Synthesis of Real-Time Implementation from UML-RT Models

Z. Gu and K. G. Shin
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122, USA
zgu@eecs.umich.edu

Abstract

ROOM (Real-Time Object-Oriented Modeling) is an architecture description language widely used in the telecommunications industry to develop embedded software. The concepts of ROOM have been incorporated into the CASE tool Rational Rose Real-Time (RoseRT) in the form of a UML profile, commonly called UML-RT. However, UML-RT itself does not provide any support for performing real-time scheduling analysis and generating an implementation that meets timing constraints. It is the job of the engineer to map the software functional model to RTOS threads and choose a scheduling discipline to satisfy timing constraints. In this paper, we summarize existing approaches for implementation synthesis of UML-RT models, and describe real-time schedulability analysis technique for the native implementation model of the RoseRT CASE tool.

1. Introduction

ROOM (Real-Time Object-Oriented Modeling) [10] is an *architecture description language* widely used in the telecommunications industry to develop embedded software. The concepts of ROOM have been incorporated into the CASE tool Rational Rose Real-Time (RoseRT) in the form of a UML profile, commonly called UML-RT. As shown in Figure 1, UML-RT has the following key concepts:

- A *capsule* is an active object with its own logical thread of control, typically representing an active unit of computation.. A capsule typically has a behavior description in the form of an object-oriented version of Statechart called *ROOMChart*, which differs from conventional Statecharts by removing certain features that are difficult to implement in an object-oriented framework, like instantaneous broadcast of data among parallel/concurrent state machines. Instead, concurrent state machines are modeled as separate capsules communicating with buffered asynchronous message passing. A capsule may contain other capsules to form a structural hierarchy.
- Explicit representation of *ports*, *protocols* and *connectors* enables construction of architectural models from a collection of capsules.
- A runtime framework called *TargetRTS* (Target Runtime System) that serves as a virtual machine supporting the model of computation defined by the UML-RT modeling language. It runs on top of a RTOS to hide the vendor-specific details of execution platform and present a uniform set of APIs to the engineer.

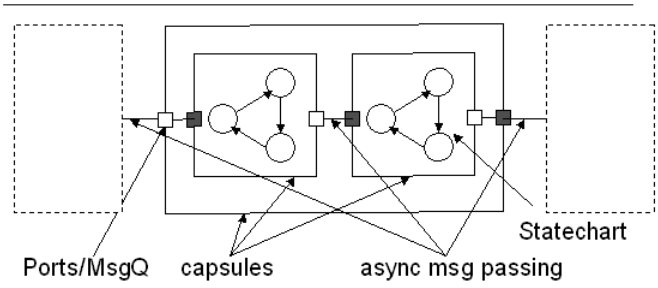

Figure 1. The basic model of computation for UML-RT.

The runtime model of UML-RT follows the *Run-To-Completion* (RTC) semantics for each capsule. Once triggered by a message at its input port, the capsule must execute the triggered action to completion before processing the next message. Messages can be assigned priorities and queued in priority order instead of FIFO order. This means that each capsule is a mutually-exclusive shared resource, and scheduled with the *priority-based non-preemptive* discipline. One or more capsules can be grouped together and assigned to OS threads. Each OS thread processes incom-

ing messages in a non-preemptive manner, consistent with the RTC semantics of capsules assigned to it. However, there can be preemptions between different threads/tasks in a multi-threaded system. (We use the words *thread* and *task* interchangeably in this paper.) A capsule executing in the context of a higher-priority thread can preempt another capsule executing in a lower-priority thread.

