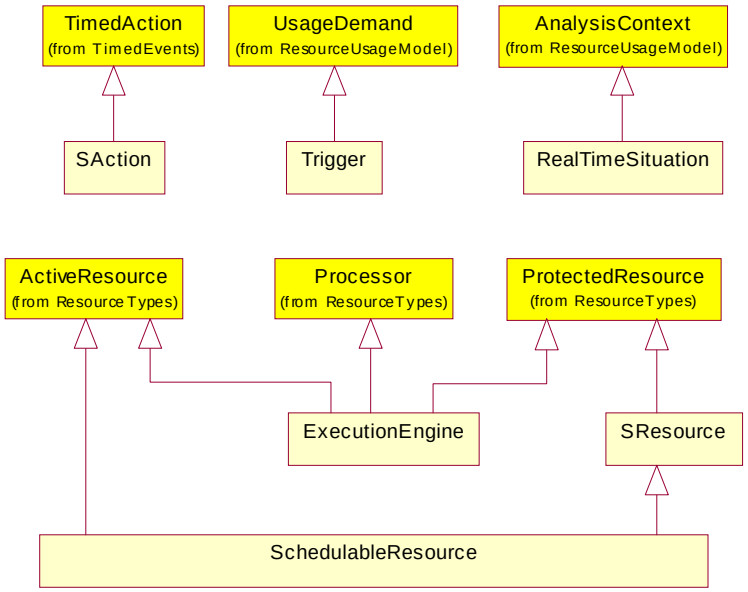


Figure 7-2 Derivation from the GRM

In addition, we include here a model of the schedulability mechanisms that are encountered in real-time operating systems (Figure 7-3). This too is based on the GRM. A scheduler is responsible for generating a schedule which is used to schedule one or more scheduling jobs. A schedule may schedule jobs for one or more execution engines. In that case, the global scheduler may consist of a set of local schedulers, one per execution engine. The scheduler is a kind of resource broker that is responsible for allocating execution engine resources. Each scheduler produces a schedule based on a scheduling policy..

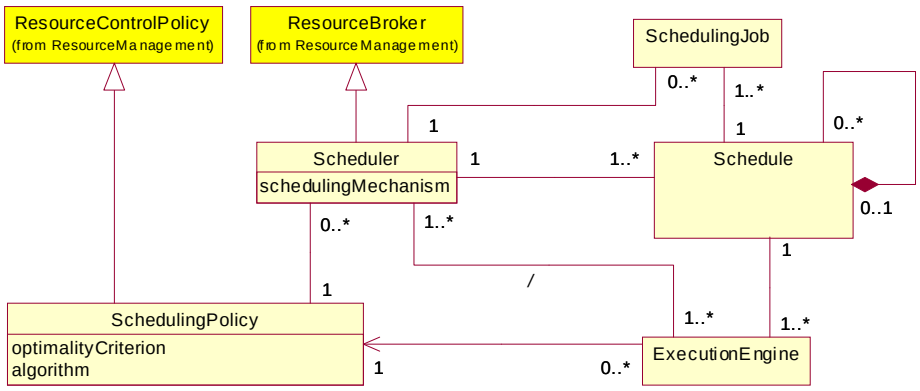


Figure 7-3 Infrastructure concepts and mechanisms pertaining to schedulability

7.1.3.1 Overview

The conceptual model of the domain has three primary types of entity:

- *Scheduling Job*, which represent the load on the system.
- *Shareable Resource*, which the scheduling jobs need to use during execution.
- *Execution Engine*, which provides the computing power.

Each scheduling job consists of a response, which defines the amount of work in the load, and a trigger, which defines how often that work needs to be done.

There are two types of shareable resources:

- A general resource type that the scheduling jobs need in order to fulfil their function. These are often sources/sinks of events and information, such as queues, data-stores etc.
- A schedulable resource that the scheduling jobs need in order to execute on the execution engine. In a typical real-time operating system, these resources are tasks or processes. Often, each such resource is responsible for executing a single scheduling job, but in some implementations, they may be shared between a number of scheduling jobs, in which case they distribute their execution time between them. The resources, particularly the schedulable resources, are realized by (deployed on) a set of execution engines, which typically are physical CPUs, although they often represent “virtual” CPUs’ such as heavyweight processes, and virtual machines.

7.1.3.2 SAction[1]

A behavior which may be characterized by its own required QoS characteristics. Note that in the GRM, actions are kinds of scenarios, which means that they are nested constructs (i.e., actions may contain other actions). The execution time of an action corresponds to the sum of the execution times for the set of its subordinate actions. The implication is that the whole subordinate set is intended to execute within the duration of the containing action.

Attributes

Priority	the priority of the action from a scheduling perspective. It may be set as a result of static analysis or by dynamic scheduling software. Some CPU scheduling policies, such as Harbour, Klein, Lehoczky (HKL) allow varying priorities of actions within a response, but most do not.
Blocking Time	the length of time that the action is blocked waiting for resources.
Ready Time	the effective Release Time expressed as the length of time since the beginning of a period; in effect a delay between the time an entity is eligible for execution and the actual beginning of execution.
Delay Time	the length of time that an action that is eligible for execution waits while acquiring and releasing resources.
Release Time	the instant of time at which a scheduling job becomes eligible for execution.
Preempted Time	the length of time that the action is preempted, when runnable, to make way for a higher priority action.
Worst-case Completion Time	the overall time taken to execute the action, including all overheads.
Laxity	specifies the type of deadline, hard or soft.
Absolute Deadline	specifies the final instant by which the action *must be* complete, defined as the Relative Deadline + Release Time. This may be either a hard or a soft deadline.
Relative Deadline	for soft deadlines, specifies the desired time by which the action should be complete.
start	(inherited from Timed Action – see Section 5.1.5.9, “TimedAction,” on page 5-13) the start time of the action.
end	(inherited from Timed Action) the completion time of the action.
duration	(inherited from Timed Action) the total duration of the action (not used if start and end times are defined).
isAtomic	(inherited from Concurrent Action – see Section 6.1.2.1, “ActionExecution (extended),” on page 6-3) identifies whether the action can be pre-empted or not.

1. We have added the prefix “S” to the term, to distinguish this specialization of the concept from the general one that is defined in the GRM.

Associations

trigger	the set of triggers that can only execute once this action has completed.
usedResources	a specialization of the GRM association that identifies the set of resources that this action uses during execution.
host	the schedulable resource that the action executes on; this is only defined if all the internal SActions that constitute the SAction execute on the same schedulable resource. This association is, in effect, a *deployment relationship*, as defined in "The Deploys Mapping" on page 4-30, between the action execution and the schedulable resource (i.e., the action is deployed on the schedulable resource). For modeling convenience, it is represented here by a stereotype of an association.

7.1.3.3 Execution Engine

An execution engine is an active, protected, executing-type resource that is allocated to the execution of schedulable resources, and hence any actions that use those schedulable resources to execute. In general, they are processor, network or device.

Attributes

Processing Rate	a relative speed factor for the execution engine expressed as a percentage. The execution times specified for the entities in the schedulability model assume a normative value of 1.0 (100%).
Context Switch Time	the length of time (overhead) that it takes to switch from one scheduling job to another.
Priority Range	the set of valid priorities for this execution engine (often dependent on the operating system). These are used to define the scheduling priorities of actions.
isPreemptible	indicates whether or not the execution engine is preemptible once it begins execution of an action.
Utilization	this is a result of model analysis and indicates the computed utilization of the processing resource expressed as a percentage.
scheduler	the set of scheduler instances that schedule jobs on this execution engine.
isSchedulable	this is the result of model analysis and indicates whether or not the processing resource is schedulable under the conditions specified in the model.

Associations

schedulableResource	the set of schedulable resources that this execution engine is charged with executing; this is, in effect, a deployment relationship (as defined in "The Deploys Mapping" on page 4-30) between the execution engine and the schedulable resources that it deploys. For convenience, it is modeled here as a stereotyped association instead of as an explicit deployment association.

ownedResources	the resources that this execution engine owns exclusively, i.e., doesn't share with other execution engines.
schedulingJob	the set of scheduling jobs that this execution engine needs to accommodate in its schedule.
schedulingPolicy	the set of rules for assigning execution engine time to a set of scheduling jobs: Rate Monotonic, Deadline Monotonic, Earliest Deadline First, etc.
accessControlPolicy	(inherited from SResource – see Section 7.1.3.12, "SResource," on page 7-12) the set of rules that govern when and under what conditions each request for the execution engine is granted, and how actions requiring resources are scheduled. This overrides the setting for individual resources owned by the execution engine.
real-TimeSituation	the set of analysis contexts in which this execution engine participates.

7.1.3.4 Real-time Situation

Provides a context for the model analysis. The various resources may be shared between different real-time situations, but the scheduling jobs are unique to each.

Associations

sResource	the set of resource instances that are used by the scheduling jobs in the situation.
executionEngine	the set of execution engines that provide the processing power in the situation.
schedulingJob	the set of scheduling jobs that comprise the processing load of this situation.

7.1.3.5 Response

A response models a sequence of action steps that is separately schedulable on an execution engine. Note that it may or may not correspond to a physical thread on an execution engine. Its most important characteristic is that it defines the root of a step sequence, which allows cumulative totals for end-to-end execution time. It inherits most of its attributes from SAction (see Section 7.1.3.2, "SAction," on page 7-7 – hence the specialization in the conceptual model). It also adds the following specific attributes.

Attributes

Utilization	the percentage of the period of the trigger during which the response is using the schedulable resource.
Slack Time	the difference between the amount of work remaining and the amount of time left in the period.
Spare Capacity	the amount of execution time that can be added to a scheduling job without affecting the schedulability of lower-priority scheduling jobs in the system.
Overlaps	in case of soft deadlines, this indicates how many instances may

overlap their execution because of missed deadlines.

Associations

cause	the trigger that causes this response to execute.
schedulingJob	the scheduling job for which this is the unit of work.

7.1.3.6 Schedule [abstract]

A specification of a particular ordering of execution of a set of scheduling job instances over a set of execution engines. This concept is not explicitly modelled, but is provided here for completeness.

Associations

schedule	a composite schedule may be composed of a set of finer-grained schedules, e.g., one for every execution engine.
schedulingJob	the set of jobs scheduled by this schedule instance.
executionEngine	the set of execution engines assumed by the schedule.

7.1.3.7 Scheduler

A kind of resource broker for execution engines that is responsible for deriving a schedule that allocates a set of scheduling jobs to its set of execution engines.

Associations

schedule	a composite schedule may be composed of a set of finer-grained schedules, e.g., one for every execution engine.

7.1.3.8 Scheduling Job [abstract]

A unit of work with a defined execution pattern, which contends for use of the execution resource that executes the schedule. In other words a representation of system load. This is a conceptual entity only, which is represented by its concrete components: Trigger and Response.

Associations

trigger	the event occurrences that cause the scheduling job to execute.
response	the work to be done when the trigger occurs.
executionEngine	the execution engines which realize this scheduling job.
real-timeSituation	the analysis context in which this job is defined.
scheduler	the scheduler instance responsible for scheduling this job.

7.1.3.9 Scheduling Policy

A kind of resource allocation policy used to schedule scheduling jobs to execution engines. This is an offered QoS concept that is manifested as an attribute rather than as a separate model element.

Attributes

optimalityCriterion — a criterion used to determine a schedule (e.g., meet all hard deadlines, minimize the number of missed deadlines, minimize the mean tardiness, maximize flow, etc.).

algorithm — the scheduling algorithm. The following scheduling algorithms are predefined:

RateMonotonic
DeadlineMonotonic
HKL = Harbour, Klein, Lehoczky
FixedPriority
MinimumLaxityFirst
MaximizeAccruedUtility
MinimumSlackTime

Associations

scheduler — the set of scheduler instances that use this policy.

executionEngine — the set of execution engines that are characterized by this scheduling policy.

7.1.3.10 Schedulable Resource

A kind of active protected resource that is used to execute an action. In a real-time operating systems this is the mechanism that represents a unit of concurrent execution, such as a task, a process, or a thread. It may be shared by multiple concurrent actions and, therefore, must be protected by a locking mechanism. It inherits most of its attributes from Protected Resource (see Section 7.1.3.12, “SResource,” on page 7-12), and requires a special access control policy, which assigns its priority based on the highest priority scheduling job. However, as a resource its capacity is always equal to 1 and is never preemptible.

Associations

sAction — the actions that execute upon this schedulable resource; in other words, these are the action executions that are deployed on this schedulable resource – hence, this is, in effect, a deployment relationship as defined in “The Deploys Mapping” on page 4-30. For convenience, it is represented here as a stereotyped association.

host — the execution engine that owns and possibly executes the code for the services of this schedulable resource; in other words, this is the set of execution engines on which this schedulable resource is deployed. In effect, this association is a deployment relationship as defined in “The Deploys Mapping” on page 4-30, but is rendered here as a stereotyped association for modeling convenience.

7.1.3.11 Trigger

An event occurrence that causes the execution of a separately schedulable sequence of actions (a response) and represents a usage demand (see Section 4.1.9.38, “UsageDemand,” on page 4-27). The combination of a trigger and a response is called a scheduling job.

Attributes

isSchedulable	a model analysis result that indicates whether the trigger can be scheduled.
endToendTime	the worst case completion time for the complete chain of dependent responses measured from the arrival of the trigger
occurrencePattern	the pattern of interarrival times between consecutive occurrences of the trigger.

Associations

effect	the response to this trigger.
schedulingJob	the scheduling job, which is triggered by this trigger.
precedents	the set of actions that must be executed before this event occurrence.

7.1.3.12 SResource[2]

A kind of protected resource[3] (e.g., a semaphore) that is accessed during the execution of a scheduling job. It may be shared by multiple concurrent actions and must be protected by a locking mechanism. It may represent a physical device or a logical exclusive-access object (e.g., a queue). All of its characteristics represent offered QoS values.

Attributes

accessControlPolicy	(inherited from ProtectedResource – see Section 4.1.9.20, “ProtectedResource,” on page 4-21) the access control policy for handling requests from scheduling jobs. The following policies are pre-defined: *FIFO* = first-in-first-out *PriorityInheritance* = priority inheritance *NoPreemption* = no pre-emption *HighestLockers* = highest lockers *PriorityCeiling* = priority ceiling (note that in this case, we also need to provide the actual value of the priority ceiling, which generally is the priority of the highest priority scenario step that uses the resource)
Capacity	the number of permissible concurrent users, for example using a

2. The “S”prefix has been added to avoid confusion with the GRM concept of the same name.
3. Non-exclusive resources are not considered here since they do not result in resource contention that can affect the results of model analysis.

	counting semaphore.
Acquisition Time	the time delay suffered by an action between being granting access to a resource and the availability of the resource.
Deacquisition Time	the time delay suffered by an action between initiating release of a resource and the action becoming eligible for execution again.
isConsumable	indicates that the resource is consumed by use.
Priority Ceiling	a value calculated in accordance with an access control policy. Generally, it is the priority of the highest priority scheduling job that uses the resource.
isPreemptible	indicates if the resource can be preempted while it is being used.
Associations	
sAction	the set of actions that use this resource during their execution.
executionEngine	the execution engine that has exclusive use of this resource. Non-exclusive resources may be used from many execution engines and generally need a different form of access control.
realTimeSituation	the set of real-time situations in which this resource appears.

7.2 UML Viewpoint

We now examine how the domain concepts can be represented (mapped) in the UML modeling environment. In general, this is done by applying the stereotypes defined in Section 7.2.2, “UML Extensions,” on page 7-15. To provide the flexibility required by the RFP for this specification, the same stereotypes may be applied to a number of different kinds of modeling elements.

7.2.1 Mapping Schedulability Domain Concepts into UML Equivalents

7.2.1.1 The Collaboration-Based Approach

There are two variants in this approach: one based on collaborations and the other on collaboration instances. Either form may be used since there are no semantic differences as far as the interpretation of the results is concerned[4]. The choice depends on circumstances (i.e., whichever model is more readily available) or individual preference of the modeler.

4. It can be argued that a collaboration is a generic specification for a *set* of different executions, whereas a collaboration instance set might specifies a single execution. However, we do not make such a distinction in this profile.

Real-time Situation

A real-time situation is modeled as a stereotype «SAsituation» of a UML collaboration (or collaboration instance set). This means that all interactions specified in that collaboration represent scheduling jobs (in fact the response part of a scheduling job), although this is not the only way of representing responses.

Scheduling Job

A scheduling job maps to an interaction (or interaction instance set). However, since interactions are not explicit graphical elements in UML diagrams (they are represented by collections of graphical elements), we will not define an explicit stereotype. Instead, a scheduling job will be represented by a trigger, which is used to stereotype the first Message (Stimulus) in the Interaction, and a response, which is used to stereotype the associated Action (Action Execution) of the trigger (although for diagrammatic convenience this might also stereotype the initial Message also). The relationship between trigger and response is either via collocation of the stereotypes, or by the UML metaassociation between Message (Stimulus) and Action (Action Execution).

The schedulable resource that the response uses to execute can be derived from the “owner” of the action with the «SAresponse» stereotype.

Resources and Execution Engines

These are indicated by stereotyping classifier roles (instances) with the appropriate stereotypes («SAengine», «SAschedulable», «SAresource»). The relationship between execution engines and the “shareable” resources is established using the «SAowns» and «GRMdeploys» stereotypes of realization (i.e., abstraction).

Actions

Actions are either stereotyped Actions (Action Executions) or, for diagrammatic convenience where appropriate, stereotyped Messages (Stimuli). The Uses relationship can often be derived from the destination of the Message, but in some cases, where the level of granularity is coarse, and messaging actions are not being used, then an explicit usage dependency (stereotyped «SAuses») may be inserted between the appropriate Action (Action Execution) and the resource.

The precedence relationship to any scheduling job triggers could potentially be modeled using an Interaction to model many scheduling jobs, in which case any action preceding a Message(Stimulus) tagged as a «SAtrigger» would indicate the presence of a “precedes” association. However, we have chosen to represent each scheduling job as separate Interactions and so an explicit usage dependency («SAprecedes») is necessary.

7.2.2 UML Extensions

7.2.2.1 Conventions

To minimize the possibility of confusion with other names, we will prefix all names of extension elements that are based on the schedulability analysis model with the string "SA".

Several tag types are defined as enumeration lists. These lists also represent declarations of standard string names that can be included as value parts for the corresponding tagged values.

7.2.2.2 Profiles

For convenience, all extensions related to this schedulability model that are defined in this section of the document are packaged in a single profile package (i.e., a «profile» stereotype of Package) called "SAprofile". This profile should be imported by other model analysis profiles that are based on it. This profile imports the GRM profile described in the *General Resource Modeling* chapter.

7.2.2.3 Stereotypes

The following stereotypes and tagged values are defined for the schedulability sub-profile. The semantics of these stereotypes are defined in Section 7.1.3, "Domain Concepts Details," on page 7-4.

«SAaction»

Represents the Action domain concept. (see Section 7.1.3.2, "SAction," on page 7-7).

Stereotype	Base Class	Parent	Tags
«SAaction»	Action	«RTaction» «CRaction»	SApriority SAblocking SAready SAdelay SArelease SApreempted SAworstCase SAlaxity SAabsDeadline SArelDeadline SAusedResource SAhost
	ActionExecution		
	Message		
	Stimulus		
	Method		
	ActionState		
	SubactivityState		
	Transition		

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
SApriority	Integer	[0..1]	SAction::Priority
SAblocking	RTtimeValue	[0..1]	SAction::Blocking Time
SAdelay	RTtimeValue	[0..1]	SAction::Delay Time
SApreempted	RTtimeValue	[0..1]	SAction::Preempted Time
SAready	RTtimeValue	[0..1]	SAction::Ready Time
SArelease	RTtimeValue	[0..1]	SAction::Release Time
SAworstCase	RTtimeValue	[0..1]	SAction::Worst Case Completion Time
SAabsDeadline	RTtimeValue	[0..1]	SAction::Absolute Deadline
SAlaxity	Enumeration: {'Hard','Soft'}	[0..1]	SAction::Laxity
SArelDeadline	RTtimeValue	[0..1]	SAction::Relative Deadline
SAusedResource	Reference to a model element that is stereotyped as «SAresource»	[0..*]	SAction::usedResources[1]
SAhost	Reference to a model element that is stereotypes as «SAschedulable»	[0..1]	SAaction::host[2]

1. Alternatively, this can be modeled using the «SAuses» stereotype.

2. Alternatively, this can be modeled using the «SAusedHost» stereotype.

The type RTtimeValue is defined in "RTtimeValue" on page 5-31.

«SAengine»

This stereotype represents the execution engine concept (see Section 7.1.3.3, “Execution Engine,” on page 7-8).

Stereotype	Base Class	Tags
«SAengine»	[1]Classifier	SAschedulingPolicy SAaccessPolicy, SAaccessPolParam SArate SAcontextSwitch SApriorityRange SApreemptible SAutilization SAschedulable SAresources
	ClassifierRole	
	Node	
	Instance	
	Object	

1. only the deployable types, Component, Artifact, Node and Class

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
SAaccessPolicy	Enumeration: {‘FIFO’, ‘PriorityInheritance’, ‘NoPreemption’, ‘HighestLockers’, ‘PriorityCeiling’}	[0..1]	Execution Engine::Access Control Policy
SAaccessPolParam	Real	[0..*]	Execution Engine::Access Control Policy (for providing numerical values associated with the access control policy, such as the priority ceiling value)
SAcontextSwitch	RTtimeValue	[0..1]	Execution Engine::Context Switch Time
SAschedulable	Boolean	[0..1]	Execution Engine::isSchedulable
SApreemptible	Boolean	[0..1]	Execution Engine::isPreemptible
SApriorityRange	Integer Range	[0..1]	Execution Engine::Priority Range
SArate	Real	[0..1]	Execution Engine::Processing Rate

SAschedulingPolicy	Enumeration: {'FIFO', 'RateMonotonic', 'DeadlineMonotonic', 'HKL', 'FixedPriority', 'MinimumLaxityFirst', 'MaximizeAccruedUtility', 'MinimumSlackTime'}	[0..1]	Execution Engine::Scheduling Policy
SAutilization	Percentage (Real)	[0..1]	Execution Engine::Utilization
SAresources	Reference to an element stereotyped as «SAresource»	[0..*]	Execution Engine::ownedResources[1]

1. Note that this can also be modeled using the «SAowns» stereotype

«SAowns»

A kind of "realizes" dependency (see Section 4.2.1, "Modeling Realization Relationships," on page 4-27) that is used to identify which resources are owned by which execution engines. This corresponds to the "ownedResources" role of the association between ExecutionEngine and SResource in the conceptual model in Figure 7-1 (see also Sectio n7.1.3.3, "Execution Engine," on page 7-8). It is modeled as a subclass of the GRM realizes stereotype.

Stereotype	Base Class	Parent
«SAowns»	Abstraction	«GRMrealize»

Note that this association can also be modeled using the SAresources tag (whose value is a reference to a resource).

«SAprecedes»

A kind of usage dependency that is used to identify any precedence relationship between Actions and triggers. This corresponds to the "precedents" role of the association between SAction and Triger in the conceptual model in Figure 7-1.

Stereotype	Base Class
«SAprecedes»	Usage

Note that this can also be represented using the SAprecedents tagged value associated with the «SAtrigger» stereotype.

«SAresource»

Represents the resource concept as defined in Section 7.1.3.12, “SResource,” on page 7-12.

