INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

NOTE TO USERS

This reproduction is the best copy available

UMI

TIMING AND SCHEDULING ANALYSIS OF REAL-TIME OBJECT-ORIENTED MODELS

PAWEŁ RODZIEWICZ

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

SEPTEMBER 1998
© PAWEŁ RODZIEWICZ, 1998

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-39491-3

Canadä

NOTE TO USERS

Page(s) not included in the original manuscript are unavailable from the author or university. The manuscript was microfilmed as received.

ii

This reproduction is the best copy available.

UMI

Abstract

Timing and Scheduling Analysis of Real-Time Object-Oriented Models

Paweł Rodziewicz

The increasing complexity of real-time software has led to a recent trend in the use of high-level modelling languages for the development of real-time software. One representative example is the Real-Time Object Oriented Modeling (ROOM) language, which provides features such as encapsulation, polymorphism, inheritance, state machine descriptions of system behavior, formal semantics for executability of models and the possibility of automated code generation. The full benefits of the ROOM language are obtained through the use of the ObjecTime toolset, designed to support the ROOM language and its development process in order to automatically create an executable for a target platform equipped with a real-time operating system. However, the ROOM language and the ObjecTime toolset largely ignore the temporal aspects of real-time systems, and fail to provide any guidance to the designer about predicting and analyzing the temporal behavior of their intended applications.

The main objective of this thesis is to develop ways to perform such timing and scheduling analysis for single- and multi-threaded ROOM models. This work builds on results presented in [SFR97] and [SPFR98], where guidelines for the design and implementation of real-time object oriented (ROOM) models were developed and tested.

In this thesis, we shall consider three orthogonal timing analysis methods: real-time scheduling theory, symbolic model checking and discreet task simulation. We formulate design guidelines for single- and multi-threaded ROOM executables. Then we develop and validate canonical scheduling models for each type of executable. Finally, we present a case study of an automobile cruise control system to further illustrate the concepts presented in this thesis.

Acknowledgments

The two most important people who helped me to put this thesis together are Manas Saksena and Paul Freedman. Thanks to their incredible efforts, this thesis became something of which I can be proud. I want to thank Manas for his invaluable comments, constant guidance, and for his tremendous theoretical contributions reflected in his recent RTSS publications. Without his help, I would have needed a decade to come up with the results presented in this thesis. Great thanks go to Paul Freedman, whose insistence on getting the structure right, while chasing the details, went a long way to make this thesis a reality. His constant support and willingness to prolong the CRIM financial support carried me through during the last three years. I would like to thank every member of my thesis committee for providing me with a tremendous amount of feedback during the defense.

Secondly, thanks galore go to my friends at Concordia University and CRIM. Balázs Kégl, my tennis and skiing partner, who kept me company during summer and winter weekends. During the last month he lodged me at his place to help me finish my thesis without leaving Montréal, thanks Balázs. Next, I would like to thank Paul MacKenzie, who kept things interesting with his karate stories; Alan Ptak, a co-author of one of our RTSS publications; Nathalie Drouin and Judith Bracke for their administrative support – both of whom were my first-class french mentors; Jean-François Lapointe, my ex-co-worker; and Emna Menif, who sacrificed almost a week to help me with the SOLA project at CRIM. Space does not permit me to list all the nice people I met at Concordia and CRIM, but a small sampling includes: John Linton, Serge Marelli, Virginie Pierrot, Roland Younes, Tarlok Birdi, Shawn Delaney, Anna Czubik, Greg Bobrowski, Wojtek Ptaszyński and Yves Gonthier.

Specjalne podziękowania przesyłam mojej rodzinie w Polsce, szczególnie mojemu tacie i babci, oraz mojej rodzinie w Toronto, która mnie motywowała do dalszego studiowania, a później dodawała mi siły i pewności siebie przez cały okres mojego pobytu w Montrealu.

And to my girlfriend, Amalita Rey, who went through a lot of heartaches to help me

obtain my skiing certificate. Her support and understanding carried me through. Muchas gracias Amalita.

Contents

List of Figures ix

List of Tables x

1 Introduction 1

1.1 Contributions 2

1.2 Related Work 4

1.3 Thesis Outline 5

2 Real-Time Object Oriented Modeling 7

2.1 Overview of ROOM Concepts 9

2.1.1 Structural Model 9

2.1.2 Behavioral Model 12

2.2 Message Sequence Charts and Transactions 12

2.3 Target Run-Time System 14

3 Choosing a Scheduling Analysis Method for ROOM Models 16

3.1 Overview of Real-Time Scheduling Theory 17

3.1.1 Worst Case Response Time 17

3.1.2 Release Jitter 18

3.1.3 Time and Space Complexity 19

3.2 Overview of Symbolic Model Checking 20

3.2.1 Timing and Scheduling Analysis of Tasks 20

3.3 Overview of Discrete Task Simulation 23

3.4 Empirical Complexity Results 23

3.4.1 Independent Tasks 24

3.4.2 Tasks with Release Jitter 26
3.5 Conclusions 28

4 **ROOM Design Guidelines for Hard Real-Time Systems** 29
4.1 Assigning Actors to Threads 30
4.1.1 Single-Threaded Executable 30
4.1.2 Multi-Threaded Executable 31
4.2 Assigning Message Priorities 34
4.3 Blocking due to Message Passing 35
4.4 Blocking due to Run-to-Completion Semantics 36