The main target application domain of UML-RT is telecommunication systems, which are generally soft real-time in nature. Perhaps due to this reason, the designers of the RoseRT CASE tool have not put much emphasis on real-time issues when implementing a UML-RT model on the target platform. The default execution model is single-threaded, that is, all capsules are mapped into the same thread of execution. Messages are queued and scheduled non-preemptively in priority-order. It is desirable to introduce more parallelism and concurrency into the system to improve predictability by adopting a multi-threaded execution architecture. It is important to distinguish between the concepts of design-level concurrency and implementation-level concurrency [7, 8, 9]. At the design level, each capsule conceptually contains its own logical thread of execution, but each logical thread does not necessarily have to be mapped into an OS thread at the implementation level. Although it is possible for each capsule to have its own OS thread, it may incur too much context-switching overhead if there are a large number of capsules, perhaps thousands or more in a realistic application. A number of alternatives have been proposed for mapping a UML-RT design model into a multi-threaded executable. In this paper, we discuss these alternatives, and our own approach to schedulability analysis of the implementation model of the RoseRT CASE tool. Note that the interaction style of active objects communicating through asynchronous message passing is very prevalent in real-time software, for example, the Quantum Framework [6] advocates this programming style without using expensive CASE tools. It makes a lot of sense in terms of good software engineering principles such as modularity, encapsulation, decoupling of interactions, etc. Therefore, the issues discussed in this paper has much wider applicability than just the RoseRT CASE tool.

2. Implementation Alternatives for UML-RT Models

Suppose we have a logical UML-RT model as shown in Figure 2, consisting of three capsules O_1, O_2, O_3 and two application scenarios t_1, t_2. Scenario t_1 is initially triggered by a periodic timeout message with period 10m that triggers an action t_{11} in capsule O_1, which in turn sends a message to capsule O_2 and triggers action t_{12} in O_2. Finally,

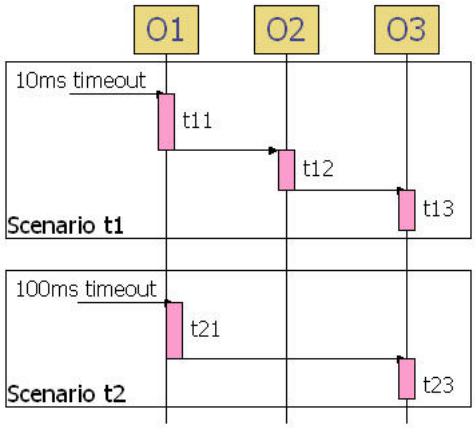

Figure 2. An example application scenario. The system consists of three objects O_1, O_2 and O_3, and two end-to-end scenarios t_1 and t_2. Each scenario consists of multiple subtasks, which are actions triggered at the objects. Scenario t_1 consists of subtasks t_{11}, t_{12} and t_{13}, and scenario t_2 consists of subtasks t_{21} and t_{23}.

O_2 sends a message to O_3 and triggers action t_{13}. We can view this scenario as a logical end-to-end task t_1 consisting of three precedence-constrained subtasks t_{11}, t_{12} and t_{13}. Similarly, the scenario t_2 is an end-to-end task consisting of two subtasks t_{21} and t_{23} triggered by a 100ms periodic timeout message. There are multiple ways of implementing this model on a multi-tasking RTOS, as discussed in the following sections.

2.1. Capsule-Based Multi-threading, Capsule-Based Priority-Assignment

This is the default implementation approach of RoseRT CASE tool. As shown in Figure 3, one or more capsules are grouped into threads with fixed priorities. Two extreme cases are mapping all capsules into a single thread, or mapping each capsule into its own thread. We call this approach *Capsule-based Multi-threading, Capsule-based Priority-assignment* (CMCP). In addition to application threads, additional *system threads* may be used to implement framework services such as a periodic timer. We do not consider system threads in this paper.

2.2. Capsule-Based Multi-threading, Scenario-Based Priority-Assignment

This is proposed by Saksena in [7]. As shown in Figure 4, similar to the RoseRT CASE tool, one or more capsules are

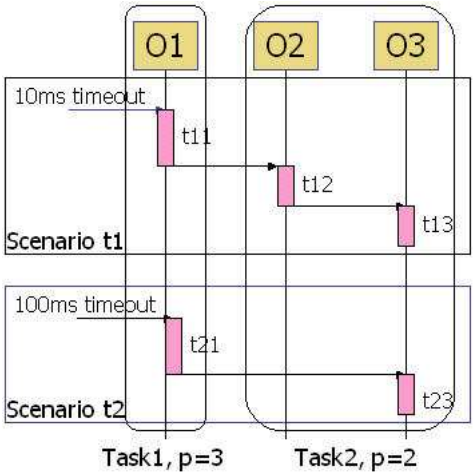

Figure 3. *Capsule-Based Multi-threading, Capsule-Based Priority-Assignment*(CMCP), as implemented in the RoseRT CASE tool. One or more capsules are grouped into a thread with uniform priority. The figure only shows one of many possibilities for grouping capsules into threads. The classic rate monotonic analysis technique [4] is not applicable, and the algorithm described in this paper is needed to perform schedulability analysis.

