By Hassan Gomaa,
Department of Information and Software Engineering,
George Mason University.

Designing Real-Time and Embedded Systems with the COMET/UML method

Most object-oriented analysis and design methods only address the design of sequential systems or omit the important design issues that need to be addressed when designing real-time and distributed applications [Bacon97, Douglas99, Selic94]. It is essential to blend object-oriented concepts with the concepts of concurrent processing [MageeKramer99] in order to successfully design these applications. This paper describes some of the key aspects of the COMET method for designing real-time and embedded systems [Gomaa00], which integrates object-oriented and concurrent processing concepts and uses the UML notation [Booch98, Rumbaugh99]. Examples are given from an Elevator Control System [Gomaa00].

THE COMET METHOD

COMET is a UML based Concurrent Object Modeling and Architectural Design Method for the development of concurrent applications, in particular distributed and real-time applications [Gomaa00]. The COMET Object-Oriented Software Life Cycle is highly iterative.

In the Requirements Modeling phase, a use case model is developed in which the functional requirements of the system are defined in terms of actors and use cases. In the Analysis Modeling phase, static and dynamic models of the system are developed. The static model defines the structural relationships among problem domain classes. Object structuring criteria are used to determine the objects to be considered for the analysis model. A dynamic model is then developed in which the use cases from the requirements model are refined to show the objects that participate in each use case and how they interact with each other. In the dynamic model, state dependent objects are defined using statecharts.

In the Design Modeling phase, an Architectural Design Model is developed. Subsystem structuring criteria are provided to design the overall software architecture. For distributed applications, a component based development approach is taken, in which each subsystem is designed as a distributed self-contained component. The emphasis is on the division of responsibility between clients and servers, including issues concerning the centralization vs. distribution of data and control, and the design of message communication interfaces, including synchronous, asynchronous, brokered, and group communication. Each concurrent subsystem is then designed, in terms of active objects (tasks) and passive objects. Task communication and synchronization interfaces are defined. The performance of real-time designs is estimated using an approach based on rate monotonic analysis [SEI93].

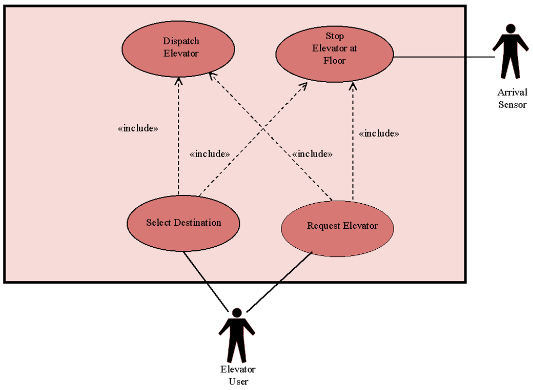

Figure 1. Use case model with abstract use cases.

REQUIREMENTS MODELING WITH UML

In the Requirements Model, the system is considered as a black box. The Use Case Model (Fig. 1) is developed in which the functional requirements of the system are defined in terms of use cases and actors. An actor is very often a human user. In real-time and embedded systems, an actor can also be an external I/O device or a timer. External I/O devices and timer actors are particularly prevalent in real-time embedded systems, where the system interacts with the external environment through sensors and actuators.

ANALYSIS MODELING WITH UML

Static Modeling

For real-time and embedded systems, it is particularly important to understand the interface between the system and the external environment, which referred to as the system context. In UML, the system context may be depicted using either a static model or a collaboration model [Douglass99]. A system context class diagram provides a more detailed view of the system boundary for a real-time system than a use case diagram.

Using the UML notation for the static model, the system context is depicted showing the system as an aggregate class with the stereotype "system", and the external environment is depicted as external classes to

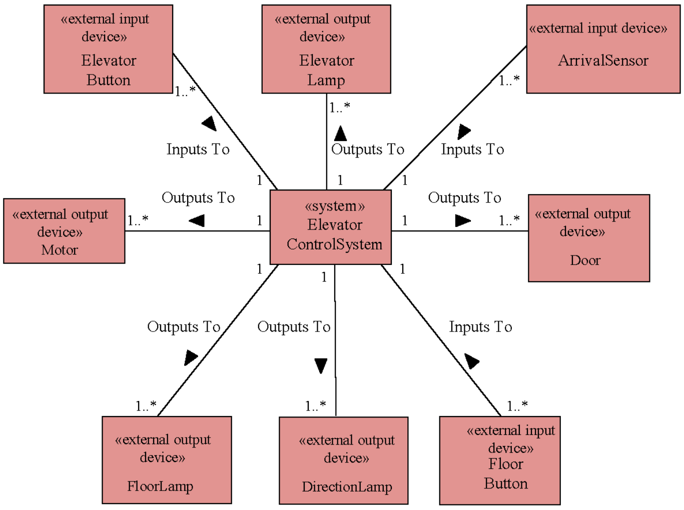

Figure 2. Elevator Control System context class diagram.

