SCHEDULABILITY ANALYSIS AND AUTOMATED IMPLEMENTATION OF REAL-TIME OBJECT-ORIENTED DESIGN MODELS

PANAGIOTA KARVELAS

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

MAY 2000

0-612-54334-X

Abstract

Schedulability Analysis and Automated Implementation of Real-Time Object-Oriented Design Models

Panagiota Karvelas

There is a growing interest in adopting object technologies for the development of real-time systems. Several commercial tools, currently available, provide object-oriented modeling and design support for real-time systems. While these products provide many useful facilities, such as visualization tools and automatic code generation, they are all weak in addressing the central characteristic of real-time system design, i.e., providing support for a designer to reason about timeliness properties.

We believe an approach that integrates the advancements in both object modeling and design methods, and real-time scheduling theory is the key to successful use of object technology for real-time software. We propose a methodology based on this idea for uni-processor multi-threaded environments. Specifically, given an application design model and end-to-end timing requirements, we synthesize a feasible implementation model using a built-in schedulability analysis tool. The synthesis process is supported by automatic code generation that can take the application design model and the synthesized implementation model and generate code for the target platform.

In this thesis, I have designed and implemented some of the key components to support this methodology. First, I have developed a schedulability test that determines whether a particular implementation model satisfies the real-time requirements of an application. This can be used during the automatic synthesis process. Second, I have developed an initial implementation supporting automatic code generation, which takes textual specifications of the application design model and a synthesized implementation model, and automatically generates executable code for it.

Acknowledgments

First, I'd like to thank my supervisor, Manas Saksena, for helping me a great deal on my thesis, always keeping his door open to his students (including me) no matter what his workload and just for being an extraordinary teacher. Of course, thanks to Yun Wang and Alex Nikolaev for all their insightful ideas during our research group meetings. It was always an interesting and draining experience ;)

Also, I'd like to thank Paul Di Marco for his constant support, his exceptional proof-reading skills and his way of making me forget all the cares in the world :)
Angus Graham, for his comical side that provided a good balance of work and play in the lab (most of the time) and often made me cry from the laughter.
My parents and brother Nick, for having to deal with my mood swings (or should I say this to everybody), for taking care of me unconditionally and for leaving the house whenever I needed to work in peace and quiet.
My friends and family, for forgiving me whenever I over-neglected them because I was sitting in the lab doing work.
And the Concordia Bubble Bath Dragon Boat team, for letting me take out my frustrations during practices. You guys are the **BEST** !!!

Finally, to the CS people of the 9th floor, I'd like to thank you for enhancing the working environment by being kind and friendly people.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time systems are differentiated from other types of systems by the timing requirements associated with some or all of their computations. As a result, validating such systems requires that these additional timing constraints also be satisfied. This verification is especially necessary for *hard* real-time systems, where fatal situations may occur if any timing constraint is not met.

Typically, designers of real-time systems have dealt with these timeliness properties by using their intuitive engineering skills to design such a system and then, by substantiating their design through system simulation. While this method produces the desired effect after possibly several iterations, it greatly relies on the abilities of the designer and unnecessarily consumes an elaborate amount of time and effort.

To eliminate these shortcomings, we have proposed a methodology [SKW00] that further supports the design of real-time systems. For simplicity, we have fully developed the design strategy for hard real-time systems consisting of a single processor that may contain one or more threads. This design methodology addresses predictability issues in the early stages of design and releases the designer from making decisions that could impact the feasibility of the system. It also eliminates many tedious and costly fine-tuning steps by supporting automation wherever possible.

This thesis focusses on the development of some essential components to support this methodology. Specifically, I design and implement a schedulability test that determines whether a particular complete system specification satisfies the timeliness requirements of the application. Also, I develop an initial implementation that supports automatic code generation given a complete system specification.

In this introductory chapter, I present the motivation behind our research, a detailed description of the proposed design strategy, my contributions to this work as well as the organization of the rest of the thesis.

1.1 Motivation

Most often, real-time systems are embedded in larger applications, where they interact with the physical world. Examples include devices we rely upon and use on a daily basis, such as cellular phones, microwaves and security systems, as well as others we may only read about, such as air traffic control systems and robot arms.

The interaction with the physical world can be quite complex to model, especially since events can occur at any time and in any order. As was stated in [SS99], this non-deterministic environment behavior induces the need to synchronize concurrent activities and unfortunately, concurrency conflicts with the inherently serial, cause-and-effect flow of human reasoning. Designers of real-time systems must also deal with issues of dependability, including correct and continuous performance of these systems. Moreover, the design of such systems must be "readable" so that these systems can be maintained and extended with ease and low cost.

Thus, it is easily observed that designing real-time systems is a daunting and expensive task. However, steps have been taken towards the incorporation of more effective technologies in real-time software development. Of these, the more fruitful have been the efforts to re-orient object-oriented technologies towards the modeling and design of real-time systems.

Object-oriented analysis and design models offer modeling abstractions that are closer to the problem space, thus facilitating the design process and promoting a better understanding of the design, especially when used in conjunction with visualizing tools. This approach has even greater benefits if the design models can be automatically translated into an implementation for a desired target platform.

A number of commercially-available real-time object-oriented development environments have already made their mark. Examples of such tools include Rational Rose Real-Time from Rational Software[1] and ObjecTime[2], ObjecTime Developer from ObjecTime,

[1]Rational Software is located at http://www.rational.com/
[2]ObjecTime is located at http://www.objectime.com/

Rhapsody from iLogix[3], ObjectGeode from Verilog[4], Real-Time Studio from Artisan Software[5] and BridgePoint from Project Technologies[6]. However, despite their claim of being "real-time" software development tools, they are all weak in addressing the central characteristic of real-time system design, i.e., providing support to reason about timeliness properties.

This inability to reason about timeliness significantly restricts the usefulness of these tools for resource-constrained systems with stringent timing requirements. Various implementation choices impact the response times in a real-time system and, in such systems, the selection of these choices becomes critical in obtaining a correct design, i.e., one that meets its timeliness requirements. In the absence of tools that allow reasoning about the impact of these implementation choices on the timeliness requirements, a designer must make choices based on rather loose and ad-hoc guidelines or intuition, and hope that the requirements will be met. This can easily result in lengthy fine-tuning cycles to iterate to a correct implementation. Furthermore, this process must be repeated frequently in evolving systems, whenever new functionality is added or when timing requirements change.

At the other end of the spectrum, schedulability analysis tools have been developed, which allow the user to analyze the feasibility of a system with respect to its timeliness requirements. Such tools include TimeWiz from TimeSys[7] and RapidRMA from TriPacific Software[8]. However, their underlying computational models lack the design power offered by object-oriented design technologies. Hence, these tools cannot provide timing analysis in the early stages of design when using object-oriented methods. Rather, they come into play once the design phase is complete. Worse still, it is not even possible to directly use these schedulability analysis tools to perform timing analysis for designs developed using the real-time object-oriented development tools, because of the incompatibilities of the underlying computational models.

One explanation for this lack of support is that the developments in scheduling theory, and more generally performance analysis techniques, have taken place largely in isolation from the developments in object-oriented design methods. The underlying computational model of implementations, generated by tools such as ObjecTime Developer, is based on

[3]iLogix is located at http://www.ilogix.com/
[4]Verilog is located at http://www.verilogusa.com/
[5]Artisan Software is located at http://www.artisansw.com/
[6]Project Technologies is located at http://www.projtech.com/
[7]TimeSys is located at http://www.timesys.com/
[8]TriPacific Software is located at http://www.tripac.com/

an event-triggered architecture, where tasks are implemented as event handlers. Thus, each task is ultimately part of multiple end-to-end computations and timing constraints, quite unlike the tasking models assumed in real-time scheduling theory, where tasks of the same end-to-end computation are essentially maintained as a whole.

Finding a middle point between these two extremes has already been proposed. One prominent and representative example is HRT-HOOD [BW94], which is heavily influenced by the developments in real-time scheduling theory. However, in concentrating on the timeliness aspects, the method is weak on behavioral modeling concepts, making it difficult to use when the reactive behavior is complex.

Hence, the general problem involves providing complete support for real-time system design. Specifically, treating timeliness as a first-class concern in the design phase by using schedulability analysis to guide the design process. In addition, incorporating features already present in most real-time object-oriented software development environments, such as a powerful modeling language for designing the functionality of real-time systems, visual tools for facilitating the design process and a code generation strategy for automating the implementation from the design stage.

1.2 Solution Overview

The solution to our problem requires the formulation of a design strategy (and its implementation in a toolset) that would enable designers to concentrate on the modeling of the system rather than the intricacies involved when designing real-time systems. In other words, a designer would not have to worry about ensuring the system's feasibility with respect to schedulability, because the tools would automatically yield a feasible system, if such a system could be manufactured.

Specifically, the integration of timeliness support involves the following steps. First, we need to isolate the design decisions that affect timeliness but, at the same time, do not affect any functional requirements of the system. Then, we must systematically explore these design choices, using schedulability analysis, to find a suitable combination that satisfies the timeliness properties.

My research group[9] has undertaken the task of developing this methodology for uni-processor hard real-time systems. In this section, I first provide an overview of the general approach, followed by a more detailed discussion of our constrained methodology.

1.2.1 General Approach

As was mentioned earlier, the methodology requires the selection of implementation choices be left as open issues by the designer. We accomplish this goal by modeling the system using two distinct views, namely the application and implementation views. The latter idea is based on the Shlaer-Mellor domain concept [SM97], where a designer can create several views of an application and each of these targets a specific focus.

The application view focuses primarily on the functional requirements of the system under development. The designer models the application view without including any implementation artifacts, which bind the modeled objects to a particular real-time execution environment. In addition, the designer includes any system interactions originating from external sources. In this way, the application model, containing only the design of objects and their interactions, is free of any physical mapping that may result in an unschedulable system, i.e., one that does not meet its timing requirements, if these mappings had otherwise been included.

The implementation view is mainly concerned with the strategies undertaken to implement the particular application. In general, the designer must model the portion of the implementation view that describes any limiting execution environment parameters within which the system will run. The latter includes variables such as whether the system is centralized or distributed and whether machines are uni-processor or multi-processor. These parameters reveal the maximum resources available to the designer for the creation of the particular system. We denote this part of the implementation model as the *fixed implementation model.* The remaining portion of the implementation model, denoted as the *variable implementation model*, will be automatically synthesized.

The problem of synthesizing a feasible implementation model is a complex combinatorial optimization problem. It involves the production of a feasible solution for the real-time system by automatically generating the variable implementation model, which entails both

[9]The research group is headed by Dr. Manas Saksena, who is now teaching at the University of Pittsburgh, and contains the PhD. student Yun Wang, as well as the Master student Alex Nikolaev and myself. Thus, any mention of "we" implies the above-mentioned people making up the group.

making any additional execution environment choices that are left unsettled and determining which implementation constructs to use in order to link the specified application model with the underlying machine. It is easily observed that a fixed implementation model specifies a family of implementation models and identifies the search space for the synthesis process. Consequently, the synthesis procedure searches through the space of candidate implementation models and finds one (if any) that is determined to be feasible using a built-in schedulability analysis tool.

Once an implementation model is synthesized, code generation proceeds, using both the application and implementation models. Notice that a similar generic architectural framework can be created for each implementation model that is based on a common fixed implementation model. Thus, to facilitate the code generation process, the strategy can also provide a set of run-time libraries, each of which implements the primitives used in the corresponding fixed implementation model and application model.

Figure 1 illustrates the developed methodclogy. Although we have constructed the tool integrating this methodology assuming uni-processor hard real-time systems, the components making up the tool are still valid for the development of any generalized real-time system. The tool can be decomposed into two subsystems, namely the *Synthesis subsystem* and the *Automatic Code Generation subsystem.*

The Synthesis subsystem, as shown in Figure 2, is made up of three components: the *Schedulability component*, the *Analysis Extractor component* and the *Search Engine component.* The Search Engine component is used to find a candidate implementation model in some heuristic approach. Then, the Analysis Extractor component builds the analysis model by extracting necessary information from both the implementation and application models and by adding the resource demands wherever necessary. The resulting analysis model is inputted to the Schedulability component, which determines whether or not the system satisfies its timeliness requirements by comparing the given timing requirements with the calculated worst-case response times for the time-constrained computations. If the considered implementation model gives rise to a feasible system, the Synthesis subsystem feeds its outputs, i.e., the implementation and application models, to the Automatic Code Generation subsystem. If not, the Search Engine component is prompted to search for the next probable solution, again in some heuristic approach.

In this case, notice that the designer must explicitly provide the system resource demands to enable the use of a schedulability analysis tool. These may be estimates in the

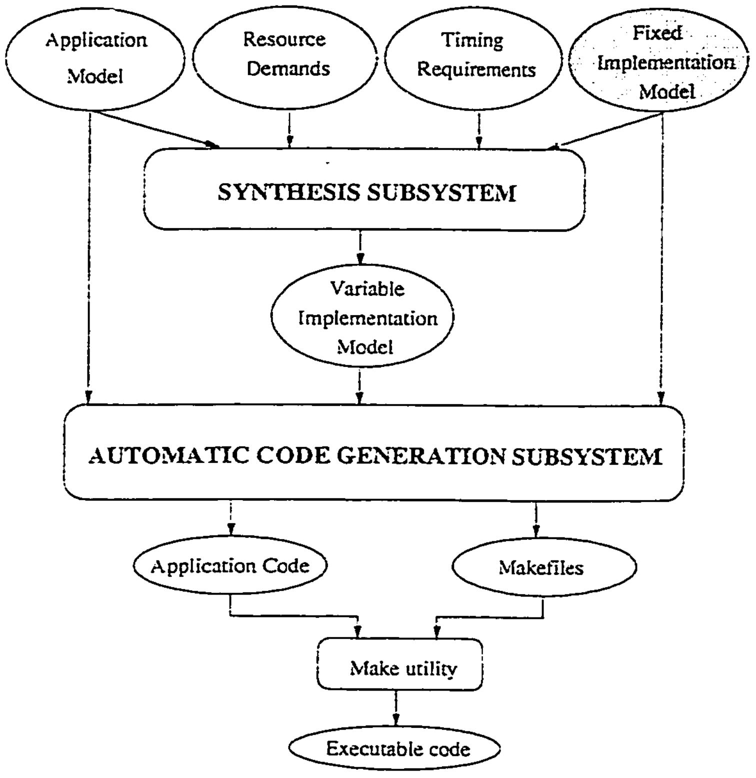


Figure 1: Methodology Overview of the Development Process

early stages of the development life cycle and, during the later stages, may be based on execution time analysis or measurements. Also, observe that measurements can be obtained easily with an automatic code generator present.

The Automatic Code Generation subsystem, as shown in Figure 3, comprises the *Translation component* and the *Run-Time Libraries*. The Translation component uses the implementation and application models to produce the corresponding application code and any necessary makefiles. The translation process follows the guidelines outlined by the generic architecture, which are specified by the fixed implementation model and are implemented as a framework provided by the run-time libraries. At this point, the designer can use the generated makefiles to compile the produced application code along with the run-time libraries, resulting in the creation of an executable for that particular application.

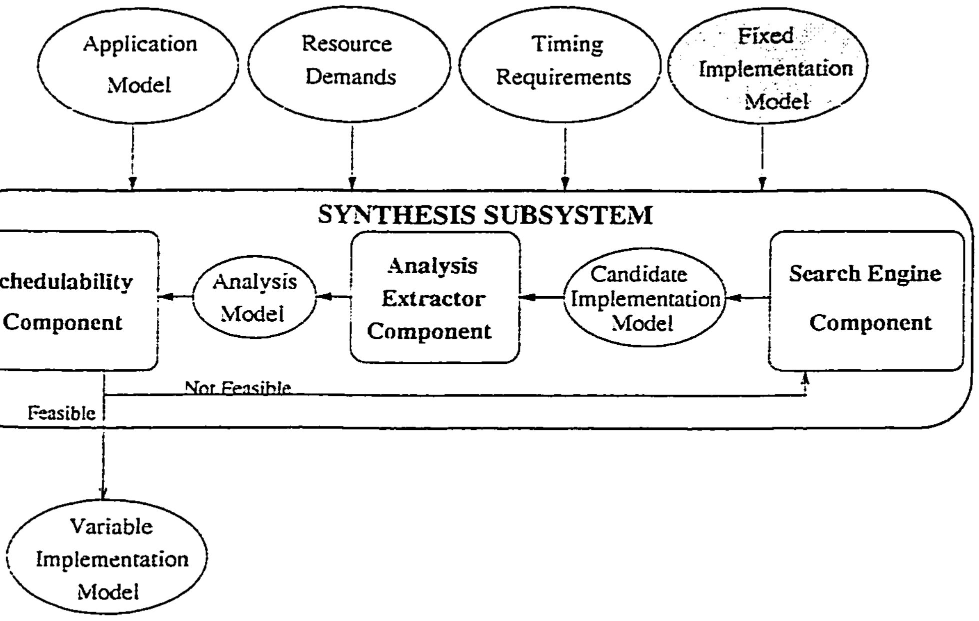
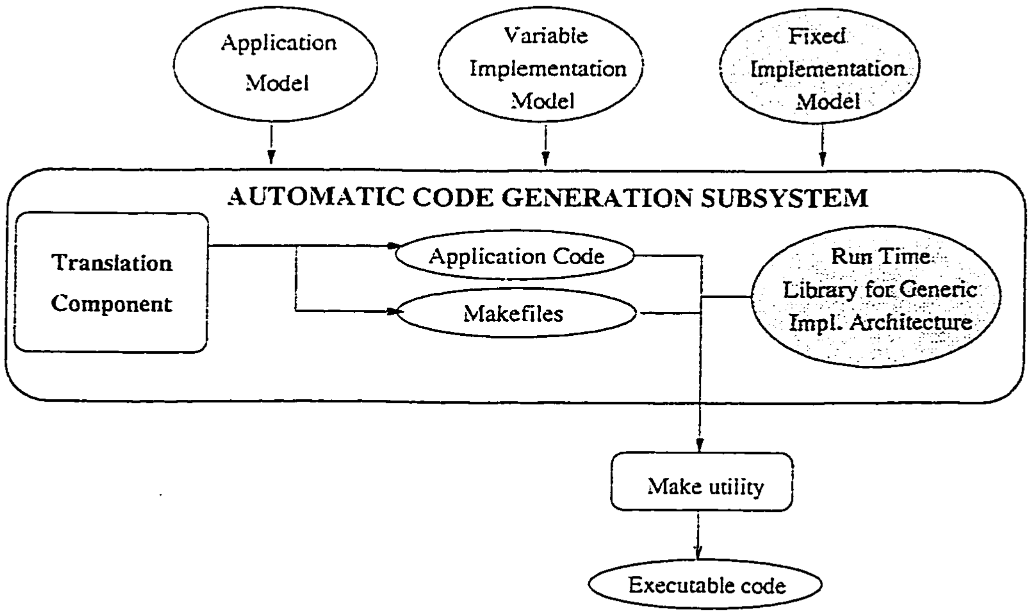


Figure 2: Overview of the Synthesis Subsystem



Application Model
Variable Implementation Model
Fixed Implementation Model
AUTOMATIC CODE GENERATION SUBSYSTEM
Translation Component
Application Code
Makefiles
Run Time Library for Generic Impl. Architecture
Make utility
Executable code

Figure 3: Overview of the Automatic Code Generation Subsystem

1.2.2 Our Constrained Approach

Our developed methodology restricts to hard real-time systems in a uni-processor, multi-threaded environment. I will now give a description of the elements already dealt with, while developing this strategy.

In the application specification, we model active objects, as in the ROOM methodology [SGW94]. These objects communicate through prioritized events. Events may be sent either synchronously or asynchronously. The former method requires the sender object wait for a reply from the receiver while, in the latter technique, the sender object may continue its execution immediately after the event is sent. The behavior of an active object is modeled using a finite state machine. The object remains dormant until an event arrives. Any incoming event may trigger a transition within the finite state machine, resulting perhaps in the execution of an action. We also enforce a run-to-completion semantic on objects, i.e., there is at most one event processing being performed for every object at any given time. In addition, the application view contains information about each external event, such as its type and arrival pattern.

As a result of the predetermined execution environment, the fixed implementation model is completely specified and thus, the designer does not specify any part of the implementation view whatsoever. We represent this fact in Figure 1 as well as Figure 2 by shading the fixed implementation model. Moreover, because of the single fixed implementation model, it is also the case that there is only one generic architectural framework to be implemented within the run-time libraries for code generation. For this reason, we also shade the run-time libraries in Figure 3.

Our run-time library constructs a priority-based preemptive multi-tasking implementation architecture. It utilizes light-weight threads as event handlers, where events are queued in priority order. Each thread remains dormant until an event arrives requiring processing within the particular thread. The thread then processes any queued event by invoking the finite state machine behavior on the recipient object. We also impose a run-to-completion semantic on threads, resulting in a non-preemptive scheduling of events [10], i.e., once processing for an event has begun, processing for another event in the same thread cannot be started until the former has completed.

In a single-threaded implementation, events are processed in a single thread and thus, a

[10] Later, I will explain why synchronous events are the exception to this rule.

higher priority event can be awaiting processing for at most one lower priority event processing. On the contrary, multi-threaded implementations can lead to unbounded blocking. Because of the run-to-completion paradigms in place, it may happen that a lower priority event is being processed while a higher priority event is awaiting processing, as in the single-threaded case. Unboundedness arises since events requiring execution in other threads with priority ranging between the priorities of these two events also end up executing before the higher priority event. This situation is denoted priority inversion. We utilize the scheduling algorithm in [WS99] to bound priority inversion. This scheduling policy supports a dual priority by assigning two scheduling attributes, namely a nominal priority as well as a preemption threshold, to each event. With this policy, an event awaits processing at its nominal priority and, when it acquires the thread for processing, its priority changes to its preemption threshold. Thread priority management occurs whenever an event processing generates another event as well as within the event handling loop in such a way that thread priority equals:

- the highest priority pending event when the thread is waiting for events to process, and
- the preemption threshold of an event when the thread is in the midst of processing that event.

Consequently, thread scheduling works as follows:

1. The operating system picks the thread to execute based on thread priorities.
2. The thread event queue picks the event to be processed based on event nominal priorities.

Our schedulability analysis for this scheduling policy is based on the tasking model with critical instant/busy period analysis [LL73, LSD89, HKL91] for fixed-priority scheduling. The critical instant for a particular task, which indicates the task arrival pattern leading to the worst-case response time, arises when all higher (nominal) priority tasks have just arrived and the lower nominal priority [11] task has just begun processing. To accommodate for the dual priority, we divide the busy period into the early and late busy period. In

[11] For multi-threaded implementations, this lower nominal priority task must also have higher preemption threshold value.

both cases, we must account for blocking from "lower priority" tasks and interference from "higher priority" tasks. These terms are different depending on whether the system is single- or multi-threaded and whether they are being calculated as part of the early or late busy period.