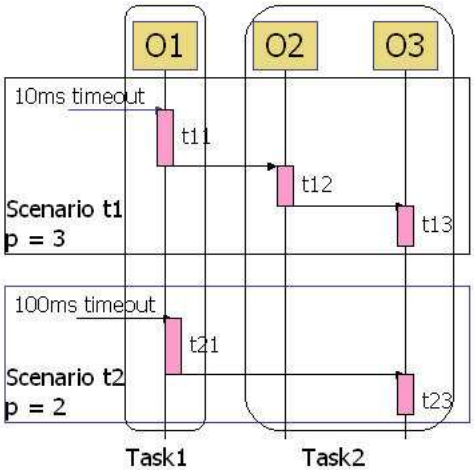

Figure 4. *Capsule-Based Multi-threading, Scenario-Based Priority-Assignment* (CMSP), proposed by Saksena [7]. One or more capsules are grouped into the same thread. Thread priority is adjusted dynamically to maintain a uniform priority across each application scenario.

grouped together into threads. However, priorities are associated with the end-to-end scenarios, and the task priorities are adjusted dynamically to maintain the scenario priorities. We call this approach *Capsule-based Multi-threading, Scenario-based Priority-assignment* (CMSP).

2.3. Scenario-Based Multi-Threading, Scenario-Based Priority-Assignment

This is proposed by Saehwa Kim in [5]. As shown in Figure 5, each application scenario is mapped into a thread. Priorities are associated with the end-to-end threads, with statically-assigned priorities. We call this approach *Scenario-based Multi-threading, Scenario-based Priority-assignment* (SMSP).

2.4. Discussions

Saksena's and Kim's approaches associate priorities with application scenarios instead of capsules, which may be more intuitive from the perspective of the application. However, depending on application characteristics, it may be appropriate to adopt different implementation alternatives. If there is very little interaction between different application scenarios, then Scenario-Based Multi-Threading is appropriate. This is the case for Avionics Mission Computing software discussed in [2]. However, if there is intensive interaction among different scenarios, then Capsule-Based Multi-Threading is more appropriate in order to avoid excessive locking and unlocking of shared capsules.

Saksena's approach requires the engineer to stick to a programming discipline of dynamically adjusting capsule priorities to reflect the priority of the currently executing end-to-end transaction. This approach hurts the encapsulation of capsules by mixing system-level concerns (scenarios) with component-level concerns (capsules). It also involves runtime system-call overheads that may or may not be acceptable to certain resource-constrained embedded systems. Certain small RTOSes may not even provide APIs to dynamically change thread priorities. Kim's approach involves modifying the RoseRT runtime kernel *TargetRTS* to have scenario-based instead of capsule-based multi-threading. It creates shared data when multiple scenarios cut through the same capsule, and necessitates error-prone concurrency control mechanisms, such as mutex, semaphore and monitor. This breaks a key advantage of UML-RT, which is to use buffered asynchronous message passing as the main communication mechanism among capsules instead of shared data in order to minimize the need for concurrency control. Note that even in the native UML-RT model, there are *passive objects*, protected by mutexes, that are used to encapsulate shared data in addition to the capsules/active objects. The number of such passive objects

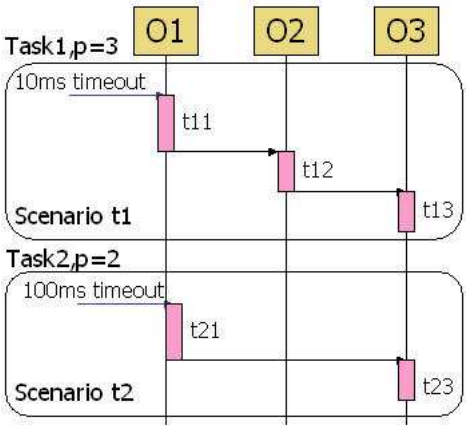

Figure 5. *Scenario-Based Multi-Threading, Scenario-Based Priority-Assignment* (SMSP), proposed by S. Kim [5]. Each application scenario is mapped into a separate thread with uniform priority. This eliminates the need for dynamic priority adjustments, but creates shared data and necessitates error-prone concurrency control mechanisms.

should be minimized in relation to the number of capsules.