which the system has to interface. External classes are categorized using stereotypes. An external class can be an "external input device", an "external output device", an "external I/O device", an "external user", an "external system", or an "external timer". For a real-time system, it is desirable to identify low level external classes that correspond to the physical I/O devices to which the system has to interface. These external classes are depicted with the stereotype "external I/O device". Standard association names are used on system context class diagrams (Fig. 2) as follows:

- "external input device" **inputs to** "system"
- "system" **outputs to** "external output device"
- "external user" **interacts with** "system"
- "external system" **interfaces to** "system"
- "external timer" **awakens** "system"

Dynamic Modeling

For concurrent, distributed, and real-time applications, dynamic modeling is of particular importance. UML does not emphasize consistency checking between multiple views of the various models. During dynamic modeling, it is important to understand how the finite state machine model, depicted using a statechart that is executed by a state dependent control object, relates to the interaction model, which depicts the interaction of this object with other objects.

State Dependent Dynamic Analysis addresses the interaction among objects that participate in state dependent use cases. A state dependent use case has a state dependent control object, which executes a statechart, providing the overall control and sequencing of the use case. The interaction among the objects that participate in the use case is depicted on a collaboration diagram or sequence diagram.

The statechart (Fig. 3) needs to be considered in conjunction with the collaboration diagram. In particular, it is necessary to consider the messages that are received and sent by the control object, which executes the statechart. An input event into the control object on the collaboration diagram must be consis-

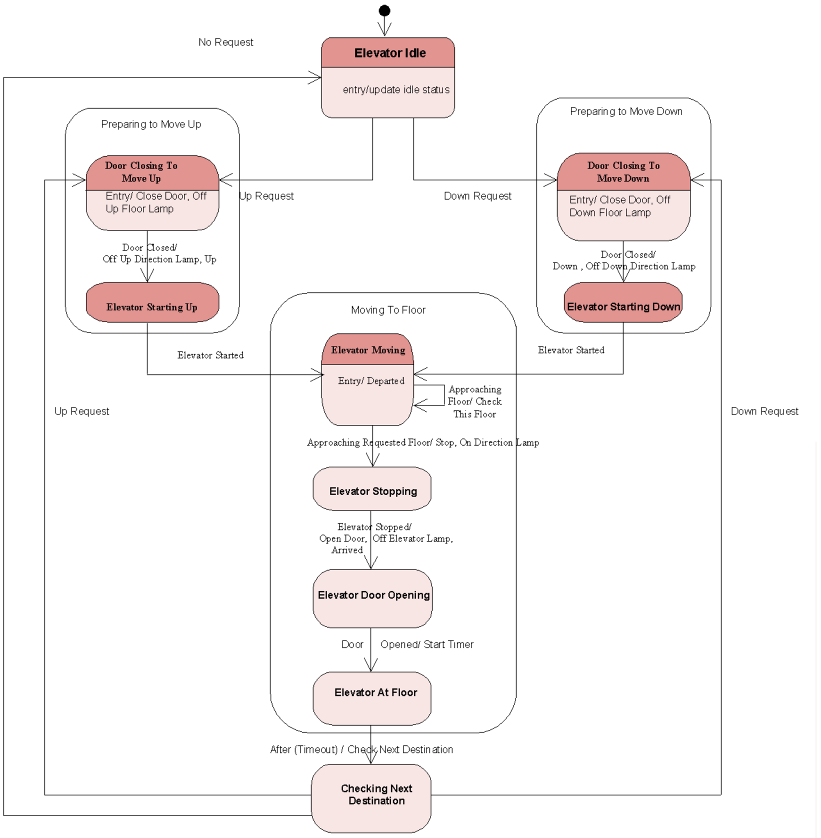
No Request
Elevator Idle
entry/update idle status
Preparing to Move Up
Door Closing To Move Up
Entry/ Close Door, Off Up Floor Lamp
Up Request
Down Request
Preparing to Move Down
Door Closing To Move Down
Entry/ Close Door, Off Down Floor Lamp
Door Closed/ Off Up Direction Lamp, Up
Door Closed/ Down , Off Down Direction Lamp
Elevator Starting Up
Elevator Starting Down
Moving To Floor
Elevator Started
Elevator Started
Elevator Moving
Entry/ Departed
Approaching Floor/ Check This Floor
Up Request
Down Request
Approaching Requested Floor/ Stop, On Direction Lamp
Elevator Stopping
Elevator Stopped/ Open Door, Off Elevator Lamp, Arrived
Elevator Door Opening
Door Opened/ Start Timer
Elevator At Floor
After (Timeout) / Check Next Destination
Checking Next Destination