The implementation architecture described above specifies a family of implementation models with varying environmental factor being the number of threads used. Thus, the synthesis process must determine:

- the number of threads in the environment,
- the assignment of scheduling attributes [12] to events, and
- the mapping of these events to the threads.

Making these choices is not fully understood at the moment. However, members of our research group have been working on this area and some results have already been obtained [WS99, SW00, Wan00]. Although the results have been primarily proven for a simplified model, where there is no communication between objects, the author of [Wan00] is extrapolating these results to fit the more generalized model being considered in this thesis. These investigations have lead to the following assertions and solutions:

- Scheduling under the preemption threshold assignment subsumes both a fixed-priority preemptive and non-preemptive scheduling model.
- If a feasible implementation model exists, i.e., one that produces a schedulable system, then creating another implementation model, which has each event mapped to its own thread and the same scheduling attribute assignments for every event, also produces a feasible solution.
- Given that each event is mapped to its own thread, address the problem of finding an optimal scheduling attribute assignment that ensures schedulability. In [WS99], the optimal solution developed there has been shown to be inefficient. To counter this effect, sub-optimal solutions have also been proposed in [SW00] that are much more practical. These algorithms have varying computational complexity and therefore, the approach is to try each algorithm, in sequence of increasing complexity, until a

[12]Nominal priority and preemption threshold for multi-threaded implementations, while only nominal priority for single-threaded implementations.

solution is found. I first present the three sub-optimal solutions, in order of increasing complexity, followed by the inefficient optimal solution.

Pre-Assigned Preemption Threshold Approach concentrates solely on nominal priority assignment because it assumes that event preemption thresholds have been predetermined. The approach requires searching for feasibility under fixed-priority preemptive and non-preemptive priority assignments. If any of these produces a feasible solution, the resulting implementation model is also feasible under preemption threshold scheduling. Fortunately, an optimal ordering algorithm with search space $\mathcal{O}(n^2)$ solves the problem for preemptive scheduling [Aud91, TBW94] and has been found to be applicable to non-preemptive scheduling as well [GRS96]. Note however that both preemptive and non-preemptive priority assignments need to be tried, as neither dominates over the other [GRS96].

Greedy Approach follows a two-stage process. First, we need to assign event priorities and then, we use the priority assignment to find a preemption threshold assignment that will result in system feasibility. This approach is based on our optimal, yet inefficient, algorithm discussed later.

- The greedy nominal priority assignment algorithm divides the events into the assigned lower priority events and the unassigned (but necessarily higher priority) events. Starting with the lowest priority value and iteratively going through each event in the unsorted list, the algorithm tries to find the next best event to assign the current priority value and move to the sorted list. The greedy heuristic function for finding the next best event consists of calculating the worst-case response time of each event and then choosing the one with the greatest "slack", i.e., the leniency that this event has before it misses its deadline, as the next best event. Worst-case response time calculations are valid in this case because we assume that preemption threshold of all events are set to the highest value; thus, blocking can be exactly determined as originating from one of the sorted lower priority events and interference occurs from every element in the unsorted higher priority events.

- The preemption threshold assignment is based on an efficient optimal algorithm presented in [WS99] with search space $\mathcal{O}(n^2)$. The algorithm consists of computing the worst-case response time for each event by setting the preemption threshold of all other events to the highest possible value and, initially, the preemption threshold of the current event to its nominal priority. While the system is unschedulable, the preemption threshold value of the current event is incremented and the worst-case response time calculation is repeated. If the preemption threshold of an event has reached the highest possible value and the system is still unschedulable, then we can say that the system can never be schedulable with the given nominal priority assignment.

Simulated Annealing is a global optimization technique that attempts to find the lowest point in an energy landscape [KGV83]. This algorithm assigns an "energy" to a complete nominal priority assignment candidate at every step and the objective is to find a minimum energy solution. The energy of an event is calculated as the maximum value between 0 and the "negative slack" [13] and thus, the energy of a candidate assignment is simply the sum of all event energies. Hence, an energy level of 0 for an event implies the event meets its deadline as well as an energy level of 0 for an entire candidate solution implies a feasible nominal priority assignment. Initially, the nominal priority assignment conforms with the deadline monotonic priority assignment. The negative slack of each event is computed by (1) assigning a preemption threshold value to all events, (2) computing the worst-case response time for each event and (3) comparing the latter result with the corresponding event deadline. Step (1) is accomplished by using the preemption threshold assignment algorithm presented in the Greedy Approach except for a slight modification: if no preemption threshold assignment makes an event meet its deadline, then its preemption threshold value is set to the maximum possible value. Simulated annealing moves from one priority assignment to the next by swapping the nominal priorities of two randomly selected events. If the new solution has a lower energy, this is a good local choice that may or may not lead to the global optimal solution and we select

[13]The slack was described above as being the subtraction of the response time from the deadline.

this new solution as the next candidate. Otherwise, this is not the greatest local choice one would make, but nevertheless this may lead to a global optimal solution later. To investigate whether to take such upward energy jumps, a control parameter is introduced, namely temperature. The upward energy jump is taken only if the difference between the new and old energies is less than the temperature. Initially, the temperature value is high enough so that every upward energy jump is taken, but it is slowly reduced when a so-called "thermal equilibrium" is reached. That is, as we get closer to the optimum, we want to take as little upward energy jumps as possible because we are narrowing in on the solution. At any time, if a solution with zero energy is found, we terminate because a feasible nominal priority assignment has been achieved.

Our Optimal Algorithm consists of a simple branch-and-bound search algorithm, where we first assign nominal priorities to events and then, use the priority assignment to set preemption threshold values in order to produce a schedulable system. This approach is also the basis for the greedy algorithm described above.

- The nominal priority assignment algorithm divides the events into the assigned lower priority events and the unassigned (but necessarily higher priority) events. Starting with the lowest priority value and iteratively going through each event in the unsorted list, the algorithm examines each event, selecting the most promising candidate first by using the same heuristic function described in the Greedy Algorithm above. If the algorithm fails to find a solution with that partial assignment, it will backtrack and then select the next event.
- Once each event has been assigned a nominal priority, the optimal preemption threshold assignment, presented in the Greedy Algorithm, is performed to assign preemption threshold values.

Besides ensuring feasibility, search algorithms may also attempt to optimize on some cost function, such as minimizing (1) the number of threads used, (2) inter-thread message passing and (3) events with the same recipient object mapped to multiple threads. A synthesis process, aimed towards obtaining a schedulable system by using a combination of these types of cost functions, enhances the performance of the search algorithm, because the cost functions ensure that the least amount of computation will be added to the system.

In [WS99, SW00], the feasibility problem and optimization problem have been completely separated. Below, I present the solutions obtained in solving some primitive optimization problems once the feasibility problem has been solved.

- Given a feasible implementation model, address the problem of optimizing it by increasing the preemption threshold values as much as possible so as to minimize the number of preemptions while still achieving schedulability. The algorithm iteratively goes through each event, in increasing nominal priority order, and increments the preemption threshold value for that event until either the highest threshold value is attained [14] or the system becomes unschedulable [15].

- Given a feasible implementation model, address the problem of optimizing it by decreasing the number of threads required as much as possible so as to minimize memory costs.

  1. The first step is to reduce the number of unnecessary preemptions from the given (feasible) implementation model by using the algorithm discussed above.
  2. The second step consists of grouping together mutually non-preemptive events, i.e., events that do not interfere with each other's execution, in the same thread. The algorithm begins by creating a list of sorted events, in increasing order of their preemption threshold. Then, we remove the first event and form a new group with that event. For each event in the ordered list, we check if it is mutually non-preemptive with the events of the newly-formed group. If so, the event is moved from the ordered list to the group; otherwise, it remains in the ordered list. Once every element in the ordered list is checked, a new group is formed with the first event that has remained in the ordered list and the process continues until the ordered list of events is empty.
  3. Then, each group of events are said to be mutually non-preemptive and thus, can be assigned to the same thread.

[14]The preemption threshold equals the maximum preemption threshold value.

[15]The preemption threshold equals the highest preemption threshold value that kept the system schedulable.

1.3 Thesis Contributions

My contribution to this work revolves around the design and implementation of some of the key components to support the above-mentioned strategy. Specifically, I developed the Schedulability component as well as the Automatic Code Generation subsystem.

The schedulability component determines whether a centralized system within a uniprocessor multi-threaded environment satisfies its timeliness requirements. This is accomplished by computing the response times of the resource-demanding entities that make up the system. We assume that the resource demands of such entities are determined and fixed in advance. I have developed a schedulability analysis tool, which takes textual specifications of the supported analysis model. Chapter 4 explains the schedulability component in more detail.

The automatic code generation component uses the application and implementation models to translate the design model into executable code. To facilitate the code generation task, I have built a library that implements the generic implementation architecture for a uni-processor multi-threaded environment as well as generic entities found within the application model. Chapter 3 fully describes the details.

1.4 Thesis Organization

The organization of the rest of the thesis is as follows. In Chapter 2, I provide an overview of related work performed in both automatic code generation and schedulability analysis. Then, in Chapter 3, I describe the details of my work on the process of automatic code generation. In Chapter 4, I explain the design and implementation of the schedulability analysis process. Finally, I conclude in Chapter 5, where I provide a summary of my work as well as future extensions.

Chapter 2

Background and Related Work

Employing the term "real-time" to describe a system generates a particular connotation: the correctness of some or all system computations depend not only on the results of the computations themselves, but also on the time by which these results are available to the system's environment. In other words, there are timeliness properties associated with system computations that must be satisfied as another step in validating the design of a real-time system.

Our everyday activities involve the usage of real-time systems, whether we are aware of it or not. Some of these systems are used for monitoring and controlling purposes, where catastrophic situations may arise to the degree of endangering human life, if the timing requirements are not satisfied. For instance, in an automobile cruise control system [Rod98], when the user wants to disable the cruise control, the return of control to the user must be done within a short interval of time to avoid the occurrence of an accident, say if there were another automobile ahead of the user that suddenly began braking.

On the other hand, other systems may necessitate these timing requirements to guarantee certain results, such as assuring some predetermined quality of service. An example of this type of system is a movie-on-demand system, where the user of such a system selects a movie for viewing and the provider relays the movie to the user. Although no great disaster will occur if the timing constraints are not met in this case, the latter are still important in some respect. In the above example, the timeliness properties need to be guaranteed so that jerkiness as well as other distortions can be avoided while viewing the movie, which ensures the customer's satisfaction relative to this aspect.

No matter what the reason for these timing constraints, a designer needs to be able to predict whether or not such systems will meet their requirements. Deciding this issue is not such an easy task, but several schedulability tests have been devised to overcome this difficulty. However, these tests come in various shapes and sizes tailored to fit some particular category of real-time system. In other words, a system termed real-time is not self-explanatory. Several distinguishing characteristics further classify real-time systems.

In this chapter, I first describe the prominent features of real-time systems. Using these, basic principles of scheduling theory will be presented, with a greater interest in fixed-priority scheduling theory. Finally, I provide some background material related to the the design of real-time systems.

2.1 Real-Time System Characteristics

In general, real-time scheduling theory models are centered around end-to-end system behavior, which are modeled using the notion of tasks. A task represents an entity requiring execution in some specified environment and it has several characteristics affiliated with it. Basically, scheduling theory modeling expresses a real-time system as a collection of tasks.

2.1.1 Task Characteristics

Associated with a task is its *computation time*, i.e., the processor time required to fully execute, as well as its *deadline*, i.e., the time by which a task should finish executing.

A system is said to be *hard* if there is no deadline-leniency for any tasks within the entire task set. In this type of system, each task must meet its deadline at all times. Thus, it is crucial that some schedulability test be performed in this case. Also, worst-case scenarios are assumed in order to ensure the system will always meet its deadlines. On the other hand, *soft* real-time systems can tolerate the occasional missed deadline. In this case, other techniques are used to determine the degree at which the timeliness requirements were satisfied, such as stochastic methods and simulations. The remaining task attributes can be divided into *arrival information* and *dependency information* as described in the sections below.

2.1.1.1 Arrival Information

Each task may have an *arrival time*, *release jitter* and *arrival pattern* associated with it. The release jitter is defined as the difference between the arrival time, i.e., the time at which a task invocation wishes to start running, and its actual *release time*, i.e., the time at which the scheduler acknowledges the fact that the task is ready-to-run. The task does not necessarily begin running at the release time, although it could.

The arrival pattern distinguishes tasks as follows: a task can either be *non-recurring*, in which case the task is invoked only once, or *recurring*, in which case multiple instances of the same task will exist.

Recurring tasks can be classified further as *periodic*, where the task arrives at regular intervals, or *aperiodic*, where the task is activated only when a particular event occurs [SR94]. The *inter-arrival time*, i.e., difference between two consecutive instance arrivals of the same task, of a periodic task is a constant value, referred to as the *period*. For aperiodic tasks, the inter-arrival time is not regular. However, if the latter is bound below by some value, which we call the *minimum inter-arrival time*, a specialized set of aperiodic tasks are formed, denoted *sporadic* tasks [BF96]. The remaining aperiodic tasks, i.e., those not sporadic, can arrive at any time and thus, performing any schedulability analysis techniques on such tasks is quite impossible.

2.1.1.2 Dependency Information

Dependency information creates links between tasks within a task set. These relationships may arise so that tasks can communicate with one another, denoted *communication relationships*, or because a task requires results from one or more tasks before it can start its execution, called *precedence relationships*.

Communication is usually based on either *shared memory*, where tasks indirectly communicate with others by accessing portions of the memory that are shared, or *message passing*, where explicit communication between tasks occurs. When shared memory is used between tasks, concurrent access can lead to inconsistent updates. However, *synchronization mechanisms* [SS99] can eliminate such problems altogether. Nonetheless, a task may become *blocked* by another because the latter is in the process of using shared memory. Note that processor sharing can also be classified as resource sharing and this also causes tasks to become blocked. In message passing communication, tasks may communicate *synchronously*, where the sender task must wait until the receiver task finishes processing the message sent before it can resume, or *asynchronously*, where the sender task continues its execution immediately after it sends the message. In pure synchronous communication, the sender must *rendezvous* with the receiver to exchange information. [SS99]

Thus, an *independent* task set refers to tasks that do not communicate with each other, do not share resources other than maybe the implicit sharing of the processor[1], and do not have precedence relationships.

2.1.2 Environment Characteristics

We must also consider the environment of the real-time system as a defining factor. First, the system may either be *centralized*, i.e., the entire system resides on one machine, or *distributed*, i.e., the task set is dispersed on more than one machine.

Whether in a centralized or distributed environment, each machine can consist of one or more processors. With multiple processors, true concurrency and parallelism can be achieved. Likewise, within a single processor, the creation of multiple threads produces pseudo-concurrency.

2.1.3 Scheduling Algorithms

Task scheduling is another distinguishing feature of real-time systems. Schedulers can be classified as either being *static* or *dynamic*. Static schedulers create the task execution pattern, or *schedule*, off-line and the latter is then used to dispatch tasks at run-time. In contrast, dynamic schedulers determine the schedule on-line, based on specific task characteristics.

Common dynamic schedulers are based on task priorities. *Priority-driven* scheduling algorithms assign *priorities* to tasks according to some policy. This priority assignment policy may be static, in which case each task is attributed with a priority value, determined off-line, and this value remains fixed at run-time. Such assignments are called *fixed-priority*. Alternatively, priority assignments that are determined at run-time, typically when the task is invoked, are denoted *dynamic-priority* scheduling algorithms. Ultimately, the goal for any priority-driven scheduling algorithm is to find a priority assignment for tasks, which will satisfy the timing requirements of the real-time system.

[1]Note that, in the case where there is no processor sharing among tasks, schedulability analysis becomes trivial.

Schedulers may also be categorized as *non-preemptive* or *preemptive*. In a non-preemptive scheduler, a scheduled task keeps the processor until it decides to relinquish it voluntarily. On the other hand, in a preemptive scheduler, a scheduled task may be suspended, or *preempted*, so that the processor can be allocated to another task.

Although various task set scheduling algorithms exist, only a few are redeemed appropriate for real-time systems. In the past, programmers resorted to using non-preemptive static scheduling techniques, such as *cyclic scheduling* followed by its extension, *cyclic executive scheduling*. In cyclic scheduling, a timer is set up to expire at the next task's arrival time. When this happens, the next task starts using the processor. This process goes on in an iterative method until all tasks are scheduled and then, the process is repeated for recurring tasks. The major problem with this method is that every task must finish in time before the arrival of the next task but, at the same time, there should not be large gaps between the finish of one task and the arrival of the next otherwise the processor is unnecessarily idle. In order to lessen this problem, the cyclic executive scheduling was conceived from the previous technique, in which case time is broken into major and minor frames. Each task is scheduled to arrive at either a minor or major frame. The timer expirations are set up at major frames and each of these contains one or more minor frames. Thus, tasks scheduled within minor frames of a specific major frame will run until completion and then, the subsequent task will automatically start running once the earlier task finishes executing.

The static ordering of tasks in these types of scheduling techniques produces a high level of determinism. In this way, the times at which tasks complete could be easily predicted in advance, leading to simplistic schedulability tests. However, several problems with this approach have been discovered [Loc92], such as the maintainability of the system, the inefficient use of resources and the inflexibility for supporting future needs. Thus, such methods have now been superseded by priority-driven scheduling approaches.

The dispatching process of priority-driven scheduling algorithms consists of scheduling higher priority tasks before lower priority ones. A preemptive approach will preempt a lower priority task whenever a higher priority task becomes ready-to-run. However, some situations do arise, whether with preemptive or non-preemptive, where a lower priority task may be executing while a higher priority task is waiting to run. This phenomenon is called *priority inversion*. For instance, say the processor is executing the task with the current highest priority in a system using a non-preemptive scheduler and a task with greater priority arrives. In this case, the non-preemptive nature of the scheduler will result in the higher priority task to wait for the lower priority task to finish. As another illustration, synchronization mechanisms that are used to eliminate data corruption for shared memory can also create priority inversion. In this case, notice that medium priority tasks can interrupt the execution of the lower priority task, thus unrestricting the amount of priority inversion. Schemes have been devised to minimize priority inversion, which will be discussed in the subsequent section. Nevertheless, precise schedulability tests can be created so long as priority inversion is bound.

Because of all these distinguishing features a real-time system can possess, several types of tests have been developed in order to determine the feasibility of a system with respect to its timeliness requirements. Some of these have been *deterministic*, i.e., it is possible to determine whether or not the timeliness needs are satisfied at design time with complete certainty, while others have been *statistical*, i.e., the determination of a system's feasibility is given with a certain amount of probability. Deterministic approaches, mostly needed for hard real-time systems, require the characteristics of the system to be known in advance.

Historically, two distinct approaches for schedulability testing have come about. Namely, the calculation of *processor utilization*, i.e., the percentage of processor time that may be occupied by tasks, and the computation of *response times*, i.e., the exact duration by which tasks may be delayed. The feasibility is then determined, in the former case, by ensuring the processor utilization does not exceed the upper bound value defined, and in the latter case, by comparing the worst-case response times with the assigned *deadlines*, introduced to quantify the system's timeliness requirements.

2.2 Schedulability Tests

In the pioneering work [LL73], the authors developed optimal static and dynamic priority scheduling algorithms for a set of periodic, independent tasks with hard real-time constraints and deadlines at the end of periods in a multi-threaded uni-processor environment, where the mapping of tasks to threads was given as an assumption. Specifically, they showed that the *rate monotonic priority* assignment is optimal in fixed-priority scheduling with an upper bound for processor utilization approximately 70% for large task sets. Similarly for dynamic scheduling, they proved that 100% processor utilization can be achieved with dynamic priority scheduling algorithms, such as the optimal *earliest deadline first* priority assignment.

Since then, there has been extensive research to generalize and improve the initial model considered. Specifically, these algorithms have been extended to deal with the effect of tasks with arbitrary deadlines, dependent tasks - both communicating and resource-sharing, tasks with release jitter, sporadic tasks. In the next section, I will only deal with these issues for uni-processor multi-threaded fixed-priority scheduling algorithms. However, there have been advancements for the case of multiple processor systems as well as distributed systems and these will be mentioned briefly in a subsequent section.

2.2.1 Fixed-Priority Scheduling Theory

Fixed-priority schedulability analysis for uni-processor systems has received a great deal of attention. The seminal paper by [LL73] initiated it all when the authors formulated the *rate monotonic priority* assignment RMA, i.e., assign a task with a shorter period, a higher priority. They showed that RMA was optimal with respect to fixed-priority scheduling algorithms in the sense that, if a task set can be scheduled by some other fixed priority assignment, it must also be schedulable using RMA. They also devised the upper bound utilization term as a function of the number of tasks within the system:

> For a set of m tasks with fixed priority order, the least upper bound to processor utilization is $U = m(2^{1/m} - 1)$.

Leung and Whitehead [LW82] extended the above model to include task sets where the deadline of each task can be less than the period. They formulated the *inverse-deadline priority* assignment, i.e., assign a higher priority to tasks with smaller deadlines, as an optimal priority assignment for the extended model. This priority assignment reduces to the original when deadlines are at the end of periods.

When synchronization primitives are used for resource sharing among tasks, unbound priority inversion may occur, which jeopardizes the satisfaction of the system's timeliness requirements. Priority inversion occurs when a higher priority task must wait for a lower priority task before executing. In [SRL90], the authors investigated a few protocols of the priority inheritance class, namely the basic priority inheritance protocol and the priority ceiling protocol. Both protocols bound priority inversion by allowing a lower priority task to inherit the priority of a higher priority task while in a critical section. However, deadlocks and chained blocking, i.e., a blocking term occurs for each synchronization primitive that the task must access sequentially, are still present in the basic priority inheritance protocol. The priority ceiling protocol reduces the blocking to exactly the longest critical section of a lower priority task as well as prevents deadlocks from occurring by deferring access to critical sections in certain circumstances.

In the works [JP86, LSD89, SRL90, Leh90], the authors developed exact schedulability analysis to determine worst-case timing behavior for tasks with hard real-time constraints in the $\mathcal{RMA}$ model considered in the initial work [LL73] as well as extended models in later pursuits, such as arbitrary deadlines, release jitter, sporadic as well as periodic tasks. Overview papers on schedulability analysis concerned with such issues and more can be found in [BF96, Fid98, HLR96].

Most of the deterministic schedulability analysis techniques follow the same approach. First, the notion of the *critical instant of a task* is defined to be an instant at which a request for that task will have the largest response time [LL73]. For the types of models we consider, the critical instant for any task occurs whenever the task is ready-to-run simultaneously with all higher priority tasks. Then, the notion of the *busy period at level* i is also required. The latter is defined as the time interval during which the processor is continuously processing tasks at priority i or higher.

With these concepts, the calculation of the worst-case response time of an action involves the computation of the response time for successive arrivals of the action, starting from a critical instant until the end of the busy period. Also, the response time of a particular instance of an action can be calculated by considering the effects of the *blocking* factor from lower priority actions and the *interference* factor from higher or equal priority actions, including previous instances of the same action.