Even if Saksena's or Kim's approach were widely adopted, there would still be a lot of legacy applications that are not likely to be changed. Therefore, instead of modifying the computational model of UML-RT design tools to fit real-time scheduling theory, we believe a better alternative is to adapt real-time scheduling theory to fit the computational model of UML-RT design tools. Specifically, we describe a schedulability analysis technique tailored for the computational model of *Capsule-Based Multi-Threading, Capsule-based Priority-Assignment* (CMCP) shown in Figure 3. This technique can be used as a subroutine when assigning priorities to capsules in order to achieve schedulability, or to meet other design objectives while guaranteeing schedulability.

For the CMCP aproach, the number of threads needs to be carefully managed in order to strike a balance between context-switching overheads due to a large number of threads and blocking time due to insufficient parallelism. We do not deal with the issue of grouping capsules into threads or assigning priorities to threads in this paper; rather, we focus on the problem of schedulability analysis given a set of capsule-to-thread groupings and priority assignments.

3. Schedulability Analysis Techniques for Capsule-Based Multi-Threading, Capsule-Based Priority-Assignment

The task model of CMCP is very similar to the end-to-end tasks with subtasks with varying priority as described by Harbour, Klein, Lehoczky [3]. We call the schedulability analysis algorithm introduced in [3] the HKL algorithm. Due to space limitations, we do not provide the full details of HKL algorithm, and refer the interested readers to [3] for details. We only describe our adaptation of HKL algorithm to fit the computational model of UML-RT by taking into account extra blocking time caused by RTC semantics and shared data objects.

Consider a UML-RT model consisting of m capsules or active objects $O_1, O_2, \ldots, O_m$, and n end-to-end scenarios or transactions, where each scenario is mapped into an end-to-end *virtual thread*, forming the taskset $\tau_1, \tau_2, \ldots, \tau_n$. Each end-to-end virtual thread $\tau_i, i = 1, \ldots, n$ cuts through one or more capsules, and triggers an action within each capsule, forming a chain of subtasks $\tau_{i1}, \ldots, \tau_{im(i)}$. We use $O(\tau_{ij})$ to denote the capsule that the subtask τ_{ij} belongs to, and $O(\tau_{ij})$ to denote the set of passive objects that τ_{ij} accesses. Each subtask τ_{ij} is actually an event-triggered action within a capsule $O(\tau_{ij})$. We use the word *virtual thread* because each transaction actually consists of multiple segments of event/action pairs distributed over different operating system threads. Due to run-to-completion semantics, a subtask may suffer a blocking time equal to the largest execution time of other subtasks sharing the same capsule. A capsule may also be involved in multiple sub-tasks within one end-to-end virtual thread. Each subtask τ_{ij} is characterized by a set of parameters (C_{ij}, D_{ij}, P_{ij}), where

- C_{ij} is the worst-case execution time.
- D_{ij} is deadline of τ_{ij} relative to the arrival time of task τ_i, taking 0 to be its arrival time.
- P_{ij} is the fixed priority level of τ_{ij}, equal to the priority level of the capsule that τ_{ij} belongs to.

In order to adapt the HKL algorithm to the UML-RT model, we need to take into account additional blocking time $B(i)$ caused by the RTC semantics of capsules and mutually exclusive access to passive objects, as shown in Equation (1). We can add this term in the HKL analysis equations to derive the Worst-Case Response Time (WCRT) for each end-to-end task, and compare it against the deadline (typically the same as task period) to determine schedulability.