Figure 3. Hierarchical statechart for Elevator Control.

tent with the same event depicted on the statechart. The output event (which causes an action, enable or disable activity) on the statechart must be consistent with the output event shown on the collaboration diagram.

When the state dependent dynamic analysis has been completed for the main sequence of the use case, the alternative sequences described in the use case need to be considered. For example, alternative branches are needed for error handling.

DESIGN MODELING

Software Architecture

In order to transition from analysis to design, it is necessary to synthesize an initial software design from the analysis carried out so far. In the analysis model, a collaboration diagram is developed for each use case. The consolidated collaboration diagram is a synthesis of all the collaboration diagrams developed to support the use cases. The consolidation performed at this stage is analogous to the robustness analysis performed in other methods [Jacobson92, Rosenberg99]. These other methods use the static model for robustness analysis, whereas COMET emphasizes the dynamic model, as this addresses the message communication interfaces, which is crucial in the design of real-time and distributed applications.

The consolidated collaboration diagram depicts the objects and messages from all the use case based collaboration diagrams. Objects and message interactions that appear on more than one collaboration diagram are only shown once. In the consolidated collaboration diagram, it is necessary to show the messages that are sent as a result of executing the alternative sequences in addition to the main sequence through each use case. The consolidated collaboration diagram is thus intended to be a complete description of all message communication. The consolidated collaboration diagram can get very large for a large system, and it may not be practical to show all the objects on one diagram. One approach to handling the scaleup problem is to develop consolidated collaboration diagrams for each subsystem, and develop a higher-level subsystem collaboration diagram to show the dynamic interactions between subsystems, which depicts the overall software architecture (Fig. 4).

Architectural Design of Distributed Real-Time Systems

Distributed real-time systems execute on geographically distributed nodes supported by a local or wide area network. With COMET, a distributed real-time system is structured into distributed subsystems, where a subsystem is designed as a configurable component and corresponds to a logical node. A subsystem component is defined as a collection of concurrent tasks executing on one logical node. As component subsystems potentially reside on different nodes, all communication between component subsystems must be restricted to message communication. Tasks in different subsystems may communicate with each other using several different types of message communication (Fig. 4) including asynchronous communication, synchronous communication, client/server communication, group communication, brokered communication, and negotiated communication.

Task Structuring

During the task (active object) structuring phase, each subsystem is structured into concurrent tasks and the task interfaces are defined. Task structuring criteria are provided to assist in mapping an object-oriented analysis model of the system to a concurrent tasking architecture (Fig. 5). Following the approach used for object structuring, stereotypes are used to depict the