Stereotype	Base Class	Tags
«SAresource»	Classifier	SAaccessControl, SAaccessCtrlParam SAconsumable, SAcapacity, SAacquistion, SAdeacquisition, SAptyCeiling SApreemptible
	ClassifierRole	
	Instance	
	Node	
	Object	

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
SAacquisition	RTtimeValue	[0..1]	SResource::Acquisition Time
SAcapacity	Integer	[0..1]	SResource::Capacity
SAdeacquisition	RTtimeValue	[0..1]	SResource::Deacquisition Time
SAconsumable	Boolean	[0..1]	SResource::Consumable
SAaccessControl	Enumeration: {‘FIFO’, ‘PriorityInheritance’, ‘NoPreemption’, ‘HighestLockers’, ‘PriorityCeiling’}	[0..1]	SResource::Access Control Policy
SAaccessCtrlParam	Real	[0..*]	SResource::Access Control Policy (for providing numerical values associated with the access control policy, such as the priority ceiling value)
SAptyCeiling	Integer	[0..1]	SResource::Priority Ceiling
SApreemptible	Boolean	[0..1]	SResource::isPreemptible

«SAresponse»

This stereotype represents the response concept (see Section 7.1.3.5, “Response,” on page 7-9).

Stereotype	Base Class	Parent	Tags
«SAresponse»	ActionExecution	«SAaction»	SAutilization SAspare SAslack SAoverlaps
	Action		
	Method		
	Transition		
	ActionState		
	SubactivityState		

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
SAutilization	Real (percentage)	[0..1]	Response::Utilization
SAspare	RTtimeValue	[0..1]	Response::Spare Capacity
SAslack	RTtimeValue	[0..1]	Response::Slack Time
SAoverlaps	Integer	[0..1]	Response::Overlaps

«SAschedulable»

Represents the schedulable resource concept (see Section 7.1.3.10, “Schedulable Resource,” on page 7-11). It inherits the attributes of SResource.

Stereotype	Base Class	Parent
«SAschedulable»	Classifier	«SAresource»
	ClassifierRole	
	Instance	
	Object	
	Node	

«SAscheduler»

The representation of a scheduler (see Section 7.1.3.7, “Scheduler,” on page 7-10).

Stereotype	Base Class	Tags
«SAscheduler»	Classifier	SAschedulingPolicy SAexecutionEngine
	ClassifierRole	
	Instance	
	Object	

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
SAexecutionEngine	Reference to a model element stereotyped as «SAengine»	[0..1]	Scheduler::executionEngine
SAschedulingPolicy	see definition of the similarly named tag in: “«SAengine»” on page 7-17	[0..1]	Scheduler::schedulingMechanism

«SAsituation»

Represents the concept of a real-time situation that can be used for model analysis purposes (see Section 7.1.3.4, “Real-time Situation,” on page 7-9).

Stereotype	Base Class
«SAsituation»	Collaboration
	CollaborationInstance
	ActivityGraph

«SAtrigger»

Represents the trigger concept (see Section 7.1.3.11, “Trigger,” on page 7-12).

Stereotype	Base Class	Tags
«SAtrigger»	Message	SAschedulable SAendToEnd SAprecedents SAoccurrence
	Stimulus	

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
SAendToEnd	RTtimeString	[0..1]	Trigger::endToEndTime
SAschedulable	Boolean	[0..1]	Trigger::isSchedulable
SAprecedents	Reference to a model element stereotyped as «SAaction»	[0..*]	Trigger::precedents[1]
SAoccurrence	RTarrivalPattern	[0..1]	Trigger::occurrence Pattern

1. Note that this can also be modeled using the «SAprecedes» stereotype.

«SAusedHost»

A kind of usage dependency that is used to identify explicitly which schedulable resource an action needs to execute. This corresponds to the “host” role of the association between SAction and SchedulableResource in Figure 7-1 in the conceptual model (see also Section 7.1.3.2, “SAction,” on page 7-7).

Stereotype	Base Class
«SAusedHost»	Usage

As an alternative, the same relationship can be modeled using the SAhost tagged value of the «SAaction» stereotype.

«SAuses»

A kind of usage dependency that is used to identify explicitly which shareable resources an action needs while executing. This corresponds to the “usedResources” role of the association between SAction and SResource in Figure 7-1 in the conceptual model (see also Section 7.1.3.2, “SAction,” on page 7-7).

Stereotype	Base Class
«SAuses»	Usage

As an alternative, the same relationship can be modeled using the SAusedResource tagged value of the «SAaction» stereotype.

7.2.3 Modeling Guidelines and Examples

In these examples we have chosen to attach the schedulability stereotypes to UML model elements that form a collaboration – this is only one of a number of possibilities.

7.2.3.1 Presentation Conventions

In this example, we will use the following presentation conventions for various model elements required for the model analysis:

- Stereotypes are shown against the graphical symbol for the underlying base-type if the symbol is unambiguous (i.e. only represents one model element); otherwise, the appropriate stereotypes for the symbol are shown via attached notes.
- Two choices are available for showing the QoS characteristics of stereotypes: an attached note with the stereotype, element name and QoS values; or a set of QoS values in {} following other information in the symbol. A note can contain the information for several stereotypes if attached to a symbol that represents more than one model element.

7.2.3.2 Example System Specification

Figure 7-4 shows the structural specification of a telemetry system example that we wish to analyze. It runs on a single processor and consists of three different kinds of classes. Note that this class diagram is a descriptor diagram, not an instance-based diagram. To enable schedulability analysis, the analyst must define one or more real-time situations, consisting of instances, and annotate the elements of the model with appropriate QoS values, such as maximum system load.

The choice of what to represent in a real-time situation is up to the analyst and depends on the type of model analysis and level of detail that is required. In the following we demonstrate several different choices for the same example.

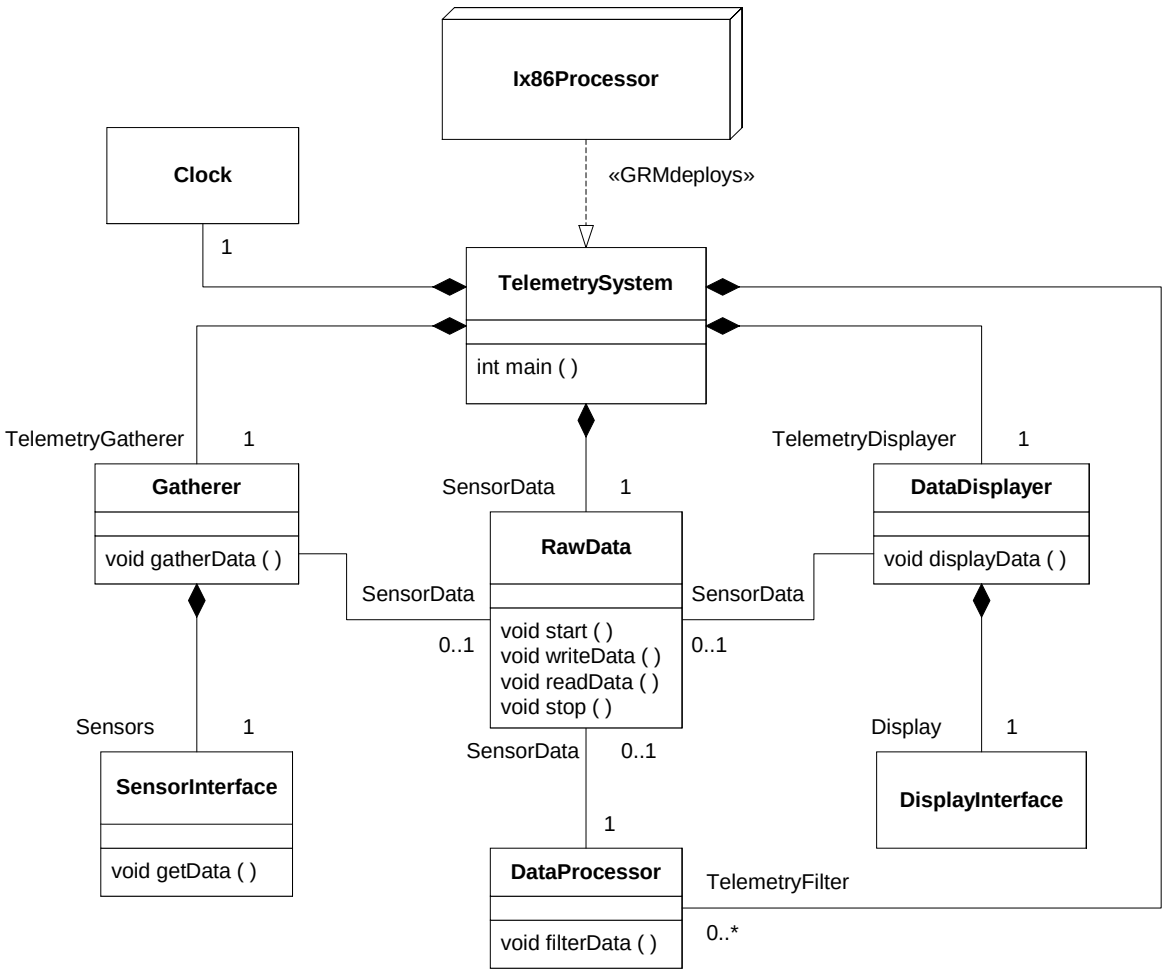


Figure 7-4 Example telemetry system specification

Note that in the system description, the realization relationship is from the node to the aggregate class that represents the system (in this case Telemetry System). At the instance level, there will be many more such realization relationships (see Figure 7-7).

7.2.3.3 *Using Sequence Diagrams to Model Real-Time Situations*

A sequence diagram is useful for the detailed description of triggers and responses. Figure 7-5 shows the data gathering scenario.

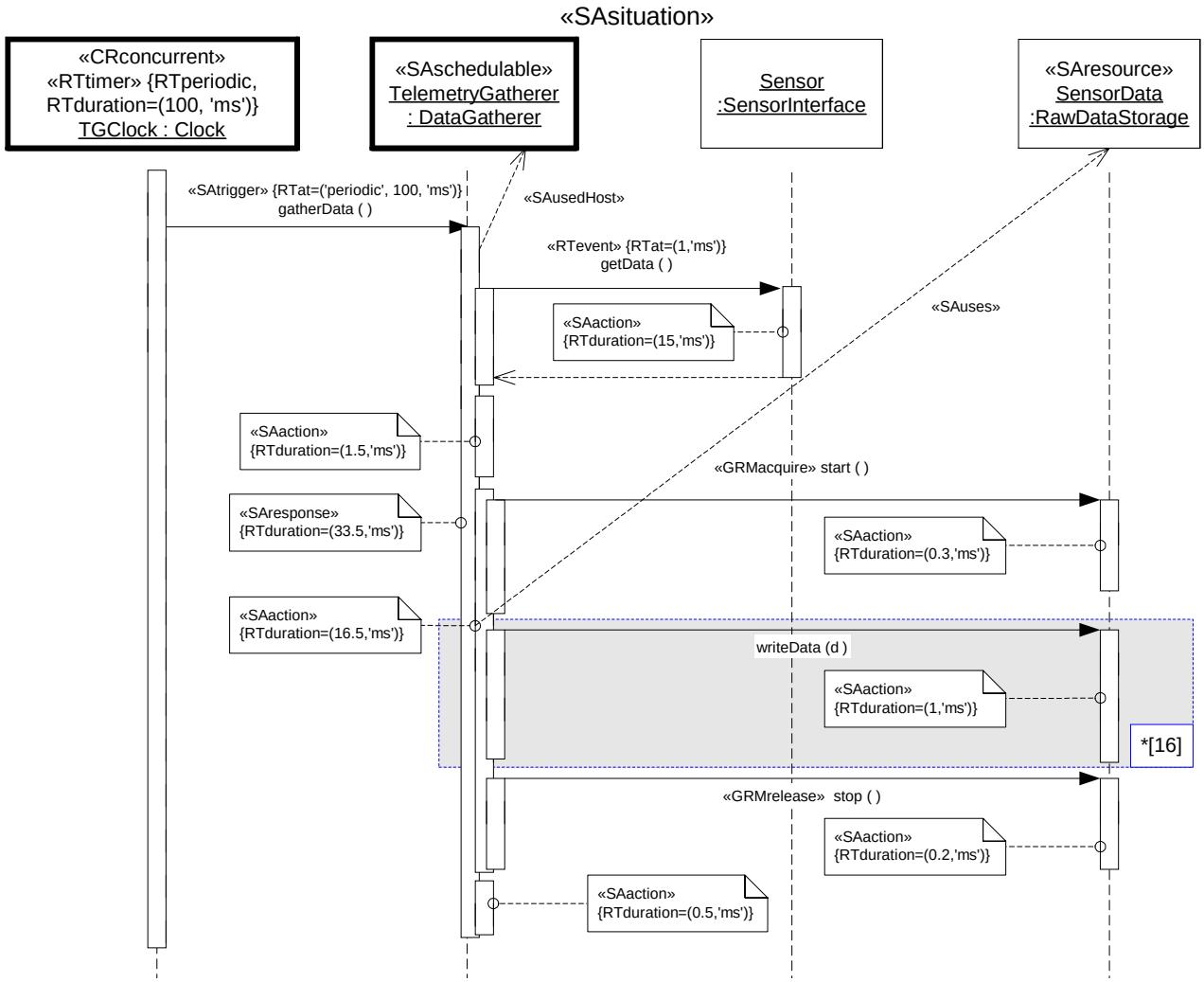


Figure 7-5 Showing a trigger and its response in a Sequence Diagram

Note that:

- This is an instance diagram, and so all of the symbols represent instance concepts.
- GRM annotations such as «GRMacquire» and «GRMrelease» are used to determine the aggregate period of use of “used resources”. In this case, the aggregate use of Sensor Data has a duration of 16.5.
- The response, as one might expect, is simply the outermost focus-of-control symbol on the “fence-post” of the schedulable resource (in this case Telemetry Gatherer).

- Dependencies such as «SAuses» may be easy to derive directly from the model (they are included here for illustrative purposes). In fact they typically map very simply onto UML, so they can easily be derived either by a tool, or by the analyst.

7.2.3.4 *Expressing Schedulability Data Using a Collaboration Diagram*

Collaboration diagrams are useful for expressing a set of triggers and responses. The diagram in Figure 7-6 shows three schedulable resources, and one (shared) used resource, as well as two that are not shared. There are three triggers and three responses marked..

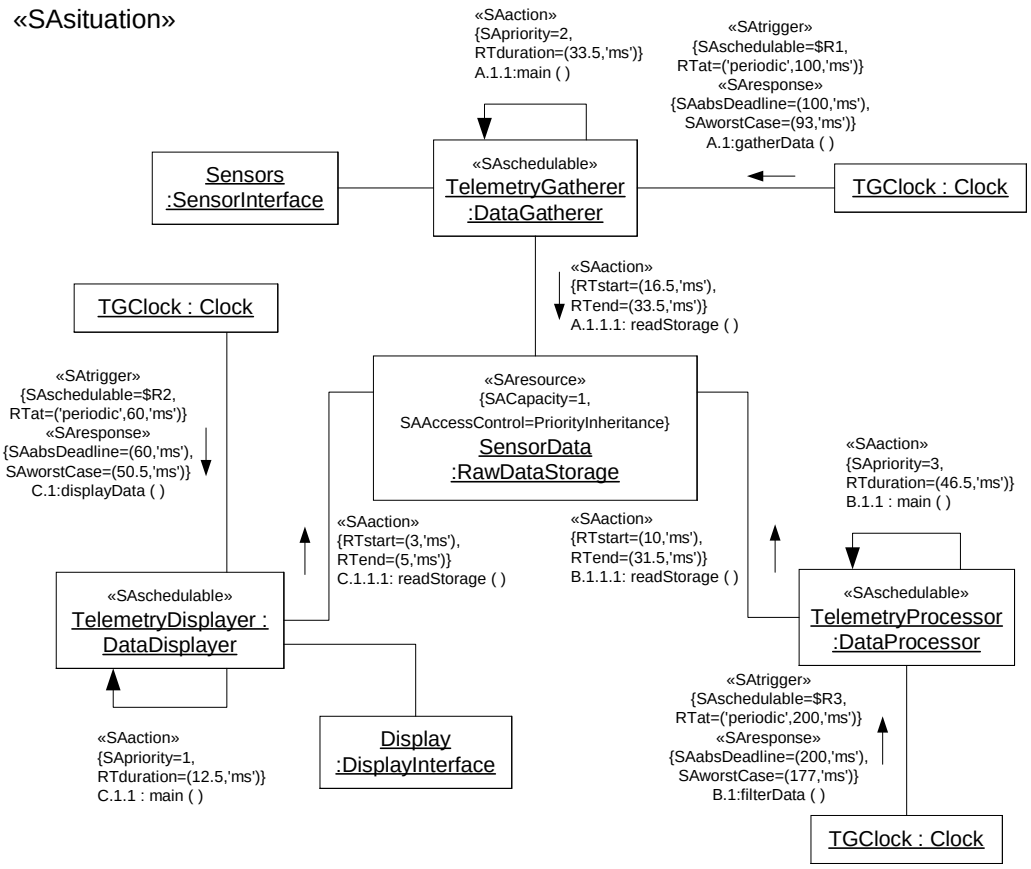


Figure 7-6 A Collaboration Diagram showing all of the schedulable triggers and responses

Note the use of TVL variables ($R1, $R2, $R3) for the items that need to be computed and returned by the model analysis tools. In this case, they are all associated with the SAschedulable tag that specify whether the corresponding response is schedulable.

7.2.3.5 Showing Resource Realization

Figure 7-7 shows another view of the model, which makes clear the relationship between the objects in the system and the single execution engine. The deployment of *instances* on a given execution engine is required in order to perform schedulability analysis.

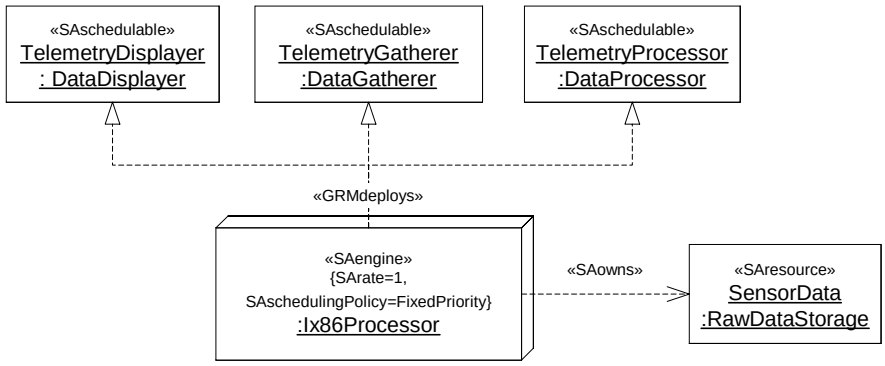


Figure 7-7 Use of the execution engine

Note that this is an object (instance) diagram, which shows explicitly how the execution capability of the processor is used. This contrasts with Figure 7-4, where the system description merely identified a realization between a node of this type and the “system” class of the telemetry system.

7.2.4 Required UML Metamodel Changes

This package assumes that the UML metamodel is modified to support the action execution concept. This is defined in Section 4.1.9.3, “ActionExecution,” on page 4-17. No other metamodel changes are assumed or required.

7.2.5 Proposed Notational Extensions

No notational extensions are proposed for this sub-profile.

Performance Modeling 8

In this section, we describe a component of the profile that is intended for general performance analysis of UML models. The profile provides facilities for:

- capturing performance *requirements* within the design context.
- associating performance-related *QoS characteristics* with selected elements of a UML model.
- specifying *execution parameters* which can be used by modeling tools to compute predicted performance characteristics.
- presenting performance results computed by modeling tools or found in testing.

Typical tools for this kind of model analysis provide two important functions. The first is to estimate the performance of a system instance, using some kind of model. The second function is assistance with determining how the system can be improved, by identifying bottlenecks or critical resources. A system designer will typically want to analyze the system under several scenarios using different parameter values for each scenario while maintaining the same overall system structure.

This chapter describes a minimal set of concepts to support the central ideas of performance analysis. The intent is to provide a base for further refinements that might perform more extensive analyses.

The structure of this section follows the convention adopted throughout this document: First, a domain viewpoint is described which identifies the basic abstractions used in performance analysis. The semantics of these abstractions and their relationships are explained with the aid of a UML model. The second part of the chapter describes how these abstractions are expressed in terms of lightweight extensions to the UML metamodel. The last section contains guidelines and examples showing common ways of applying this part of the real-time profile.

8.1 Domain Viewpoint

We start with a high-level overview of performance analyses concepts and techniques. Next, we introduce the domain model with its essential concepts and relationships.

8.1.1 Background

Scenarios define response paths whose end points are externally visible, so they represent responses with response times and throughputs. QoS requirements are placed on scenarios.

In performance-related models, each scenario is executed by a *job class or user class* with an applied load intensity, and these classes are either open or closed. To avoid confusion with software classes, such a class is called here a *workload*. An *open workload* has a stream of requests which arrive at a given rate in some predetermined pattern (such as Poisson arrivals), while a *closed workload* has a fixed number of active or potential users or jobs which cycle between executing the scenario, and spending an external delay period (some times called a Think Time) outside the system, between the end of one response and the next request.

Scenario steps or activities are the elements of scenarios, and are joined in a sequence, with predecessor-successor relationships which may include forks, joins and loops. A step may be an elementary operation at the finest granularity, or, it may be defined by a sub-scenario, to any level of nesting.

Each step also has a mean execution count, which is the mean number of times it is repeated when it is executed, and a *host execution demand* for its host device (that is, the execution time taken on its host device, in the given deployment). A scenario step may optionally have its own QoS properties.

Resource demands by a step include its host execution demand as already mentioned, and the demands of all its sub-steps. They also may include demands to resources through external resource operations (such as file I/Os) which are not defined in the UML software model, but are understood by the performance modeling tool. These demands are given as an average number of the named operations and may be interpreted appropriately by the modeling tool.

Resources are modeled as servers. Active resources are the usual servers in performance models, and have service times. Passive resources are acquired and released during scenarios, and have holding times. The *resource-operations* of a resource are the steps, or sequences of steps, which require the resource. The resource is obtained at the beginning of a resource-operation and released at the end. The resource-operations define the classes of service of the resource.

The *service time* of an active resource is defined as the *host execution demand* of the steps that are hosted by the resource. Thus different classes of service, given by different steps, may have different service times. This places the definition of service times squarely within the software specification, however a device may have a speed factor which scales all the steps that run on that resource.

Performance measures for a system include resource utilizations, waiting times, execution demands (for CPU cycles or seconds) and response time, (the actual or wall-clock time to execute a scenario step or scenario). Each measure may be defined in different *versions*, several of which may be specified in the same model, such as:

- a required value, coming from the system requirements or from a performance budget based on them (e.g., a required response time for a scenario).
- an assumed value, based on experience (e.g., for an execution demand or an external delay).
- an estimated value, calculated by a performance tool and reported back into the UML model.
- a measured value.

Further, the reported value is one of several possible statistical properties of the given measure, such as its average, maximum or xth percentile (90th percentile meaning that 90% of values are smaller than this).

Multiple statistical properties may be reported for the same measure. For example, the measured mean and the 90% and 99% values may all be given. Thus, tagged values for measures include fields to identify which version is meant, and which statistical property (for instance, ResponseTime: Required Mean 3.13 sec.).

8.1.2 Types of Performance Analysis Methods

A sub-profile for general performance analysis should support modeling tools for building different kinds of performance models. Most modeling tools deal with one or more of the following common types of models:

- Queueing models define customer classes (workloads) which execute particular aspects of the software, which are captured in different scenarios. In the simplest queueing models it is only necessary to define the class sizes or arrival rates, and the total average demands placed on each device in the system, during one execution of each scenario. In more complex queueing models the distribution of the demand may be required, there may be passive resources as well as devices, and the detailed scenario sequence may be required (for instance if it has parallel branches).
- Queueing models calculate average throughput, utilization and response times for classes overall, and layered or extended queueing models also can calculate these figures for passive resources and for parts of scenarios (scenario steps or resource-operations).
- Simulation models define multiple logical tokens which execute the software, following the detailed scenario structure and using execution time distributions for the operations of each step. There may be passive resources and they may have complex scheduling (for instance, LRU management of a cache).

  Simulation models can calculate a wide range of measures including histograms and percentiles as well as average values.