4. The Elevator Control Application Example

We use the elevator control system as an application example, taken from [1][1]. Figure 6 shows the 8 capsules and 1

$$B(i) = \sum_{k,l,j,k!=i,P_{kl}<P_{ij},O(\tau_{kl})=O(\tau_{ij})} C_{kl} + \sum_{m,n,j,m!=i,P_{mn}<P_{ij},PO(\tau_{mn})\cap PO(\tau_{ij})\neq\phi} C_{mn} \quad (1)$$

data object involved in a single-processor implementation. According to the CMCP approach, each capsule is assigned a fixed priority. There are three end-to-end scenarios consisting of subtasks of varying priorities:

1. **Stop Elevator at Floor**. The elevator is equipped with arrival sensors that trigger an interrupt to the capsule *arrival sensors interface* when the elevator approaches a floor, which in turn sends a message *approaching floor* to the capsule *elevator controller*. The *elevator controller* invokes a synchronous method call on the passive data object *elevator status and plan* object to determine whether the elevator should stop or not. We do not model method invocations to passive data objects as separate subtasks, since the passive object inherits the thread and priority from the invoking capsule, and can be viewed as an extension of the invoking capsule. But we do need to take into account blocking time caused by sharing of passive objects by multiple threads.
2. **Select Destination**. The user presses a button in the elevator to choose his/her destination, which triggers an interrupt to the capsule *elevator buttons interface*, which in turn sends a message *elevator request* to the capsule *elevator manager*. The *elevator manager* receives the message and records destination in the passive object *elevator status and plan*, which is a shared object protected by the priority ceiling protocol, and causes blocking time to the higher priority subtask.
3. **Request Elevator**. The user presses the up or down button at a floor, which triggers an interrupt to the capsule *floor buttons interface*, which in turn sends a message *service request* to the capsule *scheduler*. The capsule *scheduler* receives message and interrogates the passive object *elevator status and plan* to determine if an elevator is on its way to this floor. If not, the *scheduler* selects an elevator and sends a message *elevator request* to the capsule *elevator manager*. The rest of the sequence is identical to the **select destination** scenario.

Consider a building with 10 floors and 3 elevators. All end-to-end tasks are interrupt driven, not periodic. In order to perform schedulability analysis, we estimate the worst-case arrival rate of the interrupts and use them as approximations for periods assigned to each task. For example, the **Request Elevator** scenario is assigned a period of 200 ms by assuming that all 18 floor buttons (up and down buttons for each floor, except the top and bottom floors) are pressed within 3.6 seconds, which is likely to be the worst-case arrival rate.

We can calculate Worst-Case Response Times (WCRT) using the schedulability analysis algorithm in Section 3 for the end-to-end tasks, as shown in the WCRT column of Table 1. Note that we associate the WCRT of the end-to-end task with the last segment of the task in the table. No deadlines are missed, and the system is schedulable.

5. Conclusions and Future Work

The RoseRT CASE tool provides a code generator that generates functional code in C++ from UML-RT models, but does not take into account timing and scheduling issues. Previous work has proposed to modify the runtime model of UML-RT to make the classic Rate Monotonic Analysis [4] applicable. Instead of modifying the runtime model of UML-RT to fit the schedulability analysis algorithm, we modify the schedulability analysis algorithm to fit the native runtime model of UML-RT, i.e., *Capsule-Based Multi-threading*, *Capsule-Based Priority-Assignment* (CMCP). This should make our approach more acceptable to industry than previous work in the literature. However, depending on application characteristics, it may be appropriate to adopt different implementation alternatives.

We believe our work bridges the gap between a logical UML-RT model and its real-time implementation on the target platform by giving the engineer algorithms and tools for assessing real-time properties of different ways of grouping capsules into threads. It focuses on the nonfunctional/real-time aspect of implementation synthesis, and is complementary to the existing code generator from UML-RT into C++ code, which focuses on the functional aspect of implementation synthesis. As part of our future work, we plan to implement the algorithm discussed in this paper, either as a plug-in to RoseRT, or as a stand-alone tool that exchanges data with RoseRT through the XMI interface.

One limitation of our approach is that it can only handle linear task-chains, but not more general task-trees or task-graphs. It is an open research issue as to how to extend the HKL algorithm to deal with task-trees or graphs.

This paper has considered the problem of schedulability analysis given a system configuration of capsule-to-thread

1 Note that the analysis technique described in [1] is not entirely accurate, since it does not take into account precedence relationship among subtasks. Also the original example is not based on UML-RT, but the concepts are similar enough to be viewed as a UML-RT model.