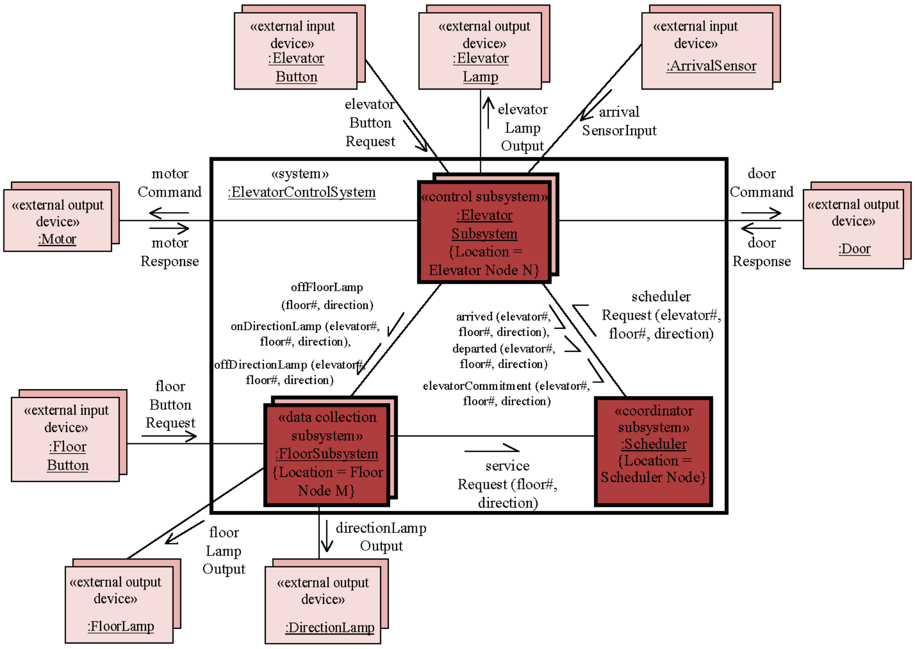
«external input device» :Elevator Button
«external output device» :Elevator Lamp
«external input device» :ArrivalSensor
elevator Button Request
elevator Lamp Output
arrival SensorInput
«external output device» :Motor
motor Command
motor Response
«system» :ElevatorControlSystem
«control subsystem» :Elevator Subsystem {Location = Elevator Node N}
door Command
door Response
«external output device» :Door
offFloorLamp (floor#, direction)
onDirectionLamp (elevator#, floor#, direction),
offDirectionLamp (elevator#, floor#, direction)
arrived (elevator#, floor#, direction),
departed (elevator#, floor#, direction)
elevatorCommitment (elevator#, floor#, direction)
scheduler Request (elevator#, floor#, direction)
«external input device» :Floor Button
floor Button Request
«data collection subsystem» :FloorSubsystem {Location = Floor Node M}
service Request (floor#, direction)
«coordinator subsystem» :Scheduler {Location = Scheduler Node}
floor Lamp Output
directionLamp Output
«external output device» :FloorLamp
«external output device» :DirectionLamp

Figure 4. Example of distributed Elevator Control System.

Copyright 2001 by Dedicated Systems Magazine - 2001 Q1 (http://www.dedicated-systems.com)

AD NATIONAL INSTRUMENTS

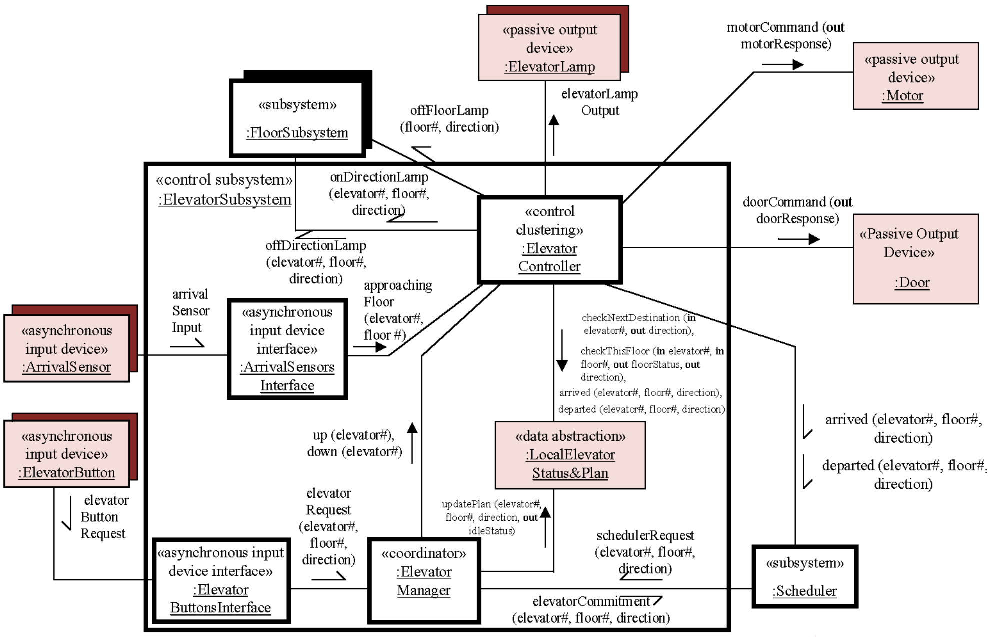
«passive output device»
:ElevatorLamp
motorCommand (out motorResponse)
«passive output device»
:Motor
«subsystem»
:FloorSubsystem
offFloorLamp (floor#, direction)
elevatorLamp Output
«control subsystem»
:ElevatorSubsystem
onDirectionLamp (elevator#, floor#, direction)
«control clustering»
:Elevator Controller
doorCommand (out doorResponse)
«Passive Output Device»
:Door
offDirectionLamp (elevator#, floor#, direction)
arrival Sensor Input
approaching Floor (elevator#, floor #)
«asynchronous input device»
:ArrivalSensor
«asynchronous input device interface»
:ArrivalSensors Interface
checkNextDestination (in elevator#, out direction),
checkThisFloor (in elevator#, in floor#, out floorStatus, out direction),
arrived (elevator#, floor#, direction),
departed (elevator#, floor#, direction)
arrived (elevator#, floor#, direction)
departed (elevator#, floor#, direction)
up (elevator#), down (elevator#)
«data abstraction»
:LocalElevator Status&Plan
«asynchronous input device»
:ElevatorButton
elevator Button Request
elevator Request (elevator#, floor#, direction)
updatePlan (elevator#, floor#, direction, out idleStatus)
schedulerRequest (elevator#, floor#, direction)
«asynchronous input device interface»
:Elevator ButtonsInterface
«coordinator»
:Elevator Manager
elevatorCommitment (elevator#, floor#, direction)
«subsystem»
:Scheduler