In addition, we are interested in the work [HKL91], where the authors consider the problem of fixed-priority scheduling of periodic tasks that may have varying priorities during specific points of execution. This can happen, for instance, when the task set consists of sequentially-constrained tasks where a task and its subsequent task are not at the same priority. Also, when the basic ceiling priority protocol is used for resource sharing, a task's priority changes for a short period of time in the midst of its execution while in a critical section. In general, systems of this type come about when the addition of a particular mechanism creates a complex priority structure. In this paper, the authors formulate schedulability analysis for such systems by decomposing tasks into sub-tasks, where each of these is characterized by an execution time, a fixed priority and, optionally, a deadline.

Then, for the response time calculation of each task, the authors go on to categorize all tasks depending on how the latter will affect the former, where three distinct possibilities can occur: a task can either have no effect on the response time computation or it can cause interference or blocking. These decisions are easily made when one realizes that sub-tasks, rather than tasks, have become the entities of execution. Consequently, the scheduling policies for tasks now apply to sub-tasks, such as sub-tasks executing at a given priority level can be preempted by any sub-task of higher priority. By deciphering through the collection of tasks, the response time of each instance of a task can be computed. Similarly, the authors make a few variations in both the critical instant and the busy period in order to work with this new model.

With the advent of these protocols, a new breed of priority assignment policies, denoted *mixed-priority assignment* policies, arise that are a mixture of both fixed- and dynamic-priority assignment policies. Another flavor of this mixed-priority assignment is *dynamic thread priorities with preemption threshold* described in [WS99]. This policy supports a dual priority by assigning both a (nominal) priority and preemption threshold value to each task. A detailed discussion of this policy will be discussed in Section 3.2.1.2.3.

2.2.2 Extensions for Multi-Processor and Distributed Systems

The scheduling problems arising from a system made up of multiple processors are two-fold: to determine when a given task executes as well as where it executes. The latter assignment problem has lead to the creation of two distinct priority-driven scheduling algorithms, namely the partitioning and non-partitioning methods. The *partitioning method* consists of dividing the tasks into separate groups, where each of these is assigned to a distinct processor. Alternatively, the *non-partitioning method* treats the set of processors as one entity and tasks are assigned in a higher priority manner to available processors.

Although tempting, it has been shown in [DL78] that optimal priority assignments for single-processor systems are not optimal in multi-processor cases. With either the partitioning or non-partitioning method for task scheduling, Leung and Whitehead [LW82] showed that the problem of determining whether a task set was schedulable, given the priority assignment, is NP-hard. They also showed that the effectiveness of these approaches is incomparable; it has been found that some task sets are schedulable using the first method but not with the second and others work in the second but not the first.

In a distributed system, further complications arise if sequence constraints among tasks exist. The general approach taken [Tin93, GGH97] is to implement each initial task triggered by some external event as a periodic task and subsequent tasks triggered as sporadic tasks. The timing analysis is then performed for each individual processor, where each task has an associated release jitter to incorporate the variable arrival time of the preceding task. In this way, the global timing properties of the entire system are accounted for and each task is thought of as independent. Because sequentially-constrained tasks across processors may be intertwined, the timing analysis process must be repeated until convergence occurs. The latter takes place when, after a single timing analysis passes through the task sets on all processors, all worst-case response times and all worst-case communication times do not increase. However, the above approach may result in pessimistic results, originating from the inclusion of unnecessary interference that cannot occur in practice [BAW97] and thus may lead to poor utilization levels. A less pessimistic approach has been devised in [GH98, BB98], where each task is now treated as dependent and is released after some time, called the *offset*. The latter is expressed using the arrival of the external event of this transaction and the ancestor tasks that the current task depends on.

2.3 Real-Time System Design

Although the principles developed in software engineering hold true for the design process of real-time systems as much as any other type of system, a shift on the importance of these principles occurs because of the timeliness issues prominently distinguishing real-time systems. Above all, the most significant principle of software engineering relative to real-time systems is *predictability*, i.e., the ability of a system to perform correctly and, sometimes even continuously. Other important issues include the maintainability and extendability of a real-time system.

2.3.1 UML and Object-Oriented Design

Recently, several efforts have been made to introduce object-oriented modeling and design to the software development process of real-time systems. The need for such techniques stems from the ever-increasing complexity of real-time systems. Such techniques allow designers to break up a complex software system into several manageable pieces.

Moreover, the object paradigm has evolved to augment the traditional programming language notions of classes and objects with higher level modeling concepts allowing designs of complex system structures and behaviors. Such modeling concepts include (1) the hierarchical specification of the software architecture of a system using objects, (2) the behavioral modeling of objects using extended finite state machines and (3) the use cases and scenarios to model end-to-end system behaviors. These modeling concepts use visual notations that greatly increase the understanding of the system structure and behavior.

The recently standardized *Unified Modeling Language* (UML) [BRJ99, RJB99] is an amalgamation of many of these concepts and notations. It has quickly becoming very popular with a number of UML-based modeling and development tools offered by various commercial vendors.

2.3.2 Code Generation

A much greater benefit from the newly-adopted approach of developing a real-time system using object-oriented technologies results if the design models can be automatically translated into an implementation for a desired target platform. With such automatic code generation, the benefits of modeling extend through the product's life-cycle.

Automatic code generation is especially critical with an iterative and incremental style of software development. In the absence of code generation, developers will often bypass the models and directly modify the code when pressed for time. Thus, models get out of sync with the code and become less relevant. When code generation is supported, models retain their usefulness and the design model becomes the implementation. [Bel98, SGW94] Moreover, when models are executable, they can be simulated to identify design flaws.

While code generation from object-oriented models has received little attention in the research literature, it has been successfully accomplished in a few commercial tools.

One of the first efforts to build design tools for real-time systems based on object-oriented modeling and to provide automatic code generation came from ObjecTime. They built the design tool ObjecTime Developer, which is based on the ROOM modeling language [SGW94]. In Section 1.1, I gave a list of the commercial tool vendors I am aware about that have been involved in similar work. They have built these design tools using UML or some variant of the latter and many support partial or total code generation.

In what follows, I will give a brief overview of many design methods proposed for real-time system development. Since we base our own modeling language on ROOM, I then follow with a discussion on ROOM.

2.3.3 Real-Time Methods Overview

Over the years, many design methods have been proposed for real-time system development, for example JSD, MASCOT, RTSA, DARTS, CODARTS, HRT-HOOD and OCTOPUS [Gom93, AkZ96]. Many recent design methods are based on object-orientation and include HRT-HOOD [BW94], OCTOPUS [AkZ96], CODARTS [Gom93], the ROOM Method [SGW94] and the Shlaer-Mellor method [SM96].

HRT-HOOD was designed for hard real-time systems and is heavily influenced by the developments in real-time scheduling theory. Its abstractions directly map to the concepts in real-time scheduling theory, thus making the designs analyzable for real-time properties. However, in concentrating on the timeliness aspects, the method is weak on behavioral modeling aspects, making it difficult to use when the reactive behavior is complex.

The Shlaer-Mellor method is representative of industrial practice and, while it is strong in code generation, it provides little support for schedulability analysis. This method provides flexibility in determining a physical architecture of the system and our generic implementation architecture is heavily influenced by this, but it provides no clear guidance to map an object design to a physical architecture.

2.3.4 ROOM Methodology/ObjecTime Developer Overview

Active objects, called *actors*, are the entities used to model a system with ROOM. These are encapsulated, concurrent objects that communicate asynchronously by sending and receiving messages through distinct interfaces called *ports*. A message consists of a signal name, an optional list of parameters and a priority. The latter identifies the significance of the message.

The behavior of actors is modeled using *ROOMcharts*, which is an extended finite state machine that may include composite states as well as guard conditions. It is based on the statechart formalism [Har87]. Sending an event to an actor may initiate the execution of an action. The action specification is thought of as a fine-grained detail and thus, can be specified using a programming language, such as C++. Code involved with message transfers are also specified within actions.

ROOM uses a *run-to-completion* paradigm for event execution on a thread, i.e., when a higher priority event arrives for execution on a thread, but another event is being processed in the same thread, the former will not get processed until the latter has completed.

ROOM also provides a generic run-time system, which is also incorporated in ObjecTime Developer toolset, a CASE tool that provides a fully integrated development environment to support the ROOM methodology, with features such as graphical and textual editing for actor construction and C++ code generation from the model [SGW94].

The generic run-time system models threads as event handlers. Within every thread resides an event queue where arriving events are queued in priority order. Actors are also mapped to threads. Hence, when a message is sent to a specific actor, the underlying machine first determines which thread the receiving actor is associated with and puts the message in the corresponding priority queue. Because the execution of an actor resides entirely in one thread and each thread executes in a run-to-completion paradigm, it is implicitly true that there is a run-to-completion paradigm with respect to each actor as well. That is, there can only be at most one processing in progress for every actor in the system. ROOM's generic run-time system also includes a dedicated thread, which is used to insert periodic or timer messages to actors.

Within single-threaded implementations, priority inversion can occur because of the run-to-completion paradigm for event execution on a thread. Nonetheless, this value is bound to the processing of exactly one event. In a multi-threaded implementation, there are two levels of priority scheduling: within the context of a single thread, where the processing of events takes place in event priority order, and across the whole system, where the operating system schedules the threads in thread priority order. However, thread priorities within ObjecTime are statically managed, which lead to unbound priority inversion. Thus, it may be possible that a lower priority message is being processed while a higher priority message is waiting and these are in different threads because the former is mapped to a thread with higher priority.

The ROOM methodology is representative of industrial practice. With the introduction of UML, ObjecTime has cooperated with Rational Software to develop UML-RT, which uses UML's in-built extensibility mechanisms to integrate ROOM concepts within UML.

Both these languages have features that, on one hand, enable high-level modeling of complex real-time applications (e.g., hierarchy in structure diagrams and in state machines, layering, dynamic structures and inter-connections) and, on the other hand, allow fine-grained details to be specified (e.g., use of a programming language like C++ to specify the action to be handled).

Chapter 3

Automated Implementation

Automatically synthesizing an implementation from design models for a desired target platform is not a deeply researched area. To our knowledge, there are no other non-proprietary strategies developed in this field, except for the well-known Shlaer-Mellor Method [SM97]. Furthermore, there are few, if any, academic tools that support code generation. Other instances of the code generation process, either partial or complete, stem from commercial tools, where the code is proprietary. For instance, the design tool ObjecTime Developer, based on the ROOM modeling language [SGW94], provides support for code generation.

In this chapter, I deal with the problem of automating the translation of design models into an implementation for a uni-processor system. Based on our design model representation discussed in the previous chapter, we can restate this problem as that of automatically merging the application view, detailing the system's functional requirements, and the implementation view, specifying the execution environment as well as the bindings of the application view to the execution environment, into executable code.

Our solution entails constructing a textual grammar based on the UML specifications to model the application and implementation views of a system. The grammar, which includes real-time notions, is restricted in certain ways, so that programming can be bearable. We then proceed by translating the entities modeled into actual code. To aid in this translation, we develop a generic implementation architecture specific for uni-processor environments as part of a library.

The details regarding the solution are presented in this chapter as follows. I specify the entities modeled in both the application and implementation views. Then, I continue with a description of the generic implementation architecture aiding the translation process. I also describe the tool that was built to conduct the translation, which includes a high-level design description of the library for the generic implementation architecture, and give a concrete but rather simplistic example of this process. Finally, I conclude this chapter by briefly discussing some issues regarding the solution.

3.1 Application-View Modeling

As in other object-oriented methods, we cast an application as a network of collaborating objects. Our application model is inspired by ROOM [SGW94], UML-RT [SR98] and a UML-based executable object model [Har87]. We make use of UML [RJB99, BRJ99] notation and terminology wherever possible.

The timing constraints associated with a real-time system are also modeled in the application view, some explicitly and others implicitly. The reason for the explicit modeling of some of these constraints is that they are not only of great importance for schedulability analysis, but also pertinent for the process of automatic translation. For completeness however, we briefly describe all the timing-constraint entities requiring modeling.

3.1.1 Object Modeling

The basic architectural modeling entity is an *active object* as in ROOM and UML-RT. Active objects are logically concurrent and communicate by sending and receiving *events*. An event is globally defined within a system by its name and, in the above-mentioned modeling languages, can have an optional list of parameters. For simplicity however, we do not model the optional list of parameters for events. Events may be sent *asynchronously*, in which case the sender object does not wait for the event to be delivered, or *synchronously*, in which case the sender object blocks and continues only after the recipient replies with a *return event*. In accordance with the UML terminology, we refer to asynchronous events as *signal events* and synchronous events as *call events*.

Communication between objects is performed via *ports*. An active object may have an optional list of ports, each of which is locally defined by its name. Message transfers within a system can occur between a pair of ports that are bound to each other. In this way, an event sent out of some port of the sender object may be received if this port is bound to another. If the port is unbound however, the event is discarded. Thus, a port binding specifies a one-to-one communication.

Active objects have their behavior modeled using a restricted *finite state machine*, based on the statechart formalism [Har87], where we disallow composite states as well as guard conditions. Each active object remains dormant until an event is received. The reception of an event may trigger a transition in the state machine, if such a transition exists from the current state given the particular event arrival. Otherwise, the event is discarded.

A transition in the state machine may have a collection of *sub-actions* associated with it. Sub-actions may also be associated with entry to a state or exit from a state. In general, an event trigger will result in an *action*, defined as a collection of *sub-actions*, that may include an exit sub-action from the current state, sub-actions associated with the transition and an entry sub-action for the next state. Hence, we can say that an action captures the processing information for an event.

Also, we expect the execution of an action be *atomic*, in the sense that it follows a *run-to-completion paradigm within the context of an object*. That is, if some action has started executing, no other action for the same object can be started elsewhere until the first action has completed. The atomicity of event processing within the context of an object ensures object consistency and greatly simplifies the design of objects through avoidance of concurrency conflicts.

Other, less disceming, elements comprised within active objects are attributes and methods, as they are conventionally defined within the context of object technology. Moreover, the definition of a method is also expressed as a collection of sub-actions.

3.1.2 Timing Constraints Modeling

The modeling of objects, their associations and their behavior seems to be sufficient for code generation. However, there are a few notions dealing with timing requirements that are also pertinent to this process. As we consider all the concepts, we will identify those meeting the latter condition.

We begin with the notion of an *external event* stream. Each external event stream is characterized by a name, arrival pattern, recipient object and initial arrival time. We differentiate between two arrival patterns: either *non-recurring one-shot* events, where the event arrives only once at the time specified by the initial arrival time, or *periodic* events, where the event first arrives at the initial arrival time and every subsequent arrival is at the time specified by $(initial\_arrival\_time) + (i * period)$. We do not explicitly model *sporadic* events.

The arrival of an external event triggers an *action* or, equivalently, a sequence of *sub-actions*, within a particular object. In other words, we can say that the external event is sent asynchronously to a particular object within the system, which may trigger a transition in the state machine, thus executing some action within the object. From this, we observe that the notion of external event is required for the translation process.

Sub-actions may be specified as code segments in a detail-level language, such as C++. These are termed *uninterpreted.* We impose certain restrictions on uninterpreted sub-actions so that, when timing analysis is performed, it may be done correctly. Specifically, uninterpreted sub-actions must be simple code segments that may not include alternate paths nor loop structures and may not cause any blocking.

We also support some special sub-actions as defined by the UML specifications. Some of these are used to denote sending of events, while others express ordinary concepts found in most programming languages. These are as follows:

- a *send* request sub-action to asynchronously send a signal event to another object,
- a *call* request sub-action to synchronously send a call event to another object,
- a *return* sub-action to reply to a call event,
- an *assignment* sub-action to assign attributes to particular values,
- a *terminate* sub-action to halt the execution of the entire system

Note that we disallow any dynamic creation or deletion of objects, which are also specified within UML as the *create* and *delete* sub-actions [RJB99, BRJ99].

Both request sub-actions consist of an event of the appropriate type (e.g., call event used for call request sub-action) and a particular object port, which ultimately specifies the recipient of the event. In addition, the return sub-action specifies an expression as a return type.

All sub-actions, excluding the assignment sub-action, must be explicitly modeled so as to perform the proper activities prior to termination of the entire program as well as to allow the inter-object behavior to be visible at the modeling level, thus facilitating the timing analysis. The assignment sub-action may alternatively be expressed by an uninterpreted sub-action.

The execution of an action may generate other internal events through send/call sub-actions, consequently resulting in the execution of other actions, and so on. The entire causal set of actions executed as a result of the external event forms a *transaction.* A transaction is not explicitly modeled in the application view. Rather, it is hidden within the representation of actions and sub-actions. This does not cause problems, because a transaction is not explicitly required for the translation process. However, it is needed for timing analysis and thus, it must be explicitly modeled in the timing analysis view, as will be explained in Chapter 4. Note however that the timing analysis model could be extracted from the application model [1] and, in this case, all the information could actually be stored in the application model.

For completeness, we mention other timing attributes not explicitly modeled in the application view. These are:

- the *computation time* associated with an action, which specifies the amount of processor time required to execute this action, and
- a *deadline* that may be associated with an action, introduced to quantify the timeliness requirements of the system.

3.2 Implementation-View Modeling

Notions contained within the implementation view specify information related to the execution environment and the binding of the application model to the execution environment. However, we have already restricted the types of applications that can be modeled to centralized ones running in a uni-processor multi-threaded environment. Also, we have assumed that our underlying implementation architecture, described later in Section 3.3, will treat threads as event handlers, where the processing of events will be performed in priority order. Hence, the *threads*, the *mapping of actions to threads* and the *assignment of event priorities* are the remaining specifications left unsettled.

3.2.1 Thread Modeling

Threads are attributed with a unique name. Moreover, by specifying the threads within a system, we implicitly also identify the *number of threads*, which leads to the determination of whether we are creating a *single-* or *multi-threaded environment* for the particular system.

[1]In Section 4.1.5, we manually extract the analysis from an example system design.

To describe the exact modeling of the mapping between actions to threads and the assignment of event priorities, we must first reason about the scheduling of actions in both single- and multi-threaded environments.

3.2.1.1 Single-Threaded Environment

A single-threaded environment automatically implies non-preemptive scheduling. When an action completes in a single-threaded environment, the current highest priority action, identified by the triggering highest priority event, is the next action to execute on the thread with no interruption until it has completed. Of course, any synchronously-called actions are the exception to this rule, i.e., the current action will suspend itself whenever a synchronously-called action is made and then, when the latter replies (with a return sub-action), the triggering action resumes its execution. We will refer to an action and any synchronously-called actions made from that action as the *synchronous set of an action*[2].

In such a system, when a higher priority action arrives while a lower priority action is executing, the former will be blocked until the synchronous set of the latter completes. Once completed, this higher priority action will execute so long as no other action with higher priority has arrived. Thus, we observe that priority inversion, or blocking, in a single-threaded environment is limited to a single synchronous set of one action.

3.2.1.2 Multi-Threaded Environment

In a multi-threaded environment, multiple threads are introduced to achieve preemptability of event processing. Assuming that event priorities have already been assigned, the objective is to process higher priority events in preference to lower priority ones. However, once an event begins processing within a single thread, it cannot be preempted by newly arrived higher priority events in the same thread. We express this non-preemptiveness quality within a single thread as the *run-to-completion paradigm within the context of a thread.* Because of this fact, priority inversion may still occur and we can think of threads as mutex resources, since only one action at a time may be actively using a thread. As in the single-threaded case, synchronously-called actions are the exception to the rule. For simplicity, we restrict synchronously-called actions to execute within the same thread as the caller action.

[2]This will be formally defined in Chapter 4 when dealing with timing analysis.

As we argue below, unless the multiple threads and their priorities are managed carefully, a solution with multiple threads may not give rise to any advantages over a single-threaded implementation. The overall aim is to drive the scheduling of events based on event priorities and to minimize priority inversion. We will begin with the assumption that actions have been mapped to threads and will consider the assignment of thread priorities using distinct priority management schemes.

3.2.1.2.1 Static Thread Priorities

The simplest approach to managing thread priorities is to assign them static values. In fact, this is the approach followed in Rational Rose RT (and other tools as well). While the designer is free to choose any priority for the threads, a good heuristic is to assign a thread a priority that is the maximum of all events that it processes. This approach is similar to the highest locker protocol for mutex resources [KRP$^+$93].

One of the problems with static thread priority assignment is that events at a lower priority in a higher priority thread have their priorities boosted up artificially. Due to this, there are possibilities of priority inversion from multiple lower priority actions. This is clearly undesirable in many situations.

Another, and potentially more important, problem with static thread priorities is the fickleness it introduces in the design. Since blocking effects can be cumulative, the chances that the addition of new low-priority functionality may affect the timeliness of time-critical transactions increases. In the single-threaded case, since blocking is limited to a single synchronous set of an action, new functionality could be safely added so long as the execution time of any new synchronous set does not exceed the previous upper bound.

3.2.1.2.2 Dynamic Thread Priorities

A solution that avoids such potentially unbound priority inversions is to dynamically manage a thread's priority such that the latter equals the highest priority pending event, including the currently processed event, if any. Note that this scheme is much like the priority inheritance protocol for mutex resources [SRL90]. When thread priorities are dynamically managed in this way, it can be shown that priority inversion is bound.

While priority inversion with the above scheme is bound, we may still get priority inversion from multiple lower priority events. In the worst case, this can occur with one lower priority event from each of the threads. This effect is similar to the chained blocking that can occur in priority inheritance protocols.

3.2.1.2.3 Dynamic Thread Priorities with Preemption Threshold

The work-around to this "chained blocking" type of priority inversion has been devised in [WS99]. In this scheme, each event is not only assigned a priority, but also a *preemption threshold priority*. To avoid any confusion, we will refer to an event's priority as its *nominal priority*. The idea is that when an event is being processed, its priority is raised to its preemption threshold priority. In this way, an event processing can only be preempted by the arrival of events in other threads that have nominal priorities higher than its preemption threshold priority. By appropriately setting the preemption threshold priority, the priority inversion problems can be minimized to a single synchronous set of an action. The preemption threshold assignment constraint is expressed in Equation 6 of Chapter 4.

3.2.2 Binding of Application-to-Environment Modeling

From the above arguments, we deduce the modeling of both the mapping of actions to threads and the assignment of event priorities. These are explicitly modeled in one giant specification, where a mapping consists of a *thread*, an *action identifier* and a *set of scheduling attributes*. We utilize an action identifier to express a unique request from one object to another. This includes the identification of the sending object, the receiving object and the (signal or call) event to be sent. The set of scheduling attributes consists of the *nominal priority* of an action identifier for a single-threaded implementation. For a multi-threaded implementation, it comprises the *nominal priority* and the *preemption threshold priority* of an action identifier.