- Discrete-state models such as Petri Nets define tokens which execute the software, following the detailed scenario structure. As in queueing models there may be open or closed classes of tokens for different scenarios. Where tokens must be differentiated they are said to be colored. Petri Nets use places to define the progress of tokens and transitions to describe decisions, and the passage of time. Resources are described by additional places and tokens, and resource scheduling by transitions which execute scheduling decisions.

  Performance Petri Nets typically calculate average measures but can provide more detailed measures such as higher moments and distributions.

8.1.3 Domain Model

Performance analysis is inherently *instance-based*. That is, it applies to models that capture either actual or hypothetical execution runs of systems consisting of sets of instances. Hence, the domain concepts all refer to instances rather than descriptors (such as classes).

Figure 8-1 depicts a general performance model that identifies the basic abstractions and relationships used in performance analysis as described in section Section 8.1.1, “Background,” on page 8-2.

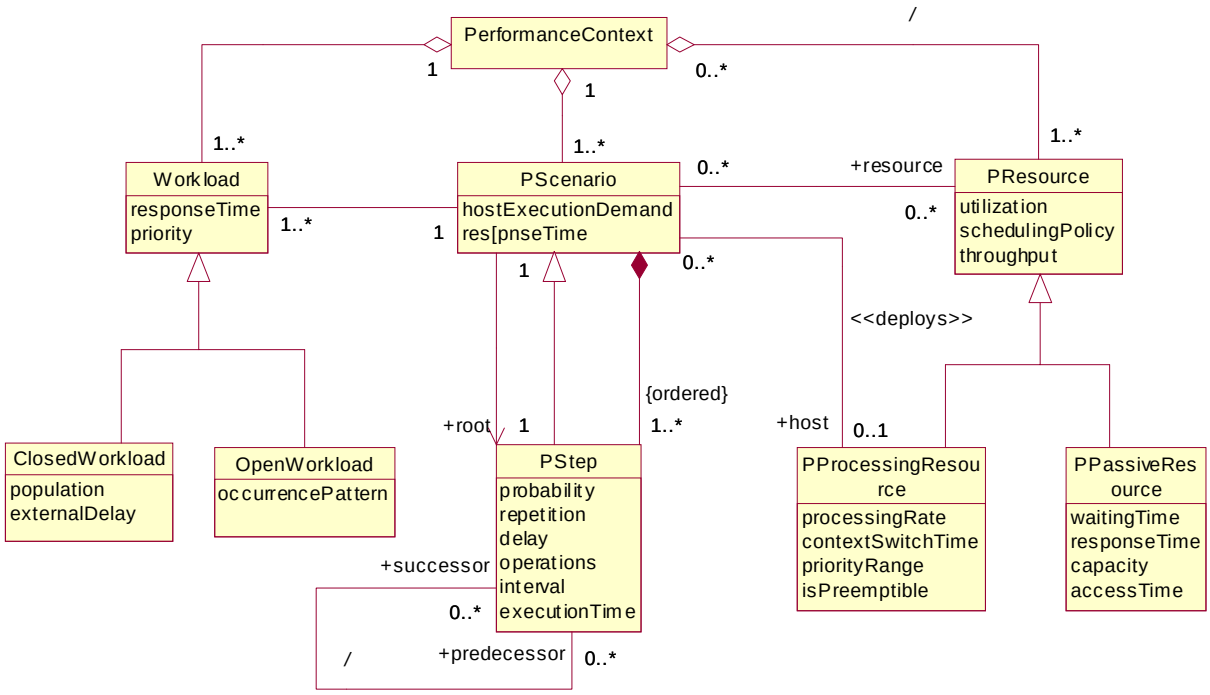


Figure 8-1 The performance analysis domain model

The concepts in this model are fully consistent with the conceptual framework defined in the generic resource model. This allows the performance sub-profile to take advantage of the mechanisms (e.g., modeling styles and stereotypes) that are provided for that framework.

The domain model is fully based on the GRM (see the *General Resource Modeling* chapter) The relationship of the performance modeling concepts to corresponding GRM concepts is depicted in the class diagram in Figure 8-2.

Since performance modeling is only concerned with resources where contention or waiting occurs, the concept of a passive resource inherits from both the passive resource and protected resource concepts of the GRM.

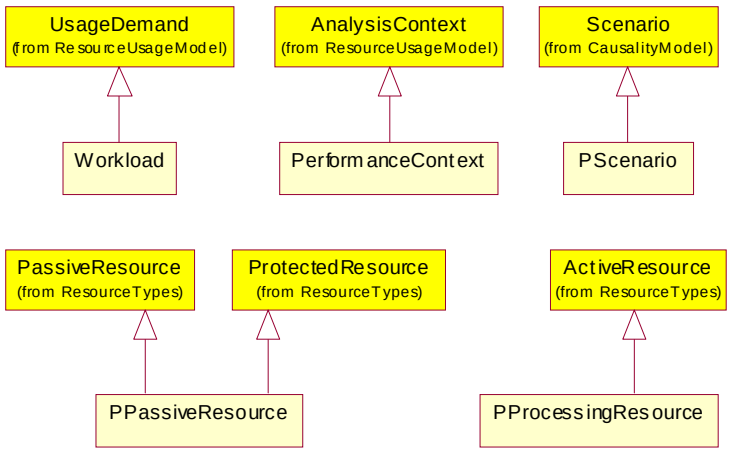


Figure 8-2 The relationship between the performance concepts and the general resource model

8.1.4 Domain Concept Details

In this section we provide a more detailed explanation of each of the concepts in the performance analysis model. Note that these are not specifications of the actual UML stereotypes, but are used as a basis for deriving such stereotypes.

For each concept, we describe all of its features and associations. We distinguish between concrete and abstract concepts. Concrete concepts are the ones used directly by the modeler, whereas abstract concepts are used to define common features of two or more related concepts. Abstract concepts are clearly identified as such below.

8.1.4.1 Performance Context

A performance context specifies one or more scenarios that are used to explore various dynamic situations involving a specific set of resources. For instance, a performance context may describe a “busy hour”, during which the maximum processor load is expected and therefore imposing the greatest likelihood of performance problems, such as missed deadlines. For a given system specification, there may be many performance contexts with overlapping resources, but the scenarios are specific to the performance context.

The QoS values considered here are load intensity and various measures of response delay. The components of a performance context may have parameterized QoS values, to enable some exploration of different QoS, but the structure of a performance context (resources and scenario steps) is fixed.

Associations

resource	the set of resources involved in this performance context; this includes both processing resources and passive resources.
scenario	the set of scenarios defined in the performance context.
workload	the set of workloads applied to the scenarios of this context; a context may not necessarily have a workload; in such cases, it derives its workload from a higher-level context in which it is embedded.

8.1.4.2 Scenario

A scenario is a sequence of one or more scenario steps. The steps are ordered and conform to a general precedence/successor relationship. Note that, in the general case, a scenario may involve multiple threads due to forking within the scenario. Therefore, a step in a scenario can have multiple successors due to forking. Similarly, a step that follows a join will have multiple predecessors.

The steps of a scenario execute on host resources, and may use passive protected resources. However, a scenario only has a host resource defined if all its sub-steps have execute on the same host (i.e. if all the steps in the scenario use the same host resource).

Attributes

hostExecutionDemand	the total execution demand of the scenario on its host resource, if defined; it is defined only if all the steps that constitute the scenario execute on the same host.
responseTime	the total time required to execute the scenario, including all resource waiting, synchronization delays and execution times; determining the value of this QoS characteristic is often the principal objective of performance analysis.

Associations

host	the processing resource on which the scenario is executed; this is only defined if all the steps that constitute the scenario execute on the same host. In effect, this association is an instance of the deploys relationship, as defined in “The Deploys Mapping” on page 4-30, but is rendered here as a stereotyped association for convenience.
step	the sequence of steps making up the scenario.
root	the first step of this scenario.
workload	the set of workloads that drive this scenario.
performanceContext	the performance context in which this scenario appears (a scenario appears in exactly one performance context).

8.1.4.3 Step

An increment in the execution of a particular scenario that takes may use resources to perform its function. In general, a step takes finite time to execute. It is related to other steps in predecessor/successor relationships.

The granularity of a step depends on the level of abstraction chosen by the modeller. If finer granularity is required, a step at one level of abstraction can be resolved into a corresponding scenario comprising finer-grained scenario steps. For this reason, a step is modelled as a kind of scenario (the simplest scenario, therefore, consists of just a single atomic step). The smallest granularity steps execute on a unique host resource or processor. A step defined by a lower-level scenario only has a defined host if all of its sub-steps have the same host.

Attributes

hostExecutionDemand (inherited from Scenario) the total execution demand of the step on its host resource, if it has a single host resource (excluding any external operations); if the step has a breakdown into a sub-scenario, with all steps on the same host, then this is the total demand of the sub-scenario.

delay the value of an inserted delay (wait or pause) within this step, for example for a user interaction.

responseTime (inherited from Scenario) the total delay to execute the step, including all resource waiting and all execution times.

probability in situations where its predecessor step has multiple successors (a choice of paths within a sequence), this is the probability that this step will be executed; in that case, the sum of probabilities of all the peer steps has to be equal to 1.

interval the time interval between successive repetitions of this step, when it is repeated within a scenario (see *repetitions* below).

repetition the number of times the step is repeated.

operations this is used to specify the set of operations of resources used in the execution of a step but which that not explicitly represented in the model; each operation attribute identifies the operation and the number of times it is repeated (the execution times of these steps are assumed to be defined externally).

Associations

predecessor one of the predecessor steps within its sequence; a step may have multiple predecessors if it occurs at a join between two or more concurrent threads of execution; if there is no predecessor, the step represents the root step of the scenario.

successor one of the successor steps within its sequence; a step may have multiple successors if it represents a fork into multiple concurrent threads of execution; a step without a successor is the final step of a scenario.

scenario the scenario in which this step is included.

host	(inherited from Scenario) the processing resource on which this step executes; if the step has a sub-scenario, then this attribute only exists if all the steps of the sub-scenario execute on the same host.
resource	the set of resources accessed by the step.

8.1.4.4 Resource [abstract]

An abstraction view of passive or active resource, which participates in one or more scenarios of the performance context.

Attributes

utilization	this is usually the result of model analysis and represents the computed utilization of the processing resource expressed as a percentage.
throughput	the rate at which the resource performs its function.
schedulingPolicy	the policy by which access to the resource is controlled.

Associations

performanceContext	the performance context in which this resource instance appears; a resource may belong to more than one performance context.

8.1.4.5 ProcessingResource

A processing resource is a device, such as a processor, interface device or storage device, which has processing steps allocated to it by the deployment of the system.

Attributes

schedulingPolicy	(inherited from Resource) the set of rules for assigning the resource to a set of steps; a number of policies are pre-defined: *FIFO* = first-in-first-out *HeadOfLine* = head-of-the-line or non-preemptible priorities *PreemptResume* = pre-empt resume *ProcSharing* = processor sharing (representing round-robin) *PrioProcSharing* = priority processor sharing *LIFO* = last-in-first-out
processingRate	a relative speed factor for the processor expressed as a percentage of some normative processor.
contextSwitchTime	the length of time (overhead) required by the processing resource to switch from the execution of one scenario to a different one.
priorityRange	the set of valid priorities for this processor (often dependent on the operating system); these are used to define the scheduling priorities of resource actions.
isPreemptable	indicates whether or not the processor is preemptible once it begins execution of an action.

Associations

scenario	the set of scenarios for which this processing resource acts as a

host; for a processing resource to be a host of a scenario, all of the steps of that scenario have to be deployed on this resource. In effect, this association is an instance of the deploys relationship, as defined in Section , “The Deploys Mapping,” on page 4-30, but is rendered here as a stereotyped association for convenience.

8.1.4.6 Passive Resource

A resource protected by an access mechanism (e.g., a semaphore), which is accessed during the execution of an operation. It may be shared by multiple concurrent resource operations. It may represent either a physical device or a logical protected-access entity.

Attributes

schedulingPolicy	(inherited from Resource) the access control policy for handling requests from scenario steps. The following policies are predefined: *FIFO* = first-in-first-out *PriorityInheritance* = priority inheritance *NoPreemption* = no pre-emption *HighestLockers* = highest lockers *PriorityCeiling* = priority ceiling (note that in this case, we also need to provide the actual value of the priority ceiling, which generally is the priority of the highest priority scenario step that uses the resource)
capacity	the number of permissible concurrent users, for example using a counting semaphore.
accessTime	the time delay suffered by a scenario or scenario step in acquiring and releasing a resource.
utilization	(inherited from Resource) the mean number of concurrent users of the resource.
responseTime	the total time expired from the moment the resource is requested to its release; this includes the time spent waiting to get access to the resource (waiting time – see below).
waitingTime	the time from the instant an access request for the resource is issued, to the time it is granted.

Associations

service	the set of resource-operations that require the resource.
resourceUsageStep	the set of steps that access this resource.

8.1.4.7 Workload (abstract)

A workload specifies the intensity of demand for the execution of a specific scenario as well as the required or estimated response times for that workload. The specification of the workload depends on its subtype.

Attributes

response Time	the delay between the instant the scenario has started and the instant

	when the scenario has completed, for the specified workload; multiple instances of this attribute may be defined, with different modifiers, to represent requirements, estimates from models, and measurements.
priority	the priority of the workload.

Associations

scenario	the scenario corresponding to this workload.

8.1.4.8 OpenWorkload

A workload that is modeled as a stream of requests that arrive at a given rate in some predetermined pattern (such as Poisson arrivals).

Attributes

occurrencePattern	for an open workload, the pattern of interarrival times between consecutive instances of the start event; this is a potentially complex specification, depending on the nature of the series of intervals.

8.1.4.9 ClosedWorkload

A workload characterized by a fixed number of active or potential users or jobs which cycle between executing the scenario, and spending an external delay period (sometimes called “think time”) outside the system, between the end of one response and the next request

Attributes

population	the size of the workload (number of system users).
externalDelay	the delay between the end of one response and the start of the next for each member of the population of system users.

8.1.4.10 Performance Values

For performance analyses to be meaningful, it is usually not sufficient to simply provide numerical values for performance-related QoS characteristics but also to identify the semantics of those values. Thus, a given value may represent an average or maximum, or, it may be a prediction or a measurement, etc. Clearly, the interpretation of an analysis result depends on such characteristics. For this reason, the values used in performance analyses take on the following general structure (expressed using standard BNF conventions):

<performance-value> ::= <source-modifier> <type-modifier> <time-value>

Where:

<source-modifier>	defines how the value was obtained and can be one of: *required*, *assumed*, *predicted*, or *measured* (with the obvious interpretations).
<type-modifier>	the statistical meaning of the value which can be one of: *average*,

variance, k^{th} moment, maximum, k^{th} percentile, or *distribution*.

<time-value> the actual time value, which may itself be complex (e.g., a probability distribution).

Furthermore, there is often a need to combine multiple such values for a single characteristic. For instance, it may be necessary to specify both the mean and variance for some QoS characteristic. Consequently, the generic form for a performance QoS characteristic may take on the following general form:

<PA-characteristic> ::= <performance-value> [<performance-Value>]*

8.2 UML Viewpoint

In this section we describe how the domain concepts can be represented in UML. First we discuss the mappings in general, an then introduce the actual UML extensions defined for this purpose.

8.2.1 Mapping Performance Domain Concepts into UML Equivalents

Since performance is a dynamic property, scenarios play a key role in determining a system's performance characteristics from its UML models. In UML, scenarios are most directly modeled either using collaborations or activity graphs. The ways in which the performance domain concepts are represented in the two approaches can be quite different. When it comes to modeling complex hierarchical scenarios, the activity based approach has some significant advantages due to both its conceptual base and also to its notational convenience.

8.2.1.1 The Collaboration-Based Approach

There are two variants in this approach: one based on collaborations and the other on collaboration instances. Either form may be used since there are no semantic differences as far as the interpretation of the results is concerned[1]. The choice depends on circumstances (i.e., whichever model is more readily available) or individual preference of the modeller.

PerformanceContext

A performance context is modelled as a stereotype «PAcontext» of a UML collaboration (or collaboration instance set). This means that all interactions specified in that collaboration represent scenarios in the sense of this sub-profile. (For an example, see Figure 8-8 on page 8-27.)

1. It can be argued that a collaboration is a generic specification for a *set* of different executions, whereas a collaboration instance set might specifies a single execution. However, we do not make such a distinction in this profile.

Scenario

A scenario maps to an interaction (or interaction instance set). However, since interactions are not explicit graphical elements in UML diagrams (they are represented by collections of graphical elements), we will not define an explicit stereotype. Instead, a scenario will be represented by its first (root) step. Thus, any performance attributes of a scenario (such as its workload specs) will be attached to this model element (see below). (For an example, see Figure 8-8 on page 8-27.)

Step

A step represents an execution of some action. There are two alternatives for identifying performance steps in collaborations (collaboration instance sets):

associate a step stereotype («PAstep») directly with an action execution model element[1] or

associate the stereotype with the message (stimulus) model element that directly causes that action execution.

If action executions are used, then the *successor* steps of a given step are represented by the set of action executions that are directly linked to the messages (stimuli) generated from that action execution. If the step is associated with a message, then the successor steps are identified by the set of successors of the message (stimulus) in the same interaction.

If the step is a root step, then it may optionally be stereotyped with the appropriate workload stereotype («PAopenLoad» or «PAclosedLoad»). (For an example, see Figure 8-8 on page 8-27).

Workload

A workload is specified as a stereotype of a step (see Section 8.1.4.3, “Step,” on page 8-7). The stereotype can only be applied to the first step of a scenario. (For an example, see Figure 8-8 on page 8-27.)

ProcessingResource

The modeling of processing resources can be done in one of two ways. The most direct is to associate the appropriate stereotype («PAhost») with any classifier role (or instance) that executes scenario steps. However, this is only useful in cases where each classifier role (instance) is executing on its own host. This is rarely the case.

Much more common is the situation where the different classifier roles (instances) are executing on different hosts with the strong possibility that some hosts will be hosting more than one role (instance). In that case, the collaboration (collaboration instance set) does not contain sufficient information to determine the allocation of roles (instances) to

1. UML 1.4 does not support the concept of an action execution. However, a proposal for amending the UML metamodel to allow this is included in this profile.

hosts. Under those circumstances, it is necessary to determine which processor resource is running which classifier role (instance) on the basis of «deploys» relationships to nodes that are stereotyped as hosts[1] (for an example see Figure 8-4).

PassiveResource

Passive resources are represented directly by classifier roles (or instances) stereotyped with the «PAresource» stereotype. Alternatively, they may be modelled indirectly by specifying one or more *PAextOp* tagged values for the steps that access those resources. (For an example, see the “sendFrame” step in Figure 8-8 on page 8-27.)

8.2.1.2 Activity-Based Approach

In this approach, a scenario is captured by an activity graph. As noted, this form has certain advantages due to the explicit hierarchical nature of activity graphs, which can decompose a subactivity into a lower-level activity graph.

For an illustration of this approach, see Figure 8-9 on page 8-28 and Figure 8-10 on page 8-29.

PerformanceContext

A performance context can be modeled by an activity graph that is stereotyped as a «PAcontext». Since an activity graph contains only one activity, this approach is restricted to just a single scenario per context. The other requirement for this approach is that the swimlanes of the diagram represent either the resources themselves or the object instances (or roles) that are participating in the activity (see below).

Note that because subactivity states allow the modeling of hierarchical activity graphs, it is possible for one performance context to have a number of related sub-performance contexts representing expanded scenario steps. The association of such performance contexts is achieved by the link between a subactivity state and its associated submachine (activity graph) as defined in the UML metamodel. The topmost performance context in this hierarchy is the only one that can have a workload defined.

Scenario

Scenarios are modeled by the set of states/activities and transitions of the activity graph. As in the collaboration-based approach, we will not define an explicit scenario stereotype, but will identify the scenario with the first step (action or subactivity state) of the activity graph. The workload information is attached to this step, which is stereotyped appropriately («PAopenLoad» or «PAclosedLoad»).

1. If such a relationship does not exist for a given classifier role (instance) in the model, the analyses cannot be performed due to insufficient information.

Step

Each action or subactivity state of the graph is stereotyped as a «PAstep»[1]. The *successor* set of a step is identified by the action or subactivity states that are directly linked to that step by transitions (or, possibly, via pseudostates).

The initial action or subactivity state may also be stereotyped as a «PAopenLoad» or «PAclosedLoad» as appropriate. However, this can only be done at the top-level performance context. The subordinate performance contexts assume the load that is imposed on them by their hierarchically superior performance contexts.

Workload

The scenario workload is modeled by the tagged values associated with the first step of the topmost performance context. In addition to being stereotyped as a scenario step, this action or subactivity state can be stereotyped as a «PAopenLoad» or «PAclosedLoad», as appropriate.

ProcessingResource

The modeling of processing resources can be done in one of two ways. The most direct is to associate the appropriate stereotype («PAhost») with an activity graph partition that is linked to the appropriate object (or classifier role) However, this is only useful in cases where each classifier role (instance) is executing on its own host.

Much more common is the situation where the different partitions represent instances that are executing on different hosts and that some instances share hosts. In that case, the activity graph does not contain sufficient information to determine the allocation of objects to hosts. Under those circumstances, it is necessary to determine which processor resource is running which instance on the basis of «deploys» relationships to nodes that are stereotyped as hosts (for an example see Figure 8-11 on page 8-29).

PassiveResources

Passive resources are represented directly by partitions (swimlanes) stereotyped with the «PAresource» stereotype. Alternatively, they may be modelled indirectly by specifying one or more PAextOp tagged values for the steps that access those resources. (For an example, see the “sendFrame” step in Figure 8-10 on page 8-29.)

8.2.2 UML Extensions

We noted above that performance analysis is based on instances. Hence, the stereotypes above were described as applying to instance-based concepts (classifier role, instance, action execution, etc.). However, sometimes it is useful for the type specifications corresponding to those instances (e.g., classes, actions, etc.) to be stereotyped with the

1. Of course, we only stereotype UML concepts and not domain concepts. However, saying that we are “stereotyping a step,” is merely a shorthand way of saying that we are applying a stereotype to a UML model element that has already been stereotyped as a «PAstep». We will use this convention throughout this section.

same stereotypes. This is a shortcut that allow us to define default values for all instances based on that type. Any such default values, however are overridden by values that appear in specific instances that appear in performance contexts.

8.2.2.1 Naming Conventions

To minimize the possibility of confusion and conflict with other profiles, we will prefix all extension element names pertaining to this portion of the profile with the “PA”prefix.

8.2.2.2 Profile Package

For convenience, all extensions related to this part of the profile are packaged in a single profile package called “PAprofile.” This profile should be imported by other model analysis profiles that are based on it.

8.2.2.3 Stereotypes and Associated Tags

Based on the modeling approach identified in section In this section we describe how the domain concepts can be represented in UML. First we discuss the mappings in general, an then introduce the actual UML extensions defined for this purpose. above, it is not necessary to define a separate stereotype for every domain concept, since some of the concepts do not appear explicitly in UML models. However, their presence can be inferred from the presence of others. For descriptions of the semantics of the stereotypes defined here and the rules for their usage, the reader should refer to sections:
Section 8.1.3, “Domain Model,” on page 8-4 and Section 8.1.4, “Domain Concept Details,” on page 8-5. In this section we describe how the domain concepts can be represented in UML. First we discuss the mappings in general, and then introduce the actual UML extensions defined for this purpose respectively.

«PAclosedLoad»

This stereotype models a closed workload (see Section 8.1.4.9, “ClosedWorkload,” on page 8-10).