Figure 5. Task Architecture of Elevator Subsystem and Task Interfaces.

different kinds of tasks. Stereotypes are also used to depict the different kinds of devices the tasks interface to. During task structuring, if an object in the analysis model is determined to be active, then it is categorized further to show its task characteristics. For example, an active ”I/O device interface” object is considered a task and categorized as one of the following: an ”asynchronous I/O device interface” task, a ”periodic I/O device interface” task, a ”passive I/O device interface” task, or a ”resource monitor” task. Similarly an ”external input device” is classified, depending on its characteristics, into an ”asynchronous input device” or ”passive input device”.

Detailed Software Design

In this step, the internals of composite tasks that containing nested objects are designed, detailed task synchronization issues are addressed, connector classes are designed that encapsulate the details of inter-task communication, and each task’s internal event sequencing logic is defined.

If a passive class is accessed by more than one task, then the class’s operations must synchronize the access to the data it encapsulates. Synchronization is achieved using the mutual exclusion or multiple readers and writers algorithms [Bacon97].

Connector classes encapsulate the details of inter-task communication, such as loosely and tightly coupled message communication. Some concurrent programming languages such as Ada and Java provide mechanisms for inter-task communication and synchronization. Neither of these languages supports loosely coupled message communication. In order to provide this capability, it is necessary to design a Message Queue connector class, which encapsulates a message queue and provides operations to access the queue. A connector is designed using a monitor, which combines the concepts of information hiding and task synchronization [Bacon97, MageeKramer99]. These monitors are used in a single processor or multiprocessor system with shared memory. Connectors may be designed to handle loosely coupled message communication, tightly coupled message communication without reply, and tightly coupled message communication with reply.

PERFORMANCE ANALYSIS OF REAL-TIME DESIGNS

Performance analysis of software designs is particularly important for real-time systems. The consequences of a real-time system failing to meet a deadline can be catastrophic.

The quantitative analysis of a real-time system design allows the early detection of potential performance problems. The analysis is for the software design conceptually executing on a given hardware configuration with a given external workload applied to it. Early detection of potential performance problems allows alternative software designs and hardware configurations to be investigated.

In COMET, performance analysis of software designs is achieved by applying real-time scheduling theory. Real-time scheduling is an approach that is particularly appropriate for hard real time systems that have deadlines that must be met [SEI93]. With this approach, the real time design is analyzed to determine whether it can meet its deadlines.