From this, we can observe that an action identifier uniquely identifies each action within our system. We already imposed the restriction that event names are unique within the system. Thus, we can say that an event also uniquely identifies each action within our system. In addition, we restrict the mapping of synchronously-called actions to execute in the same thread as the caller action in multi-threaded environments. The reason for the above constraint is so that the priority management scheme we use does not incur any added blocking.

3.3 Generic Implementation Architecture

Our generic implementation architecture uses preemptively scheduled threads and is similar to that used in both ObjecTime Developer and Rhapsody tools. However, there are two main differences: first, we transparently manage the priorities of threads in order to bound and minimize priority inversion and second, we allow general mappings of the application model to the implementation architecture.

In this architecture, a thread is modeled as an event handler. A thread maintains an event queue where arriving events are queued. The queued events are processed in a run-to-completion manner within the context of a thread. This can easily be done using an event-handling loop. If there are no queued events, the thread blocks itself, awaiting new event arrivals. Event processing is accomplished by calling the appropriate action of the destination object.

Figure 4 graphically shows the behavior of a thread. Thread priority management is shown in the right half of the figure. As was mentioned above, we use dynamic thread priorities with preemption threshold as our priority management scheme. A thread's priority is then managed as follows.

When an event is deposited into the event queue of the receiving thread within a *send* sub-action, the receiving thread's priority is set to the higher of its current priority and the event's nominal priority, as illustrated in Figure 5(a).

A similar change of priority is done for *call* sub-actions. However, in this case, the target thread's priority is set to the maximum of its current priority and the calling action's preemption threshold priority, as illustrated in Figure 5(b). Nonetheless, the thread priority value turns out to be the same as before for the send sub-action, since we have assumed that synchronously-triggered actions run in the same thread as their caller action.

When a thread retrieves a message from its event queue for processing, it sets its priority to the event's preemption threshold priority. At the end of processing, the thread priority is set to the highest nominal priority pending event in its event queue. Figure 4 shows the changing of priorities within a thread's event loop. With this priority scheme, we can show that priority inversion (defined with respect to nominal event priorities) is bound whenever the preemption threshold priorities are no less than the nominal priorities. Moreover, if the preemption threshold priority of an event is no less than the nominal priorities of all events that are processed in the thread or the object, then priority inversion can be shown to be bound by one lower priority event processing [WS99] (actually, the synchronous set

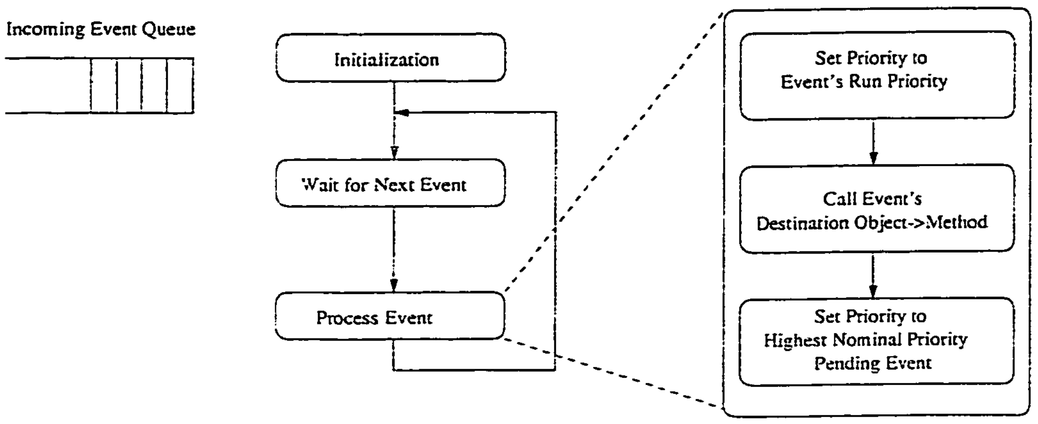

Incoming Event Queue
Initialization
Wait for Next Event
Process Event
Set Priority to Event's Run Priority
Call Event's Destination Object->Method
Set Priority to Highest Nominal Priority Pending Event

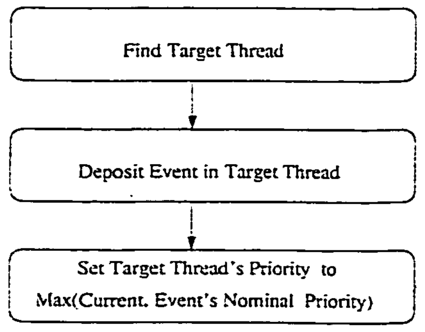
Figure 4: Thread Behavior in the Generic Implementation Architecture

of one lower priority event processing). This fact is proven later when considering the schedulability analysis.

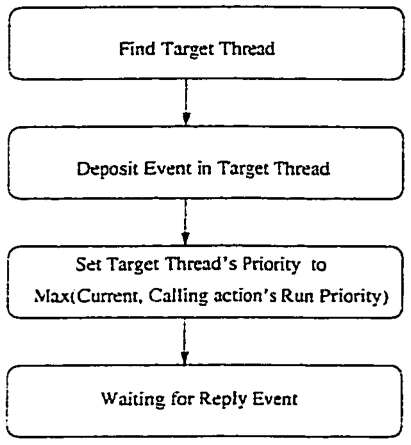
External events, including timed events, are injected into the event queue of the appropriate thread by polling for external events within a thread's event loop. Another possibility is to use a timer manager thread to manage timed events, as it is done in ObjecTime Developer. Alternatively, external events may trigger interrupts and an interrupt handler can insert the event into the target thread's event queue (or set a flag that is checked within the event loop).

Figure 5 models the implementation of the send and call sub-actions described earlier. Both send and call sub-actions must first determine the target thread where the event will be processed. This is simply determined by looking up the mapping of the action identifier to the specific thread mentioned before. This mapping is used to insert an event into the appropriate thread's event queue when it is generated. We assume that all threads share a single address space and therefore, a thread has access to other thread's event queue. Thus, the sending thread (i.e., the thread that executes the send/call sub-action) deposits the event in the receiving thread's event queue.

The generic implementation architecture does not assume or imply any *a priori* mapping of events to threads and, in general, any arbitrary mapping is allowed. Therefore, it is possible that a thread may process events destined for multiple objects. Likewise, events destined for an object may be processed in multiple threads. When this is the case, we need additional mechanisms to ensure that the execution is consistent with the run-to-completion


Find Target Thread
Deposit Event in Target Thread
Set Target Thread's Priority to Max(Current. Event's Nominal Priority)

(a) Send Action


Find Target Thread
Deposit Event in Target Thread
Set Target Thread's Priority to Max(Current, Calling action's Run Priority)
Waiting for Reply Event

(b) Call Action

Figure 5: Send and Call Sub-actions

concurrency model (i.e., within the context of an object). This can be easily done by associating a "mutex" lock with each such object, which is locked during event processing. The locking and unlocking is then done as part of the event handling loop.

Another method, which is what I used, is to take advantage of the dual priority when in a multi-threaded system to ensure that both run-to-completion paradigms are satisfied. For this to work properly however, we need to impose another restriction. Namely, an object may not make a synchronous call to itself using a call event; it must do so by calling the appropriate method within an uninterpreted sub-action. Also, call event loops, which lead back to an object that is awaiting a reply, are strictly forbidden.

3.4 Tool

The textual grammars, describing the application and implementation models referred to in this chapter, can be found as appendices A and B respectively. Although they are two separate grammars, we have combined them into one in such a way that the corresponding parser gets the input from the user from two separate files, namely one containing the application model and the other the implementation, and translates the input from these two files into executable code.

The translation component generates C++ code given valid application and implementation models. As it parses the input from both views, it constructs the translation model, which includes all the necessary information to create the complete application. Then, using the generated makefile, executable code is automatically created using the pre-existing library as well as the application code produced by the translation process. Before we proceed to describe the information stored in all these files, we will first describe the library in greater detail.

3.4.1 Library

The library, implementing the generic implementation architecture, consists of a set of C++ files. These files can be subdivided into six groups as shown in Figure 6. They are as follows:

- Synchronization Service, implementing mutex (RTMutex) and condition variable (RTSyncObject).
- Generic Class Management, implementing a generic class (RTClass), its finite state machine (RTFSMInfo) and its ports (RTPort).
- Thread and Messaging Management, implementing a priority message queue (RT-Controller) and thread functionality (RTThread).
- Memory Management, implementing a message type and the management of memory (RTMessage, RTMessageQ, RTResourceMgr).
- Timer Service, implementing a time specification, a timer event type and the management of timed events (RTTimespec, RTTimerNode, RTTimeClass).
- Generic Main, implementing the generic main function (RTMain).

3.4.1.1 Synchronization Service

Synchronization mechanisms are used to control access to shared data among threads. In our case, we use POSIX-compliant thread libraries for this task, which are guaranteed to be fully portable to other POSIX-compliant environments.

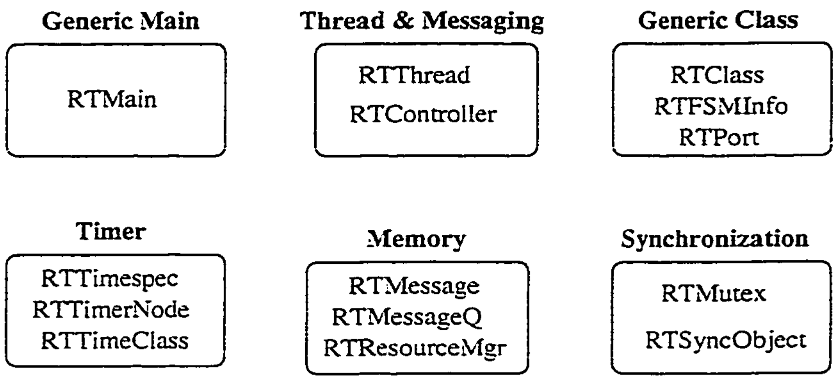


Figure 6: Run-Time Library Services

The RTMutex class implements the mutex synchronization mechanism with the methods *enter(void)* and *leave(void)* for exclusive access to a critical section. This feature is essential in our system because it permits access to global data such as the messaging management and global timer service structures in place.

The RTSyncObject class, derived from RTMutex, implements the methods *signal(void)*, *wait(void)* and *timedwait(RTTimespec)*, which are respectively used to signal a condition variable (i.e., wakeup a single thread that may be blocked on the condition variable), to wait on a condition variable (i.e., block a thread indefinitely until the condition variable is signalled) and to wait on a condition variable or until the absolute time specified by RTTimespec has expired. This mechanism ensures processor resources are not wasted unnecessarily (see Section 3.4.1.5 for more information).

3.4.1.2 Generic Class Management

Every user-defined class inherits from the abstract class RTClass. The latter contains a few abstract methods, which are automatically filled in within each derived class by the code generation process. Specifically, these are:

- Method *AddUserSignalsFSMBehavior(void)* to shape the underlying finite state machine by adding its behavior.
- Method *SpecialInitialization(void)* to include any special initialization requirements when the finite state machine is initialized via the initial transition.

- $SpecialDestruction(void)$ to add any special destruction requirements prior to the destruction of the finite state machine.

This class also defines the underlying functions that are replaced in the user-defined class implementations whenever either a *call*, *signal*, *return* or *terminate* sub-action is made. These are $request(RTMessage :: Synch, Event, Data, OutPort, Priority)$, $request(RTMessage :: Asynch, Event, Data, OutPort, Priority)$, $reply(Data)$ and $abort(void)$ respectively, where all the parameters are further described in Section 3.4.1.3.

In addition to these, RTClass has methods to create, delete, shape and access the associated finite state machine. The finite state machine object, within every RTClass object, is implemented by the RTFSMInfo class. Because there is only one object accessing the finite state machine, no mutex lock is required to access it.

Access to an entry of the finite state machine requires an event and a state. Each entry in a finite state machine includes a pointer to an entry method, a pointer to an exit method, a pointer to a transition method and a new state. Each object's finite state machine includes all user-defined and system events, whether or not this object can receive/send all the events. The behavior of an object is accessed by using the $Go(Event)$ method of its finite state machine, where $Event$ defines the event that has just been received and, of course, the finite state machine has knowledge of its current state.

If the event is received by the object and this event is invalid or there is no transition from the current state, nothing will happen. However if there is a transition involved, then the object's $performAction(PointerToMethod)$ will be called by its finite state machine first, with the exit-of-state method (if any) and then, with the transition method (if any). Then, the finite state machine will move to the appropriate new state and call the object's $performAction$ method for the entry-of-state method (if any).

A list of all ports defined within the derived class are stored in RTClass. Ports are implemented by the RTPort class. All ports have a unique identifier. That is, object a of class A and object b of the same class A each have ports, say one port, port i. Then, port i of object a has a different identifier than port i of object b. Ports may be bound using the $BindPort(RTPort)$ method, where RTPort specifies the remote port that this port will be bound to. In this way, if a message is sent from this port, it will be received by the remote port and vice-versa.

3.4.1.3 Thread and Messaging Management

The messaging service is implemented by the RTController class. This class implements a priority message queue as two distinct entities, namely the inner message queue and the outer message queue. Also, each RTController object has an associated RTSyncObject. Its utility is explained below in Section 3.4.1.5, where the timer service is described.

When a thread wants to send a message, if the message is to be processed in the same thread, then the message is just put into the inner message queue. On the other hand, if the message is to be handled in another thread, a lock on the outer message queue of the receiving thread must first be acquired, the message is then inserted and the lock is released afterwards.

In order to determine whether a message is to be processed in the same thread, we store a static look-up table within the RTController class, which maps action identifiers to controllers. These mappings, originally obtained from the implementation model, are inserted in the table during initialization.

We also associate an RTThread object and RTTimeClass object with every RTController object. An RTThread object implements methods such as $getPriority(void)$ and $setPriority(Priority)$ used for thread management purposes. The RTController class has two derived classes, namely singleController and multiController, where the former conforms to single-threaded environment design specifications while the latter to multi-threaded environments. The RTTimeClass object, described in Section 3.4.1.5, is used to insert timer events into the current RTController's message queue.

3.4.1.4 Memory Management

Memory management is implemented by the RTMessage, RTMessageQ and RTResourceMgr classes. The RTResourceMgr object exists statically for all RTControllers and manages all the memory for the entire system. Messages sent and received by objects are implemented using the RTMessage class. Each message consists of the event to be sent (either synchronous or asynchronous), the scheduling attributes of the message (nominal priority and the preemption threshold priority for multi-threaded environments) and the port the message is sent from.

3.4.1.5 Timer Service

There is one RTTimeClass object for every RTController object, and thus every thread, in the system. Each RTTimeClass object keeps track of timer events that are to be processed within this thread and ensures these timer events are sent when required to the proper destination. Timer messages, implemented by the RTTimerNode class, are sent internally within a thread, therefore there is no additional inter-thread send cost. Time is implemented using the RTTimespec class.

Since the RTTimeClass object is not running in its own thread, we cannot perform "wait" operations, because this blocks all other activities that need to be performed in this thread. Instead, specific methods of the RTTimeClass object are called to handle all local expired timer events after the completion of every message processing. If no more processing remains for the thread, we perform a "timedwait" on the condition variable associated with the message queue, which will block the current thread until the next timer event expires or a new event arrives (in the outer message queue).

3.4.2 Description of Generated Files

As we mentioned earlier, the translation component automatically generates files. Specifically, it creates:

- Class header and implementation files, one for each declared class and
- a makefile, a global header file as well as a file containing the main function.

The class header and implementation files store specific user-defined functionality for each class. They also store information required to create the finite state machine behavior associated with each class, namely the states, the transitions and the valid signals. In addition, local port details are available so that each instance of the class can access its ports without having to know their global names. Each user-defined class inherits from RTClass. Thus, any *call* or *signal* events as well as any *terminate* sub-action are translated to appropriate calls provided by the generic RTClass class. Also, in order to add user-defined behavior to the finite state machine, special abstract methods of RTClass are automatically generated within each derived class by the translation process.

All information required for both objects and the underlying system are stored in the global generic header file. These include:

- A list of all events that can possibly be generated within the system.
- A list of all objects in the system. Note that these objects may only be statically created during system initialization.
- A list of all port identifiers, each of which uniquely defines an object port.
- A list of all controllers in the system.
- A list of all initial port bindings, i.e., object ports that will be bound during system initialization.

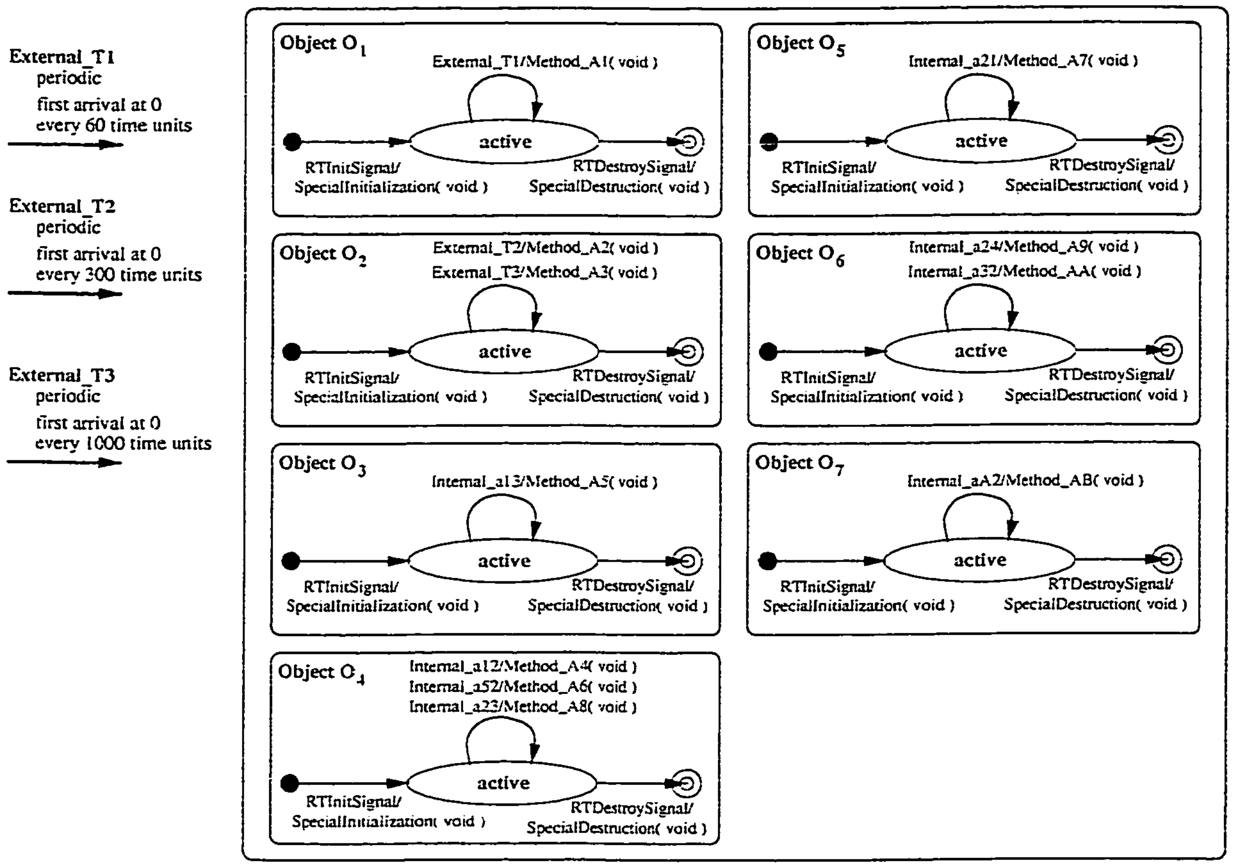
The main function performs the following actions:

1. Create all required system objects.
2. Create any additional objects specific to the particular application.
3. Bind all object ports for communicating objects.
4. Map methods (actions) to threads.
5. Add all initial messages to the system as well as any timer events.
6. Start the event handling loop for the current thread.

3.5 Simple Example

Let us now illustrate the steps a designer would follow in order to use the automated implementation. This would necessitate the specification of the application and implementation model. As the implementation model should be synthesized automatically, we will delay its specification until we need it (in Section 4.3.4). That is, we will defer the event priority assignment and, for multi-threaded implementations, the thread specification, event preemption threshold assignment as well as action-to-thread mapping. Let Figure 7 give a view of the example system and its interaction with the environment.

Our example system is made up of seven objects, where each object's finite state machine is shown. We can observe that each object has only one "real" state associated with it; any accepted event first triggers some transition action and then returns to the same state. We also notice that each object calls its $SpecialInitialization$ action during initialization,



External_T1
periodic
first arrival at 0
every 60 time units
External_T2
periodic
first arrival at 0
every 300 time units
External_T3
periodic
first arrival at 0
every 1000 time units
Object O1
External_T1/Method_A1(void)
Object O2
External_T2/Method_A2(void)
External_T3/Method_A3(void)
Object O3
Internal_a13/Method_A5(void)
Object O4
Internal_a12/Method_A4(void)
Internal_a52/Method_A6(void)
Internal_a23/Method_A8(void)
Object O5
Internal_a21/Method_A7(void)
Object O6
Internal_a24/Method_A9(void)
Internal_a32/Method_AA(void)
Object O7
Internal_aA2/Method_AB(void)
active
RTInitSignal/
SpecialInitialization(void)
RTDestroySignal/
SpecialDestruction(void)

Figure 7: Example System: General Description

through the system event *RTInitSignal*, and *SpecialDestruction* action during system shutdown, through the system event *RTDestroySignal*. In addition, there are three external events interacting with the system just described above. Each of these is periodic and initially arrives into the system at time 0. Their periods, however, differ.

We require more information about our example system even though Figure 7 was quite descriptive. Specifically, we need to know the existing communication relationships and the method descriptions; these are shown in Figure 8 and Figure 9 respectively.

Ports are used to define the communication relationships within the system. For instance, in the figure, we see that object $O1$ can communicate with objects $O3$ and $O4$. By looking at the method description of object $O1$, we see that action *Method_A*1 asynchronously [3] generates signal event *Internal_a*12 and sends it out through its port $P3$,

[3]An asynchronous request is denoted with the keyword "send" in our application model, while a synchronous request by "call".