Stereotype	Base Class	Tags
«PAclosedLoad»	Message	PArespTime PApriority PApopulation PAextDelay
	Stimulus	
	Action State	
	SubactivityState	
	Action	
	ActionExecution	
	Operation	
	Method	
	Reception	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PArespTime	PAperfValue	[0..*]	Workload::responseTime
PApriority	Integer	[0..1]	Workload::priority
PApopulation	Integer	[0..1]	ClosedWorkload::population
PAextDelay	PAperfValue	[0..1]	ClosedWorkload::externalDelay

The following constraint is defined for this stereotype:

- This stereotype can only be applied to be the first step in a performance context.

«PAcontext»

This models a performance analysis context (see Section8.1.4.1, “ Performance Context,” on page 8-5).

Stereotype	Base Class
«PAcontext»	Collaboration
	CollaborationInstanceSet
	ActivityGraph

The following constraints are defined for this stereotype:

- A performance analysis context must contain at least one element that is stereotyped as a kind of step.
- A performance analysis context based on collaborations must have exactly one model element stereotyped as a workload.
- Only a top-level performance context can have a workload defined.

«PAhost»

This stereotype models a processing resource (see Sectio n8.1.4.5, “ProcessingResource,” on page 8-8).

Stereotype	Base Class	Tags
«PAhost»	Classifier	PAutilization PAschdPolicy PArate PActxtSwT PAprioRange PApreemptable PAthroughput
	Node	
	ClassifierRole	
	Instance	
	Partition	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PAutilization	Real	[0..*]	Resource::utilization
PAschdPolicy	Enumeration: {‘FIFO’, ‘HeadOfLine’, ‘PreemptResume’, ‘ProcSharing’, ‘PrioProcSharing’, ‘LIFO’}	[0..1]	ProcessingResource::schedulingPolicy
PArate	Real	[0..1]	ProcessingResource::processingRate
PActxtSwT	PAperfValue	[0..1]	ProcessingResource::contextSwitchTime
PAprioRange	Integer range	[0..1]	ProcessingResource::priorityRange
PApreemptable	Boolean	[0..1]	ProcessingResource::isPreemptable
PAthroughput	Real	[0..1]	Resource::throughput

The following constraints are defined for this stereotype:

- This stereotype can only be applied to be the first step in a performance context

«PAopenLoad»

This models an open workload (see Section 8.1.4.8, “OpenWorkload,” on page 8-10).

Stereotype	Base Class	Tags
«PAclosedLoad»	Message	PArespTime PApriority PAoccurrence
	Stimulus	
	Action State	
	SubactivityState	
	Action	
	ActionExecution	
	Operation	
	Method	
	Reception	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PArespTime	PAperfValue	[0..*]	Workload::responseTime
PApriority	Integer	[0..1]	Workload::priority
PAoccurrence	RTarrivalPattern	[0..1]	OpenWorkload::population

The RTarrivalPattern type of the occurrence tag is defined in “RTarrivalPattern” on page 5-33.

The following constraints are defined for this stereotype:

- This stereotype can only be applied to be the first step in a performance context.

«PAresource»

This stereotype models a passive resource (see Section 8.1.4.6, “Passive Resource,” on page 8-9).

Stereotype	Base Class	Tags
«PAresource»	Classifier	PAutilization PAschdPolicy PAschdParam PAcapacity PAaxTime PArespTime PAwaitTime PAthroughput
	Node	
	ClassifierRole	
	Instance	
	Partition	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PAutilization	Real	[0..*]	Resource::utilization
PAschdPolicy	Enumeration: {‘FIFO’, ‘PriorityInheritance’, ‘NoPreemption’, ‘HighestLockers’, ‘PriorityCeiling’}	[0..1]	PassiveResource::schedulingPolicy
PAschdParam	Real	[0..*]	PassiveResource::schedulingPolicy (for providing numerical values associated with the scheduling policy, such as the priority ceiling value)
PAcapacity	Integer	[0..1]	PassiveResource::capacity
PAaxTime	PAperfValue	[0..n]	PassiveResource::accessTime
PArespTime	PAperfValue	[0..n]	PassiveResource::responseTime
PAwaitTime	PAperfValue	[0..n]	PassiveResource::waitTime
PAthroughput	Real	[0..1]	Resource::throughput

The following constraints are defined for this stereotype:

- This stereotype can only be applied to be the first step in a performance context.

«PAstep»

This models a step in a performance analysis scenario (see Section 8.1.4.3, “Step,” on page 8-7).

Stereotype	Base Class	Tags
«PAstep»	Message	PAdemand PArespTime PAprob PArep PAdelay PAextOp PAinterval
	Stimulus	
	Action State	
	SubactivityState	

Tag definitions:

Tag	Type	Multiplicity	Domain Attribute Name
PAdemand	PAperfValue	[0..*]	Step::hostExecutionDemand
PArespTime	PAperfValue	[0..*]	Step::responseTime
PAprob	Real	[0..1]	Step::probability
PArep	Integer	[0..1]	Step::repetition
PAdelay	PAperfValue	[0..*]	Step::delay
PAextOp	PAextOpValue	[0..*]	Step::operations
PAinterval	PAperfValue	[0..*]	Step::interval

8.2.2.4 *Tagged Value Types*

The following types of tag value strings are defined for use with the stereotypes above. We have used TVL to describe these often complex strings (see Appendix A - *The Tag Value Language*). They are all instances of the TVL list type. The elements of the list are typically mixtures of strings, numeric literals, TVL variable names, and TVL expressions.In representing the syntax of these types, we use the following standard BNF notational conventions:

- A string between double quotes (“) represents a literal.
- A token in angular brackets (<element>) is a non-terminal.
- A token enclosed in square brackets ([<element>]) implies an optional element of an expression.
- A token followed by an asterisk (<element>*) implies an open-ended number of repetitions of that element.
- A vertical bar indicates a choice of substitutions.

Note that TVL uses parentheses to identify arrays, commas to separate elements of arrays, and single quotes for string literals.

PAperfValue

These strings are used to specify a complex performance value as defined in Section 8.1.4.10, “Performance Values,” on page 8-10. The value is an array in the following format:

“(“ <source-modifier> “,” <type-modifier> “,” <time-value> “)”

Where:

<source-modifier> ::= ‘req’ | ‘assm’ | ‘pred’ | ‘msr’

is a string that defines the source of the value meaning respectively: required, assumed, predicted, and measured.

<type-modifier> ::= ‘mean’ | ‘sigma’ | ‘kth-mom’ , <Integer> | ‘max’ |’percentile,’ <real> | ‘dist’

is a specification of the type of value meaning: average, variance, k^{th}-moment (integer identifies value of k), percentile range (real identifies percentage value), probability distribution.

<time-value> is a time value described by the RTtimeValue type (note that this value could be a variable such as @T1).

For example, the tagged value expression:

{PAdemand = (‘msr’, ‘mean’, (20, ‘ms’))}

represents a demand in a scenario step with a measured mean value of 20 milliseconds (note the parenthesis around the last term can be removed with no change in semantics). For more examples of these performance value strings, refer to the example provided in Section 8.2.3.1, “Web Video Application,” on page 8-22.

PAextOpValue

This string is used to identify an external operation and either (a) the number of repetitions of that operation that are performed or (b) a performance time value. The time value of the operation is assumed to be defined elsewhere. The general format for this string is given as:

“(“ <String> “,” <integer> | <time-value> “)”

where the string is used to name the external operation, the integer provides a mean number of times that the operation is invoked, and time value is of type PAperfValue (see “PAperfValue” on page 8-21).

8.2.3 Modeling Guidelines and Examples

8.2.3.1 Web Video Application

This example is a web-based video-streaming application. A user sitting at a workstation needs to access a video over the internet through the services of a centralized web server. The logical structure of this application is shown in Figure 8-3.

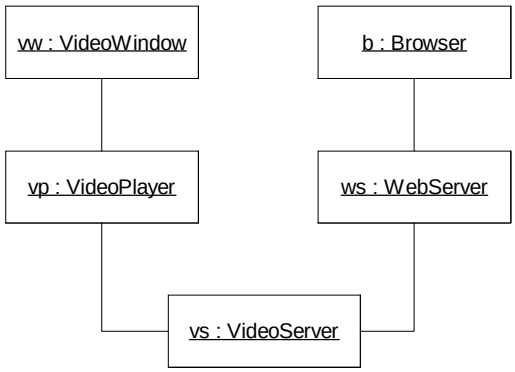


Figure 8-3 The instance structure of the web video application

The deployment of the logical elements across the engineering environment is shown in Figure 8-4..

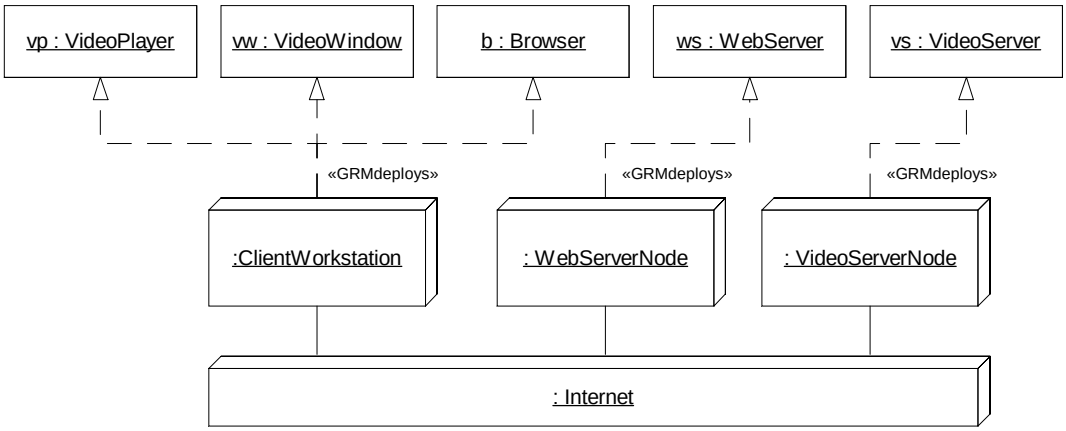


Figure 8-4 The deployment structure of the web video application

The user makes a selection through the web browser (b), asking the remote web server (ws) for a video to be fed back to the user's site. Based on the selection, the web server chooses an appropriate video server (vs), which initiates a video player (vp) on the user's site and then sends it a stream of video frames. The video frames are displayed back to the user through a dedicated video window (vw).

The scenario just described can be conveniently represented either by the sequence diagram in Figure 8-5 or, even better, by a hierarchical activity graph, as shown in Figure 8-6 and Figure 8-7.

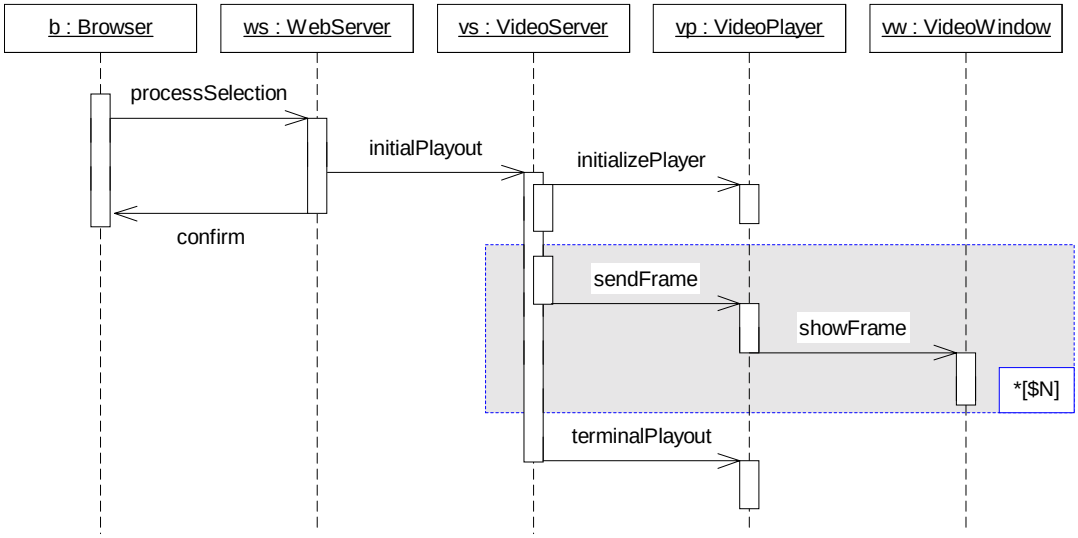


Figure 8-5 Video request and display scenario – sequence diagram representation

The advantage of activity graphs over sequence diagrams for this purpose is that they provide more direct ways of modeling

- concurrent forks and joins, such as the fork that occurs when a confirmation is sent to the browser at the same time that an initialization message is sent to the video server.
- hierarchical scenarios, since steps can be represented by subactivity states, which, in turn, have matching activity graphs. For example, the “send video” step shown in Figure 8-6 is expanded in Figure 8-7.

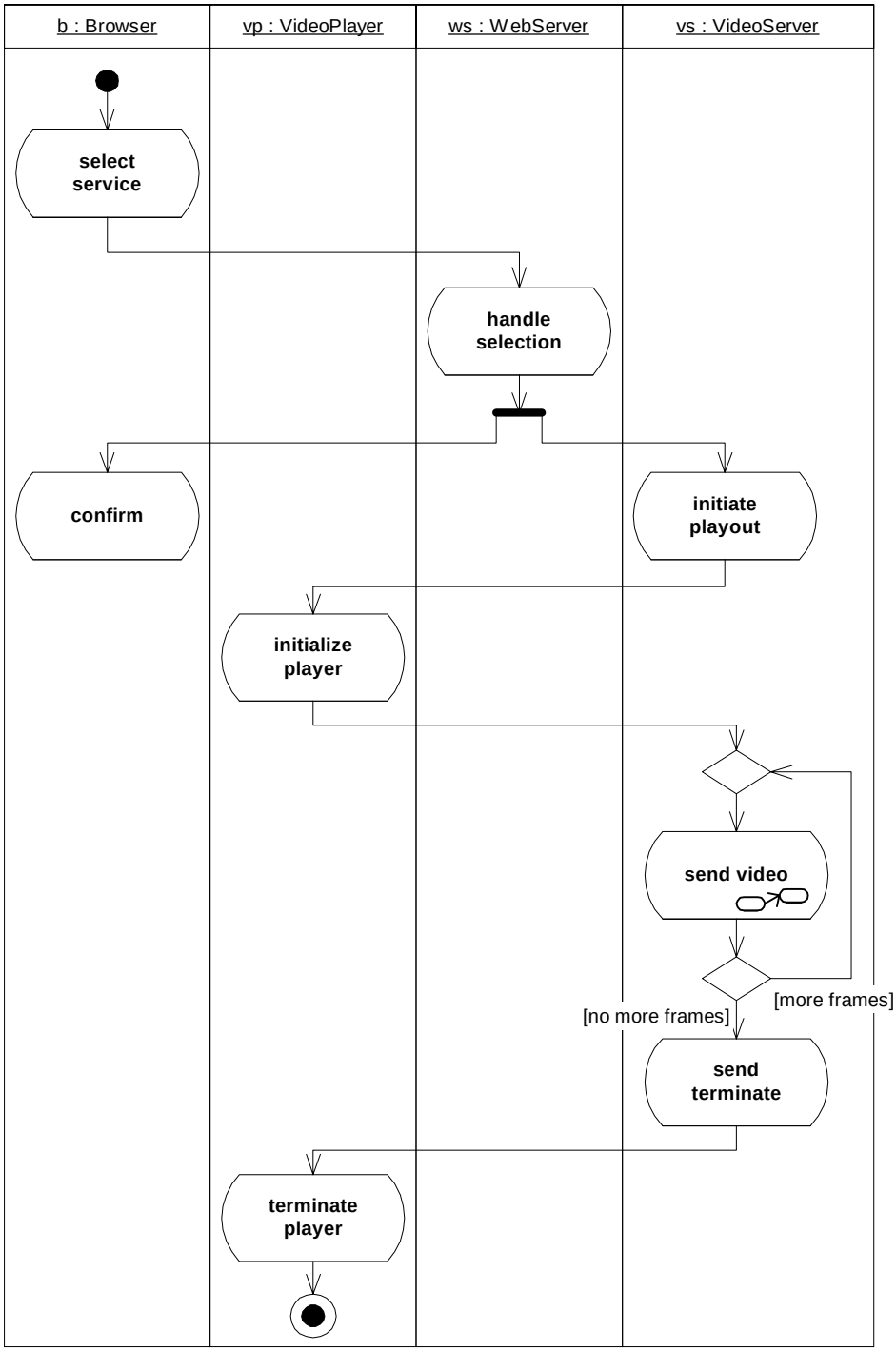


Figure 8-6 Video request and display scenario – hierarchical activity diagram representation

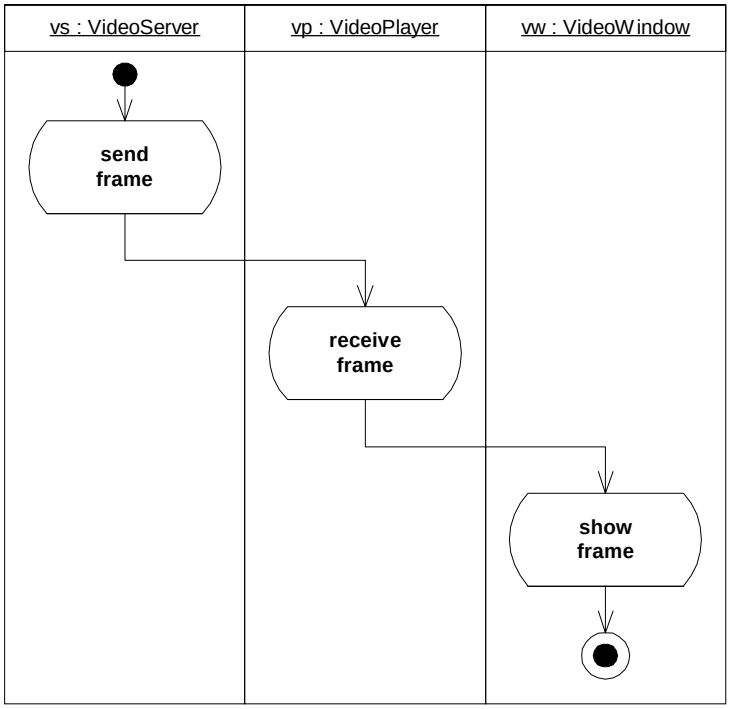


Figure 8-7 The activity graph corresponding to the “send video” subactivity in Figure 8-6

QoS Requirements

Next, we introduce the performance requirements of this application. Firstly, there is a performance requirement on the response time for the confirmation to the user that the request has been received. This requirement is specified as a probability that the delay in receiving the confirmation will not take longer than half a second in 95% of the cases:

Probability(Confirmation delay > 500 ms) < 0.05

Or, expressed as a percentile measure:

95^{th} percentile (Confirmation delay) < 500 ms

The next performance requirement is on the video stream that is fed back to the user. Frames should be displayed at regular intervals of 30 ms, and there is a performance requirement on the jitter, that the probability of a frame being displayed late is less than 1%:

Probability (Interval between frame display instants < 30 ms) > 0.99

This very high-level view is sufficient to introduce the system and the performance issues. The confirmation delay is the interval between the “Select service” and “Receive confirmation” activities. The jitter of the video reception is defined over intervals between successive showFrame operations.

To analyze performance, we would require data on the execution of the responsibilities. Suppose, purely for the purpose of exposition, that we have the following values (they are labeled as to whether they are measured values, estimates, or assumptions):

- (estimate) video server processing demand per frame: mean = 10 ms
- (estimate) video viewer processing demand per frame: mean = 15 ms, standard deviation (stdv) = 10 ms
- (assumed) network delay distribution: exponential with mean: 10 ms
- (measured) local network packets per frame: 65
- (measured) file operations per frame, for retrieval for the video server: 12

Additional parameters that are needed to complete an evaluation include the requirements, and a description of the workload intensity. Here, we will use the folowing additional parameters:

- the number of users active in the system at one time is limited to $NUsers
- external delay: each user has an average delay between ending one session and beginning another of 20 minutes
- frames in a video: $N, a variable
- video frame interval: 30 ms
- (requirement) confirmation delay: 95% value < 500 ms
- (requirement) interval between successive frame displays: 99% value < 30 ms.

If the client operations are divided between a Browser component and a VideoPlayer component, the software design has four components, which will be shown in the following analysis.

The Annotated UML Model

For convenience, we have mostly used the comment-based notation for displaying the relevant performance-related values in the example. While this increases visual clutter, it is much easier to discern which specific QoS attributes apply to which elements. Of course, it is assumed that computer-based tools can significantly reduce and possibly eliminate such clutter, by hiding elements of a graph that are not of interest at a particular point in time.

First, in Figure 8-8 we show a sequence diagram for the scenario outlined above. The QoS attributes associated with the events and actions of this diagram are taken directly from the requirements. In this case, the closed workload is associated with the first action execution, since there is no explicit user in the model (had there been one, then the workload could have been attached to the stimulus coming from the user to the browser).

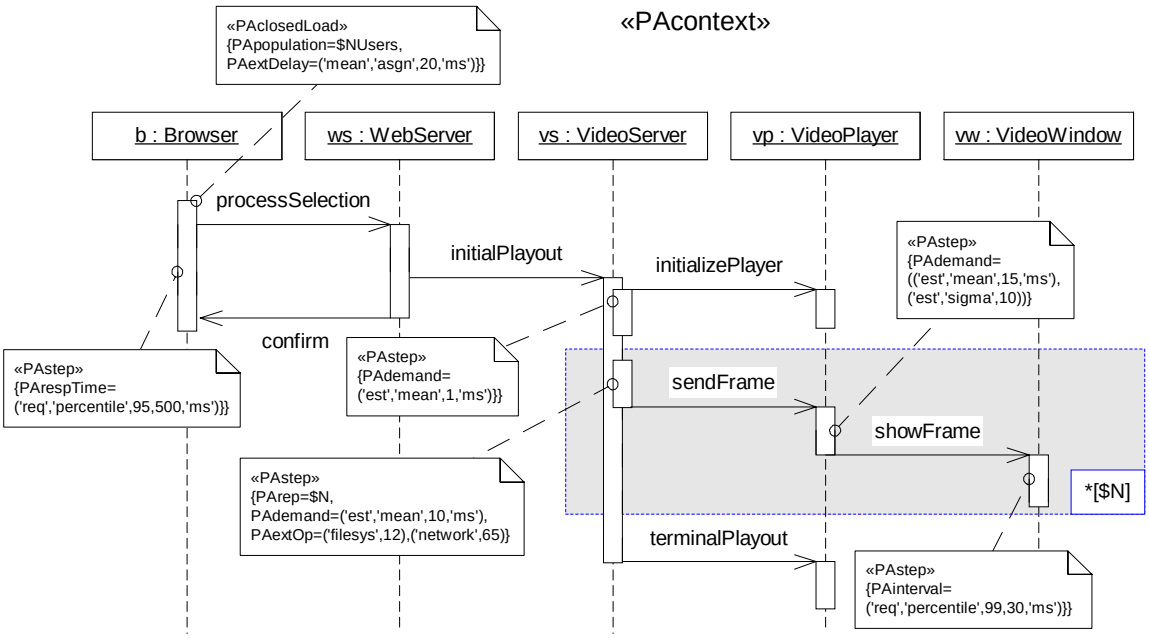


Figure 8-8 Sequence diagram of web video application with performance annotations (partial set)

Alternatively, we can annotate the sequence diagram with the same information (Figure 8-9 and Figure 8-10). The choice is up to the modeler and the circumstances on hand.