5 **Developing Canonical Scheduling Models for ROOM** 38
5.1 Generic Resource Scheduling Model 38
5.1.1 Generic Overhead for Higher Priority Transactions 40
5.1.2 Generic Blocking 41
5.1.3 Generic Overhead 42
5.2 Single-Threaded Executable 42
5.2.1 Specific Overhead for Higher Priority Transactions 42
5.2.2 Specific Blocking 43
5.2.3 Specific Overhead 43
5.2.4 Canonical Scheduling Model 43
5.3 Multi-Threaded Executable 44
5.3.1 Specific Overhead for Higher Priority Transactions 44
5.3.2 Specific Blocking 44
5.3.3 Specific Overhead 45
5.3.4 Canonical Scheduling Model 45
5.4 Validating Canonical Scheduling Models 45
5.4.1 First ROOM Model 47
5.4.2 Second ROOM Model 48

6 **Case Study of Automobile Cruise Control** 50
6.1 Structural Model 51
6.2 Behavioral Model 52
6.2.1 Description of Data Processing Actors 52
6.2.2 Description of the Cruise Control Actor 54

6.3 Transactions and Timing Constraints 57
6.3.1 Closed-Loop Feedback Control 57
6.3.2 Entering and Preparing for Automatic Cruise Control 59
6.3.3 Exiting Automatic Cruise Control 59
6.4 Schedulability Analysis 60
6.4.1 Response Time Analysis 62
6.4.2 Discussion 63

7 Conclusions **65**
7.1 Thesis Summary 65
7.2 Future Work 66

Bibliography **68**

List of Figures

1 Computer Actor 10
2 Functional Aggregate: System Actor 10
3 Functional Aggregate: Computer Actor 11
4 Message Sequence Chart for Printing 14
5 Schedulability Analysis with SMV toolset 21
6 FSM Model of a Task 22
7 SMV Execution Time for Three Independent Tasks 25
8 SMV Execution Time for Tasks with Release Jitter 27
9 ROOM Transaction Tree 34
10 Cruise Control Structural Model 52
11 Accelerator, Brake and Lever Actor Behavior 53
12 Throttle Actor Behavior 53
13 Speedometer Actor Behavior 54
14 Top-Level Behavior of Cruise Control Actor 55
15 Automatic Control Behavior of Cruise Control Actor 56
16 Resume Cruising Control Behavior 56
17 Manual Control Behavior of Cruise Control Actor 57
18 Message Sequence Chart for ControlLoop Transaction 58
19 Message Sequence Chart for EnterCruise Transaction 60
20 Message Sequence Chart for BrakePressed Transaction 61

List of Tables

1 Temporal Characteristics of Three Independent Tasks 24
2 Empirical Results for Three Independent Tasks 24
3 Temporal Characteristics of Tasks with Release Jitter 26
4 Empirical Results for Tasks with Release Jitter 26
5 Scheduling Model Characteristics 28
6 Implementation Overheads 46
7 Transaction Specifications for the First ROOM Model 47
8 Transition Specifications for the Second ROOM Model 48
9 Transaction Specifications for the Second ROOM Model 48
10 Description of Automobile Cruise Control Transactions 58
11 Transition Computation Times 60
12 Specifications of Automobile Cruise Control Transactions 63

Chapter 1

Introduction

With the rapid advances in computing and communications technology, computerized control is being widely employed in many real-time control applications. Digital control systems implemented in software have many advantages: they can perform complex control operations in a responsive manner, and can dynamically adapt to changes in the operating environment through the use of appropriate control structures. The increasing complexity and the sophisticated demands of such systems in terms of safety, reliability, and performance [Boa93] requires the use of rigorous methodologies, to ensure reliable software development, along with analytical techniques to analyze and predict the temporal behavior of applications.

It is not surprising that there is increasing interest in new generation methodologies and CASE tools which make possible the modelling and analysis of computing systems in terms of executable models [HG96]. One such product is the modelling language "ROOM" (Real-Time Object Oriented Modeling) [SGW94] and its CASE tool, "ObjecTime", which grew out of more than ten years of internal development at Bell-Northern Research (now Nortel). In addition to the analysis made possible by the simulation of executable models, ObjecTime also provides a highly integrated software development environment and code generation capabilities designed to bring some automation to the development process. Indeed, we observe that the telecommunications community has widely accepted such new generation of CASE tool support, e.g. ROOM [SGW94] and OCTOPUS [AKZ96]. We believe that such acceptance is due, in part, to the sheer size and complexity of new telecommunication products and their fundamentally event-driven nature.