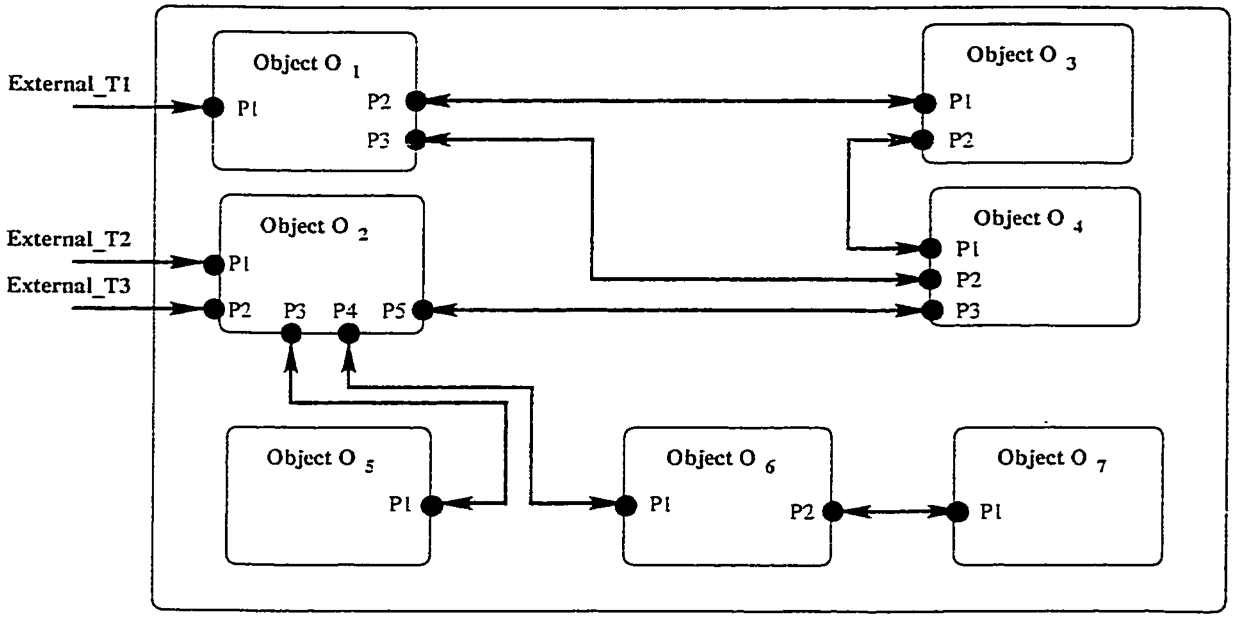


Figure 8: Example System: Port Description

which is bound to port $P2$ of object $O4$. Hence, action $Method\_A1$ of object $O1$ asynchronously generates $Internal\_a12$, which is sent to object $O4$. Now, using Figure 7, we can determine what processing will occur within object $O4$ because of this signal event. If we proceed in this way, starting from the arrival of an external event until no more events are generated, we create the end-to-end view of our system that becomes crucial in our response time analysis.

3.6 Discussion

As discussed earlier, we use a primitive modeling language for system design. Although the language attempts to conform to UML, it does not support much of its features. One obvious distinction is the fact that, in UML, an object may be categorized as active or passive. Active objects, which we explicitly model as well, have a finite state machine associated with them and, when triggered by some event arrival, execute the appropriate action on a specified thread. In contrast, passive objects have no associated finite state machine and, when called upon via an event triggering a method invocation, execute by "borrowing" the thread of the caller object. In our modeling language, we do not explicitly model passive

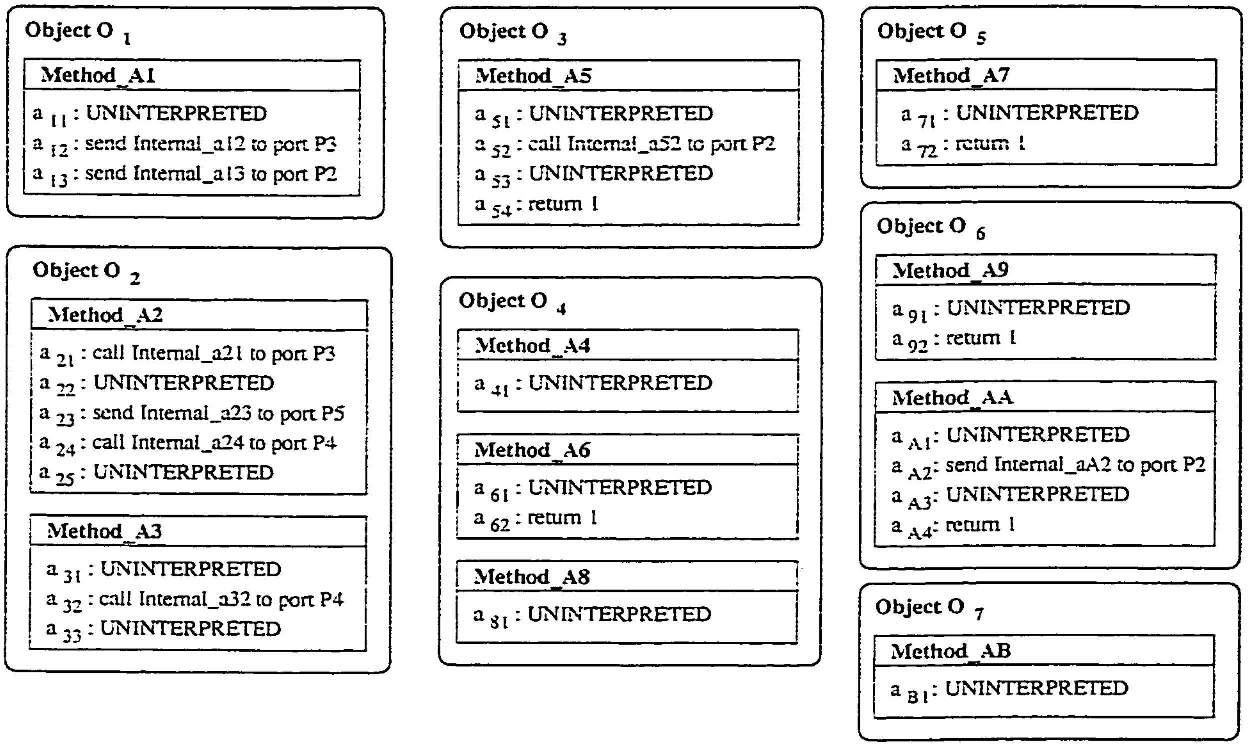


Figure 9: Example System: Method Description

object. Nonetheless, these can be incorporated by calling the required method of an object in uninterpreted sub-actions.

As a means of understanding the full extent of the restrictions imposed on our models, I will re-iterate them below by clustering similar items:

- Supported types of applications that can be modeled must be centralized with a single processor.

- The UML specifications create and delete sub-actions are not supported. All objects are static; they must be created during initialization and destroyed at system shutdown.

- An object's finite state machine is restricted by disallowing composite states and guard conditions.

- A port binding specifies a one-to-one communication relationship. Communications from a single source to multiple destinations or from multiple sources to multiple destinations are not supported.

- If an event does not trigger any transition from the current state of the recipient object, it is discarded. Deferred events are not supported.
- Parameterized events are not supported. An event is globally defined within the system. An action identifier is uniquely identified by an event. An action captures the processing information for an event.
- An asynchronously-triggered action follows two run-to-completion paradigms: one within the context of the corresponding object and the other, within the context of the corresponding thread.
- The processing of a synchronous event must not be on behalf of the same object as the event that triggered its generating action. In multi-threaded environments, a synchronous event must be processed in the same thread as its triggering event. A synchronously-triggered action follows a run-to-completion paradigm within the context of the corresponding object.
- A method is implemented as a collection of sub-actions. A collection of sub-actions is defined as an action. A method can also be denoted an action.

In our model, we also imposed uninterpreted sub-actions to be simple code segments, with no alternate paths or loop structures present. This restriction however is due to the limitations originating from our timing analysis and not our code generation process. In other words, if our timing analysis did support such sub-action traits, the latter could be included within our code generation process by representing them in C++ code.

Also recall that, in the implementation architecture, we schedule event based on event scheduling attributes. As a result, thread priorities are never directly assigned by the designer but rather, they become artifacts of the underlying implementation.

The implementation architecture can easily be specialized by constraining the action-to-thread mapping process. In fact, these specialized architectures may not only trivialize the synthesis problem, but also simplify the schedulability analysis as well. Even more, the code implementing such specialized architectures may be optimized to meet its needs. Examples of such specialized architectures are as follows:

- Mapping all events for a particular object to the same thread. This constraint generates implementation models that are supported in various CASE tools, such as ObjecTime Developer. With this restriction, code optimization could be performed on the implementation of the specialized architecture, since the logic of sending a message would be greatly simplified in this constrained implementation than it would be in the general case.

- Mapping all events in the same transaction to the same thread. This constraint eliminates inter-thread communication and makes the models similar to those used in real-time scheduling theory. Once again, code could be optimized, such as eliminating the outer message queues altogether and eliminating the need to determine where a message is to be processed.

- Mapping all events of the same priority to the same thread. This can be used to eliminate the need for dynamic priority changes.

For many applications, it may be simpler to use one of such specialized architectures. At the very least, these architectures can serve as initial starting points, and the constraints can be relaxed if no suitable solution is found.

Chapter 4

Schedulability Analysis

In this chapter, I consider the schedulability analysis of object design models. The analysis is restricted to uni-processor hard real-time systems and is applicable to the design and implementation models presented in Chapter 3. To facilitate schedulability analysis, we consider an analysis model that can be systematically derived from the design and implementation models. The analysis model presents a view of the system that focusses on end-to-end behaviors, instead of object behaviors. This is useful since timing requirements in real-time systems are often "end-to-end" in nature, i.e., from system inputs to system outputs, and thus, span a computation that may involve the collaboration of multiple objects.

4.1 Analysis Model

Our analysis model assumes that a system is made up of a set of *transactions*, where a transaction denotes a single end-to-end computation within the system. Specifically, it refers to the entire causal set of actions executed as a result of the arrival of an external event, i.e., an event originating from an external source. External event sources are typically input devices (sensors) that interrupt the CPU-running embedded software when an event has occurred. We also include timed events as external events; such events are generated by periodic and one-shot timers. Since all processing within a system is ultimately initiated by some external event, expressing the system as a collection of transactions captures all computations in the design model.

We also utilize the term *action* to capture the processing information associated with an event. Recall that, in our design model, each event is processed by invoking the recipient object's finite state machine and this processing is done in a run-to-completion manner within the context of the object. Thus, in our computation model used for the analysis, an action captures this entire run-to-completion processing for an event. The execution of an action may generate internal events that, in turn, trigger the execution of other actions (possibly in other active objects). Thus, on a more detailed level, each transaction can be expressed as a collection of actions and events, where an external event triggers the execution of the first action in a transaction and this action may generate zero or more internal events that, in turn, trigger the execution of other actions also associated with that particular transaction.

While the execution of an action is atomic within the context of the corresponding active object, its effects may be visible outside the active object even when it has only partially executed. This is true when an action generates events, especially synchronous events, as part of its execution. Therefore, we express an action as being composed of *sub-actions*. In particular, sub-actions that generate internal events are of interest. These include the *send*, *call* and *return* sub-actions. A send sub-action generates an internal event and asynchronously sends it to the recipient object. In contrast, a *call* sub-action generates an internal event, sends it to the recipient object and blocks, waiting for a reply to be sent back through a *return* sub-action by the called action. To simplify matters, we assume that if an action is triggered by a synchronous event, then that action must have a single return sub-action, and the latter must be the last sub-action in the sequence.

Sub-actions are also useful in capturing conditional behavior within an action, as may happen when the action may execute different steps depending on the state of the object or the data associated with the event. In this way, the sub-actions composing an action can create alternate paths within this action. Note that simultaneous paths, i.e., a sub-action branches into more than one sub-action and all of these paths are executed, are forbidden since one action is correlated to the processing of a message and the thread that handles this message must execute the sub-actions in sequential order. This is a true restriction, an artifact of the design model and its implementation. In the rest of the chapter, we will confine our attention to a single sequence of sub-actions. The analysis presented can be easily extended to account for conditional behavior, by considering all alternate paths.

In our analysis model, we also need to capture timing constraints. We are mainly concerned with arrival rates of external events and end-to-end deadlines. The end-to-end deadlines, introduced to quantify the system's timeliness requirements, can be specified on any action in a transaction; the deadlines are end-to-end in the sense that they are relative to the arrival of the transaction or, more precisely, the arrival of the external event associated with the transaction.

4.1.1 Notation

Let $\mathcal{E} = \{E_1, E_2, \ldots, E_n, E_{n+1}, \ldots, E_m\}$ be the set of all event streams in the system, where $E_1, E_2, \ldots, E_n$ denote external event streams and the remaining internal ones. Each external event stream E_i corresponds to a transaction $\mathcal{T}_i$. Associated with each event E_i is an action A_i. An action is decomposed into a sequence of sub-actions, i.e. $A_i = \langle a_{i,1}, a_{i,2}, \ldots, a_{i,n_i} \rangle$, where each $a_{i,j}$ denotes a primitive execution step.

Any sub-action that has externally visible side-effects must be explicitly modeled. In our current analysis, these sub-actions include the send, call and reply sub-actions. Other sub-actions may be used as well without affecting the schedulability analysis, with the restriction that these sub-actions must have bounded execution times and have no externally visible side-effects.

Each event and action is part of a particular transaction. Superscripting is employed to show the association of an entity with its transaction. For example, A_i^τ represents an action and E_j^τ an event, both of which belong to transaction τ. Adding the superscript for external events $\{E_k : k = 1, 2, \ldots, n\}$ is unnecessary since there is exactly one external event associated with each transaction, i.e., external event E_k belongs to transaction k and would be denoted as E_k^k. In this case, the superscript is omitted.

4.1.2 Event and Action Properties

Each external event stream E_i is characterized by a function $\Psi_i(t)$ that gives the maximum number of event arrivals in any interval $[x, x+t)$, where the interval is closed at the left and open at the right. Also, the notation $\Psi_i^+(t)$ indicates the maximum number of event arrivals from event source E_i in any closed interval $[x, x+t]$. For example, an event stream with a minimum inter-arrival time of T has $\Psi_i(t) = \lceil t/T \rceil$ and $\Psi_i^+(t) = \lfloor t/T \rfloor + 1$. In contrast to external events, the rates of internal events are dependent on the execution and thus, cannot be formulated here.

Each action is characterized as either asynchronously- or synchronously-triggered, depending on whether the triggering event is asynchronous or synchronous respectively. All external events are assumed to be asynchronous. Each action A_i executes within thread $\Gamma(A_i)$ and on behalf of active object $\mathcal{O}(A_i)$. The scheduling attributes of each action (or equivalently, the triggering event) include its (nominal) priority and, for multi-threaded implementations, preemption threshold priority, as described in Chapter 3. We represent these scheduling attributes for A_i as $\pi(A_i)$ and $\gamma(A_i)$ respectively. By convention, we will express a higher priority with a larger value. Each sub-action $a_{i,j}$ of A_i has an associated computation time $C(a_{i,j})$ (abbreviated $C_{i,j}$). The computation time of an action is simply the sum of its component sub-action computation times as shown in Equation 1. Also, the computation time of any sequential sub-group of sub-actions $a_{i,p}$ to $a_{i,q}$ where $p \leq q$ is described in Equation 2.

$$C(A_i) = \sum_{j=1}^{n_i} C_{i,j} \tag{1}$$

$$\sum_{j=p}^{j \leq q} C(a_{i,j}) = \sum_{j=p}^{j \leq q} C_{i,j} = \sum_{j=p}^{j \leq q} C_{i,j} :: p \leq q \tag{2}$$

4.1.3 Communication Relationships

Communication relationships within the system can be captured using binary relations between actions. There are two types of communication relationships between actions, namely *asynchronous* and *synchronous*. These can be defined as follows:

Definition 4.1.1 (Asynchronous and Synchronous Relations) *An asynchronous relation $A_i \dashrightarrow A_j$ exists between action A_i and A_j, if A_i generates an asynchronous (signal) event E_j (using a send sub-action) that triggers the execution of action A_j. Likewise, a synchronous relation $A_i \rightleftharpoons A_j$ exists between action A_i and A_j, if A_i generates a synchronous (call) event E_j (using a call sub-action) that triggers the execution of action A_j.*

A single action may (asynchronously or synchronously) trigger 0 or more actions, but each action must have a unique triggering event, which must be generated by a unique action. Hence, both the relations defined above are many-to-one. Moreover, these relations are irreflexive, anti-symmetric and intransitive.

While these relations are defined on actions, it is a sub-action that actually generates the event that establishes a relation. Sometimes, it is useful to identify the particular sub-action that generated the event. To do so, we will sometimes write the relation as $A_i(p)\ \mathcal{R}\ A_j$, where $\mathcal{R} \in \{\rightarrow, \rightleftharpoons\}$, indicating that sub-action $a_{i,p}$ of action A_i generates E_j that triggers the execution of action A_j.

In addition to these communication relationships, it is also useful to define a "causes" relation, denoted by the symbol $\leadsto$, which captures the causal relationship between actions. A causal relationship exists between two actions of a transaction, whenever one of the actions directly or indirectly causes the execution of the other. Formally, this can be defined as follows:

Definition 4.1.2 (Causes Relation)

$$A_i \leadsto A_j \overset{\text{def}}{=} (A_i \rightarrow A_j)\ \vee\ (A_i \rightleftharpoons A_j)\ \vee\ ((\exists k)((A_i \leadsto A_k) \wedge (A_k \leadsto A_j)))$$

In other words, A_i causes A_j if either it directly triggers the execution of A_j (by generating event E_j), or it indirectly causes the execution of A_j through another action A_k. Moreover, the causes relation $A_i \leadsto A_j$ indicates that A_j is a successor of A_i, A_i is a predecessor of A_j and A_i must execute (at least partially) for A_j to be triggered.

When an executing action makes a synchronous call to another action, the latter must execute completely before the former can resume. In this way, we can refer to synchronous actions as calling extensions of the triggering action. Furthermore, it is useful to define the notion of a synchronous set of an action, identifying the set of actions that form this calling extension, as well as the synchronous subset, a subset of the synchronous set.

Definition 4.1.3 (Synchronous Set and Synchronous Subset) *The synchronous set of A_i, denoted $\Upsilon(A_i)$, is a set of actions that can be built by including A_i and all actions synchronously-triggered by it. The process is recursively repeated until no more actions can be included in the set.*

Similarly, the synchronous subset of sequence $a_{i,p}$ to $a_{i,q}$ where $p \leq q$, denoted $\Upsilon(a_{i,p..q})$, is a set of actions that can be built by first including all synchronously-triggered actions originating from any call sub-actions in the sequence $a_{i,p}$ to $a_{i,q}$. Then, any synchronously-triggered action that may arise from the actions already-present in the set are also included, in a recursive manner as in synchronous sets.

With the definitions above, we can observe that each synchronous action may be part of synchronous sets of zero or more synchronously-triggered actions, but of only one asynchronously-triggered action. We will call this asynchronous action as the *asynchronous root of a synchronous set* and will denote it $\xi(A_i)$.

We can also notice that the synchronous set with a synchronous root A_j will always be a subset of the synchronous set with the asynchronous root A_i, where $A_i \rightsquigarrow A_j$. Moreover, the group of synchronous sets with asynchronous roots are non-overlapping.

For clarity, here is an example. Let action A_k have exactly one synchronous sub-action $a_{k,k(l)}$ that triggers A_l and the latter contains exactly two synchronous sub-actions $a_{l,l(m)}$ and $a_{l,l(n)}$ triggering A_m and A_n respectively, where $l(m) \leq l(n)$. Let us further assume that neither A_m nor A_n have any synchronous sub-actions. In this case, we have the following synchronous relations:

$$A_k(k(l)) \rightleftharpoons A_l \quad A_l(l(m)) \rightleftharpoons A_m \quad A_l(l(n)) \rightleftharpoons A_n$$

The synchronous sets of the actions can be constructed through recursive use of the synchronous relations as given below:

$$\begin{aligned}
\Upsilon(A_n) &= \{A_n\} \\
\Upsilon(A_m) &= \{A_m\} \\
\Upsilon(A_l) &= \{A_l\} \cup \Upsilon(A_m) \cup \Upsilon(A_n) = \{A_l, A_m, A_n\} \\
\Upsilon(A_k) &= \{A_k\} \cup \Upsilon(A_l) = \{A_k, A_l, A_m, A_n\}
\end{aligned}$$

Finally, the synchronous sets of sequences of sub-actions, as defined above can also be constructed, as given below:

$$\begin{aligned}
\Upsilon(a_{k,1..k(l)}) &= \Upsilon(A_l) \\
\Upsilon(a_{k,1..(k(l)-1)}) &= \oslash \\
\Upsilon(a_{l,1..(l(m))}) &= \Upsilon(A_m) \\
\Upsilon(a_{l,(l(m))..(l(n)-1)}) &= \Upsilon(A_m) \\
\Upsilon(a_{l,(l(m))..(l(n))}) &= \Upsilon(A_m) \cup \Upsilon(A_n)
\end{aligned}$$

Also, let $C(\Upsilon(A_i))$ denote the cumulative execution time of all the actions in the synchronous set of A_i. Mathematically, this value can be calculated as follows:

$$C(\Upsilon(A_i)) = C(A_i) + \sum_{j::A_i \rightleftharpoons A_j} C(\Upsilon(A_j)) \tag{3}$$

Similarly, let $C(\Upsilon(a_{i,p..q}))$ denote the cumulative execution time of all the actions contained in the synchronous subset of $\Upsilon(a_{i,p..q})$.

4.1.4 Model Restrictions

The schedulability analysis presented in this chapter is conducted on an application model, for which an implementation model has already been synthesized. We impose a few restrictions on the design and implementation models to simplify the analysis. Most of these restrictions are reasonable and do not impose serious limitations on the models.

1. A synchronously-triggered action has a single reply sub-action that is the last sub-action.

2. Any reply action within an asynchronously-triggered action is treated as a send action, i.e., it generates an asynchronous event.

3. In a multi-threaded implementation, we assume synchronously-triggered actions must be handled by the same thread as the caller action.

4. We assume the assignment of priorities to actions follows the rule that any successor action A_j with respect to action A_i must have a priority of equal or lesser importance than action A_i. Thus, we say:

$$(A_i \rightsquigarrow A_j) \Rightarrow (\pi(A_i) \geq \pi(A_j)) \tag{4}$$

5. We also associate priorities with synchronous actions, although these actions are executed as an extension of the calling action (i.e., are not separately scheduled). Therefore, for convenience we assume the following:

$$(A_i \rightleftharpoons A_j) \Rightarrow (\pi(A_i) = \pi(A_j)) \tag{5}$$

6. In multi-threaded implementations, a preemption threshold is used to minimize the blocking effects. The scheduling model works correctly if the following constraint is true [SW00]:

$$((\Gamma(A_i) = \Gamma(A_j)) \vee (\mathcal{O}(A_i) = \mathcal{O}(A_j))) \Rightarrow (\gamma(A_i) \geq \pi(A_j)) \ \wedge \ (\gamma(A_j) \geq \pi(A_i)) \tag{6}$$

The above equation says that, if actions A_i and A_j execute on behalf of the same object and/or within the same thread, then (1) the preemption threshold of A_i must be no less than the nominal priority of A_j and (2) the preemption threshold of A_j must be no less than the nominal priority of A_i. Without loss of generality assume

A_i starts executing first, A_i will not get preempted by A_j because the preemption threshold of A_i must be at least as much as the nominal priority of A_j. Furthermore, as was mentioned in the previous chapter, there is no need to include any special mechanisms within the implementation model so as to preserve the object and thread run-to-completion paradigms with this priority management scheme.