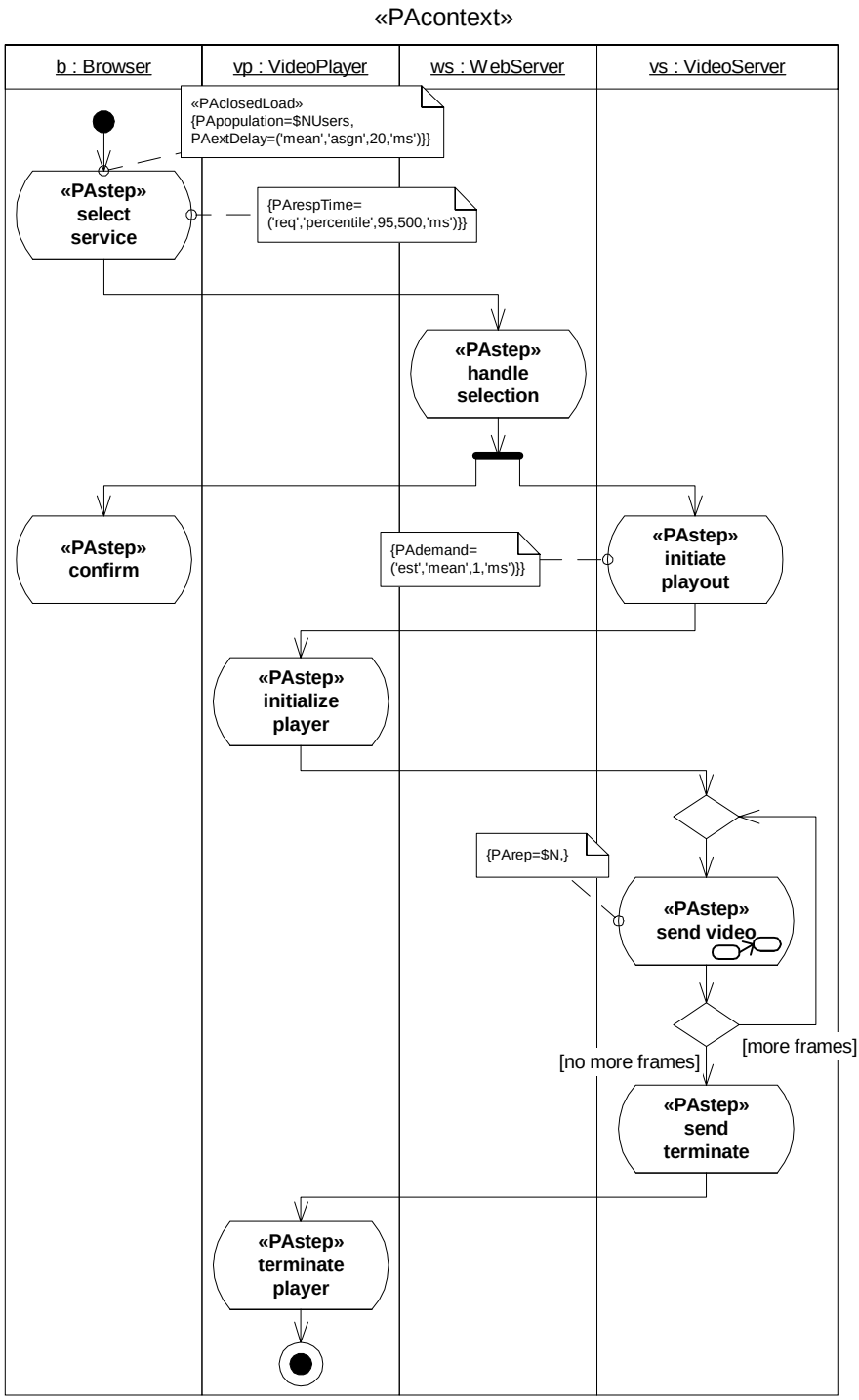


Figure 8-9 Annotated activity diagram of the web video application with performance annotations

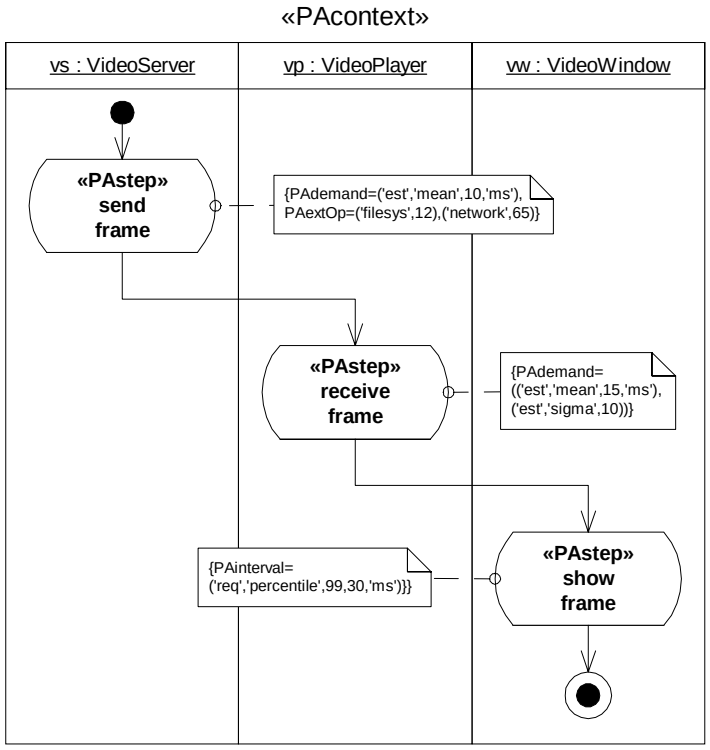


Figure 8-10 Details of the “send video” subactivity with performance annotations

Finally, in, Figure 8-11 we also depict the annotated deployment diagram. In this case, to reduce visual clutter, we have only shown the QoS attributes of one host processor. Notice the use of the $Util variable whose value is to be determined by model analysis.

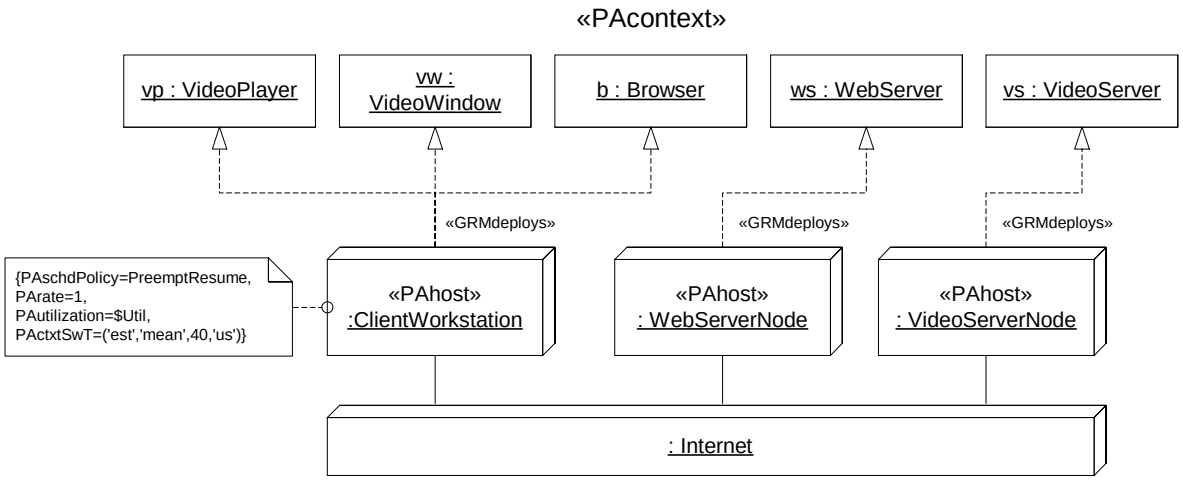


Figure 8-11 Annotated deployment model for the web video application

A model annotated in this manner contains sufficient information to be analyzed for basic properties such as processor utilization.

8.2.4 Required UML Metamodel Changes

This package assumes that the UML metamodel is modified to support the action execution concept. This is defined in Section 4.1.9.3, “ActionExecution,” on page 4-17. No other metamodel changes are assumed or required.

8.2.5 Proposed Notational Extensions

No special notation is proposed for performance based modeling.

Real-Time CORBA Applications 9

In this chapter we demonstrate how the general concepts and concrete extensions defined in this specification can be used to model applications that use OMG's Real-Time CORBA standard. Note that this is not a model of Real-Time CORBA itself, although abstract representation of some client-visible aspects of that standard are included. The purpose of these extensions is to facilitate schedulability analyses of Real-Time CORBA applications, by annotating UML models of such applications in terms of concepts defined in the schedulability framework of this specification as described in the *Schedulability Modeling* chapter. Thus, they constitute a sub-profile of the schedulability sub-profile.

(For those who need a detailed UML model of Real-Time CORBA we provide the basic class definitions in Appendix B - *Model of Real-Time CORBA*.)

9.1 Domain Viewpoint

Real-Time (RT) CORBA is a set of extensions built on top of standard CORBA. One of its primary purposes is to provide a framework for predictable end-to-end execution times of CORBA-based applications in environments that may be distributed and heterogeneous (e.g., combining several different real-time OS-based parts into a common application).

The basic CORBA model, also supported by RT CORBA, is a client-server model. A client makes a series of requests to one or more servers that may be in different address spaces from the client. In the real-time context, such execution scenarios ("activities" in RT CORBA parlance) typically have deadline constraints. Since there can be multiple scenarios running concurrently, possibly sharing some of the same servers and resources, conflicts can arise so that special action needs to be taken to ensure that deadlines are respected. This requires the knowledge of relative priorities of activities, controlled access to shared resources (to prevent deadlocks and to minimize priority inversion), etc. RT CORBA provides built-in mechanisms for handling this, provided that it is given information of the QoS characteristics of the various scenarios.

Figure 9-1 depicts a domain model of RT CORBA applications. The model is fully based on the conceptual framework provided by the scheduling domain model (see Figure 7-1 on page 7-5). This approach was chosen to simplify the schedulability analysis of UML models that represent RT CORBA application..

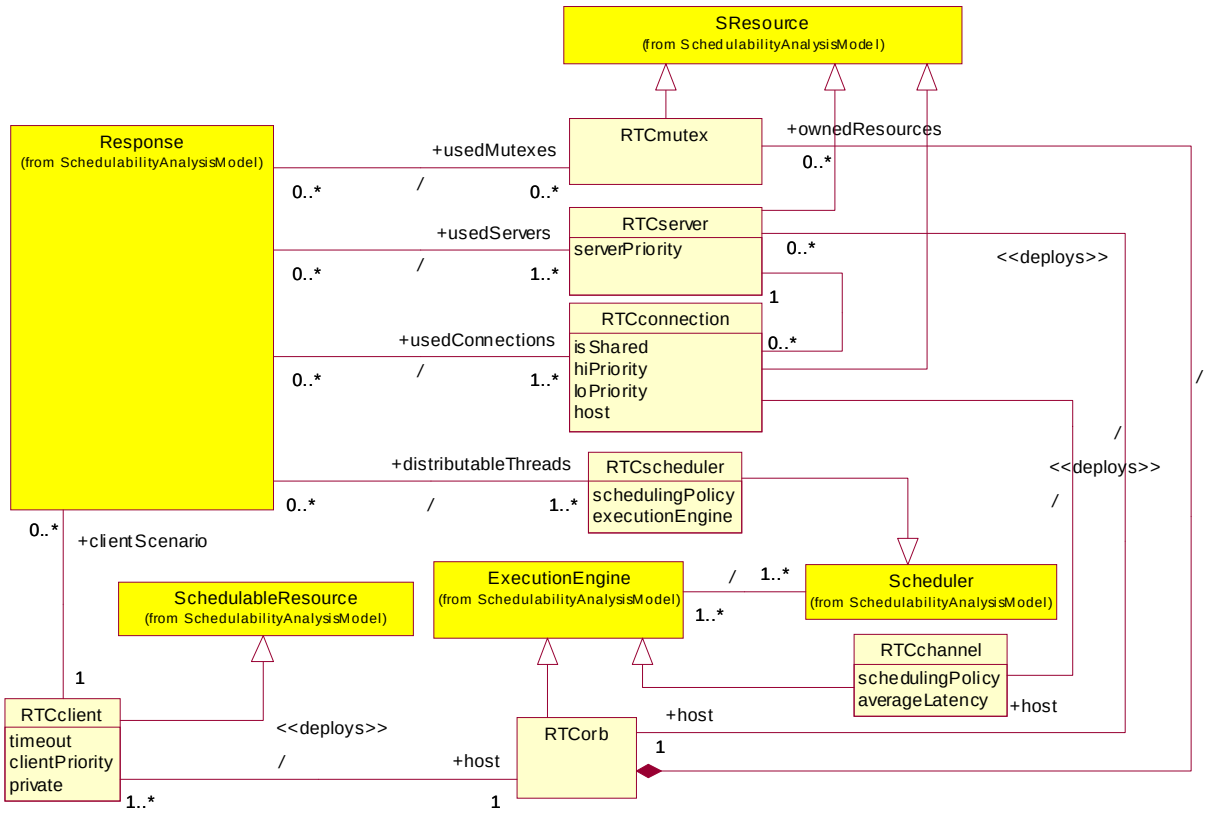


Figure 9-1 The Real-Time CORBA domain model

RT CORBA clients make requests, which represents “responses” in terms of the schedulability model in this specification. It is useful to recall here that a response is a kind of composite action (see Section 4.1.9.30, “Scenario,” on page 4-25), which means that it can be decomposed into a set of finer-grained actions, allowing a finer-grained model if desired (e.g., the individual actions may be accessing different resources). Note that we do not show the trigger that activates a response in this model, but it would certainly need to be specified in a concrete case if schedulability analysis is to be performed.

The client requests (responses) are made through one or more RT CORBA connections, which connect clients to servers. These connections may be shared or they may be exclusive to the client. In some cases, multiple shared connections may be set up to allow priority-based handling of requests. These are known as “banded connections” since each connection is dedicated to a particular interval of priority values (delineated by a high priority and a low priority).

Note that request priorities may be set either by clients (“client-propagated priority model”) or by servers (“server-declared priority model”).

An RT CORBA client may also use one or more mutexes which are used as binary semaphores.

Connections, servers, and mutexes are all examples of resources as defined in the schedulability model (note that this means that they are all protected resources). The role of the execution engine of the schedulability model is played here by the real-time ORB (RTCorb in the class diagram), since it is the platform on which all these resources run and which is responsible for administering the scheduling policy that is built into the RT ORB. (Oddly enough, RT CORBA does not specify an explicit mechanism for defining such a policy – that is outside the scope of the standard. All it requires is that it must support some type of priority inheritance or priority ceiling protocol.)

Finally, the RT CORBA client acts as a platform for the scenarios that it executes, so it is a natural match for the Schedulable Resource Concept of the schedulability model.

9.1.1 Domain Concept Details and Usage

We now look at each of the domain concepts and explain their characteristics and usage. We describe the concepts primarily from a modeling perspective and do not go into the finer details of the individual concepts. Those can be found in the appropriate OMG document.

9.1.1.1 RTCchannel

This is an instance of a communication channel between RT CORBA ORBs. It represents the transport over which tow ORBs communicate.

Attributes

averageLatency	the one-way latency (delay) for the communication channel. This is a time value expression, so it may be an integer, a distribution function, or a histogram.

Associations

schedulingPolicy	(inherited from ExecutionEngine) is the scheduling policy for this communication channel. The usual policies are FixedPriority or FIFO.
rTCconnection	the set of RT CORBA connections that are deployed on this channel. In effect, this is an instance of a deploys relationship. However, for convenience, it is rendered in the domain model as a stereotyped association.

9.1.1.2 RTCclient

This is an instance of an RT CORBA client. It encompasses both the application part of the client as well as the underlying RT CORBA infrastructure machinery (the IDL “stub”). This allows us to view it from both perspectives based on convenience. In its

infrastructure role, it represents the platform (e.g., process, task, thread), which runs the client response. This is why it is a subclass of SchedulableResource. In its application role, it executes the client functionality.

Attributes

clientPriority	This attribute is only specified in the client-propagated priority model in which priority is specified with the service request as made by the client. If this attribute is not specified then we will assume that the client propagated priority model is not being used. This priority value is used to determine the priority of the responses undertaken by this client.
private	this Boolean attribute identifies that the client has a private non-multiplexed transport connection to the server. If this attribute is omitted, then it is assumed that there is no private connection present.
timeout	the round-trip timeout value; that is, the maximum time interval in which the response is expected to complete; this value is passed as the absolute deadline of the response.

Associations

clientScenario	the set of scenario executions performed by the client instance, in response to some trigger (e.g., a real-time clock interrupt).
host	the instance of the RT ORB on the client side on which this client is deployed. In effect, this is an instance of a deploys relationship as defined in “The Deploys Mapping” on page 4-30. However, for convenience, it is rendered in the domain model as a stereotyped association.

9.1.1.3 RTCconnection

This is a kind of communication resource that is used by a client to communicate with the server. It is a kind of protected resource since it may be multiplexed across a number of clients and scenarios. The specifics of the modeling of its communication QoS characteristics (throughput, delay, etc.) are outside the scope of this specification (for one possible approach, refer to “Passive Resource” on page 8-9).

Attributes

capacity	(inherited from SResource) this attribute may be used to specify the communication QoS characteristic; as noted, the details of this are out of scope for this specification.
isShared	this Boolean attribute indicates whether this connection is shared or dedicated to a single client. In the latter case, the corresponding client has to have its “private” attribute set.
hiPriority	the highest priority of communications stimuli that can be passed through this connection (the priority is an RT CORBA priority); if the loPriority value is omitted, then this connection does not represent a banded connection.
loPriority	the lowest priority of communication stimuli that can be passed

	through this connection; note that this integer value must be less than or equal to the hiPriority attribute.

Associations

response	the set of responses that use this connection.
rTCserver	the RT CORBA server to which this connection is attached.
accessControlPolicy	(inherited from ProtectedResource via SResource) this is the access control policy that is used to access the channel.
host	the communication channel that implements this connection

9.1.1.4 RTCmutex

This is a representation of an instance of an RT CORBA mutex. This is a kind of protected resource which can only be accessed/released using appropriate acquire and release operations.

Associations

accessControlPolicy	(inherited from ProtectedResource via SResource) this is the access control policy that is used by the mutex; it must be some form of priority inheritance policy or priority ceiling policy.
response	the set of responses that use this mutex.
rTCorb	the RT ORB that owns this mutex.

9.1.1.5 RTCorb

This is an instance of an RT CORBA ORB. It represents both the server and client sides. The ORB is responsible for administering its built-in scheduling policy. It is modeled as a kind of execution engine since it is the “virtual platform” upon which the client and server are deployed.

Associations

schedulingPolicy	(inherited from Execution Engine) the scheduling policy administered by this ORB. According to the standard, it has to be some kind of priority inheritance or priority ceiling policy.
rTCserver	the set of RT CORBA objects that act as servers and which are deployed on this ORB. In effect, this is an instance of a deploys relationship as defined in “The Deploys Mapping” on page 4-30 However, for convenience, it is rendered in the domain model as a stereotyped association.
rTCclient	the set of clients that are deployed on this ORB. In effect, this is an instance of a deploys relationship as defined in “The Deploys Mapping” on page 4-30. However, for convenience, it is rendered in the domain model as a stereotyped association.
ownedResources	the set of mutexes owned by this ORB instance.

9.1.1.6 RTCscheduler

This is an instance of an RT CORBA scheduler (see Section 7.1.3.7, “Scheduler,” on page 7-10). It becomes the scheduling policy for the RT CORBA POA with which it is associated. As an infrastructure element, it is the platform for execution of the scheduler interface and serves as a resource broker for execution engines. The RT CORBA 2.0 specification defines four modules for four types of schedulers (fixed priority, earliest deadline first, least laxity first, and maximum utility (see “Scheduling Policy” on page 7-10)).

Associations

response	the set of responses scheduled by this scheduler

9.1.1.7 RTCserver

This is an instance of an RT CORBA object that performs one or more published services. Like the client, this concept represents both the application and the CORBA infrastructure element. As an infrastructure element, it is the platform for executing the server code and is, therefore, a kind of SchedulableResource (as per the schedulability model; NB: this derivation is not shown explicitly in Figure 9-1). From an application aspect, it represents the server functionality.

Attributes

serverPriority	this is the priority that is used in case of the “server declared priority model”. If this attribute is not specified, we assume that the server declared model is not being used for this server.
Capacity	(inherited from SResource) this represents the capacity to handle concurrent requests; in case of an RT CORBA server, this reflects the size of the threadpools associated with the server/POA. The ability to more precisely model the complex threadpool mechanisms of RT CORBA (which includes threadpool lanes, static and dynamic threads, request buffering, and thread borrowing) are outside the scope of this sub-profile and will require future specializations.

Associations

response	the set of responses that are using this server.
rTCconnection	the set of connections bound to this server.
host	the execution engine (RT ORB) on which this server is deployed. In effect, this is an instance of a deploys relationship as defined in “The Deploys Mapping” on page 4-30. However, for convenience, it is rendered in the domain model as a stereotyped association.

9.2 UML Viewpoint

We now examine how the domain model concepts map to UML model elements and UML extensions.

9.2.1 Mapping RT CORBA Application Concepts to UML Equivalents

Since this is a specialization of the schedulability sub-profile, the approach used is identical to the one described in Section 7.2.1, “Mapping Schedulability Domain Concepts into UML Equivalents,” on page 7-13.

9.2.2 UML Extensions

9.2.2.1 Conventions

To minimize the possibility of confusion and name clashes, we will prefix all extension elements defined for this sub-profile using the “RSA” prefix (which stands for “Real-Time CORBA Schedulability Analysis). Note that some of the tags will be inherited from stereotypes defined in other parts of this specification, notably the stereotypes defined for schedulability analysis.

9.2.2.2 Profiles

For convenience, all extensions related to this sub-profile are packaged into a single profile package called “RSAprofile”. This profile imports from the schedulability analysis profile “SAprofile” and hence must be used with that profile.

9.2.2.3 Stereotypes

The following stereotypes and associated tagged values are defined for this sub-profile.

«RSAchannel»

Represents the RT CORBA channel concept as defined in “RTCchannel” on page 9-3..

Stereotype	Base Class	Parents	Tags
«RSAchannel»	Classifier	«SAengine»	RSAschedulingPolicy RSAaverageLatency
	ClassifierRole		
	Instance		
	Object		
	Node		

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
RSAschedulingPolicy	Enumeration: {'FIFO', 'RateMonotonic', 'DeadlineMonotonic', 'HKL', 'FixedPriority', 'MinimumLaxityFirst', 'MaximizeAccruedUtility', 'MinimumSlackTime'}	[0..1]	RTCchannel::schedulingPolicy
RSAaverageLatency	RTtimeValue	[0..1]	RTCchannel::averagelatency

«RSAclient»

Represents the RT CORBA client concept as defined in "RTCclient" on page 9-3.

Stereotype	Base Class	Parents	Tags
«RSAclient»	Classifier	«SASchedulable»	RSAtimeout RSAclPrio RSAprivate RSAhost
	ClassifierRole		
	Instance		
	Object		
	Node		

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
RSAtimeout	RTtimeValue	[0..1]	RTCclient::timeout
RSAclPrio	Integer	[0..1]	RTCclient::clientPriority
RSAprivate	Boolean	[0..1]	RTCclient::private
RSAhost	Reference to an element sterotyped as «RSAorb»	[0..1]	RTCclient::host

If the RSAhost tagged value is not specified, it can still be identified if this element is involved as the supplier of a «GRMdeploys» relationship with an element stereotyped as «RSAorb».

«RSAconnection»

Represents the RT CORBA connection concept as defined in “RTCconnection” on page 9-4.

Stereotype	Base Class	Parents	Tags
«RSAconnection»	Classifier	«SASchedulable» «SAResource»	SAAccessControl RSAshared RSAhiPrio RSAhost RSAloPrio RSAserver
	ClassifierRole		
	Instance		
	Object		
	Node		

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
SAAccessControl	Enumeration of: {PriorityInheritance, DistributedPriorityCeiling}	[0..1]	RTCconnection::Access Control Policy
RSAshared	Boolean	[0..1]	RTCconnection::isShared
RSAhiPrio	Integer	[0..1]	RTCconnection::hiPriority
RSAhost	Reference to an element stereotyped as «RSAchannel»	[0..1]	RTCconnection::host
RSAloPrio	Integer	[0..1]	RTCconnection::loPriority
RSAserver	Reference to an element sterotyped as «RSAserver»	[0..1]	RTCconnection::server

If the RSAserver tagged value is not specified, it can still be identified if this element is involved in an association with a model element stereotyped as «RSAorb».