A second approach for analyzing the performance of a design is to use event sequence analysis and to integrate this with the real-time scheduling theory. Event sequence analysis considers scenarios of task (active object) collaborations and annotates them with the timing parameters for each of the active objects participating in each collaboration (Fig. 6), in addition to system overhead for inter-object communication and context switching. The equivalent period for the active objects in the collaboration is the minimum inter-arrival

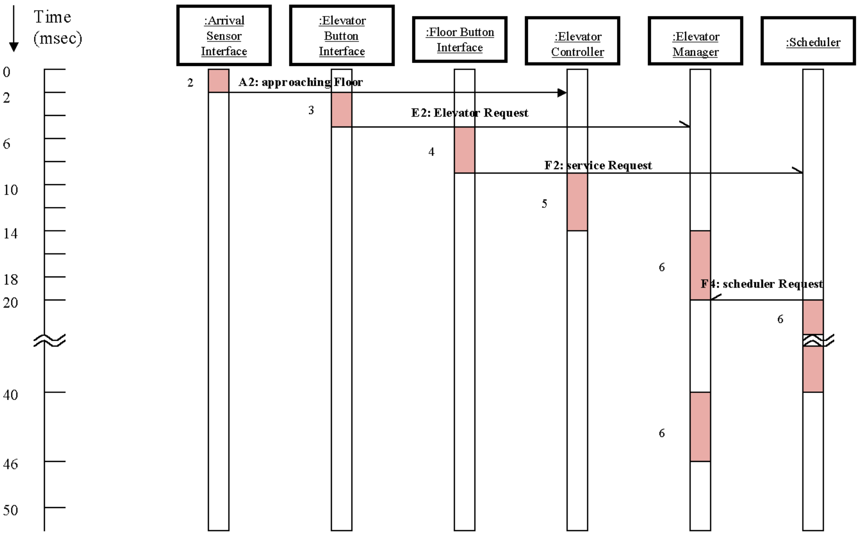

Figure 6. Non-Distributed Elevator Control System, time annotated sequence diagram.

time of the external event that initiates the collaboration.

CONCLUSIONS

When designing real-time and embedded systems, it is essential to blend object-oriented concepts with the concepts of concurrent processing. This paper has described some of the key aspects of the COMET method for designing real-time and embedded systems, which integrates object-oriented and concurrent processing concepts and uses the UML notation ■

Hassan Gomaa, a Professor of Software Engineering at George Mason University in Fairfax, Virginia, is an internationally acknowledged authority on the software design of distributed and real-time systems. His career in software engineering spans both industry and academia. He has developed concurrent, distributed, and real-time applications in industry; designed software development methods and applied them to real-world problems; and taught short courses to professional software engineers around the world. He has a B.Sc. in Electrical Engineering from University College, London, and a Ph.D. in Computer Science from Imperial College, London.

His book, "Software Design Methods for Concurrent and Real-Time Systems", was published by Addison Wesley as part of the Software Engineering Institute Series on Software Engineering, and had its fourth printing in 1999. His new book entitled "Designing Concurrent, Distributed, and Real-Time Applications with UML", was also published by Addison Wesley in August 2000. He is the developer of the DARTS, ADARTS, CODARTS, and COMET software design methods for concurrent, real-time, and distributed applications. He teaches short courses to industry and consults on software design and development projects.

REFERENCES

- [Bacon97] Bacon J., "Concurrent Systems", Second Edition, Addison Wesley, 1997.
- [Booch98] G. Booch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide", Addison Wesley, 1999.
- [Douglass99] B. P. Douglass, "Real-Time UML", Second Edition, Addison Wesley, 1999
- [Gomaa00] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley, 2000.
- [Jacobson92] I. Jacobson, Object-Oriented Software Engineering, Addison Wesley, 1992.
- [MageeKramer99] J. Magee and J. Kramer, "Concurrency: State Models & Java Programs", John Wiley & Sons, 1999.
- [Rosenberg99] D. Rosenberg and K. Scott, "Use Case Driven Object Modeling with UML", Addison Wesley, 1999.
- [Rumbaugh99] J. Rumbaugh, G. Booch, I. Jacobson, "The Unified Modeling Language Reference Manual", Addison Wesley, 1999.
- [SEI93] Carnegie Mellon University Software Engineering Institute, "A Practioner's Handbook for Real-Time Analysis - Guide to Rate Monotonic Analysis for Real-Time Systems", Kluwer Academic Publishers, Boston, 1993.
- [Selic94] B. Selic, G. Gullekson, and P. Ward, "Real-Time Object-Oriented Modeling", Wiley 1994.