4.1.5 Example: Extracting Analysis from Design Models

An automated extraction of the analysis model from design and implementation models has not been developed yet. However, at the end of Section 3.5 and in this chapter as well, I explained the idea behind it. Therefore, I take the opportunity now to provide the entire manual derivation of the simple example that was presented in Section 3.5 by using the notation described above. This is shown in Figure 10. Notice that, in this figure, I still have not specified any information about threads, the mapping of actions to threads and the scheduling attributes. Thus, we can say that this figure represents the end-to-end behavior of our system, whether it be single- or multi-threaded.

In this figure, we use ellipses to denote actions and represent the underlying sub-actions using squares. A solid directed edge from one sub-action to another represents a communication relationship; this can either be synchronous (call event) or asynchronous (signal event). On the other hand, a dashed directed edge represents a return sub-action, which implies that the event that triggered this particular action must have been a call event. To make the figure more legible, we use $Method\_Ax$ instead of A_{Method_Ax} as well as $External\_Ty$ rather than $E_{External\_Ty}$.

Our example system consists of three periodic transactions. Note that, each transaction traverses multiple objects.

4.2 Problem Statement

Given the above model and restrictions, we want to calculate the end-to-end response times of each action within the model. Once again, end-to-end simply means that the response times are computed relative to the associated transaction arrival. The well-known critical instant/busy-period analysis [LL73, LSD89, HKL91] for fixed priority scheduling is used, but it is adapted to fit the analysis model developed above. Having calculated these values, we can then compare them with the assigned end-to-end deadlines to determine the

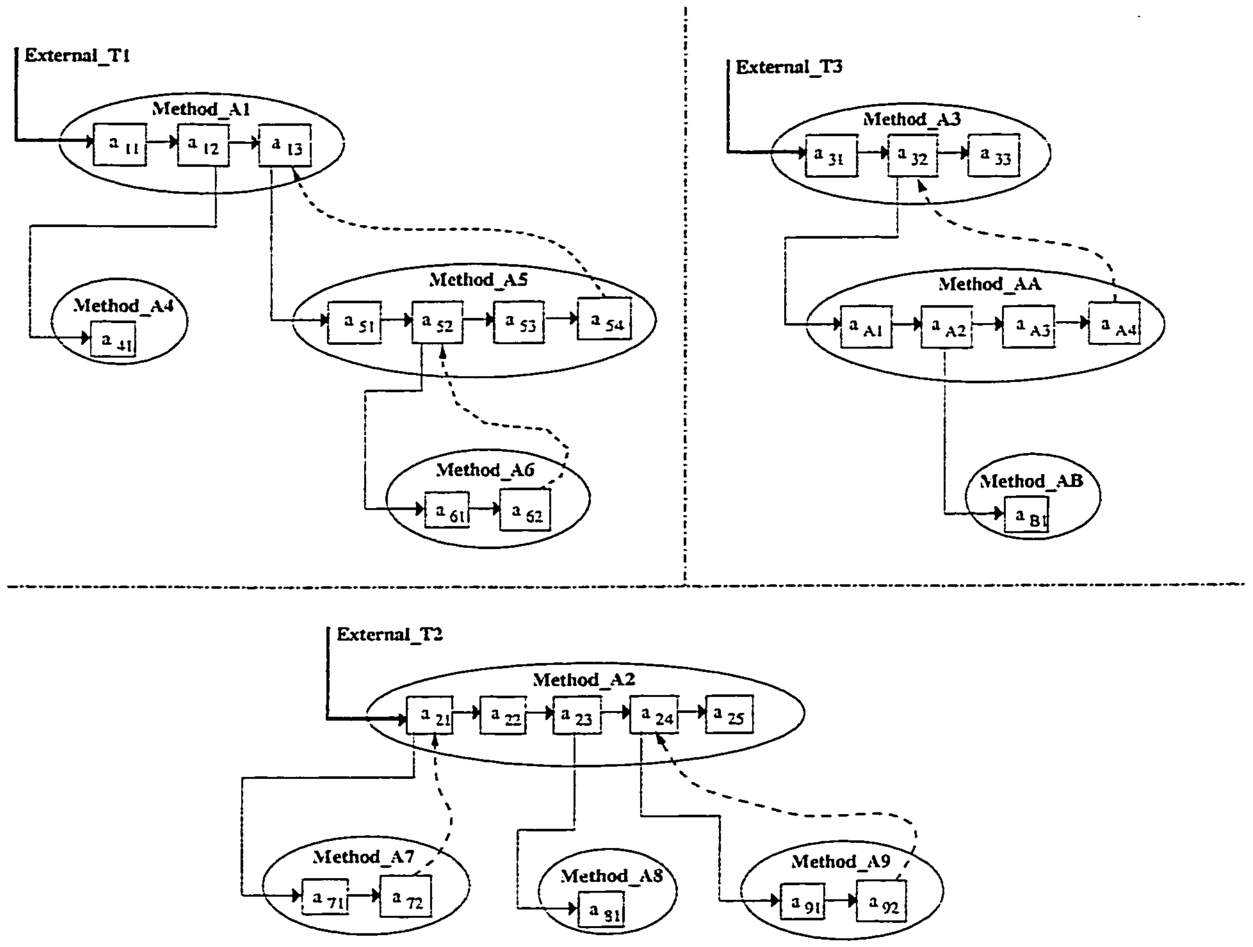


Figure 10: Example System: End-to-End Behavior View

schedulability of a system.

Definition 4.2.1 (Schedulable System) *A system is said to be schedulable if all response times obtained do not surpass the respective deadlines.*

4.3 Response Time Analysis

In our response time analysis for action A_i^T, we will compute the response time of the action for successive arrivals of the transaction, starting from a critical instant, until the end of the busy period. From these, the one with the highest value will be denoted the *worst-case response time of action* A_i^T. We do this by determining the worst-case ending time for each instance of an action within the busy period. However because of our modified busy period analysis, we also need to calculate the starting time for each instance. The starting time refers to when that instance actually gets the CPU for the first time, while the ending time refers to the time at which that instance has completely finished executing. Hence, prior to the starting time, the action is waiting to execute at the nominal priority and afterwards, it is executing (or preempted) at its preemption threshold.

Definition 4.3.1 (Response Time and Worst-Case Response Time) *Let $\mathcal{S}_i^\tau(q)$ denote the worst-case start time for instance q of action A_i^τ, starting from the critical instant at time $= 0$. Likewise, let $\mathcal{F}_i^\tau(q)$ denote the worst-case finish time, starting from the critical instant. Let $Arr_\tau(q)$ denote the arrival time of instance 'q' of external event E_τ, starting from the critical instant, assuming events arrive at their maximum rate. Thus, $Arr_\tau(q) = \min\{t :: \Psi_\tau^+(t) = q\}$.*

Iteratively compute the results of $\mathcal{S}_i^\tau(q)$ and $\mathcal{F}_i^\tau(q)$ for $q = 1, 2, 3, \ldots$ until $q = m$, such that $\mathcal{F}_i^\tau(m) \leq Arr_\tau(m+1)$. Then, the worst-case response time of action A_i^τ is given by:

$$\mathcal{R}_i^\tau = \max_{q \in [1,\ldots,m]} \mathcal{F}_i^\tau(q) - Arr_\tau(q) \tag{7}$$

More specific, the response time of an action can be calculated by considering the effects of: (1) *blocking* from lower priority actions and (2) *interference* from higher or equal priority actions, including previous instances of the same action. The blocking effects refer to priority inversion. These two factors are considered from both single- and multi-threaded implementation perspectives.

4.3.1 Properties of the Scheduling Model

In a single-threaded implementation, a single thread processes pending events in priority order. This simply results in a non-preemptive priority scheduling of actions. There is a special case however: every time a synchronous call sub-action is present, the current action pauses its execution until the synchronously-triggered action sends a reply (using a reply sub-action). However, we made an assumption earlier stating that a synchronously-called action has exactly one reply call and the latter is always the last sub-action of the entire action. Thus, the current action, which synchronously called another, pauses its execution while the synchronous action executes and only resumes once the synchronous action executes completely. Hence, synchronously-triggered actions are truly extensions of the caller action.

Since the execution of a synchronously-triggered action depends on the caller's execution and thus the caller's priority, we can say that the priority of a synchronously-triggered action is simply a matter of convenience; its priority does not have any significance for scheduling. For this reason, we previously restricted the priority of synchronously-trigger actions to equal that of its caller action.

The scheduling in multi-threaded environments is more complex. Recall from chapter 3 that we use a scheduling model in which multiple threads, if they are present, are scheduled based on a dynamic priority management scheme. With this scheme and the model restrictions given earlier, the following properties are true:

1. When a thread is processing an event, i.e., executing an action, then thread priority equals the preemption threshold of the event,
2. When a thread is not processing an event, its priority equals the priority of the highest nominal priority pending event in its event queue,
3. An action A_i can never be preempted by another action A_j if they execute in the same object and/or same thread,
4. When the processor is idle, and if multiple events arrive simultaneously, the thread with the highest priority will have the event with the highest nominal priority and the highest nominal priority event will be processed next,
5. When a thread finishes processing an event, then the next event to be processed will be the event with the highest nominal priority.

4.3.2 Blocking

Blocking refers to the effect of lower priority actions on the response time of an action. Blocking is inevitable in this architecture, due to the run-to-completion semantics of objects and the run-to-completion processing architecture of a thread. However, the scheduling model has been designed to assure that this blocking effect is bound and minimalized. In this section we show how blocking effect may be calculated for an action A_i.

Definition 4.3.2 (Blocking Effect) *Let $B(A_i)$ denote the maximum blocking effect for an action A_i from all lower priority actions.*

4.3.2.1 Case 1 : Single-Threaded Implementations

The blocking effect is easy to determine in single-threaded implementations. Since scheduling is non-preemptive, based on action (nominal) priority, priority inversion can only occur if a lower priority action is already in execution. However, if the lower priority action that causes blocking also generates synchronous events, then those actions will also get executed and cause blocking. In this way, blocking is limited to one synchronous set of actions with a lower priority asynchronous root action. This action must have started executing just before the transaction containing A_i arrives. Thus, the blocking effect for an action A_i is given by:

$$B(A_i) = \max_k\{C(\Upsilon(A_k)) :: \pi(A_i) > \pi(A_k)\} \tag{8}$$

Notice that this blocking term may come from any transaction and so, we omit the superscript denoting the transaction altogether.

4.3.2.2 Case 2: Multi-Threaded Implementations

The blocking effect is trickier to calculate in multi-threaded implementations, but is still greatly simplified due to the properties of our scheduling model. Let us consider an action A_i^τ and suppose transaction τ arrives at time 0. First, any lower priority event that has not triggered its corresponding action execution at time 0 cannot cause blocking. This is because all predecessors of A_i have a priority equal to or higher than A_i and therefore, due to the nature of the scheduling scheme, A_i will be scheduled in preference to this lower priority action. Second, if a lower priority action is in execution at time 0 and its preemption threshold is lower than $\pi(A_i^\tau)$, it will not cause any blocking since it will be preempted. Note that, this action could not have been in the same object or thread as A_i^τ, because of Restriction 6 and therefore, preemption is possible.

Again, as in the single-threaded case, if an action A_j can cause blocking, then its entire synchronous set can. We already mentioned that a synchronous set from an asynchronous root is a superset of one with a synchronous root. Trivially then, we can restrict our attention to synchronous sets with asynchronous roots in considering the blocking effect.

Finally, we can show that the blocking effect must arise from a single such synchronous set with a lower priority asynchronous root. Let us suppose that action A_i gets blocking from two actions, namely A_j and A_k. Again, we know that $\pi(A_j) < \pi(A_i) \leq \gamma(A_j)$ and $\pi(A_k) < \pi(A_i) \leq \gamma(A_k)$. Thus, for them to both block A_i, it must be true that they are both in execution with priority set to their respective preemption threshold. Without loss of generality, assume that A_j started execution first. Then, A_k cannot also be in execution unless $\gamma(A_j) < \pi(A_k)$. However, if this is the case, we have $\gamma(A_j) < \pi(A_k) < \pi(A_i) \Rightarrow \gamma(A_j) < \pi(A_i)$, which contradicts the assumption that A_j blocks A_i.

Based on the above description, we can derive the blocking effect for action A_i as follows:

$$B(A_i) = \max_k\{C(\Upsilon(A_k)) :: \gamma(A_k) \geq \pi(A_i) > \pi(A_k)\} \tag{9}$$

4.3.3 Interference Effects and Busy Period Analysis

As is usual in real-time scheduling literature, the term "interference" is used to denote the effect of higher priority events/actions on the response time of an action. The interference effects cause those higher priority actions to execute in preference to the action under consideration. Unlike blocking however, a higher priority action may cause interference multiple times, if it becomes ready more than once. Since dependencies between actions determine when an action gets ready, we will consider the interference effects of transactions, rather than individual actions, in presenting the response time analysis.

Consider the response time computation for an action $A_i^{\mathcal{T}}$. As explained earlier in this chapter, we perform the busy period analysis by considering instances $q = 1, 2, \ldots, m$ of action A_i and iteratively computing $\mathcal{S}_i^{\mathcal{T}}(q)$ and $\mathcal{F}_i^{\mathcal{T}}(q)$ for each instance, assuming that the first instance arrives at time 0. Accordingly, we define two functions, one to consider the interference effect prior to the starting time of instance q and another for the interference effect once instance q first starts utilizing the CPU until it has completely finished its processing.

Definition 4.3.3 (Interference Effects) *Let the* early interference function $Early_k^{A_i^{\mathcal{T}}(q)}(t)$ *denote the interference effect of transaction k prior to $\mathcal{S}_i^{\mathcal{T}}(q)$, assuming that $\mathcal{S}_i^{\mathcal{T}}(q) = t$. Likewise, let the* late interference function $Late_k^{A_i^{\mathcal{T}}(q)}(t)$ *be a function that gives the interference effect of transaction k for the interval $[\mathcal{S}_i^{\mathcal{T}}(q), \mathcal{F}_i^{\mathcal{T}}(q)]$, assuming that $\mathcal{F}_i^{\mathcal{T}}(q) = t$.*

We will defer a description on how these functions may be determined. We compute the value of $\mathcal{S}_i^{\mathcal{T}}(q)$ by considering all actions that can execute before the action under consideration. Assuming these functions have been determined, the value for $\mathcal{S}_i^{\mathcal{T}}(q)$ can then be determined as follows:

$$\mathcal{S}_i^\tau(q) = \min W :: \quad W \;=\; B(A_i^\tau) + \sum_{k::1 \le k \le n} Early_k^{A_i^\tau(q)}(W) \tag{10}$$

That is, an action (instance) will start, in the worst case, at a time W if the sum of the blocking and interference effects equals W, where W is the first time instance when this becomes true. Note that the term W occurs on both sides of the equation. This equation can be solved by iteratively refining W using the right side of the equation, starting from an initial lower bound value, $B(A_i^\tau)$ in this case, as explained in [TBW94].

Once $\mathcal{S}_i^\tau(q)$ is known, we can compute $\mathcal{F}_i^\tau(q)$. This is done by considering the additional interference effects from higher or equal priority actions that can *preempt* $A_i^\tau(q)$ and therefore, can further delay the execution of $A_i^\tau(q)$. Then, we can write the equation for $\mathcal{F}_i^\tau(q)$ as:

$$\mathcal{F}_i^\tau(q) = \min W :: \quad W \;=\; \mathcal{S}_i^\tau(q) \;+\; C(\Upsilon(A_i^\tau)) \;+ \sum_{k::1 \le k \le n} Late_k^{A_i^\tau(q)}(W) \tag{11}$$

The equation adds any additional computation beyond what has already been accounted for in $\mathcal{S}_i^\tau(q)$. This includes, at a minimum, the computation time of the action, including any actions in its synchronous set. In addition, we add any interference that comes from actions that may preempt $A_i^\tau(q)$, which is given by $Late_k^{A_i^\tau(q)}(W)$, a function which we will derive later in this section.

4.3.3.0.1 Early Interference Function. Let us now consider the determination of the early interference function $Early_k^{A_i^\tau(q)}(t)$. The function depends on whether we are considering interference from another transaction $k \neq \tau$, or from actions in the same transaction, i.e., $k = \tau$.

First consider the case when $k \neq \tau$. In this case, for any arrival of the transaction k in the interval $[0, t]$, we have to consider the computation times of all higher or equal priority[1] actions making up transaction k. Thus, the interference function is given by:

$$Early_k^{A_i^\tau(q)}(t) \;=\; \Psi_k^+(t) \;\cdot\; \sum_l (\; C(A_l^k) :: \pi(A_l^k) \geq \pi(A_i^\tau) \;) \tag{12}$$

We note that interference from other transactions will be considered for all event arrivals in the interval $[0, \mathcal{S}_i^\tau(q)]$, where the closed interval is considered, because if a higher priority

[1]Equal priority actions are considered as well to ensure that we get the worst-case. However only some equal priority actions will interfere because events are queued in FIFO ordering. Taking all equal priority actions into account for interference gives rise to pessimistic analytical results.

action becomes enabled at time $S_i^\tau(q)$, then $A_i^\tau(q)$ cannot begin executing. We also note that all synchronously-called actions that can interfere are also included due to the way we have assigned priorities to them.

Now consider the case when $k = \tau$, i.e., interference from actions within the same transaction. In this case, it is important to distinguish between previous instances, i.e., $1, 2, \ldots, q-1$ of the transaction, and all other instances after that. Accordingly, we write:

$$Early_{\tau}^{A_i^\tau(q)}(t) \;=\; Early_{\tau-}^{A_i^\tau(q)}(t) \;+\; Early_{\tau+}^{A_i^\tau(q)}(t) \tag{13}$$

where $Early_{\tau-}^{A_i^\tau(q)}(t)$ is the interference effect from past instances $(1, 2, \ldots, q-1)$ and $Early_{\tau+}^{A_i^\tau(q)}(t)$ is the interference effect of all other instances $q, q+1, \ldots$ that may have arrived in $[0, S_i^\tau]$. The past instances of the transaction have a similar effect as other transactions, since any higher or equal priority actions of the transaction must execute prior to $A_i^\tau(q)$. Thus, this can be given as:

$$Early_{\tau-}^{A_i^\tau(q)}(t) \;=\; (q-1) \;\cdot\; \sum_{l}(\; C(A_l^\tau) :: \pi(A_l^\tau) \geq \pi(A_i^\tau) \;) \tag{14}$$

The interference effect of instances q onwards must not count the effect of any action A_l^τ if $A_i^\tau \rightsquigarrow A_l^\tau$ since if $A_i^\tau(q)$ has not executed, any action that is caused by it could not have executed either. Furthermore, we assume that multiple instances of the same action execute in order and thus, this is true for instances $q+1$ onwards as well. There are additional considerations if A_i^τ is synchronously-triggered, which we will show shortly. However, if action A_i^τ is asynchronously-triggered, the following equation gives the interference effect.

$$Early_{\tau+}^{A_i^\tau(q)}(t) \;=\; (\Psi_\tau^+(W) - q + 1) \;\cdot\; \sum_{l}(\; C(A_l^\tau) :: \neg(A_i^\tau \rightsquigarrow A_l^\tau) \;\wedge\; \pi(A_l^\tau) \geq \pi(A_i^\tau) \;) \tag{15}$$

Let us now consider a synchronously-triggered action A_i^τ and let $A_g^\tau = \xi(A_i^\tau)$ be the asynchronous root of the synchronous set containing A_i^τ. Then, we have a chain of actions, starting from A_g^τ up to A_i^τ, that partially execute and then, get blocked until A_i^τ completes. Thus, there may be sub-actions in this chain of actions that have not had a chance to execute and the effects of those sub-actions must be discounted. Furthermore, any actions caused by those sub-actions must also be discounted.

The above statement changes the interference term $Early(\mathcal{I}_{\tau+}(A_i^\tau))$, i.e., the interference for instances $q, q+1, \ldots, \Psi_\tau^+(S_i^\tau(q))$. For instance q, only a subset of the actions making up $\Upsilon(A_g^\tau)$ will contribute to interference and, of these, only some of their computations will be counted. Specifically, say sub-action $a_{g,k}$ is the synchronous action that

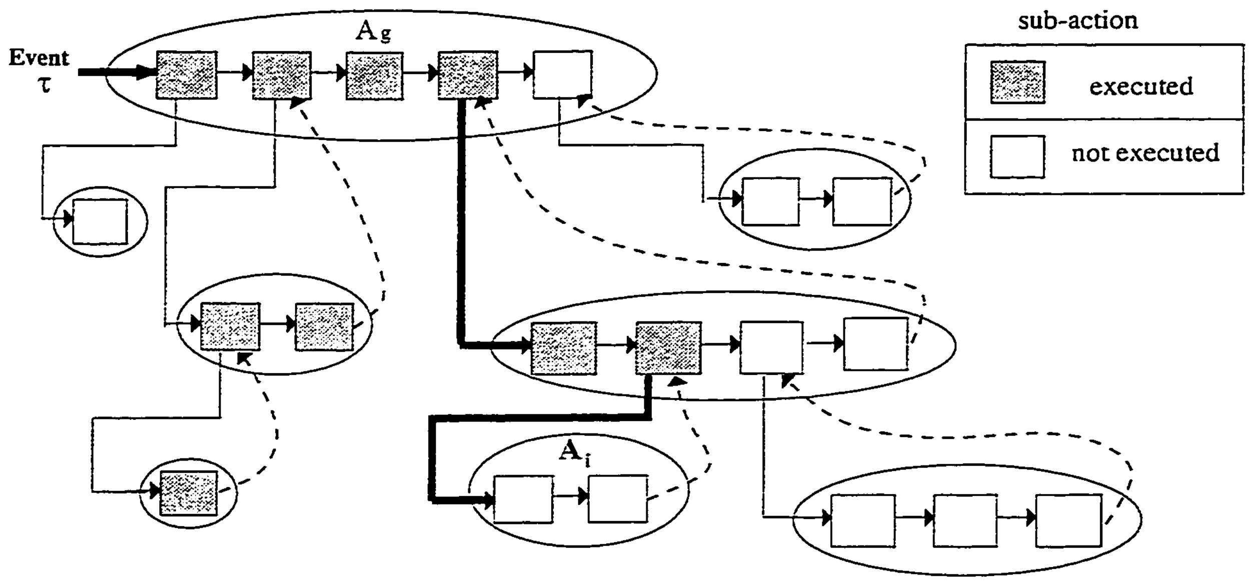


Figure 11: Execution (Partial or Complete) of Actions Prior to the Execution of A_i

eventually leads to the execution of A_i, then the interference added by A_g is $\sum_{q=1}^{q\leq k}(C_{g,q})$ in addition to the contribution of any synchronous action triggered by a sub-action in the set $\{a_{g,1..k-1}\}$. Once again, the synchronous action triggered by $a_{g,k}$ that leads closer to the execution of A_i, say A_l, will only partially execute before it makes the synchronous call that brings us closer to the execution of A_i. Thus, we see that actions leading to the direct path from A_g to A_i will partially contribute and any synchronous action triggered prior to the execution of the sub-action that leads to the direct path will contribute completely. Pictorially, this can be represented as in Figure 11, where the direct path from the asynchronously-triggered predecessor A_g to the action in question A_i is visible. For instances $q+1$ onwards, none of the actions in $\Upsilon(A_g^\tau)$ can cause interference, since their previous instance (q) is blocked.