«RSAmutex»

Represents the RT CORBA mutex concept as defined in “RTCmutex” on page 9-5.

Stereotype	Base Class	Parents	Tags
«RSAmutex»	Classifier	«SAResource»	SAAccessControl RSAhost
	ClassifierRole		
	Instance		
	Object		
	Node		

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
SAAccessControl	Enumeration of: {‘PriorityInheritance’, ‘DistributedPriorityCeiling’}	[0..1]	RTCmutex::Access Control Policy
RSAhost	Reference to an element sterotyped as «RSAorb»	[0..1]	RTCmutex::rTCorb

If the RSAhost tagged value is not specified, it can still be identified if this element is involved as the contained entity in a composition relationship with an element stereotyped as «RSAorb».

«RSAorb»

Represents the RT CORBA ORB concept as defined in “RTCorb” on page 9-5.

Stereotype	Base Class	Parents	Tags
«RSAorb»	[1]Classifier	«SAEngine»	SASchedulingPolicy
	ClassifierRole		
	Instance		
	Object		
	Node		

1. only the deployable type: Component, Artifact, Node, and Class

The defined tag is:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
SASchedulingPolicy	(see “«SAengine»” on page 7-17)	[0..1]	Execution Engine::SchedulingPolicy

«RSAserver»

Represents the RT CORBA server concept as defined in “RTCserver” on page 9-6.

Stereotype	Base Class	Parents	Tags
«RSAserver»	Classifier	«SAResource»	RSAsrvPrio SACapacity
	ClassifierRole		
	Instance		
	Object		
	Node		

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Attribute Name
RSAsrvPrio	Integer	[0..1]	RTCserver::serverPriority
SACapacity	Integer	[0..1]	RTCserver::Capacity

9.2.3 Modeling Guidelines and Example

We illustrate the use of these extensions on an elementary example consisting of one RT CORBA client and two RT CORBA servers. Clearly, such an example does not pose a challenging schedulability analysis problem, but our objective is to explain how the stereotypes are to be used without complicating things with the intricacies of schedulability analysis. The basic real-time situation is shown in Figure 9-2.

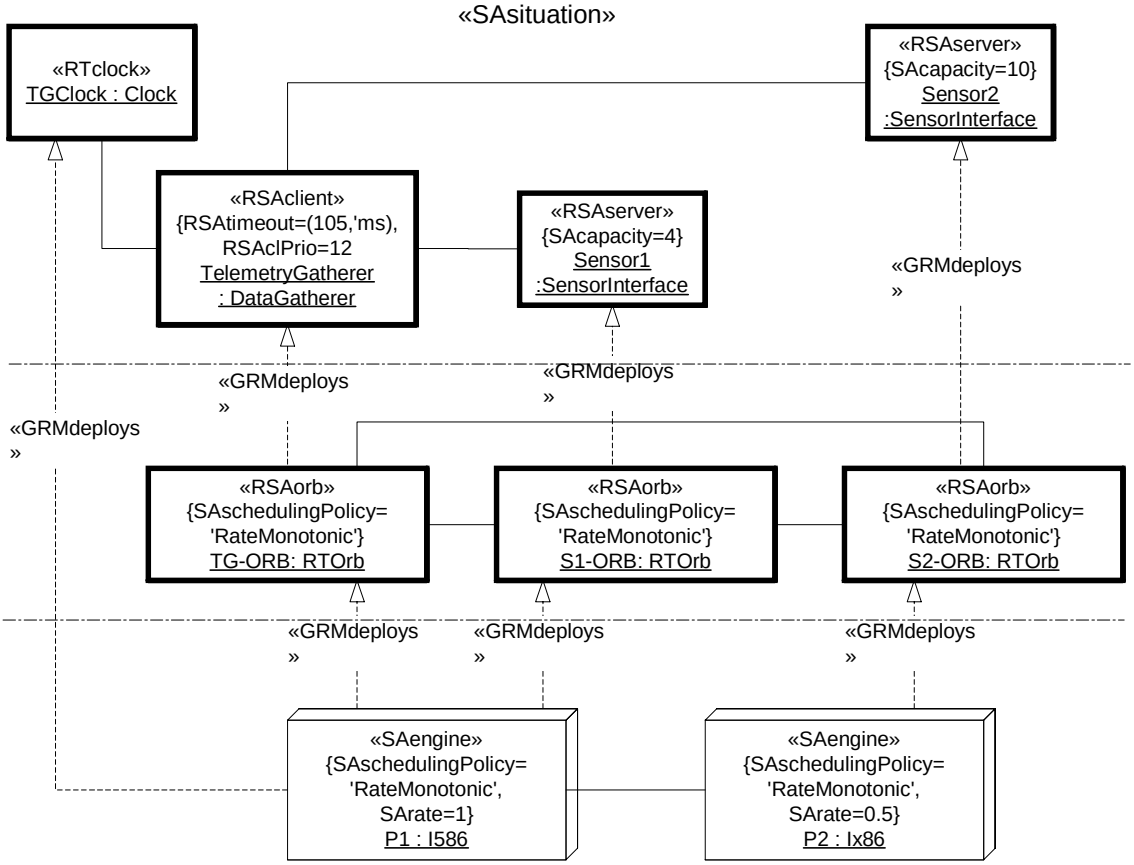


Figure 9-2 Example RT CORBA application

This model shows three distinct layers of the application in one diagram (successive layers are separated by a horizontal intermittent line). The top layer contains the “pure” application. This is a logical model that, except for the stereotype annotations, could be based on any suitable run-time technology, not necessarily RT CORBA. (Note that we could apply a different profile, say a DCOM profile, to this part of the model and its structure would remain the same.)

The next level down is the engineering environment that realizes the application layer. It happens to be based on RT CORBA technology comprising a set of interconnected ORBs. Of course, a different run-time technology would have a different structure and the deployment mappings to the elements of the layer above would also be quite different in that case. Hence, the content and structure of this layer is more technology specific.

At the bottom, we see the actual hardware configuration, which consists of two different kinds of processors, each realizing a portion of the functionality of the RT CORBA layer. In this case, the RT CORBA layer is the logical model and the hardware is the engineering model.

Note that we skipped the operating system level in this example. This is a matter of modeling preference. Such a model could have been interspersed between the RT CORBA layer and the hardware, but that would have made the problem more complex. Such a model would be required in situations where fine-grained detail is relevant to the model analysis.

In our case, we have abstracted away the operating system and absorbed its effects into the QoS characteristics of the RT CORBA layer. Of course, the QoS values at each level are a function of the values of the layer below. For example, the worst case completion time of a server depends on the characteristics of the ORB technology used, which in turn, depends on the QoS characteristics of the hardware. For this example, we have chosen not to show these dependencies. However, in practice, such dependencies must be factored in if a layered model such as this is being analyzed. (In many cases, it is sufficient to analyze a system at only one layer.)

When this occurs, the dependencies need to be factored in. For example, if we define a nominal CPU processing rate using the TVL variable $CPUrate, then the actual timings might be expressed as a function of that rate, for example:

```
SAWorstCase=(1.5 * $CPUrate,'ms')
```

where the value of $CPUrate might be supplied from outside the model.

The application works as follows: a real-time clock, TG Clock, prompts the RT CORBA client, which uses RT CORBA to tale a reading from each of two sensors. Once it has received a reply, it compares the two and makes an appropriate decision (the details of that are not relevant here). The scenario that ensues when the clock triggers the client can be seen in Figure 9-3.

By itself, this application-level diagram may not contain sufficient information to perform model analysis. For instance, we do not know which scheduling policy is being used. However, if we look down into the layer below, we can determine that the ORBs are using a priority inheritance type of policy.

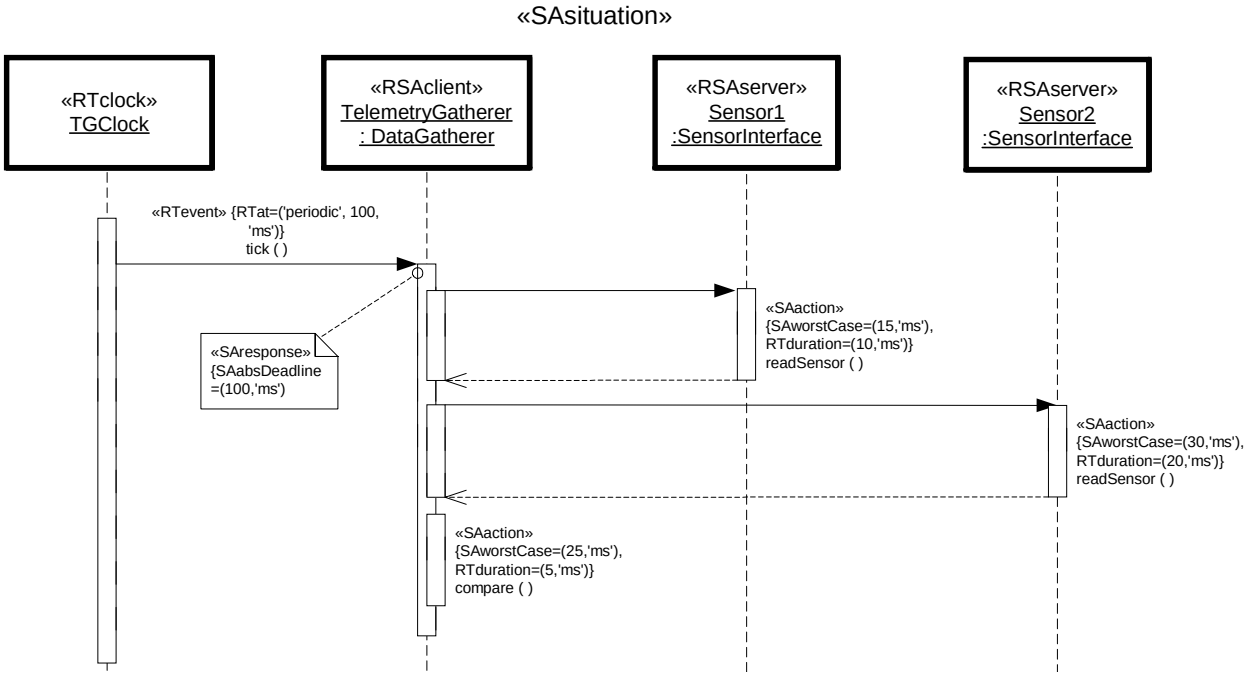


Figure 9-3 The “response” scenario to a clock trigger

A complete model analysis would start at the bottom and work out the QoS characteristics by moving upwards a layer at a time. One way of avoiding this potentially lengthy and complex procedure is to define a “required environment” for a given level using the GRMrequires relationship, as described in “The Requires Mapping” on page 4-31. In that case, model analysis can start proceed on the assumption that the required environment will be met so that the QoS values of this environment can be used as a basis for model analysis of all the layers above. This is particularly useful in the early phases of system design when the details of the engineering technologies in the lower layers are either completely unknown or not known in sufficient detail.

9.2.4 Required Metamodel Changes

This sub-profile assumes that the UML metamodel is modified to support the action execution concept. This is described in detail in Section 4.2.4, “Required UML Metamodel Changes,” on page 4-38. No other metamodel changes are assume or required.

9.2.5 Proposed Notational Extensions

No special notation is proposed for this sub-profile.

Model Processing 10

Model processing is the process by which a UML model is either analyzed by some model analysis technique or supplemented by some synthesis method. Since no specific decomposition of tool functionality is prescribed or assumed, it is necessary to define a standard format for communicating information between tools. In this chapter we describe the mechanisms by which this can be achieved.

10.1 Domain Viewpoint

10.1.1 Use Cases

The following are the basic use cases envisaged for model processing in increasing order of complexity:

- The most elementary use case is a *simple model analysis.* First, the UML model is annotated using the appropriate stereotypes as defined in this document. In some cases, additional parts of the model may need to be defined to represent specific analysis contexts. Note that a given model may be annotated in a number of different ways. Once the annotation is complete, the model is passed from the model editing tool to the model analysis tool where it is analyzed and the results fed back to the editing tool (this allows viewing of the results in the same environment and form in which the model was produced).
- A more complex scenario is one of *manual experimentation*: the model is annotated in such a way that some aspects are left unspecified. The result is a parametrized model and the intent is to experiment with different parameter values until a suitable set of parameter values is discovered.
- The most sophisticated scenario is an automated variant of the manual experimentation approach. However, instead of manually searching for appropriate parameter values, the search is automated on the basis of some optimality criterion. We refer to this as the *synthesis* use case, since the parameter values are derived automatically.

What can be parameterized in a model for the latter two cases. The most obvious candidates are various QoS values. Another possibility is deployment mappings (see Section 4.1.8.3, “Deployment,” on page 4-15). This can serve as a basis for determining an optimal engineering platform for a given logical model.

10.1.2 Domain Concepts

The diagram in Figure 10-1 represents a conceptual model of the general process envisaged for analyzing or synthesizing models. It is important to note that this diagram merely identifies the kinds of functionality and information required for this process and is not intended to prescribe how, or even whether, that functionality is to be partitioned among tools. At one extreme, a single complex tool may realize all of these functions or, at the other extreme, a different tool may realize each one.

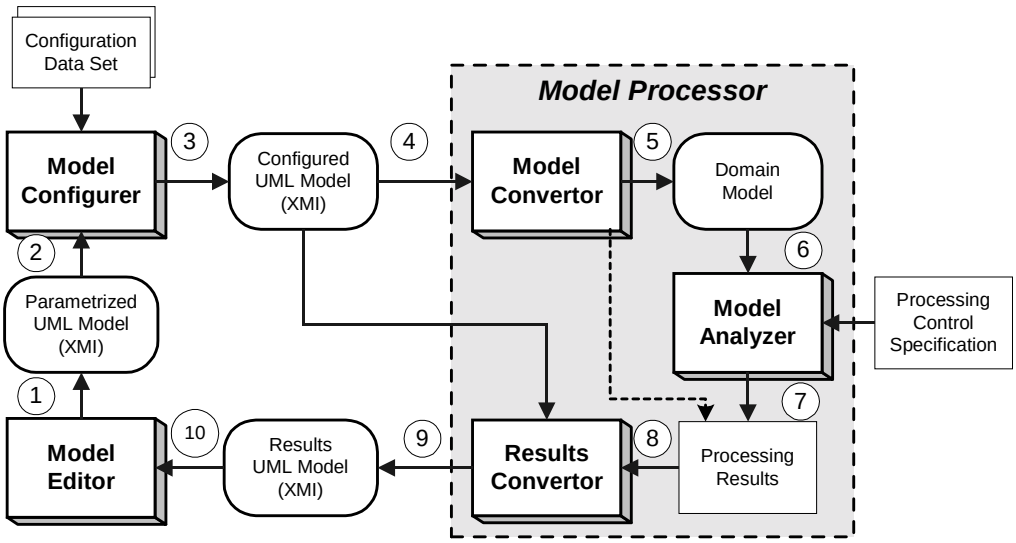


Figure 10-1 General model-processing framework

The arrows in the diagram denote data as well as control flows from one functional box to another. The data exchanged between them consists of models, control data, and results.

The *model editor* function is used to create and modify UML models. In principle, the model editor does not have to be aware of the semantics of the extensions defined in this profile. However, it should be able to enforce the rules and constraints defined in this profile, such as restrictions on which kind of stereotype can be applied to a which kind of UML model element. (This capability should be standard on any tool that supports the UML standard.)

Of course, it may be useful for a UML modeling tool intended specifically for the real-time domain to provide custom support for the profile and its elements. Such a tool could perform domain-specific validations of the model and provide more convenient support for real-time concepts and notations.

As noted above, a model produced by a model editor may be parameterized. This allows a single model to be analyzed for different parameter values without having to produce a new model every time the values change. However, this creates a need for a *model configurer* function, a function that is similar in purpose and form to the C pre-processor. It takes a parameterized model and, by substituting an appropriate set of parameter values, it produces a different model.

In general, it is convenient to package sets of values that go together into distinct configuration data sets. Each set represents the values that are to be applied to one model processing (analysis) run. The analyst simply has to choose the set that is of interest for the model analysis at hand.

The *model processor* function takes in a UML model analyzes it and generates the *analysis results*. We discuss this function in more detail below. A *processing control specification* may be used to control any options in the model analysis (or synthesis) process. Given that different model analysis tools produce different results and given that it is desirable to minimize the amount of expertise of individual tools and model analysis methods that software developers need to have, the results of model analysis must be returned in a format that is understandable to the developer and the model editing tool.

10.1.3 The Model Configurer

The model configurer works simply: it looks for TVL expressions in the value field of tagged values and evaluates them (see Appendix A - *The Tag Value Language* for the specification of TVL). If the expression involves a scalar or array variable, and the value of that variable is provided in the configuration data set that is being used, it substitutes the value for the variable. For instance, a tag value might be expressed as follows:

```
{RTduration = ($serverTO, 'ms')}
```

Assuming that the chosen configuration data set contains the Perl[1] statement

```
$serverTO = 55;
```

the result of the substitution will be

```
{RTduration = (55, 'ms')}
```

Note that TVL allows conditional expressions such as

```
($timeType == 'discrete') ? int($myTimeValue) : $myTimeValue
```

which returns the integer portion of the value of $myTimeValue if the $timeType variable is equal to 'discrete' and the unaltered value otherwise.

1.TVL is a small subset of the standard Perl language.

In fact, configuration data sets represent Perl scripts which can be arbitrarily complex (but will typically consist mostly of simple variable assignment statements). The purpose of these scripts is to compute the desired variable values. The model configurer works as follows:

- it first invokes the Perl interpreter and runs the configuration data set script; this script calculates the necessary values.
- it parses the XMI of the UML model seeking tagged value values that contain TVL expressions (this means that it has to recognize the stereotypes defined in this profile).
- it evaluates such expressions using the Perl interpreter and, if the expression can be computed successfully, it substitutes the result of evaluation (as a string) in place of the expression string in the XMI.
- any value expressions whose values cannot be computed are left unmodified.

Expressions whose values cannot computed by the model configurer are used for one of two purposes:

- they may represent model analysis result variables, whose values are substituted into the model once model analysis completes.
- they may represent configuration values that are to be computed by the model processor prior to analysis; the data required to compute these values may be supplied with the processor control specifications.

The last point means that the model processor may have to contain its own model configurer. However, this configurer is likely to be analysis specific rather than generic. For example, a given time value may be defined by an array that specifies a minimum value, a maximum value, and an average. Depending on the type of model analysis desired, the model processor might choose one based on the needs of the moment, or it may run three separate analyses, one for each value.

Note that the model configurer is only required if configuration data sets are to be used. If not, then it can be omitted altogether and a simpler (but more manual) process can be used instead, as depicted in Figure 10-2.

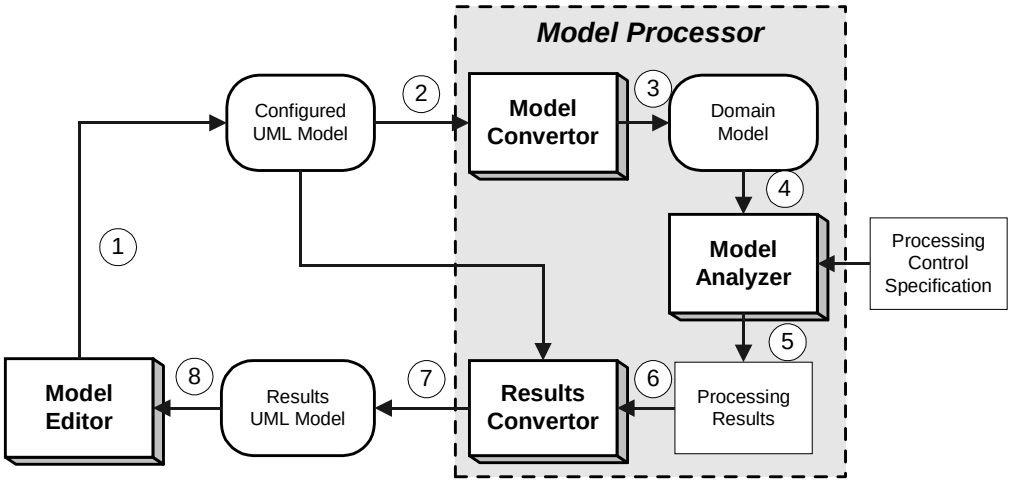


Figure 10-2 A simplified system without a model configurer

10.1.4 The Model Processor

The *model processor* is where the model analysis (or synthesis) is performed. As shown in Figure 10-3 on page 10-6, it is decomposed into units of finer functionality.

10.1.4.1 The Model Convertor

In general, the input function of a model processor is a pre-processor or *model converter.* This function might include a domain- and tool-specific configurer as described above, but its primary function is to convert the offered UML model into a domain-specific model suitable for model analysis. It does this by first detecting the appropriate analysis contexts (see Section 4.1.9.5, “AnalysisContext,” on page 4-17) in the model. Using the elements in these contexts as the roots of its exploration of the UML model, the convertor extracts the information that it needs. It primarily seeks elements that have been stereotyped with stereotypes that it recognizes. Note, however, that the model convertor cannot depend exclusively on stereotypes, but must be at least partially cognizant of the semantics of UML.

For example, let us assume that we are doing a schedulability analysis of some UML model whose fragment is shown in Figure 10-3. Let us say that the exploration of the model has led the convertor to the Ix86Processor Node element which is stereotyped as a kind of «SAEngine» and, hence, relevant to the domain model. From here, it will need to follow any composition links or associations attached to that model element, *even if they are not stereotyped*, to determine if any of them lead to other elements that may be relevant to model analysis. In this way it will detect, the attached SensorData element and, because it is stereotyped as an «SAResources», it will include that in its model as

well. However, upon following a similar link to the display unit it will not include that element in its domain since that element has not been identified as relevant to schedulability analysis.

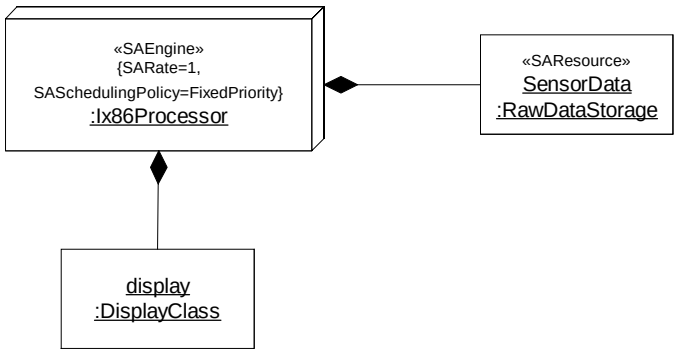


Figure 10-3 An example of links that need to be followed by a model processor

So how will the convertor know which links to follow and which ones to ignore? This is determined by the relationships defined in the domain model. When it finds an element that it knows belongs to the domain model, it needs to look for any relationships that this domain element might have and find their UML equivalents. These mappings are described in the individual model analysis sub-profiles.

For instance, the above composition link between «SAEngine» and «SAResources» maps directly to the composition association (“ownedResource”) between ExecutionEngine and SResource shown in Figure 7-1 on page 7-5.

The model converter is responsible for detecting any inconsistencies of the QoS annotations that it uses. Any such inconsistencies that are detected must be fed back to the UML model editor via the results convertor, possibly bypassing the model analyzer if the inconsistency is critical to model analysis as shown by the dashed line in Figur e10-1.