In contrast, the embedded control systems community continues to be much more "conservative". Once again this is due, in part, to the smaller size and complexity of these products and their dominating time-driven behavior. In the time-driven context, programming features, such as the task construct in Ada95, were developed to handle periodic behavior of embedded control systems along with real-time scheduling theory [KRP+93] which provides guidance, at design time, about predicting the time-driven behavior of such systems. Nevertheless, as embedded control systems become more complex in terms of their reactive behavior, they become harder to implement with traditional programming features, such as `case` and `if ... else` programming constructs. Because ROOM/ObjecTime supports both time- and event-driven styles of software, it is particularly attractive as a potential development methodology for embedded control systems, so long as appropriate timing and scheduling analysis methods are developed.

1.1 Contributions

This thesis presents new results to help provide a priori timing and scheduling analysis of real-time object oriented (ROOM) models. To perform this analysis, timing constraints are imposed on end-to-end computations, i.e. deadlines are imposed on ROOM transactions which are formally defined in Chapter 2, prior to system design and development. For example, these timing constraints specify how quickly an automobile cruise control system should be disabled when a break or accelerator pedal is pressed or how often a closed-loop feedback control should be executed in order to maintain the speed of the automobile at the desired cruising speed.

In our work, we assume that the timing constraints are already known. The results presented in this thesis enable the software designer to test a ROOM model of a real-time system against its timing constraints before its actual implementation. Although a general structure of the ROOM model (actors, message sequence charts) must be completed as well as transaction execution times must be estimated prior to performing the scheduling analysis, the designer can detect timing problems early in the development cycle. If some of the timing constraints are not met, either the ROOM model or the timing requirements should be modified.

While our work was executed within the context of a specific software development methodology, i.e. ROOM, we believe that our results and observations are more generally

applicable. In particular, the results developed here are equally applicable to the Unified Modeling Language (UML) developed by a consortium of companies, including Rational, i-Logix and ObjecTime, and recently standardized by Object Management Group (OMG). Indeed, a recent white paper [SR98] from ObjecTime and Rational states that "Rational and ObjecTime are defining a comprehensive approach for the application of UML to the development of complex real-time systems", in which modelling constructs defined in the ROOM language are specified using the UML standard. Finally, we believe that our results can also be applied to Statemate from i-Logix (`http://www.ilogix.com`), the Libero toolset from iMatix (`http://www.imatix.com/html/libero/index.htm`) and ObjectGeode from Verilog (`http://www.verilogusa.com/og/og.htm`), or any other finite-state machine (FSM) based CASE toolset. Let us summarize the main contributions of this thesis.

1. Selection of an appropriate timing and scheduling method for ROOM models.
   (a) Overview of real-time scheduling theory, symbolic model checking and discreet task simulation.
   (b) Comparison of their time and space complexity results.
2. ROOM design and implementation guidelines.
   (a) Overview and comparison of single- and multi-threaded implementations of ROOM models.
   (b) Specification of a suitable thread priority assignment for single- and multi-threaded ROOM executables.
   (c) Specification of a desirable message priority assignment.
   (d) Bounding priority inversion of transactions due to inter-thread message passing mechanism.
   (e) Definition of an explicit rule which allows adding new functionality to an actor, without affecting the schedulability of the higher priority transactions associated with that actor.
3. ROOM scheduling models.
   (a) Development of a generic resource scheduling model.

(b) Specification and measurements of implementation cost parameters for the single- and multi-threaded ROOM executables.

(c) Development and validation of canonical resource scheduling models for the single- and multi-threaded ROOM executables.

1.2 Related Work

In 1995-1996 work began at the Centre de recherche informatique de Montréal (CRIM) to investigate the suitability of ROOM/ObjecTime and its software development automation for the development of embedded control systems, in cooperation with Bombardier's Transportation Equipment Group, one of the North America's leading mass transport companies. In [GF96], Gaudreau and Freedman described how ROOM models could be subjected to Generalized Rate Monotonic Analysis (GRMA) [KRP+93], using a simplified cruise control example adopted from [Gom93]. They wanted to promote GRMA as a method for the temporal analysis of ROOM models at design time, to the software development team. Although, their work only applies to ROOM models with static thread priority assignments and it does not take implementation costs such as context-switch between threads into consideration, it was a solid starting point for the follow-up research.

In September 1996, another project began at CRIM jointly founded by Bombardier and the Natural Sciences and Engineering Research Council (NSERC) to include the University of Sherbrooke and Concordia University. As a result of CRIM-Concordia collaboration, another paper [SFR97] was published. In [SFR97], Saksena *et al.* showed that it is possible to perform scheduling analysis on ROOM models, based on the key insight that even though a given application may be processing many external messages in a state-dependent manner, only a relatively small number of transactions [1] are time-critical and need to be analyzed for timeliness. The authors presented implementation guidelines that minimize priority inversion of transactions due to lower priority transactions by managing thread priorities dynamically, bound priority inversion of transactions due to inter-thread message passing mechanism by using the Immediate Inheritance protocol [SRL90], and allow adding new functionality to an actor [2] without affecting the schedulability of higher priority transactions associated with that actor. Also, another closely related paper [SPFR98], was written

[1] A transaction is a chained sequence of transition actions triggered by an external incoming (stimulus) message and will be formally defined in Chapter 2.