Thus, based on the above discussion, we can derive the following equation:

$$\begin{aligned} Early_{\tau+}^{A_i^\tau(q)}(t) &= \sum_{x=0}^{i-1}[\ C(\Upsilon(a_{(h(x)),(1..(j(h(x))-1))})) + \sum_{y=1}^{j(h(x))} C(a_{(h(x)),(y)}) \\ &\quad :: (A_{h(x)}((j(h(x)))) \rightleftharpoons A_{h(x+1)}) \\ &\quad \wedge (A_g^\tau \equiv A_{h(0)}) \\ &\quad \wedge (A_i^\tau \equiv A_{h(i)})\ ] \end{aligned} \tag{16}$$

4.3.3.0.2 Late Interference Function. The late interference function incorporates the effects of any actions after the action under consideration (A_i^τ) has begun executing. It is necessarily true that any such effect will come from actions that (1) arrive after $\mathcal{S}_i^\tau$, and (2) can preempt (A_i^τ).

Since actions are executed in a non-preemptive manner for single-threaded implementations, there can be no preemption effects after an action has started executing. Thus, we have the following trivial function:

$$Late_k^{A_i^\tau(q)}(t) = 0 \tag{17}$$

Consequently, the worst-case finish time for instance q of action A_i is given by the following simplified equation:

$$\mathcal{F}_i^\tau(q) \;=\; \mathcal{S}_i^\tau(q) + C(\Upsilon(A_i)) \tag{18}$$

In a multi-threaded implementation, preemption from other actions is possible. For an action A_j to preempt A_i, several conditions must be satisfied. First, its nominal priority must be no less than the preemption threshold of A_i. Furthermore, A_i and A_j must be assigned to execute in different threads and on behalf of different objects. Finally, these must also hold for A_j's triggering action as well, otherwise the event that triggers the execution of A_j cannot be generated. The last condition must actually be recursively true up to the root of the transaction, i.e., the action triggered by the external event. We note that this also eliminates any successors of A_i^τ, so we do not need to consider that separately. If A_i is synchronously-triggered, this also eliminates any actions that are part of the synchronous chain from $\xi(A_i)$ to A_i and therefore their successors, since we have assumed that they must all be in the same thread. Based on the above, we can write:

$$Late_k A_i^\tau(q) \;=\; (\Psi_k(W) - \Psi_k^+(\mathcal{S}_i^\tau(q))) \;\cdot\; L_k(A_i^\tau) \tag{19}$$

where $L_k(A_i^\tau)$ sums up the computation time of all the actions within transaction k that meet the conditions specified above, and $\Psi_k(W) - \Psi_k^+(\mathcal{S}_i^\tau(q))$ gives the number of new arrivals of event E_k. Formally, $L_k(A_i^\tau)$ can be computed as in the following equation, where we assume that the associated event E_k triggers the execution of action A_1^k:

$$\begin{aligned} L_k(A_i^\tau) &= L(A_i^\tau, A_1^k) \\ L(A_i^\tau, A_j^k) &= \begin{cases} 0 \quad \text{if} \quad (\Gamma(A_j^k) = \Gamma(A_i^\tau)) \vee (\mathcal{O}(A_j^k) = \mathcal{O}(A_i^\tau)) \vee (\pi(A_j^k) < \gamma(A_i^\tau)) \\ \\ C_{A_j^k}, + \sum_{g::A_j^k \rightarrow A_g^k \vee A_j^k \rightleftharpoons A_g^k} L(A_i^\tau, A_g^k) \qquad \text{else} \end{cases} \end{aligned} \tag{20}$$

Here, we cannot say that if an action A_j causes interference to A_i, then all its synchronous successors also cause interference. This is because a synchronously-triggered action may be part of the same object as action A_i even though we are sure that the synchronously-triggered action will be in a different thread than A_i (because we assume it to be in the same thread as A_j).

4.3.4 Example: Calculating Worst-Case Response Times

We now continue developing our simple example by performing schedulability analysis on it. As was explained in Section 4.1.5, the example system is made up of three periodic transactions. Please refer to Figure 10 to view the end-to-end behavior of the system.

At this point, we need to specify other information that has not been mentioned yet. This includes the event priority assignment, and, for multi-threaded implementations, the thread specification, event preemption threshold assignment as well as action-to-thread mapping.

We are primarily interested in showing how to compute worst-case response times. Therefore, we do not require any deadline specifications. We present the analysis for both single- and multi-threaded implementations.

We first compute the worst-case response times for the single-threaded implementation. Let Table 1 provide the information required for the single-threaded case. For completeness, we include the information already available from all the figures already presented. Here, we use A_x to denote action A_{Method_Ax} and transaction T_y to correspond to external event $E_{External\_Ty}$. Recall that a priority with higher value indicates a greater importance.

Using the tool described below, I calculate the worst-case response times. The results are displayed in Table 2. For completeness, I also provide the worst-case response time calculations of all instances in Appendix D. Recall that all these response times are actually

Transaction	Period	Action	Priority	Object	Sub-Actions	Computation Time
T_1	60	A_1	10	$\mathcal{O}_1$	$\langle a_{1,1}, a_{1,2}, a_{1,3} \rangle$	$\langle 5, 1, 1 \rangle$
		A_4	6	$\mathcal{O}_4$	$\langle a_{4,1} \rangle$	$\langle 5 \rangle$
		A_5	10	$\mathcal{O}_3$	$\langle a_{5,1}, a_{5,2}, a_{5,3}, a_{5,4} \rangle$	$\langle 2, 1, 2, 1 \rangle$
		A_6	10	$\mathcal{O}_4$	$\langle a_{6,1}, a_{6,2} \rangle$	$\langle 4, 1 \rangle$
T_2	300	A_2	9	$\mathcal{O}_2$	$\langle a_{2,1}, a_{2,2}, a_{2,3}, a_{2,4}, a_{2,5} \rangle$	$\langle 1, 3, 1, 1, 4 \rangle$
		A_7	9	$\mathcal{O}_5$	$\langle a_{7,1}, a_{7,2} \rangle$	$\langle 9, 1 \rangle$
		A_8	7	$\mathcal{O}_4$	$\langle a_{8,1} \rangle$	$\langle 10 \rangle$
		A_9	9	$\mathcal{O}_6$	$\langle a_{9,1}, a_{9,2} \rangle$	$\langle 50, 1 \rangle$
T_3	1000	A_3	8	$\mathcal{O}_2$	$\langle a_{3,1}, a_{3,2}, a_{3,3} \rangle$	$\langle 4, 1, 5 \rangle$
		A_A	8	$\mathcal{O}_6$	$\langle a_{A,1}, a_{A,2}, a_{A,3}, a_{A,4} \rangle$	$\langle 4, 1, 5, 1 \rangle$
		A_B	5	$\mathcal{O}_7$	$\langle a_{B,1} \rangle$	$\langle 250 \rangle$

Table 1: Example System: Single-Threaded Implementation

Transaction	Action	End-to-End Response Time
T_1		753
	A_1	268
	A_4	753
	A_5	261
	A_6	255
T_2		603
	A_2	429
	A_7	368
	A_8	603
	A_9	409
T_3		593
	A_3	593
	A_A	583
	A_B	421

Table 2: Example System: Single-Threaded Response Times

relative to the arrival of the entire transaction and not the individual actions.

All new information required for multi-threaded implementations are contained within Table 3 below. Please notice that we also include preemption threshold values, which satisfy Restriction 6 in Section 4.1.4. To ensure we meet this constraint, we assign preemption thresholds as follows:

1. We first find the "maximum priority" of an object by assigning it the same value as the greatest priority of all actions that execute on behalf of this particular object. Thus, $MaxPrio(\mathcal{O}(x)) = max\{\pi(A_i) :: \mathcal{O}(A_i) = \mathcal{O}(x)\}$.

2. In the same way, we also find the "maximum priority" of a thread by assigning it the same value as the greatest priority of all actions that have been mapped to this particular thread. Hence, $MaxPrio(\Gamma(y)) = max\{\pi(A_i) :: \Gamma(A_i) = \Gamma(y)\}$.

Action	Thread	Preemption Threshold
A_1	Γ_1	10
A_4	Γ_2	10
A_5	Γ_2	10
A_6	Γ_2	10
A_2	Γ_3	9
A_7	Γ_3	9
A_8	Γ_2	10
A_9	Γ_3	9
A_3	Γ_3	9
A_A	Γ_3	9
A_B	Γ_4	5

Table 3: Example System: Multi-Threaded Implementation

Transaction	Action	End-to-End Response Time
T_1		161
	A_1	28
	A_4	161
	A_5	21
	A_6	15
T_2		161
	A_2	67
	A_7	49
	A_8	161
	A_9	108
T_3		582
	A_3	145
	A_A	146
	A_B	582

Table 4: Example System: Multi-Threaded Response Times

3. Finally, we assign the preemption threshold of each action to the highest priority between the maximum priority of its corresponding object and its thread. Mathematically, $\gamma(A_i) = max\{MaxPrio(\mathcal{O}(A_i)), MaxPrio(\Gamma(A_i))\}$.

Pictorially, the execution of our multi-threaded implementation is shown in Figure 12.

Again, I calculate the worst-case response times by inputting the information above into the tool described in the next section. Table 4 presents the results obtained and, as before, Appendix E shows the complete set of worst-case response times for each action instance.

Comparing Tables 2 and 4, we can observe that the end-to-end response time of all transactions have decreased, some more dramatically than others. The introduction of multiple threads has given rise to preemptions.

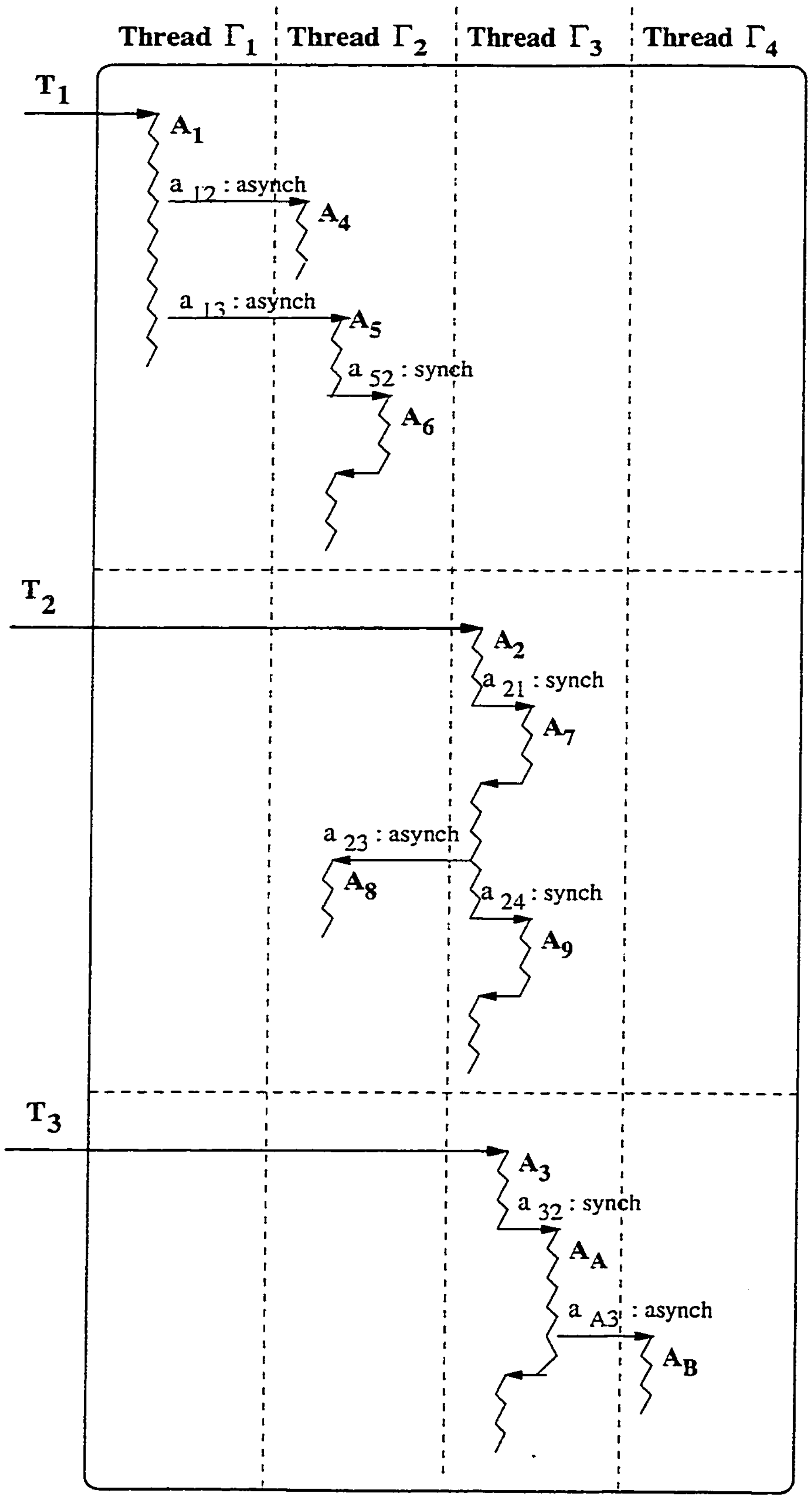


Figure 12: Example System: Execution of our Multi-Threaded System

4.4 Tool

The developed tool computes the worst-case response times of actions making up transactions. These are compared to the deadlines associated with the actions (if any) and if each response time does not surpass the corresponding deadline, the system is said to meet its timeliness requirements.

The analysis model described in this chapter was transformed into an understandable input for the schedulability tool by using a textual grammar. The grammar itself is found as Appendix C.

The tool first parses the user input based on the analysis grammar. Based on its findings, it creates the structure for the model. The analysis model structure is illustrated in Figure 13. In this figure, italicized words imply that these will be calculated by the program, while all others should be given as input by the user. From this, we see that actions hold a great deal of precomputed information, which is later used to calculate the response time. The precomputed information includes the blocking term, both the early and late interference terms, one such structure for each transaction within the system. Using the equations for blocking and interference, the tool goes on to compute the precalculated values. Then, it uses these values to calculate the end-to-end response times of each instance of an action until the end of the busy period, again as described by the response time analysis in this chapter.

The schedulability analysis component is made up of 3 main functions as shown in Figure 14. The more important of the three, calculating the response time, is also shown in greater detail to the right of this figure.

4.5 Discussion

As was mentioned earlier, a system is made up of a set of transactions. This set may be really large if the system is extremely complicated and some of these transactions may not be associated with timing constraints whatsoever. Because of this, it is often useful to restrict our attention to a subset of the entire collection of transactions, namely those associated with time-critical requirements. Since we are using a priority scheduling scheme, a detailed modeling of the discarded transactions is not necessary if they are executed (in their entirety) at lower priorities as compared to the transactions being considered. This assumption is truly reasonable because, in a system where priorities decide the scheduling process,

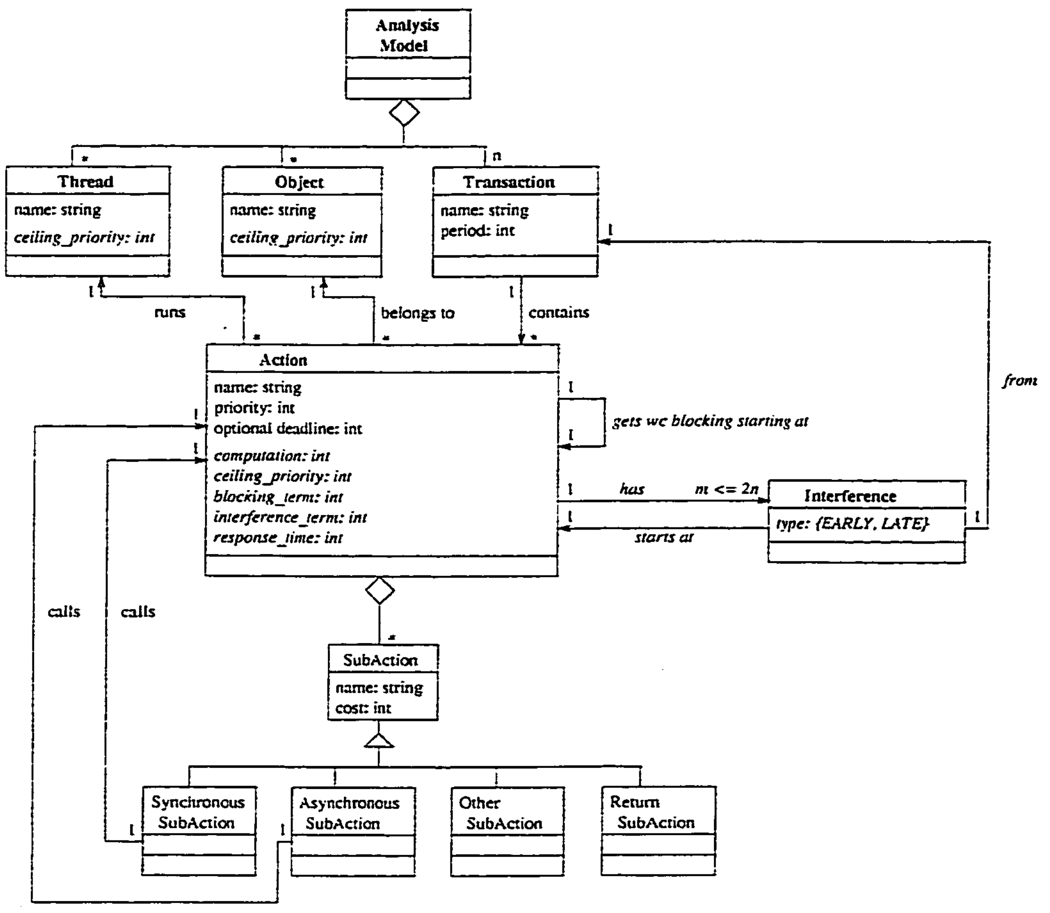
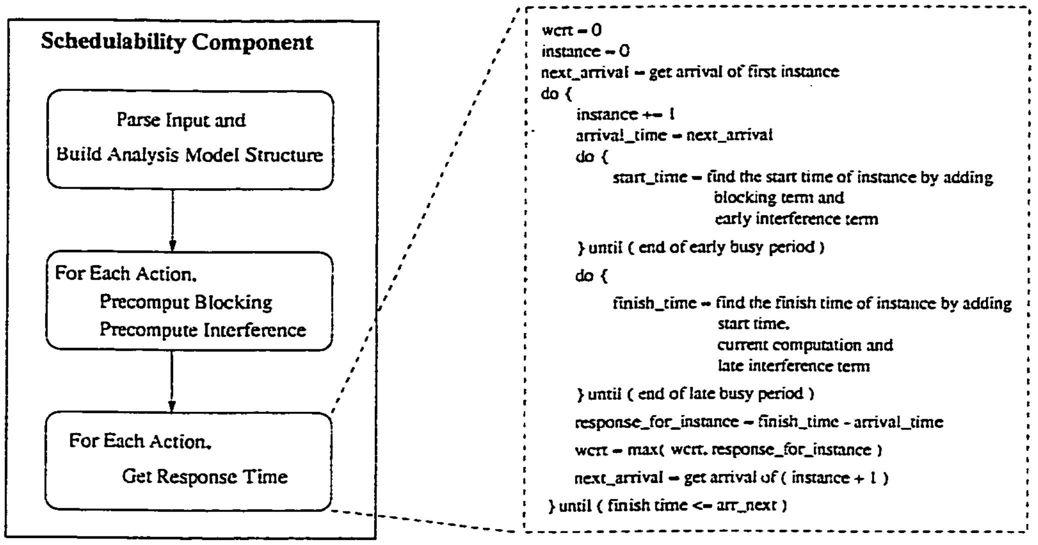


Figure 13: Analysis Program Structure

it would make complete sense to give non-time-critical transactions lower priorities than time-critical transactions.

In practice, however, such transactions cannot be completely disregarded due to priority inversions. In the scheduling scheme we employ, such priority inversion considerations are limited to a single lower-priority action. Therefore, it may be necessary to identify actions with large computation times and ensure that any priority inversion effects associated with those actions are accounted for in the analysis. Also, there are other overheads associated with these transactions such as the costs associated with executing the interrupt handler, which will typically execute at high priorities. Such overheads must also be accounted for in the analysis. We have ignored the effects of any such overheads and priority inversions, although, we note it is relatively straightforward to incorporate these effects into the analysis.



```
wcrt = 0
instance = 0
next_arrival = get arrival of first instance
do {
     instance += 1
     arrival_time = next_arrival
     do {
          start_time = find the start time of instance by adding
                            blocking term and
                            early interference term
     } until ( end of early busy period )
     do {
          finish_time = find the finish time of instance by adding
                            start time,
                            current computation and
                            late interference term
     } until ( end of late busy period )
     response_for_instance = finish_time - arrival_time
     wcrt = max( wcrt, response_for_instance )
     next_arrival = get arrival of ( instance + 1 )
 } until ( finish time <= arr_next )
```

Figure 14: Analysis Tool Behavior

The schedulability analysis and the associated tool presented in this chapter have a few limitations that we now enumerate. Clearly, one limitation is the restrictions that we made on the models. While not severely limiting, the analysis greatly relies on those restrictions and becomes much more difficult if those restrictions are removed.

One limitation of the tools developed is that the transaction information must be manually extracted from the design models, although automation of this process should be quite feasible, because we can now decomposed an action into sub-actions. Note also that there are many "pre-specified" sub-actions that are automatically generated with the code. These sub-actions also include calls to the real-time execution framework. An automatic generation process can easily include these actions as well.

Chapter 5

Concluding Remarks

Software design has become more and more important within the real-time design process ever since functionality implementation gradually migrated from hardware to software and, even more so, because of the increase in software complexity. Consequently, several commercial tools have become available that provide an integrated development environment for real-time systems with object-oriented technology to facilitate the design phase. However, these tools lack the "real-time" support required by many of these systems, especially those with stringent timing requirements and limited resources.