Once the domain model has been extracted, it is passed on to the *model analyzer.*

10.1.4.2 Model Analyzer

The model analyzer encapsulates the pure model analysis function. It does not require any knowledge of UML. The *processing specification* that may be fed to it identifies specific control information for the *model processor* and is unique to each vendor.

The results of the model analysis, of course, are also independent of UML and are not only analysis specific but also tool specific. For this reason, we need another adaptor function: the *results convertor.*

10.1.4.3 The Results Converter

The purpose of this function is to convert the results of model analysis back into a UML model that can be returned to the model editor where it can be viewed in “source” form by the software developer.

The model analyzer produces three kinds of results data that needs to be fed back to the user. The simplest method is to use the *tagged value variable* notation. In that case, the *results converter* replaces the dependent variable string with the actual value generated by the model analyzer. Producing more complex results that can be expressed within a UML model can be returned in the form of the contents of a note. This may be in XML or HTML, allowing complex results (e.g., graphs, charts) to be displayed directly in the model editor. The final class of "data" for results is to augment the UML with additional synthesized model elements. For example, the results of model analysis may add new elements to the domain model, which have to be converted by the results convertor into corresponding classes, packages, or other UML elements to create a new "results" model.

In the latter case, it is desirable that the inserted elements be identifiable as changes to the existing model. Model differencing or some similar technique would be very helpful. The changes should be reversible so that the original model can be preserved. A classification system for changes could show which changes are "normal" and which are "problem areas."

10.2 UML Viewpoint

10.2.1 UML Extensions

The use of TVL as a method for specifying tagged values is described in Appendix A - The Tag Value Language. Strictly speaking, this is not an extension to UML since it adheres fully to the format supported by the current definition of tagged values.

The only useful extension that is necessary to support the model processing framework described here is a new stereotype of the UML Comment concept to allow tools to recognize a comment that is not simple text, but HTML text instead.

«HTMLtext»

Represents a Comment whose body represents HTML text.

Stereotype	Base Class
«HTMLtext»	Comment

10.2.2 Required Metamodel Changes

No metamodel changes are assumed by the concepts defined in this chapter.

10.2.3 Proposed Notational Extensions

No special UML notation is proposed for the concepts defined in this chapter.

Tag Value Language *A*

This appendix defines a formal language for specifying the value fields of tagged values. While tagged values are often assumed to be simple values there are certain cases where it may be necessary to express such values in a more complex way. For example, it may be required for one tagged value to be related in some way to another. This requires both a way of referencing the value of another tag as well as the ability to use expressions, such as arithmetic expressions.

We propose to base the syntax and semantics of this Tag Value Language (or, TVL) on a simple subset of the Perl language. This has two principal advantages: (1) since Perl is a widely used language, it is likely to be familiar to many users and (2) Perl is supported by a wide range of readily available freeware and shareware tools and utilities.

In this first release, we intentionally restrict TVL to a very small subset of Perl. Once there is enough experience with the use of TVL, we anticipate that additional Perl features will be included.

TVL expressions are used to specify the value part of a tagged value according to the following syntax:

```
{<tag-name> = <TVL-expression>}
```

The expression could be a simple literal, such as a number, or it could be a complex expression that involves variables and arithmetic functions. Whatever the expression, the desired value is produced when the expression is evaluated.

The use of expressions and variables clearly presumes that there is a pre-processor that evaluates the TVL-based tagged values before a model can be analyzed. It also requires a mechanism for supplying the values of independent variables, since TVL itself does not have an assignment operation. However, these additional mechanisms are necessary for any system that allows tagged values to be expressed through variables and are not a consequence of using TVL. The nature of these mechanisms is described in Section 10.1.3, “The Model Configurer,” on page 10-3.

A

A.1 Literals

Naturally, the ability to describe literals is a fundamental capability for a language that serves exclusively to specify values.

A.1.1 Numbers

Numbers are represented, in decimal form only. Both integer and real numbers are allowed, as are both positive and negative numbers. No whitespaces or commas are allowed within numbers, but an underscore can be used to improve readability (that is, the underscore character in the middle of a literal is ignored). Real numbers may be expressed using the scientific notation.

The following are typical examples:

```
12345       #positive integer
–123        #negative integer
1234.56     #positive real
1.2E3       #scientific notation
1_000_000   #represents the value one million
```

The following is an example of a tagged value specified as a numeric literal:

```
{timeout = 60}
```

A.1.2 Booleans

Boolean values are not expressed through literals, but through the values of two predefined global variables **$true** and **$false**. For example:

```
{isPeriodic = $true}
```

A.1.3 Strings

Strings are specified by bracketing a stream of printable characters between single quotes (‘). Any printable character can be included in a string. To include the single quote character itself, the two-character combination of backslash and quote (\’) is used, while a backslash character can be inserted into a string literal using a double backslash combination (\\).

The following are typical examples:

```
‘A simple string’
‘A string with a quote literal (\’) included within.’
‘The backslash-quote combination (\\\’) appearing literally
in a string’
```

There are no predefined upper limits on the size of strings.

A.1.4 Lists

Literals of heterogeneous types can be combined into a list of items between a set of parentheses with individual items separated by commas. There are no predefined limits on the size of lists.

The following are typical examples:

```
(1, 2, 3)      #a simple numerical list
('a', 2.5)     #a heterogeneous list
((1,2,3), ('a', 2.5))#a list of lists (equivalent to the
flattened list)
```

For example, the following is a tagged value whose value is represented by a list:

```
{timeout = (60, 'sec')}
```

Note that this is a more realistic example than the earlier one since it explicitly identifies the physical measure that the number expresses (i.e., time expressed in seconds). Most tagged values, that express physical quantities will likely be expressed using lists rather than simple values.

Ordered lists can be specified as ranges using the "range" operator, which returns a list of values counting by one from the left value to the right value. For instance, the list (1..5) is a shorthand form for the list (1, 2, 3, 4, 5).

Individual items in a list can be accessed by indexing. This is achieved by specifying the index of the desired list element to the right of the list, with the left-most element at index value 0. For instance, the result of evaluating the following expression:

```
('a', 'b', 'c', 'd') [2]
```

is the literal '**c**', which is in index position **2**.

A.2 Variables

There are two kinds of variables in TVL, scalars and arrays. Scalar variables store a single value, whereas arrays store lists of values. Scalar variables begin with the "$" character, whereas lists are identified by the "@" symbol as the first character. The second character must be a letter or underscore, followed by a combination of alphanumeric characters (including underscore characters) up to a maximum of 255 characters. Uppercase and lowercase characters are distinct.

All variable names are global to the UML model in which they appear, although the scalar and array namespaces are distinct (i.e., $A and @A represent different variables).

A variable is declared by its use in the value part of a tagged value; e.g.:

```
{clockRate = ($clock_rate, 'ns')}
```

The variable above may then appear in some other TVL expression, such as

```
{adjustedClockRate = (1.4 * $clock_rate, 'ns')}
```

A.3 Expressions

An expression can be a simple literal or variable, or it can be a compound expression formed by combining expressions through operators. The latter provides a relatively sophisticated capability to express values that are related to each other in possibly very complex ways.

For instance, the following somewhat contrived example shows a complex case where the value of a timeout will depend exponentially on the number of clients configured in the system (**`$clients`**), unless that number is greater than **`6`**, in which case a single maximum value is used:

```
{timeout = (int ( ($clients < 6) ? (0.5 * exp ($clients)):
(0.5 * exp (6))), 'seconds')}
```

A.3.1 Arithmetic Operators

TVL supports the following arithmetic operators, listed in order of decreasing precedence:

```
( )         bracketed expression
**          exponentiation
-           arithmetical negation
*   /       multiplication and division
+   -       addition and subtraction
```

Note that parentheses can be used to change precedence in the usual way. For example, the following expression will evaluate to 1:

```
- (1 + 2)**2 + 10
```

A.3.2 Relational Operators

The relational operators are, in decreasing precedence order:

```
<   >   <=   >=less than, greater than, less than or equal,
greater than or equal
==   !=     equal, not equal
```

The relational operators are all below the arithmetic operators in the operator precedence order.

The result of applying these operators is a Boolean value.

A.3.3 Boolean Operators

The Boolean operators, which are at the bottom of the precedence hierarchy are:

```
not          logical negation
and          logical conjunction
or  xor      logical disjunction and exclusive disjunction
```

Like the relational operators, the result of applying these operators is a Boolean value.

A.3.4 Conditional Operator

This operator works like an if-then-else statement, but, it can be used inside an expression. It has the following format:

<Boolean-expression> ? <if-true-expression> : <if-false-expression>

The result of evaluating this expression will be the result of the evaluation of the **<if-true-expression>** if the **<Boolean-expression>** is true. Otherwise, the result will be the result of the **<if-false-expression>**. For instance:

```
($clock_rate > 5) ? (5) : ($clock_rate)
```

returns either the value of **$clock_rate** or **5**, whichever is smaller.

The conditional operator has a precedence that is below the relational operators, but above the Boolean operators.

A.3.5 Numeric Functions

The following standard numeric functions are pre-defined and can be used in value expressions:

```
abs (arg)   returns the absolute value of the argument
exp (arg)   returns e to the power of the argument
int (arg)   returns the integer portion of the argument
log (arg)   returns the base e logarithm of the argument
sqrt (arg)  returns the square root of the argument
```

A

Model of Real-Time CORBA *B*

In this section we describe a custom model library[1] that defines UML extensions for modeling the internal elements of Real-Time CORBA [35]. This, in turn, is based on the OMG standard UML Profile for CORBA. The overall package structure is shown in Figure B-1.

In the remainder of this appendix, we provide definitions of the contents of these packages. However, we do not go into a detailed description of any of them, since they are all defined in the appropriate OMG CORBA specifications [31], [34], [35].

1.A "model library" is defined in the UML 1.4 standard as *"a stereotyped package that contains model elements which are intended to be reused by other packages. A model library differs from a profile in that a model library does not extend the metamodel using stereotypes and tagged definitions."*

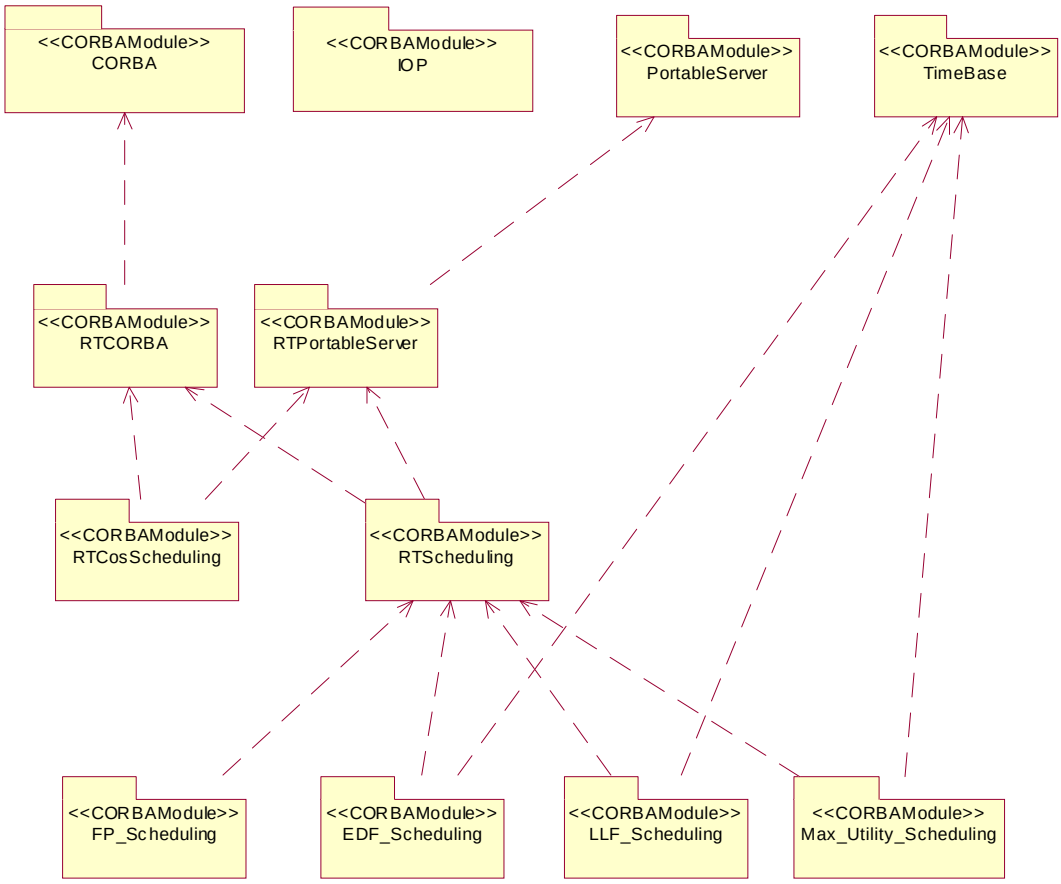


Figure B-1 The CORBA and Real-Time CORBA packages and their dependencies.

The elements of the CORBA and IOP packages are shown in Figure B-2 and Figure B-3 respectively.

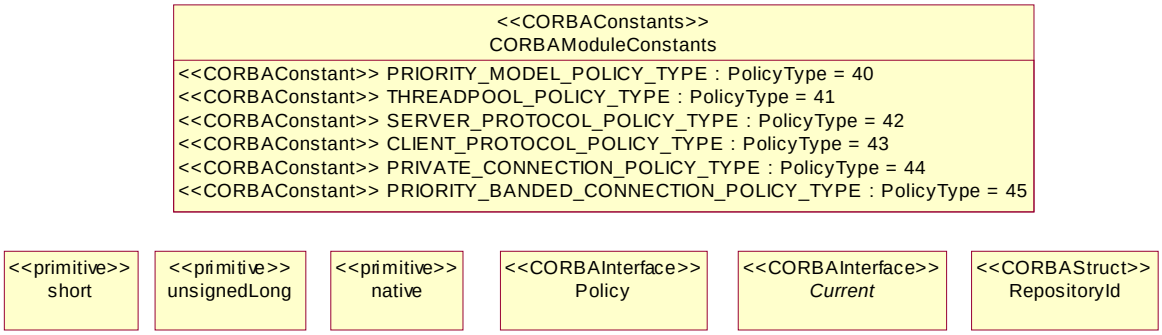


Figure B-2 CORBA module definitions

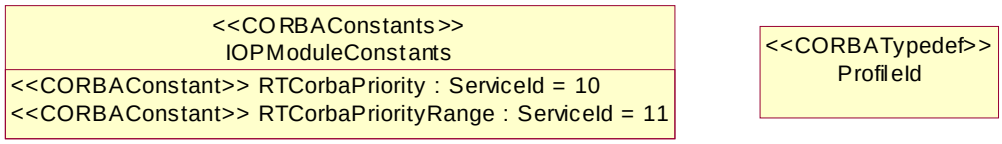


Figure B-3 IOP module definitions

The types and data structures for the RT CORBA module are based on types inherited from the CORBA and IOP modules.

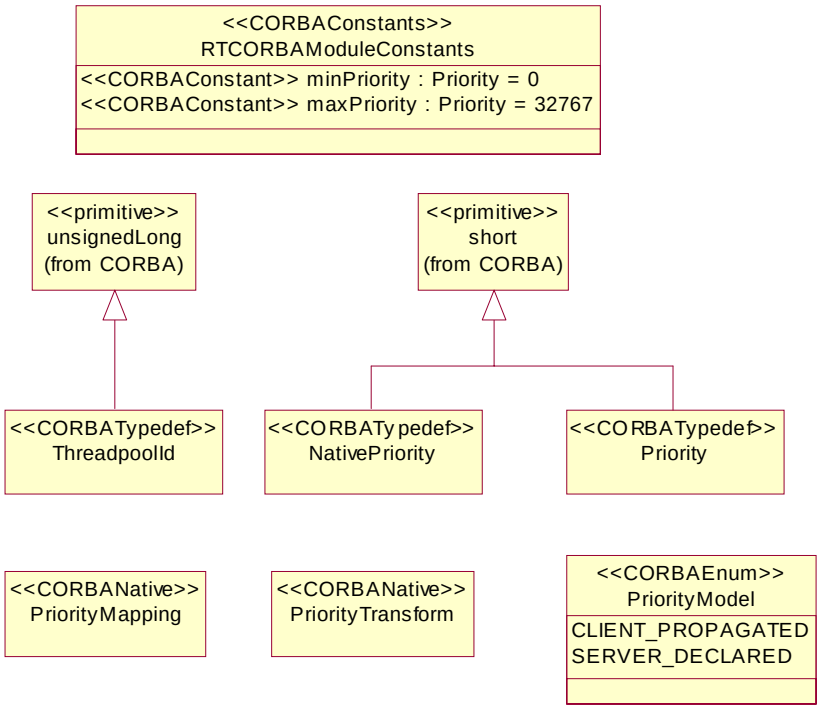


Figure B-4 RTCORBA typedefs and constants

The structures and interfaces associated with threadpools are shown in Figure B-5, and the structures associated with protocols are defined in Figure B-6.

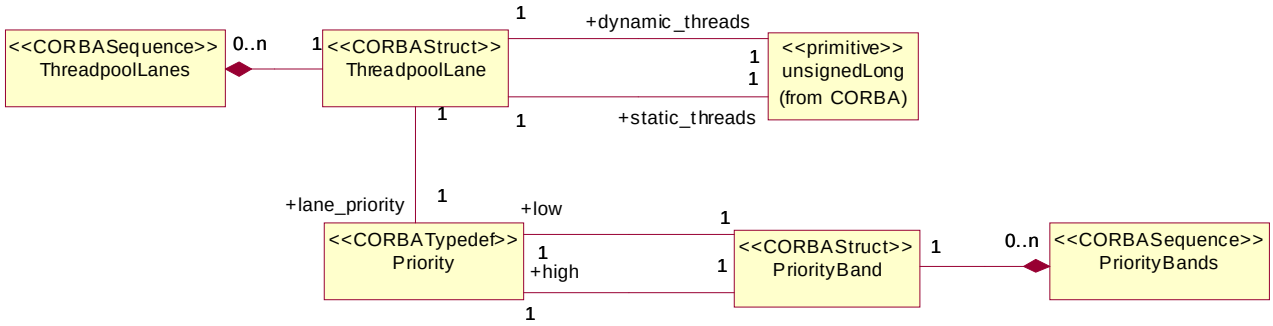


Figure B-5 Structs related to threadpools

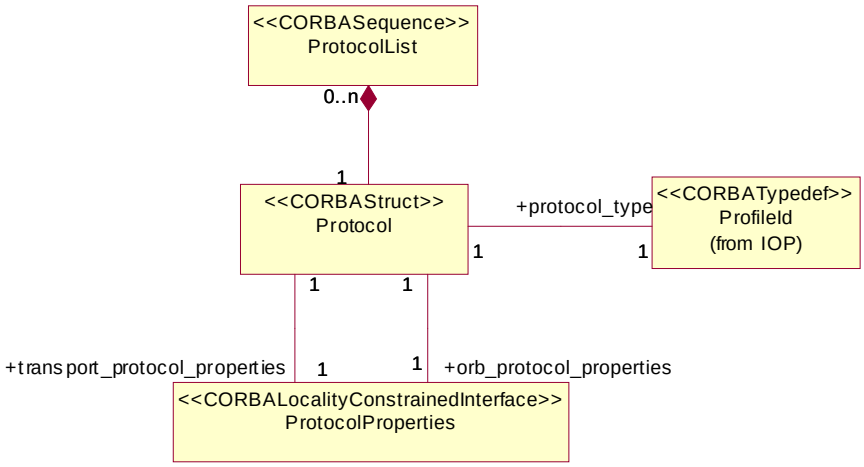


Figure B-6 Structs related to protocols

The primary interface for Real-Time CORBA is the Real-Time ORB, shown in Figure B-7 (note that, to reduce clutter, we have omitted the full set of individual operation parameters since they can be found in the standard definition).

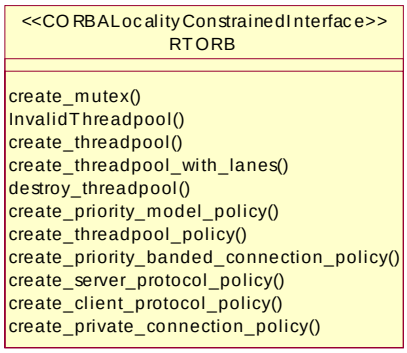


Figure B-7 The Real-Time ORB interface specification

The remaining interfaces are for setting policies of various kinds that apply in RT CORBA (Figure B-8 and Figure B-9).

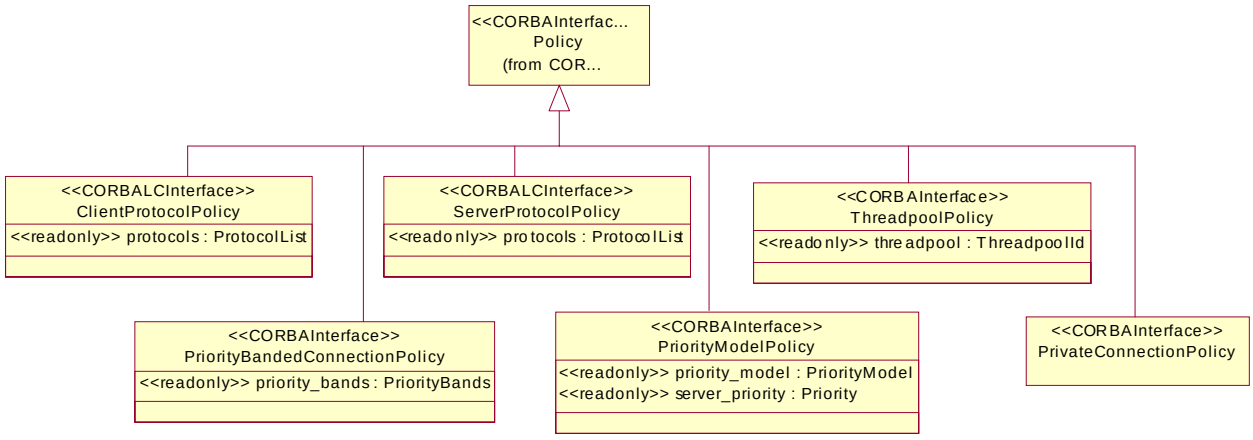


Figure B-8 RTCORBA interfaces for setting policies

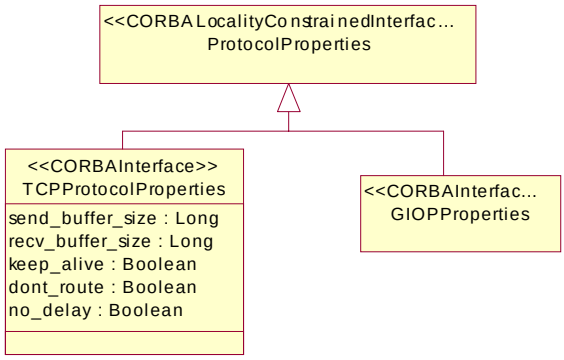


Figure B-9 RTCORBA locality constrained interfaces for policies

Figure B-10, depicts the interfaces for accessing RTCORBA::Current and for using the RTCORBA::Mutex interface.

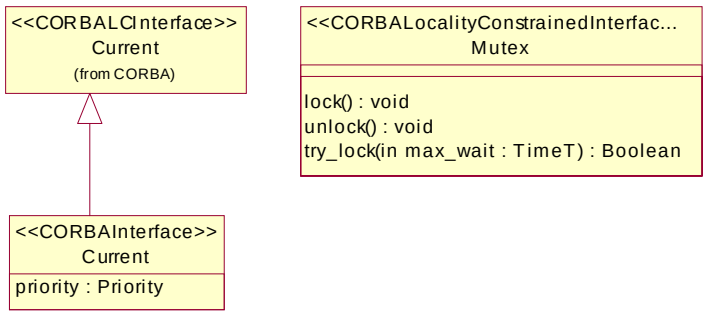


Figure B-10 RTCORBA::Current and RTCORBA::Mutex interfaces

Another part of the RT CORBA interface is the Scheduling Service, shown in Figure B-11.