[2] Actors are the primary structural element in ROOM and will be also described in Chapter 2.

as part of continuing CRIM-Concordia collaborative efforts which presented experimental evidence of the soundness of the ideas described in [SFR97]. To do that, the ObjecTime run-time system (TargetRTS) was modified in accordance with the guidelines presented in [SFR97], a schedulability analysis model was developed taking into account implementation costs of TargetRTS, and the approach was validated using a simple instrumented ROOM model implementing a set of periodic transactions through measurements. To illustrate the generality of the scheduling analysis model, a case study was presented of a train tilting system adopted from earlier work performed at CRIM with Bombardier's Transportation Equipment Group. The train tilting system represented a realistic computerized control system and illustrated the scalability of the approach. Note that previous work on the application of ROOM/ObjecTime for the development of the train tilting system [Fre98] had revealed serious timing problems associated with TargetRTS and the use of static thread priorities.

1.3 Thesis Outline

This thesis contains seven chapters. This chapter provides an overview of the problems associated with real-time embedded control applications, in particular the need for rigorous development methodologies, for reliable software development, along with analytical techniques to analyze and predict the temporal behavior of such applications. We mentioned one such methodology (ROOM) along with its ObjecTime CASE tool, and explained why it is widely accepted by telecommunications industry and why its use could be encouraged within the more "conservative" embedded control systems industry.

Chapter 2 reviews the advantages of using the ROOM methodology and ObjecTime toolset for the development of real-time computerized control systems over more traditional approaches such as cyclic executives or task models. Then we present an overview of two ROOM specification levels: the schematic level, subdivided into the structural model and behavioral model, and the detailed level. The chapter formalizes the notion of transactions and finally describes ObjecTime's TargetRTS.

In **Chapter 3**, we examine and select a suitable timing and scheduling analysis method for ROOM models among three orthogonal timing analysis methods: real-time scheduling theory, symbolic model checking and discrete task simulation. In this chapter, we only consider schedulability tests for fixed priority algorithms, as these are the most relevant to

the ROOM models presented in this thesis. We also present an overview of each method, followed by a comparison of their time and space complexity results.

Chapter 4 develops guidelines to apply the real-time scheduling theory presented in the previous chapter to ROOM models. As we shall see, the biggest challenge comes from the various sources of priority inversion that can result in large and possible unbounded blocking times. In this chapter, we describe single- and multi-threaded implementations of ROOM models and we compare their relative advantages and disadvantages. For each implementation, a guideline is formulated which specifies a suitable thread priority assignment. An additional guideline then defines a desirable message priority assignment. Finally, we present implementation guidelines that bound priority inversion of transactions due to inter-thread message passing mechanism and that allow adding new functionality to an actor without affecting schedulability of the higher priority transactions associated with that actor.

In **Chapter 5**, we develop a generic resource scheduling model for the scheduling analysis of ROOM models that includes the overhead and blocking costs of `TargetRTS` and the underlying real-time operating system (RTOS) scheduler. Then we develop the specific resource scheduling models for the single- and multi-threaded ROOM executables and finally we validate them, using a simple instrumented ROOM model through measurements.

Chapter 6 illustrates the concepts developed in earlier chapters by studying a variant of the automobile cruise control system [3] presented in [GF96]. The chapter provides an overview of the cruise control system, specifies its time-critical transactions and presents schedulability analysis results for these transactions.

Finally, **Chapter 7** summarizes the most significant results of this thesis and identifies possible topics for future research.

[3]Automobile cruise control is a well studied example which has been used to illustrate other real-time design methods such as Octopus, ADARTS and CODARTS.

Chapter 2

Real-Time Object Oriented Modeling

Developing real-time computerized control systems, and especially embedded systems, is particularly challenging since the complexity of the physical environment has to be accommodated along with stringent time constraints. Until recently, the majority of real-time applications have been developed with the *cyclic executive* model [Loc92], [BS89]. With this model, real-time software engineers are forced to fit their computational tasks, i.e. concurrent sets of sequential actions which can take place within a system, into a predetermined cycle. The cycle is repeated once per period and is called the *major cycle*. The major cycle is in turn subdivided into a smaller period, called the *minor cycle*. This model imposes several restrictions in terms of the development process of real-time applications. The developer is burdened with fitting tasks into the minor cycle segments, sacrificing code clarity, and making it more difficult to develop and maintain the software. Timing correctness for the cyclic executive model is determined statically by the design of the timeline. Clearly, there is a "tight coupling" between timing analysis and the design of the task set.

The increasing complexity and the sophisticated demands of real-time systems in terms of development time, software maintenance and reliability gave rise to real-time scheduling theory [LL73], which decouples the design of a task set from its timing or scheduling analysis. With this method, real-time designers can develop a time-critical system as a set of tasks, and then use a mathematical *feasibility* test to examine timing aspects of the task set, such as comparing CPU utilization to its allowable upper bound, or comparing each individual task's worst case response time to its deadline.