As a result, we proposed a methodology for the integration of schedulability analysis techniques within real-time object-oriented modeling techniques. We accomplish the above by visualizing a system using two separate views: (1) the application model, detailing the functional requirements of the system, is completely specified by the designer and (2) the implementation model is partly specified by the designer to indicate the maximum resources available for the creation of the system and partly synthesized to bind the modeled objects to the particular real-time execution environment. We showed how this synthesis process can be automated to yield a feasible implementation model, i.e., one that meets the specified timing requirements. The automatic synthesis process releases the designer from the burden of selecting various implementation artifacts, such as priorities of events and physical mapping of events, in much the same way automatic code generation releases a designer from the burden of deciding how to implement the modeling abstractions.

We began the construction of a prototype toolset for uni-processor hard real-time systems, implementing the above methodology. Specifically, I dealt with the creation of a schedulability analysis tool and an initial implementation supporting automatic code generation.

The initial implementation automatically generates code from design models using primitives that have been defined within a generic implementation framework. For simplicity, our modeling language does not support sophisticated features, such as hierarchical decomposition, inheritance of state machines and dynamic objects, available in other powerful modeling languages like ROOM. The implementation framework supports multiple threads with threads serving as event handlers and objects communicating by sending prioritized events to eachother.

The schedulability analysis tool is used to determine a system's feasibility, with respect to the timeliness requirements, once an implementation model has been synthesized. In addition to the application and implementation views, the designer needs to input the system resource demands to calculate worst-case response times and the timing requirements to compare against the calculated response times. The analysis is based on a slight modification of the well-known critical instant/busy period analysis [LL73, LSD89, HKL91] for fixed-priority scheduling. In our analysis, we consider two busy periods to calculate the response time for each action: (1) the time prior to the considered action's execution (for the first time), where the action is waiting at its nominal priority, and (2) the interval from the time the action starts executing for the first time until it completely finishes; in a single-threaded implementation, the action cannot be interrupted from this point on while, in a multi-threaded case, the action executes at its preemption threshold priority.

5.1 Future Work

The major area for advancements in our developed methodology is mostly centered around the synthesis problem. In particular, we want to fine-tune the synthesis algorithms to integrate complex cost functions that would maximize system performance, however the latter is defined.

Also on a broader scale, constructing the methodology for more than just uni-processor real-time systems would, of course, be the ultimate design tool. This could be done by using self-contained components, one for each type of environment setting, that can be mixed and matched with others to provide the appropriate design tool. Components for each aspect of the methodology could be divided into those for hard or soft real-time systems, centralized or distributed systems as well as uni-processor or multi-processor systems. Other components that would enhance this design tool even more includes adaptive and dynamic systems.

However, this thesis deals with some aspects of the methodology and, as a result, I limit the discussion to the future work of these topics. First, the methodology would be greatly enhanced if the analysis model (transaction information) could be automatically extracted from the design models. Also, we want to investigate the impact of specialized architectures (as mentioned in Section 3.6) compared to the generalized architecture we have developed. Moreover, the development of a real-world example would provide much better insight on the viability of our methodology.

In what follows, I briefly outline the modifications to both the automated implementation and schedulability analysis that arise when certain imposed restrictions are eliminated. I will only consider the elimination of restrictions that are not realistic and, thus, diminish the value of our methodology [1].

5.1.1 Extensions for Sub-Action Types

Types of sub-actions that would be greatly appreciated include alternate paths and loop structures. We do not support these types of sub-actions in our methodology, since resource demands of such structures are "fuzzy" and thus, performing schedulability analysis on such systems could be misleading.

Alternate paths could be represented within our modeling language by creating a *case* or $if-then-else$ sub-action as well as adding guard conditions to the finite state machines of objects. The translation of such sub-actions to code would be straight-forward. Now, the problem would be how to accurately calculate response times if alternate paths are included. In the simplest case, we could include the computations of all paths in the response time calculations. In this case, we have overestimated these values and, possibly, not found any schedulable solution when one might have existed. A better way would be to distinguish the different paths and include the one that leads to the largest computation. However, when considering the response time of an action that is part of an alternate path, then the following considerations need to be taken into account:

[1]The restrictions fitting this description that do not require much thought will, of course, not be discussed.

- Previous instances of the same transaction can execute any of the paths and so, the one leading to the largest computation is assumed (to get worst-case response times).
- The current instance of the same transaction must execute the path that the considered action is in (otherwise, the action could not have executed).
- Future instances of the same transaction can execute any of the paths and so, the computation added is the same as that for previous instances.

Loop structures are a little trickier to include because of the unknown number of iterations that a loop may need to execute. If the latter value is known (or an upper bound of it can be determined), then unrolling the loop would be the easiest way for schedulability analysis. Note that this unrolling could be automatically done by the future extraction process. Again, a loop structure can easily be represented in our modeling language and be translated to actual code.

5.1.2 Extensions for Transaction Priorities

If we lift the restriction that transactions are made up of non-increasing priority actions, many schedulability analysis terms are invalidated and would need to be redefined. These include the critical instant, the blocking and interference effects, which are all crucial in the calculation of the response time analysis. No necessary changes would be required for the automated implementation.

5.1.3 Extensions for Synchronously-Triggered Actions

Without the supposition that a synchronously-triggered action has exactly one return sub-action and the latter is the last sub-action in the sequence, we can do the following. If there are more than one return sub-actions, then the first will be treated as a reply and any others as asynchronous events to the caller action's object. Additional information would need to be kept in the automated implementation to express when a reply is made. Also, if the return sub-action is not the last one in the sequence, then we would need to modify the schedulability analysis to include this situation. Particularly, we could consider such a synchronously-triggered actions as two actions; the first, considered as a synchronously-triggered action, would contain all sub-actions up to and including the first return sub-action and the second, considered as an asynchronously-triggered action, would contain the remaining sub-action sequence. At this point, response times could be calculated using the same formula as described in this thesis.

If we assume that synchronously-triggered actions can have a priority different from that of its caller action, several complications arise. Making a synchronous event have lower priority would disturb the entire notion of a synchronous event given that our scheduling policy conforms to preemptive priority scheduling. On the other hand, making the synchronous event have higher priority could be possible and, in this case, a schedulability analysis even more complicated than the one with transaction priorities extensions (as mentioned in Section 5.1.2) would be required.

In addition, abolishing the restriction that a synchronously-triggered action must execute in the same thread as its caller action would necessarily complicate the system. First, in our automated implementation, we would need to include blocking code for actions that are waiting for a reply when it synchronously triggers an action that executes in another thread. This ensures that no action can start executing in a thread that just generated a synchronous event to another thread. Also, this same stipulation must be taken into account when considering schedulability analysis. Specifically, the interference effect for a synchronously-triggered action could be smaller because it will not include any action executing in the same thread as the caller action. Also, the blocking effect would be severely modified, because an action A_k may have been previously running in some thread when another action A_i interrupts its execution. This action A_i could possibly generate a synchronous event destined for the same thread that A_k is executing on. In this case, unbounded priority inversion arises that can be remedied by redefining the preemption threshold priority assignment.

Bibliography

[AkZ96] M. Awad, J. kuusela, and J. Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach using OMT and Fusion.* Prentice Hall, 1996.

[Aud91] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS 164, Department of Computer Science, University of York, England, December 1991.

[BAW97] A. Burns, N. Audsley, and A. Wellings. On the schedulability analysis for distributed real-time systems. In *Proceedings of Euromicro Conference on Real-Time Systems*, pages 136–143, 1997.

[BB98] I. Bate and A. Burns. Investigation of the pessimism in distributed systems timing analysis. In *Proceedings of the 10th Euromicro Workshop on Real Time Systems, Berlin, Germany*, pages 107–114, June 1998.

[Bel98] R. Bell. Code generation from object models. *Embedded Systems Programming*, 11(3), March 1998.

[BF96] A. Bertossi and A. Fusiello. Rate-monotonic scheduling for hard-real-time systems. *European Journal of Operational Research*, pages 429–443, June 1996.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[BW94] A. Burns and A. J. Wellings. Hrt-hood: A design method for hard real-time. *Real-Time Systems*, 6(1):73–114, 1994.

[DL78] S. Dhall and C.L. Liu. On a real-time scheduling problem. *Operations Research*, pages 127–141, September 1978.

[Fid98] C.J. Fidge. Real-time schedulability tests for preemptive multitasking. *The Journal of Real-Time Systems*, pages 61–93, 1998.

[GGH97] J. Gutierrez, J. Garcia, and M. Harbour. On the schedulability analysis for distributed real-time systems. In *Proceedings of Euromicro Conference on Real-Time Systems*, pages 136–143, 1997.

[GH98] J. Gutierrez and M. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *RTSS*, December 1998.

[Gom93] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley Publishing Company, 1993.

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uni-processor scheduling. Technical Report N^o 2966, INRIA, France, September 1996.

[Har87] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 1987.

[HKL91] M. Harbour, M. Klein, and J. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In *Proceedings, IEEE Real-Time Systems Symposium*, pages 116–128, December 1991.

[HLR96] J.-F. Hermant, L. Leboucher, and N. Rivierre. Real-time fixed and dynamic priority driven scheduling algorithms: Theory and experience. Technical Report N^o 3081, INRIA, France, December 1996.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, pages 390–395, 1986.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[KRP$^+$93] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer Academic Publishers, 1993.

[Leh90] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 201–209. IEEE Computer Society Press, December 1990.

[LL73] C. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[Loc92] C. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems (Netherlands)*, 4:37–53, March 1992.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society Press, December 1989.

[LW82] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation (Netherlands)*, 2:237–250, 1982.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[Rod98] P. Rodziewicz. Timing and scheduling analysis of real-time object-oriented models. Technical report, Department of Computer Science, Concordia University, August 1998.

[SGW94] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[SKW00] M. Saksena, P. Karvelas, and Y. Wang. Automatic synthesis of multi-tasking implementations from real-time object-oriented models. In *Proceedings of IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, March 2000.

[SM96] S. Shlaer and S. Mellor. The shlaer-mellor method. Technical Paper, Project Technology, Inc. 1996. Available from http://www.projtech.com, 1996.

[SM97] S. Shlaer and S. J. Mellor. Recursive design of an application-independent architecture. *IEEE Software*, 14(1), January 1997.

[SR94] K. Shin and P. Ramanathan. Real-time computing: A new discipline of computer science and engineering. In *IEEE Proceedings*, pages 6–23. IEEE Computer Society Press, January 1994.

[SR98] B. Selic and J. Rumbaugh. Using uml for modeling complex real-time systems. White Paper, Published by ObjecTime and available from www.objectime.com, March 1998.

[SRL90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, September 1990.

[SS99] M. Saksena and B. Selic. Real-time software design: State of the art and future challenges. *IEEE Canadian Review*, 32:5–8, Summer 1999.

[SW00] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proceedings of IEEE Real-Time Systems Symposium*, December 2000.

[TBW94] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *The Journal of Real-Time Systems*, 6(2):133–152, March 1994.

[Tin93] K. Tindell. Holistic schedulability analysis for distributed hard real-time systems. *Department of Computer Science, University of York*, 1993.

[Wan00] Y. Wang. Scheduling fixed priority tasks with preemption threshold and its application in automatic synthesis of object-oriented real-time systems. Technical report, Department of Computer Science, Concordia University, August 2000.

[WS99] Y. Wang and M. Saksena. Fixed priority scheduling with preemption threshold. In *Proceedings, IEEE International Conference on Real-Time Computing Systems and Applications*, December 1999.

Appendix A

Application-View Model Grammar

```
%start applicationModelSpec

applicationModelSpec
   : 'Application Model'
     '{' classList objectList
     bindingList optExternalEvents '}'
   ;

optExternalEvents
   : /* empty */
   | 'External Events' '{' externalEventList
      externalEventSpec '}'
   ;

externalEventList
   : /* empty */
   | externalEventList externalEventSpec ';'
   ;

externalEventSpec
   : eventName optEventArrivalPattern
     startTime recipientObject
   ;

optEventArrivalPattern
   : /* empty */
   | 'One-shot'
   | periodicArrival
   ;

periodicArrival
```

```
  : 'Period' '=' INTEGER
  ;

recipientObject
  : objectName
  ;

startTime
  : 'Start' '=' INTEGER
  ;

classList
  : /* empty */
  | classList classSpec
  ;

objectList
  : /* empty */
  | objectList objectSpec ';'
  ;

objectSpec
  : 'Object' objectName ':' className
  ;

bindingList
  : /* empty */
  | bindingList bindingSpec ';'
  ;

bindingSpec
  : 'Bind' fullPortName '=' fullPortName
  ;

fullPortName
  : objectName '.' portName
  ;

classSpec
  : 'Class' className '{'
    optClassAttributes optClassMethods
    optPorts optSignalEvents optCallEvents optFsmSpec '}'
  ;
```

```
optPorts
   : /* empty */
   | 'Ports' '{' portList portSpec '}'
   ;

portList
   : /* empty */
   | portList portSpec ';'
   ;

portSpec
   : portName
   ;

portName
   : NAME
   ;

optCallEvents
   : /* empty */
   | 'Call Events' '{' callEventList callEventSpec '}'
   ;

callEventList
   : /* empty */
   | callEventList callEventSpec ';'
   ;

callEventSpec
   : eventName
   ;

optFsmSpec
   : /* empty */
   | 'FSM' '{' optStates initialTransition optTransitions '}'
   ;

initialTransition
   : 'Initial Transition' 'to' targetState optAction
   ;

optAction
```

```
    : /* empty */
    | '/' actionList
    ;

optStates
    : /* empty */
    | 'States' '{' stateList stateSpec '}'
    ;

stateList
    : /* empty */
    | stateList stateSpec ';'
    ;

stateSpec
    : stateName optEntryAction optExitAction
    ;

eventName
    : NAME
    ;

optEntryAction
    : /* empty */
    | 'Entry' '/' actionList
    ;

optExitAction
    : /* empty */
    | 'Exit' '/' actionList
    ;

optTransitions
    : /* empty */
    | 'Transitions' '{' transitionList transitionSpec'}'
    ;

transitionList
    : /* empty */
    | transitionList transitionSpec ';'
    ;

transitionSpec
```

```
    : transitionName 'from' sourceState 'to' targetState
      eventName optAction
    ;

stateName
    : NAME
    ;

transitionName
    : NAME
    ;

sourceState
    : stateName
    ;

targetState
    : stateName
    ;

optSignalEvents
    : /* empty */
      | 'Signal Events' '{' signalList signalSpec '}'
    ;

signalList
    : /* empty */
    | signalList signalSpec ';'
    ;

signalSpec
    : eventName
    ;

type
    : className optPointer
    | defaultType optPointer
    ;

defaultType
    : 'void'
    | 'int'
    | 'char'
```

```
   | 'double'
   | 'float'
   | 'enum'
   | 'long'
   | 'short'
   ;

optPointer
   : '*'
   | '*' '*'
   ;

optAbstract
   : /* empty */
   | 'Abstract'
   ;

className
   : NAME
   ;

optClassAttributes
   : /* empty */
   | 'Attributes' '{' attributeList '}'
   ;

attributeList
   : /* empty */
   | attributeList attributeSpec ';'
   ;

attributeSpec
   : optVisibility optScope attributeName
     optAttributeType optInitialDefaultValue
   ;

optInitialDefaultValue
   : /* empty */
   | '=' expression
   ;

expression
   : NAME
```

```
    | INTEGER
    | FLOAT
    | STRING
    | CHAR
    | boolean
    ;

boolean
    : TRUE
    | FALSE
    ;

optVisibility
    : /* empty */
    | 'public'
    | 'protected'
    | 'private'
    ;

optScope
    : /* empty */
    | 'scope instance'
    | 'scope classifier'
    ;

attributeName
    : NAME
    ;

optAttributeType
    : /* empty */
    | ':' type
    ;

optClassMethods
    : /* empty */
    | 'Methods' '{' methodList '}'
    ;

methodList
    : /* empty */
    | methodList methodSpec ';'
    ;
```

```
methodSpec
   : optVisibility optScope optAbstract methodName
     optMethodParameters optReturnType optMethodCode
   ;

optReturnType
   : /* empty */
   | ':' type
   ;

optMethodCode
   : /* empty */
   | '{' actionList '}'
   ;

methodName
   : NAME
   ;

optMethodParameters
   : /* empty */
   | '(' methodParameterList methodParameterSpec ')'
   ;

methodParameterList
   : /* empty */
   | methodParameterList methodParameterSpec ';'
   ;

methodParameterSpec
   : optDirection attributeName
     optAttributeType optInitialDefaultValue
   ;

optDirection
   : /* empty */
   | 'in'
   | 'out'
   | 'inout'
   ;

actionList
```

```
   : /* empty */
   | actionList actionSpec
   ;

actionSpec
   : 'Action' actionName '{' subActionList subActionSpec '}'
   ;

actionName
   : NAME
   ;

subActionList
   : /* empty */
   | subActionList subActionSpec ';'
   ;

subActionSpec
   : assignmentAction
   | requestAction
   | returnAction
   | terminateAction
   | uninterpretedAction
   ;

assignmentAction
   : variable ':=' expression
   ;

variable
   : NAME
   ;

objectName
   : optObjectPath simpleObjectName
   ;

optObjectPath
   : /*empty*/
   | 'PATH' className ':'
   ;

simpleObjectName
```

```
  : NAME
  ;

returnAction
  : 'return' expression
  ;

requestName
  : NAME
  ;

requestAction
  : 'call' requestName 'to' portName
  | 'send' requestName 'to' portName
  ;

terminateAction
  : 'terminate'
  ;

uninterpretedAction
  : STRING
  ;
```

Appendix B

Implementation-View Model Grammar

```
%start implementationModelSpec

implementationModelSpec
  : 'Implementation Model' '{' threads mapping '}'
  ;

threads
  : 'Threads' '{' threadList threadSpec '}'
  ;

threadList
  : /* empty */
  | threadList threadSpec ';'
  ;

mapping
  : 'Mapping' '{' mappingList mappingSpec '}'
  ;

mappingList
  : /* empty */
  | mappingList mappingSpec ';'
  ;

mappingSpec
  : threadName '=' ActionIdentifier
    'at' '(' schedulingAttributes ')'
  ;

schedulingAttributes
  : whatPriority optWhatPreemptionThresholdPriority
  ;

ActionIdentifier
  : '(' sendObjectName ',' recvObjectName ',' eventName ')'
  ;

sendObjectName
  : objectName
  ;

recvObjectName
  : objectName
  ;

objectName
  : optObjectPath simpleObjectName
  ;

optObjectPath
  : /*empty*/
  | 'PATH' className ':'
  ;

className
  : NAME
  ;

simpleObjectName
  : NAME
  ;

eventName
  : NAME
  ;

threadSpec
  : threadName
  ;

threadName
  : NAME
  ;
```

```
whatPriority
   : 'Nominal Priority' '=' INTEGER
   ;

optWhatPreemptionThresholdPriority
   : /*empty*/
   | ',' 'Preemption Threshold Priority' '=' INTEGER
   ;
```

Appendix C

Schedulability Analysis Model Grammar

```
%start analysisSpec

analysisSpec
  : 'Analysis' '{' threads objects transactions actions '}'
  ;

threads
  : 'Threads' '{' thread_list thread_desc '}'
  ;

thread_list
  : /*empty*/
  | thread_list thread_desc ','
  ;

thread_desc
  : NAME
  ;

objects
  : 'Objects' '{' object_list object_desc '}'
  ;

object_list
  : /*empty*/
  | object_list object_desc ','
  ;
```

```
object_desc
  : NAME
  ;

transactions
  : 'Transactions' '{' transaction_list transaction_desc '}'
  ;

transaction_list
  : /*empty*/
  | transaction_list transaction_desc
  ;

transaction_desc
  : NAME ':' period_expr ',' init_action_expr
  ;

period_expr
  : 'period' '=' INTEGER
  ;

init_action_expr
  : 'initial action' '=' actionName
  ;

actions
  : 'Actions' '{' action_list action_desc '}'
  ;

action_list
  : /*empty*/
  | action_list action_desc
  ;

action_desc
  : NAME ':' priority_expr ',' optDeadline_expr
             thread_expr ',' object_expr ','
             subactions
  ;

subactions
  : 'SubActions' '=' '{' subaction_list subaction_desc '}'
  ;
```

```
subaction_list
  : /*empty*/
  | subaction_list subaction_desc
  ;

subaction_desc
  : NAME ':' subactionType_expr
  ;

subactionType_expr
  : 'send' actionName ',' cost_expr /* Asynch Event */
  | 'call' actionName ',' cost_expr /* Synch Event */
  | 'return' ',' cost_expr
  | 'uninterpreted' ',' cost_expr
  ;

actionName
  : NAME
  ;

cost_expr
  : 'cost' '=' INTEGER
  ;

priority_expr
  : 'priority' '=' INTEGER
  ;

optDeadline_expr
  : /*empty*/
  | 'deadline' '=' INTEGER ','
  ;

thread_expr
  : 'thread' '=' NAME
  ;

object_expr
  : 'object' '=' NAME
  ;
```

Appendix D

Response Times Calculations: Single-Threaded Implementation

Transaction	Action	Instance	Arrival	Finish	End-to-End Response Time
T_1					753
	A_1	1	0	268	268
		2	60	286	226
		3	120	304	184
		4	180	322	142
		5	240	340	100
		6	300	358	58
	A_4	1	0	753	753
		2	60	758	698
		3	120	763	643
		4	180	768	588
		5	240	773	533
		6	300	778	478
		7	360	783	423
		8	420	806	386
		9	480	811	331
		10	540	816	276
		11	600	821	221
		12	660	826	166
		13	720	831	111
		14	780	836	56
	A_5	1	0	261	261
		2	60	279	219
		3	120	297	177
		4	180	315	135
		5	240	333	93
		6	300	351	51
	A_6	1	0	255	255
		2	60	273	213
		3	120	291	171
		4	180	309	129
		5	240	327	87
		6	300	345	45
T_2					603
	A_2	1	0	429	429
		2	300	536	236
	A_7	1	0	368	368
		2	300	475	175
	A_8	1	0	603	603
		2	300	720	420
		3	600	748	148
	A_9	1	0	409	409
		2	300	516	216
T_3					593
	A_3	1	0	593	593
	A_A	1	0	583	583
	A_B	1	0	421	421

Table 5: Single-Threaded Response Time Calculations: Full Results

Appendix E

Response Times Calculations: Multi-Threaded Implementation

Transaction	Action	Instance	Arrival	Finish	End-to-End Response Time
T_1					161
	A_1	1	0	28	28
	A_4	1	0	161	161
		2	60	166	106
		3	120	171	51
	A_5	1	0	21	21
	A_6	1	0	15	15
T_2					161
	A_2	1	0	67	67
	A_7	1	0	49	49
	A_8	1	0	161	161
	A_9	1	0	108	108
T_3					582
	A_3	1	0	145	145
	A_A	1	0	146	146
	A_B	1	0	582	582

Table 6: Multi-Threaded Response Time Calculations: Full Results