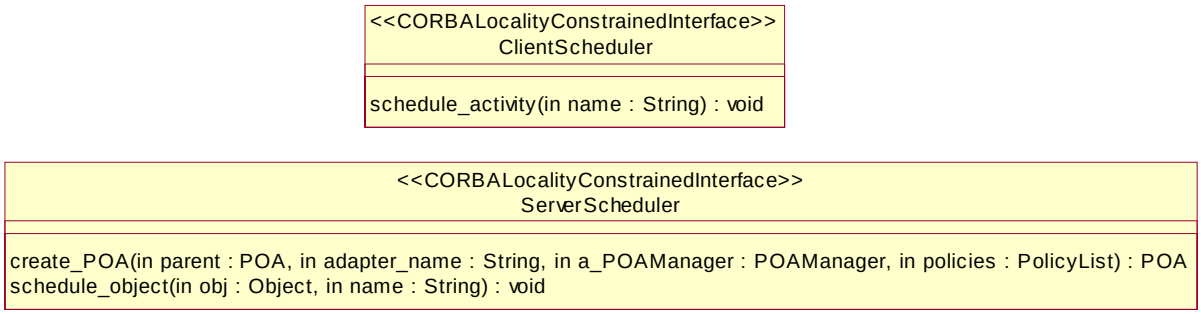


Figure B-11 RTCORBA scheduling service interfaces

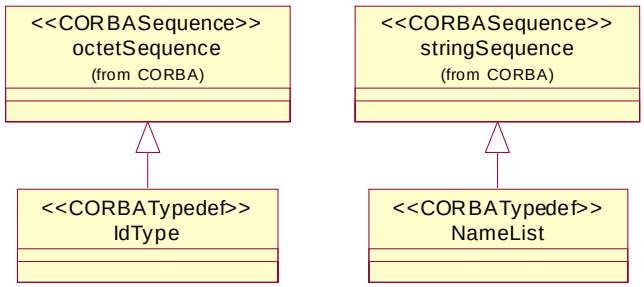


Figure B-12 RTScheduling typedefs

The following four figures describe the interfaces for four well-known scheduling disciplines: fixed priority, earliest deadline first, least laxity first, and maximize accrued utility. These interfaces are used in conjunction with the RTScheduling::Scheduler interface.

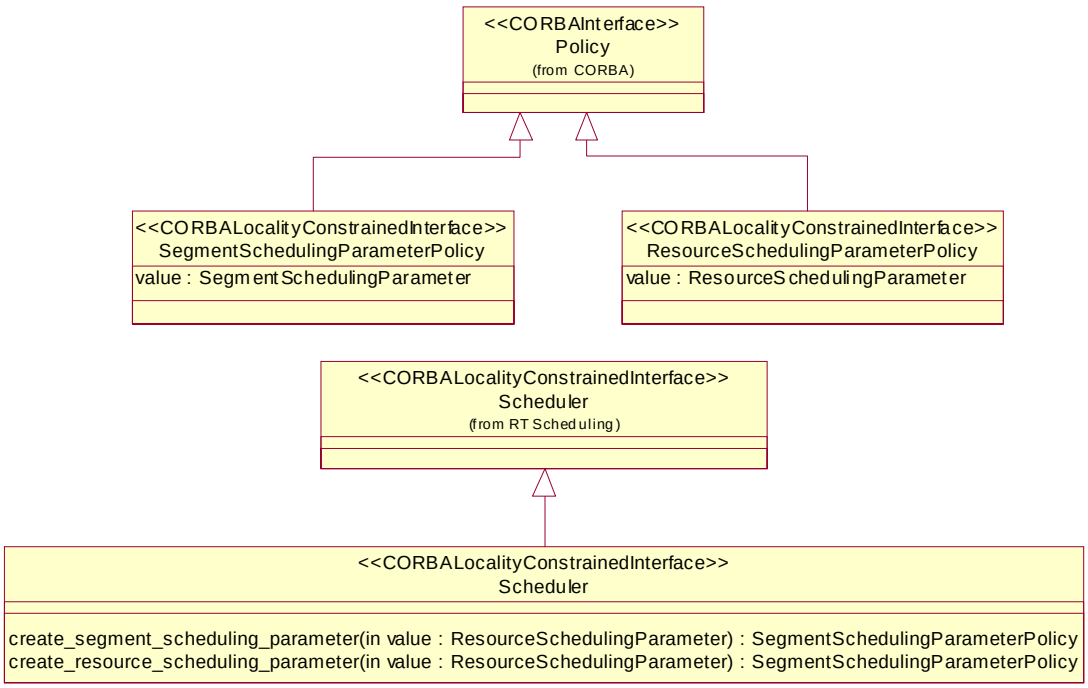


Figure B-13 Fixed priority scheduling structs and interfaces

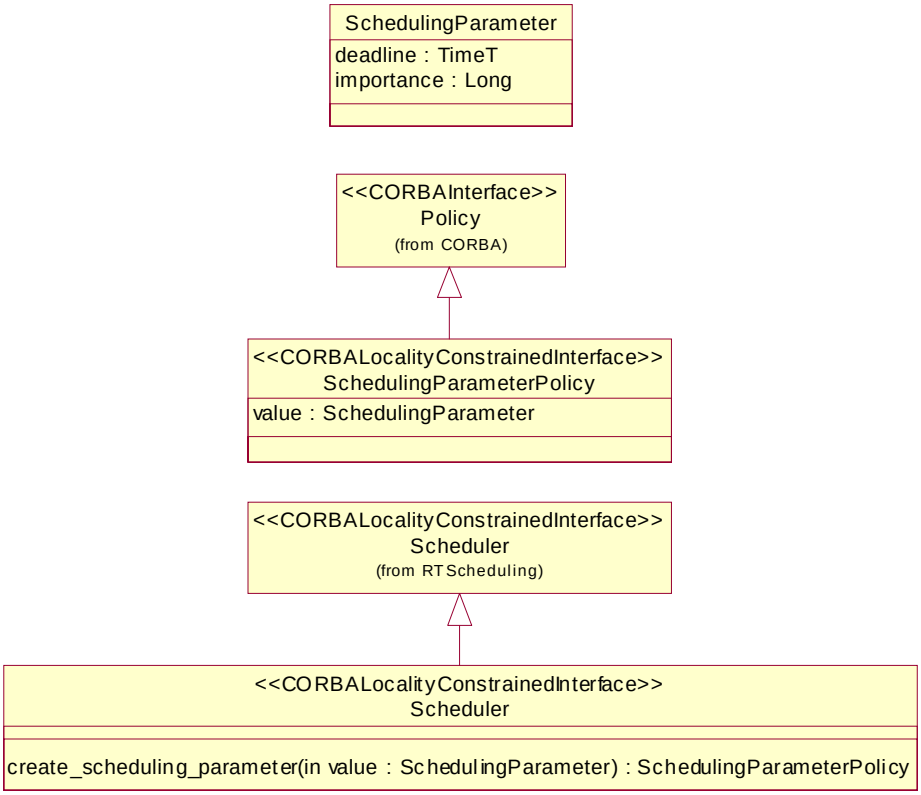


Figure B-14 Earliest deadline first interface

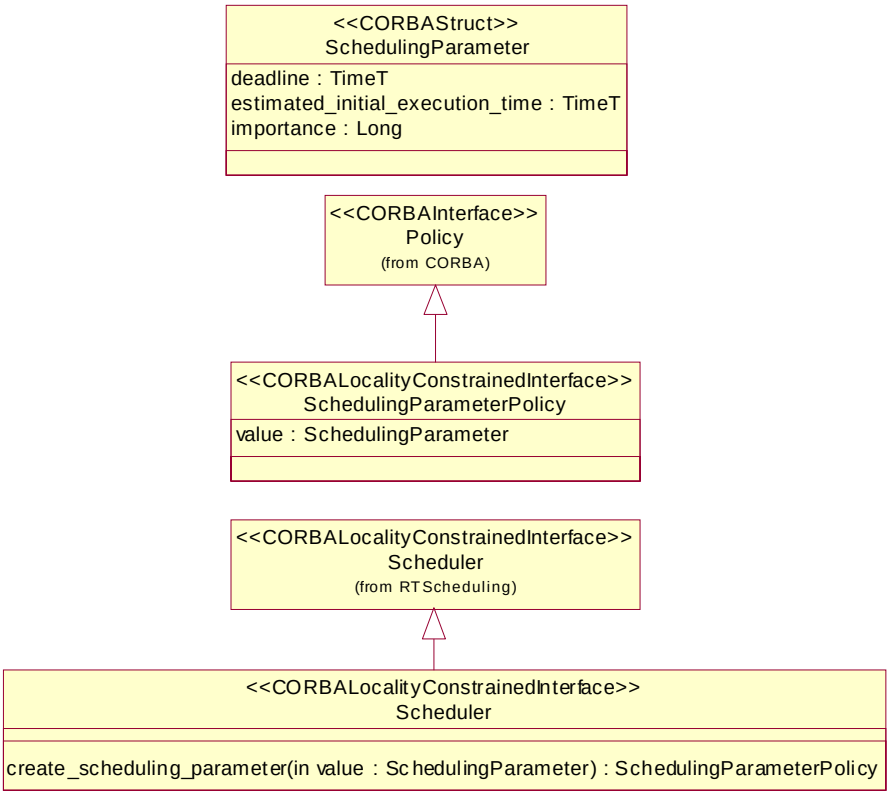


Figure B-15 Least laxity first interface

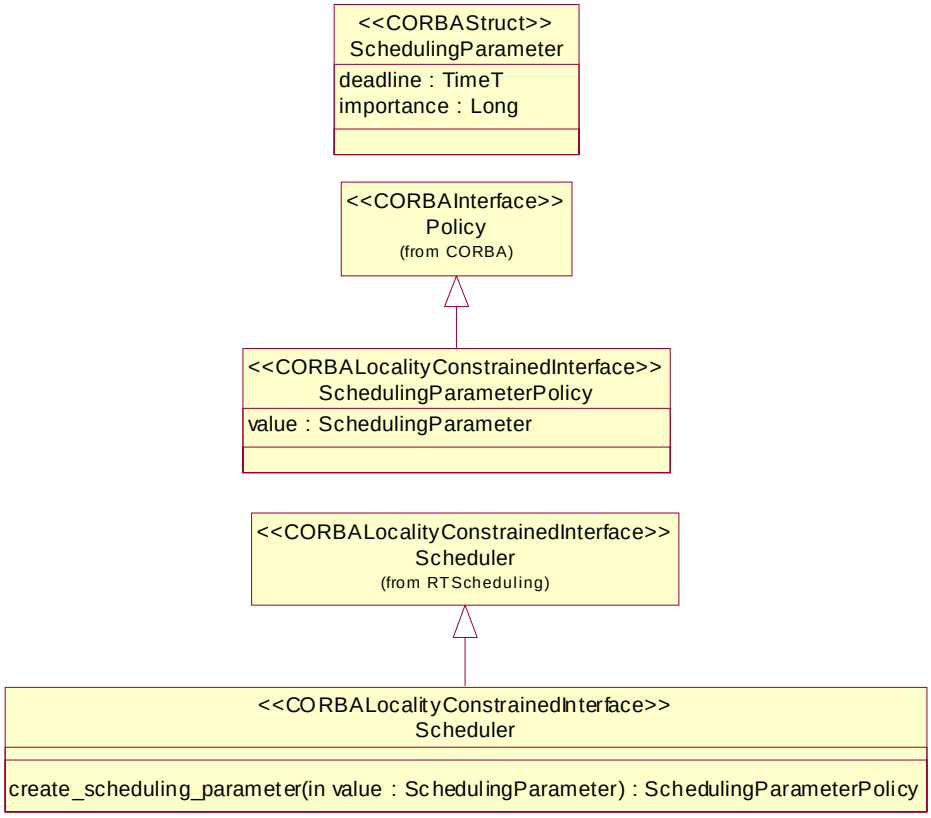


Figure B-16 Maximize accrued utility interface

Figure B-17 depicts the interfaces that implement the dynamic scheduling functions for RTCORBA. RTScheduling::Scheduler creates and sets the scheduling policy for RTScheduling::Current. RTScheduling::ThreadAction has a single virtual operation RTScheduling::ThreadAction::do() that is the entry point for an application. RTScheduling::Current controls the allocation of the distributable thread (RTScheduling::DistributableThread) that is the scheduling unit for RTCORBA. RTScheduling::ResourceManager is an abstract interface for defining a scheduling discipline's resource manager.

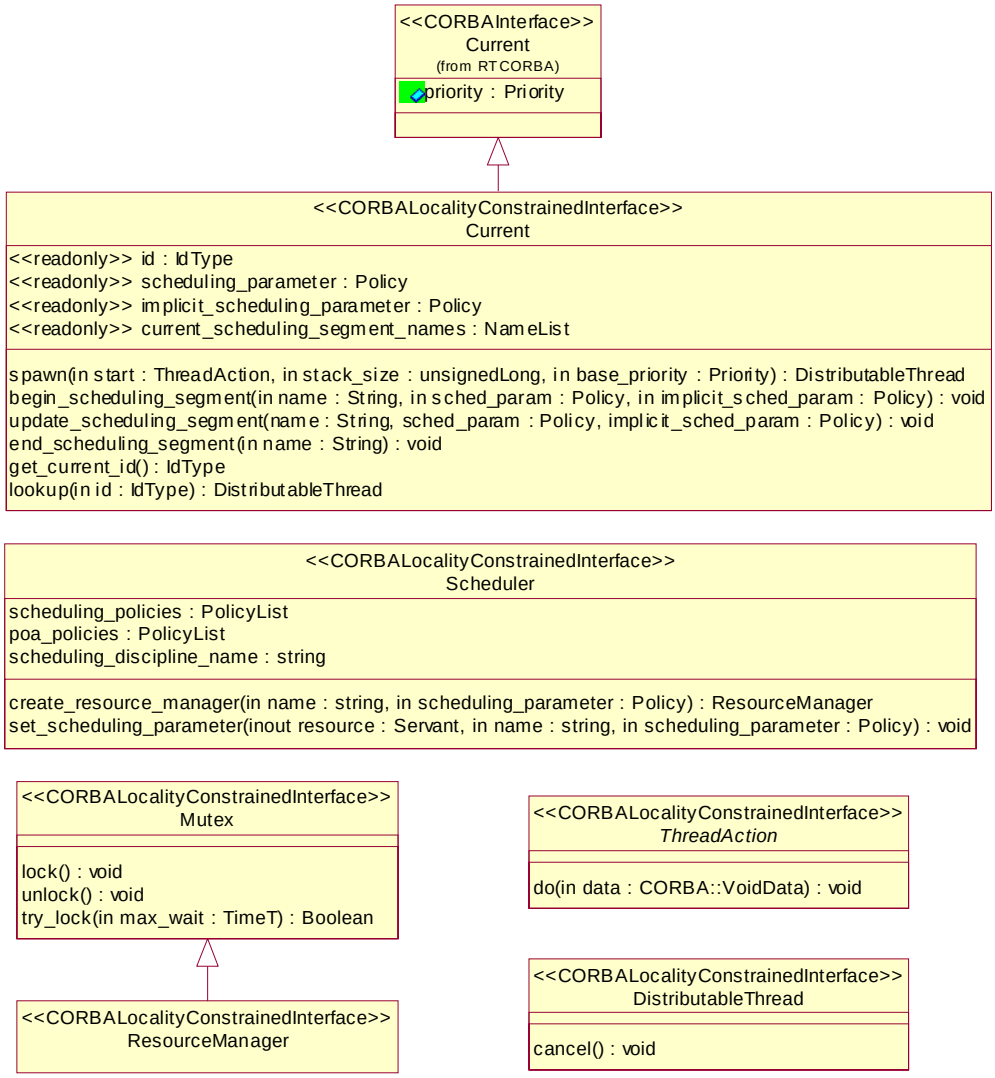


Figure B-17 RTScheduling interfaces (RTScheduling::Current, RTScheduling::Scheduler, RTScheduling::ResourceManager, RTScheduling::ThreadAction, and RTScheduling::DistributableThread)

Bibliography *C*

[1] Awad, M., Kuusela, and J., Ziegler, J., *Object-Oriented Technology for Real-Time Systems*, Prentice-Hall Inc., 1996.

[2] Beckwith, T. and Moore, A., “UML for real-time? Yes, but…,” Embedded Systems Engineering, April/May 1998.

[3] Briand, L. P. and Roy, D. M., *Meeting Deadlines in Hard Real-Time Systems*, IEEE Computer Society Press, 1999.

[4] Burns, A., and A. Wellings, *Real-Time Systems and Programming Languages (2nd ed.)*, Addison-Wesley, 1997.

[5] Cooling, N., and Moore, A., “Real-Time Perspective – Foundation,” Artisan Software White Paper, 1998.

[6] Cortelessa V. and R. Mirandola, “Deriving a Queueing Network based Performance Model from UML Diagrams,” Proc. 2nd International Workshop on Software and Performance (WOSP 2000), ACM, 2000.

[7] Cortelessa V. and R. Mirandola, “UML Based Performance Modeling of Distributed Systems,” Proc. <<UML-2000>> Conference, SpringerVerlag, 2000.

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

[9] Douglass, B., *Real-Time UML, Developing Efficient Objects for Embedded Systems – Second Edition*, Addison Wesley, 2000.

[10] Drake, J., Gonzalez Harbour, M., Guiterrez, J., and Palencia, J., “Modeling and Analysis Suite for Real-Time Applications (MAST), Grupo de Computadores y Tiempo Real, Universidad de Cantabria (internal report), 2000.

[11] Gallmeister, B., *POSIX.4: Programming for the Real World*, O’Reilly and Associates, Inc. 1995.

[12] Gomaa, H., *Designing Concurrent, Real-Time and Distributed Applications with*

UML, Addison Wesley, 2000.

[13] Herzberg, D., "UML-RT as a Candidate for Modeling Embedded Real-Time Systems in the Telecommunication Domain," Proc. 2nd International Conference on the Unified Modeling Language («UML»' 99), Springer (LNCS vol. 1723), 1999 (pp.331-338).

[14] Hoeben, F. , "Using UML Models for Performance Calculation," Proc. 2nd International Workshop on Software and Performance (WOSP 2000), ACM, 2000.

[15] ISO/IEC/ANSI, *Ada 95 Reference Manual: Annex D: Real-Time systems* ISO/IEC/ANSI 8652:1995.

[16] ISO/IEC, *ITU-T X.901 | ISO/IEC 10746-1 ODP Reference Model Part 1: Overview,* ISO/IEC JTC1/SC21/WG7 (SC21 N8926rev), 1995.

[17] ISO/IEC, *CD 15935 – Information Technology: Open Distributed Processing – Reference Model – Quality of Service,* ISO/IEC JTC1/SC7 N1996, 1998.

[18] Kabous, L. and Neber, W., "Modeling Hard Real Time Systems with UML: The OOHARTS Approach," Proc. 2nd International Conference on the Unified Modeling Language («UML»' 99), Springer (LNCS vol. 1723), 1999 (pp.339-355).

[19] Käkhkipuro, P., "UML Based Performance Modeling Framework for Object-Oriented Distributed Systems," Proc. 2nd International Conference on the Unified Modeling Language («UML»' 99), Springer (LNCS vol. 1723), 1999 (pp.356-371).

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

[21] King, P. and R. Pooley, "Using UML to Derive Stochastic Petri Net Models," Proc. 15th UK Performance Engineering Workshop, U. of Bristol, July 1999.

[22] Kopetz, H., *Real-Time Systems: Design Principles for Distributed Embedded Applications,* Kluwer Academic Publishers, 1997.

[23] Lanusse, A., Gerard, S., and Terrier, F., "Real-Time Modeling with UML: The ACCORD Approach," Proc. 1st International Conference on the Unified Modeling Language («UML»' 98), Springer (LNCS vol. 1618), 1998 (pp.319-335).

[24] Liu, J. W. S., *Real-Time Systems,* Prentice-Hall, Inc., 2000.

[25] McLaughlin, M. and Moore, A., "Real-Time Extensions to UML", Dr. Dobbs Journal, December 1998 (pp. 82-93).

[26] de Miguel, M. et al., "UML Extensions for the Specification and Evaluation of Latency Constraints in Architectural Models," Proc. 2nd International Workshop on Software and Performance (WOSP 2000), ACM, 2000.

[27] Moorehead, P., "Grow Real-Time UML Through Innovation," Embedded Systems Development (no 46), October 1998.

[28] Motus, L., and M. G. Rodd, *Timing Analysis of Real-Time Software,* Pergamon Press, 1994

[29] Niehaus, D., Stankovic, J., and Ramamritham, K., "A Real-Time System

Description Language," Proc. Of the 1995 IEEE Real-Time Technology and Applications Symposium, May 1995 (pp.104-115).

[30] Object Management Group, *UML 1.4 with Action Semantics*, OMG document number ad/02-01-09 (January 2002).

[31] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, OMG document number formal/00-10-01 (October 2000).

[32] Object Management Group, *Enhanced View of Time Specification (version 1.1)*, OMG document number formal/02-05-07 (May 2002).

[33] Object Management Group, *Object Time Service*, OMG document number 95-11-8 (December 1995).

[34] Object Management Group, *Real-Time CORBA (version 1.1)*, OMG document number formal/02-08-02 (August 2002).

[35] Object Management Group, *Real-Time CORBA 2.0: Dynamic Scheduling Specification*, OMG document number ptc/01-08-34 (September 2001).

[36] Object Management Group, *RFP for Scheduling, Performance, and Time, OMG* document number ad/99-03-13 (March 1999).

[37] Object Management Group, *Unified Modeling Language Specification v. 1.4*, OMG document number formal/2001-09-67 (September 2001).

[38] Petriu, D. and Y. Sun, "Consistent Behaviour Representation in Activity and Sequence Diagrams," Proc. <<UML-2000>> Conference, Springer Verlag, 2000.

[39] Porres Paltor, I., Lilius, J., *Digital Sound Recorder: A case study on designing embedded systems using the UML notation*, TUCS Technical Report No. 234, Turku Centre for Computer Science, January 1999.

[40] Ramamritham, K., Stankovic, J., and Zhao, W., "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Transactions on Computers (vol. 38 no. 8)*, August 1989 (pp. 1110-1123).

[41] Roubtsova, E., van Katwijk, J., Toetenel, W., Pronk, C., and de Rooij, R., "Specification of Real-Time Systems in UML,", Proc. 1st Workshop on Models for Time-Critical Systems (MTCS 2000), Pennsylvania State University, August 2000 (`http://univaq.it/~mtcs2000/proceedings.html`).

[42] Rumbaugh, J., Jacobson, I., Booch, G., *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[43] Schneider, F., *On Concurrent Programming*, Springer-Verlag, 1997.

[44] Selic, B., "A Quality of Service Framework for Object-Oriented Architectures," International Journal of Software Engineering and Knowledge Engineering (vol.8 No.3), 1998 (pp. 315-331).

[45] Selic, B., "Turning Clockwise: Using UML in the Real-Time Domain," Communications of the ACM (vol. 42, No.10), October 1999 (pp.46-54).

[46] Selic, B., "A Generic Framework for Modeling Resources with UML," IEEE

Computer (vol. 33 no.6), June 2000 (pp.64-69).

[47] Smith, C. and L. Willliams, “Software Performance Engineering for Object-Oriented Systems: A Use-Case Approach,” Technical Report, Software Engineering Services, 1998.

[48] Woodisde, C., “Resource Architectures from Software Design,” Chapter 3 in *Software Resource Architecture and Performance Oriented Patterns* (manuscript in preparation).

[49] Xu, J. and Parnas, D., “On Satisfying Timing Constraints in Hard-Real-Time Systems,” *IEEE Transactions on Software Engineering (vol. 19, no.1)*, January 1993 (pp.70- 84).