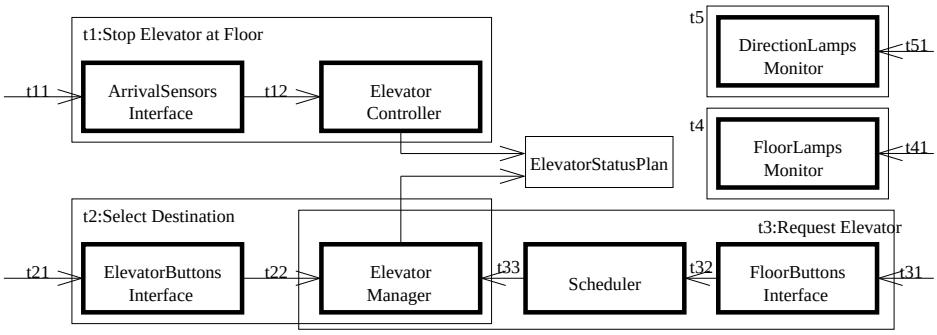

Figure 6. The event sequence diagram for the single-processor elevator control system.

Task	Period	WCET	Priority	WCRT
t_1**: Stop elevator at floor**				
t_{11}: Arrival Sensors Interface	50	2	9	-
t_{12}: Elevator Controller	50	5	6	15
t_2**: Select Destination**				
t_{21}: Elevator Buttons Interface	100	3	8	-
t_{22}: Elevator Manager	100	6	5	22
t_3**: Request Elevator**				
t_{31}: Floor Buttons Interface	200	4	7	-
t_{32}: Scheduler	200	20	4	-
t_{33}: Elevator Manager	200	6	4	46
t_4**,** t_5**: Other Tasks**				
t_{41}: Floor Lamps Monitor	500	5	3	58
t_{51}: Direction Lamps Monitor	500	5	2	63

Table 1. The taskset of the single-processor elevator control system. Not shown in the table, is blocking time caused by the shared object *ElevatorStatusPlan*, which we assume to be 1ms. Also note that it is a common practice to assign a higher priority to the interrupt handler task [4], that is, the *Interface* subtasks here, in order to avoid losing any interrupts.

grouping and thread priority assignment, but it is still an open issue as to how to arrive at such a configuration. Exhaustive search is not feasible in general because the size of design space grows exponentially with the number of capsules or priorities. There may be some guidelines to follow, such as "assign higher priority to interrupt service routines to avoid losing interrupts". We plan to investigate applicability of optimization techniques such as branch-and-bound, simulated annealing and genetic algorithms to exploration of the design space in order to achieve a close-to-optimal design in terms of objectives such as minimized number of threads or minimized response time for critical application scenarios. The schedulability analysis algorithm discussed in this paper can be used as a subroutine during the design space exploration process for priority assignment and implementation synthesis of UML-RT models.

References

[1] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object Oriented Modeling*. Addison Wesley, 1994.

[2] M. Saksena and P. Karvelas, "Designing for schedulability: integrating schedulability analysis with object-oriented design," in *Proc. IEEE Euro-Micro Conference on Real-Time Systems*, 2000, pp. 101–108.

[3] M. Saksena, P. Karvelas, and Y. Wang, "Automatic synthesis of multi-tasking implementations from real-time object-oriented models," in *Proc. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 200, pp. 360 – 367.

[4] M. Saksena, A. Ptak, P. Freedman, and P. Rodziewicz, "Schedulability analysis for automated implementations of real-time object-oriented models," in *Proc. IEEE Real-Time Systems Symposium*, 1998, pp. 92 – 102.

[5] (2004) The Quantum Framework website. [Online]. Available: http://www.quantum-leaps.com/qf.htm

[6] M. H. Klein, T. Ralya, B. Pollak, and R. Obenza, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[7] J. Masse, S. Kim, and S. Hong, "Tool set implementation for scenario-based multithreading of uml-rt models and experimental validation," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003, pp. 70–77.

[8] Z. Gu, S. Kodase, S. Wang, and K. G. Shin, "A model-based approach to system-level dependency and real-time analysis of embedded software," in *Proc. IEEE Real-Time Technology and Applications Symposium*, 2003, pp. 78–85.

[9] M. Harbour, M. H. Klein, and J. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," *IEEE Trans. Software Eng.*, vol. 20, no. 2, pp. 13–28, 1994.

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