Although, the use of real-time scheduling theory has greatly improved development time and maintenance of real-time applications, there is still a need for a formal design

method. In the ROOM tutorial [Sel96], Selic presents arguments for using ROOM methodology along with its CASE toolset, ObjecTime, during design, implementation, maintenance and evolutionary development phases of real-time applications. To do that, he defines the term *architectural decay*, the phenomenon of successive deterioration of system architecture during software implementation and maintenance phases. Because architectures are typically specified informally using block diagrams supplemented by prose, such specifications are often misunderstood or even neglected by real-time application programmers. This is amplified during the maintenance phase, which is often done by the least experienced professionals who, faced with enormous amount of code, again, tend to disregard high level concepts of real-time applications. Later on, we will describe how ROOM addresses the *architectural decay* problem.

In a recent article entitled "The Challenges of Real-Time Software Design" [SW96], Selic and Ward describe two basic "styles" of real-time software. The *time-driven* style corresponds to using cyclic activities triggered by time and is well suited to the implementation of periodic activities, e.g. control loops within embedded systems. In contrast, the *event-driven* software typically waits for an external event to occur (for a message to arrive), performs an appropriate action, and enters another state waiting for the next event. This style is well suited to the implementation of reactive behavior, associated with unpredictable, discreet events which may occur in a non-deterministic way, e.g. sporadic operator input, component failure, in the external system.

Various techniques such as rate-monotonic and deadline-monotonic scheduling, were developed by real-time scheduling theory pioneers to perform timing and scheduling analysis of time-driven real-time systems. Unfortunately, little attention has been paid to the integration of scheduling analysis into event-driven systems. Until now, these systems were targeted to deal with the complexity arising from the asynchronous and concurrent nature of event-driven applications, neglecting its temporal characteristics. Because ROOM supports both styles of real-time software, it appears to be a very attractive formal method for the development of increasingly complex real-time systems. In the following sections, we will describe the concepts used by ROOM to facilitate the implementation of reactive systems (event-driven systems), and in Chapter 5 we will develop canonical scheduling models to perform timing and scheduling analysis of ROOM models.

2.1 Overview of ROOM Concepts

To address the *architectural decay* problem, the ROOM language defines system specifications at two distinct, but formally related levels. The upper level, called the *schematic level*, is used to specify the architectural aspects of the system and is subdivided into the *structural* model and *behavioral* model. The lower level, called the *detailed level*, is used to define the finer grained implementation detail. The detailed level specifications are then passed to the C++ programming language code generator. An important characteristic of ROOM is that it is an object-oriented modelling language, implementing encapsulation, polymorphism and inheritance at both levels of its specifications.

2.1.1 Structural Model

The primary structural element in ROOM is an *actor*, which is an encapsulated, concurrent object responsible for performing some specific function. Each actor object is an instance of a corresponding actor class capable of utilizing polymorphism and inheritance, prominent features of object-oriented methodology. Inter-actor communication is performed exclusively by sending and receiving *messages* via interface objects called *ports*. A message is a tuple consisting of a signal name, an optional message body and an optional message priority. A port is an instance of a *protocol class*, which defines a reusable message protocol specification. The same protocol specification can be used by many different actors by simply creating protocol class instances. Specifying protocols as classes also enables real-time developers to relate them through inheritance, creating a protocol class hierarchy. This provides the ability to define abstract protocols[1] that can be specialized in different ways by subclassing. A simple actor representing a computer with three ports is depicted in Figure 1. The `LPT1` port is an instance of the `Parallel` protocol class and the `COM1` and `COM2` ports are instances of the `Serial` protocol class.

In ROOM, it is possible to combine one or more actors into more complex functional aggregates, i.e. more complex actors, by means of three compositional mechanisms: *binding*, *layer connections* and *containment*. A binding models a communication channel between two compatible ports, where one of the ports is the inverse of its peer port (the white fill on the port indicates that the port uses a conjugated protocol class). In Figure 2 we show how the binding mechanism can be used to add peripheral actors to our previous model of

[1] Abstract protocols represent general protocol specifications for which ports can not exist.

LPT1

COM1 COM2

Figure 1: Computer Actor

the Computer actor.

COM2

modem

serial

printer

computer

parallel LPT1

COM1

serial mouse

Figure 2: Functional Aggregate: System Actor

In contrast to bindings, layer connections are directed relationships between actors in the form of a client-server model, where actors in the lower layer provide a set of services to the entities in the layer above. The linkage between layers is done at discrete contact points, which are called *service access points* (SAPs) in the upper layer using the services, and *service provision points* (SPPs) at the layer providing the services. Each SAP must be connected to a SPP in the layer below (there can be many SAPs connected to a single SPP), and the end points of each such connection must have matching service points. Like ports, SAPs and SPPs are instances of message protocol classes. The bottom layer in ROOM

models is called the ROOM *virtual machine* (which is implemented by the ObjecTime run-time system). In addition to executing ROOM specifications, the virtual machine provides a set of system services, among which there is a *communication service* and a *timing service*. The communication service establishes and manages connections between actors. The timing service is used to set and cancel timers, both one-shot and periodic [Sel95]. The system services are accessed through SAPs, like other services, however their corresponding SPPs are implicit. The ROOM virtual machine is also responsible for interfacing with other (non-ROOM) environments, such as specialized hardware or "foreign" software systems that may be part of the computing environment.

The last compositional mechanism in ROOM is containment. It involves decomposing an actor into aggregates of more elementary actors. As an example, Figure 3 shows the decomposition of the Computer actor.

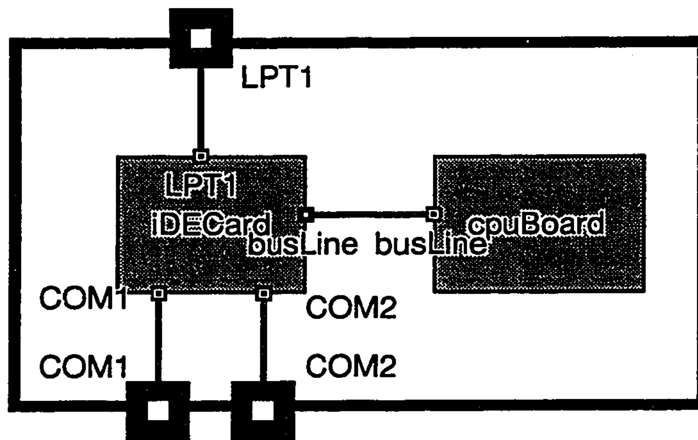

Figure 3: Functional Aggregate: Computer Actor

The actor structures that we have reviewed until now are static, i.e. they are created once when the ROOM application is started and they exist throughout its entire execution. To handle dynamic creation of actors, ROOM provides the concept of an *optional actor*, indicated by the hashed actor component. Optional actors are not created automatically, instead they are created and destroyed, with the help of the *framing service*, as the need arises. Although, we do not need to create actors dynamically, we will define optional

actors to assign them to different logical threads[2].

2.1.2 Behavioral Model

The actor's behavior is represented by an extended state machine called a *ROOMchart*, based on statechart formalism [Har87]. Each actor remains dormant in its current state until a message is received by an actor. Incoming messages trigger transitions to new states as defined by the actor's finite state machine. Actions may be associated with state transitions, as well as entry and exit points of a state. The sending of messages to other actors is initiated by an action. The finite state machine behavior model imposes that only one transition at a time can be executed by each actor. As a consequence, a *run-to-completion* paradigm applies to state transitions. This implies that the processing of a message cannot be preempted by the arrival of a new (higher priority) message for the same actor. However, as we will explain in Chapter 4, in a multi-threaded implementation, the processing may be preempted by processing associated with actors in other higher priority threads.

ROOMcharts also support the object-oriented paradigm. As mentioned in the previous section, each actor derived from its parent automatically inherits its parent's structure, as well as the extended finite state machine behavior. The encapsulation in ROOMcharts is implemented with the notion of a *composite* state, which can be decomposed into substates. Decomposition of a state into substates can be taken up to an arbitrary level in a recursive manner. The current state of such a system is defined by a nested chain of states called a *state context*. The actor is said to be simultaneously "in" all of these states. Transitions of the innermost current scope take precedence over equivalent transitions in higher scopes. A message for which no transition is triggered at all levels of the state hierarchy is discarded, unless it is explicitly deferred.

2.2 Message Sequence Charts and Transactions

Interactions between selected ROOM actor instances can be specified at design time in a Message Sequence Chart (MSC), conforming to a subset of CCITT recommendations Z.120 [Int94]. ROOM MSCs may graphically depict message flow between actors, transition actions and changes in actor states. However, ROOM itself does not provide any mechanisms to specify and enforce timing constraints imposed on each MSC scenario.

[2]Only optional actors may be placed in threads other than the main thread.

Thus, in order to perform our scheduling analysis, we will add additional annotations to indicate these timing constraints [BAL97]. We will call these annotated MSCs *transactions*, and we will use them to indicate time critical end-to-end computations, for which timing constraints such as periodicity and deadlines may be specified. To deal with temporal requirements of distributed real-time systems, Burns and Wellings also defined the notion of a transaction [BW96] to link input and output activities that share associated deadlines. They were used to reflect end-to-end properties, i.e. from an input event trigger to an output event response, of a real-time system. These transactions were implemented using tasks and *protected shared objects* (PSOs) which ensured exclusive access to data shared between the tasks. We shall also define a transaction tree as a chained sequence of transition actions triggered by an external incoming (stimulus) message. The triggering messages may be aperiodic (often corresponding to device interrupts or generated by the one-shot timer) or periodic (generated by the periodic timer). Each branch in the transaction tree will represent a ROOM transaction. A transaction tree Υ_i has an associated activation *period* T_i, which represents either the inter-arrival time (period) of the periodic timer, or a minimum inter-arrival time for aperiodically triggered (sporadic) interrupt.

The following timing constraints are specified for each transaction Γ_i (for each branch in the transaction tree Υ_i) associated with the transaction tree Υ_i.

- The worst case *execution time* C_i, representing the sum of the worst case execution times of its transition actions.
- An end-to-end *deadline* D_i on the response time of the transaction.
- A transaction *priority* P_i, assigned to each transaction according to its importance.

Then, a ROOM transaction tree set consisting of n transaction trees is $\Upsilon_1, \Upsilon_2, \ldots, \Upsilon_n$. If all the external incoming messages can be identified, then the ROOM transaction tree set can be mechanically generated by starting from these messages, and then forming transition action sequences by recursively considering the output message set for the actors handling the messages. The definition of a transaction tree allows two transactions to be triggered by the same external incoming message, thus having a common initial transition action sequence. Such transactions would then have the same activation period equal to the activation period of their transaction tree, but may have different deadlines and different priorities. On the other hand, two transactions can share a common final transition action

sequence. Such transactions may have different activation periods equal to the activation periods of their transaction trees, end-to-end deadlines and different priorities.

As mentioned before, MSCs in ROOM are created at design time, i.e. they are generated by the real-time designer as an additional way to specify system requirements. Thus, they can diverge from the actual executable ROOM model. In Figure 4 we show a sample MSC depicting interactions between the IDECard and CPUBoard actors (Note that time flows from top to bottom).

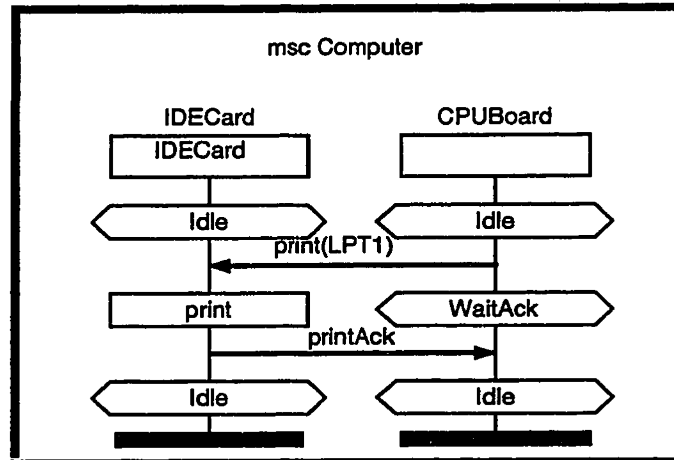

Figure 4: Message Sequence Chart for Printing

2.3 Target Run-Time System

The ObjecTime toolset provides a run-time system in the form of an implementation of the ROOM virtual machine, responsible for providing the mechanisms that support the ROOM paradigm as well as the services needed by ROOM models. The ObjecTime Developer Toolset is 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. The ObjecTime toolset includes a target run-time system (`TargetRTS`) [Obj97], which is linked with the application code to provide a stand-alone executable that may be executed on either a Unix workstation

environment, e.g. Solaris, or on a target environment with an underlying real-time operating system such as VxWorks, QNX, pSOS, and VRTX. In addition to executing ROOM specifications, `TargetRTS` provides a set of system services, among which there is a *communication service* and a *timing service*. The timing service is used to set and cancel timers, both one-shot and periodic [Sel95].

In ROOM, actors are potentially concurrent objects having a private address space. In implementing ROOM models, one must deal with the mapping of ROOM actors to the underlying operating system's execution abstractions. Operating systems typically provide two such abstractions: a heavy weight *process*, which has a private address space, and communicates with other processes using inter-process communication services such as sockets, pipes or RPC calls, and a light weight *thread*, which shares memory with other threads within the same process. Because communication between processes is expensive, and ROOM models rely heavily on inter-actor communication, mapping actors to processes would be inefficient in most cases. Therefore, a ROOM model is implemented as a multi-threaded process, where actors are grouped into one or more lightweight threads. `TargetRTS` implements each thread as a message handler, which executes a main loop waiting for messages to arrive, and processes the arriving messages in priority order. This implies that message processing cannot be preempted by another message arrival within the same thread, and it is consistent with the run-to-completion semantics of ROOM. However, a thread may be preempted by other threads depending on thread priorities and the scheduling of threads by the underlying operating system.

As mentioned before, the transaction period represents an inter-arrival time of the periodic timer, or a minimum inter-arrival time of the one-shot timer [3]. The scheduling of transactions in `TargetRTS` always happens following the arrival of a timing signal, which represents an external incoming message. Because the timing service is a part of the ROOM virtual machine, the transaction timing source is external to the CPU clock, i.e. it is generated by the `TargetRTS` timing service and is asynchronous to the CPU clock. In a ROOM/ObjecTime model, the `TargetRTS` timing service is polled by OS timing services. Because of the asynchronous nature of these services, polling leads to the *blocking* affect we shall discuss in Chapter 3. This type of transaction scheduling is defined as *timer-driven* scheduling and is further described in [Kat94].

[3] Restricting the transaction period to a minimum inter-arrival time of the one-shot timer is a limitation necessary for our timing and scheduling analysis.

Chapter 3

Choosing a Scheduling Analysis Method for ROOM Models

Task scheduling analysis is a fundamental issue in developing *hard-real-time* systems, where all the tasks must finish executing before their corresponding deadlines. Examples of hard-real-time systems are flight and traffic control, nuclear plant control, robotics and automobile cruise control systems, the last one described in [Gom93] and further studied in Chapter 6. When hard-real-time systems are employed to regulate physical environments imposing computational resource constraints, such systems are called *embedded control* systems. Because of the stringent constraints imposed on computing resources, it is necessary to optimize their utilization while being able to predict task schedulability at design time.

As mentioned in the previous chapter, the timing and scheduling analysis of ROOM models is based on time critical end-to-end computations about which timing constraints such as periodicity and deadlines may be specified. These end-to-end computations were formally defined as transactions, thus they can be subjected to common timing analysis methods such as the real-time scheduling theory [BW96] or symbolic model checking.

In this chapter, we will examine and select a suitable timing and scheduling analysis method for ROOM models, where we will consider three orthogonal timing analysis methods: real-time scheduling theory, symbolic model checking and discrete task simulation. We will only consider schedulability tests for fixed priority algorithms, as these are more frequently used in embedded control systems than dynamic priority algorithms due to their simpler scheduler and lower implementation overheads. We will present an overview of

each method, followed by the time and space complexity costs of using them, and we will summarize the chapter with concluding remarks.

3.1 Overview of Real-Time Scheduling Theory

A first contribution to task scheduling analysis was made by Liu and Layland [LL73] who developed optimal static and dynamic priority scheduling algorithms for *hard-real-time* sets of independent tasks. They showed that the dynamic priority scheduling algorithms, i.e. earliest deadline, can achieve 100% CPU schedulable utilization, while the rate-monotonic algorithm has a least upper bound of 69% of CPU schedulable utilization. They provided a necessary and sufficient utilization-based test for earliest deadline scheduling, and a sufficient utilization-based test for rate-monotonic scheduling. The Liu and Layland's rate-monotonic test is sufficient in that the task set which does not pass the test can be still schedulable, as long as the utilization is less than 100%. Since then, significant progress has been made on generalizing and improving the schedulability analysis. Necessary and sufficient conditions for fixed priority algorithms have also been developed [LSD87] and [JP86]. These conditions construct the worst case phasing for each task, i.e. task critical instance, and test the task scheduling under this configuration. These conditions also release the requirement that tasks are scheduled according to the rate-monotonic algorithm.

3.1.1 Worst Case Response Time

The Joseph and Pandya work [JP86] provided a necessary and sufficient timing analysis test for a set of tasks scheduled by the deadline-monotonic or rate-monotonic algorithm. They came up with the idea of calculating the worst case response time [1] of a task. Once we have the worst case response time for each task, the scheduling test is trivial: we just look to see that $R_i \leq D_i$ for each task. R_i is computed by the following recursive equation:

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_j \qquad (1)$$

where $hp(i)$ denotes the set of tasks of higher priority than task τ_i, and the term R_i^{n+1} is the $n+1$ approximation to the worst case response time, R_i, and is found in terms of the nth

[1]The worst case response time R_i of an independent periodic task τ_i (with $D_i \leq T_i$) occurs when all tasks are released (i.e. become ready to run) at the critical instant $t = 0$.

approximation. The recursive equation 1 either converges to $R_i^{n+1} = R_i^n = R_i$ or exceeds D_i. It can be solved by successive iteration starting from $R_i^0 = C_i$. With this analysis, a value of R_i is valid only if $D_i \leq T_i$, which is what we have assumed while specifying the timing constraints of a transaction in a ROOM model.

As mentioned before, by looking at the worst case response time of each task, we can determine if the task set is schedulable. Moreover, if the worst case response time of each task is smaller than its deadline, we know that the CPU is not fully utilized. Thus, the above analysis allows us to test task schedulability and optimize the processor utilization by considering the worst case scenario, when all the tasks become ready to run at the critical instant.

3.1.2 Release Jitter

Another very important contribution to real-time scheduling theory was made by Audsley *et al.* [ABR+93], who created a scheduling test for a hybrid event- and timer-driven scheduler which was unique to the system they modelled. In this system, some tasks would undergo the timer jitter effect, while others did not. The jitter effect was modelled as task *release jitter*, which is the difference between the earliest and latest releases [2] of a task relative to the invocation [3] of the task. The worst case effects of release jitter were accounted for by phasing tasks to maximize the worst case response time of each task. If the amount of jitter for task τ_i is J_i, then the time taken for the task to complete once it has been released is:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n + J_j}{T_j} \right\rceil \cdot C_j \qquad (2)$$

The notation 'w' is used because the task is constrained to execute within a time *window* during which higher priority tasks can preempt it. The worst case response time for task τ_i is given by:

$$R_i = J_i + w_i \qquad (3)$$

If no task has jitter, equations 2 and 3 reduce to equation 1 and model the worst case response time for tasks scheduled by event-driven schedulers. If all tasks have equal jitter J, the two equations model timer-driven schedulers with the timer jitter effect $J_{timer} = J$.

[2] A task is released if it is placed by a scheduler in a priority queue of runnable tasks.
[3] A task is invoked if it could logically be made runnable at or after that time.

The above equation provides only a sufficient test for task scheduling, in that the task τ_i for which $R_i > D_i$ can still be schedulable. Notice that the equations do not contain implementation costs, such as task context-switch overhead or priority inversion for the active task when lower priority tasks are being scheduled. To include the implementation costs of ROOM models, the generic resource scheduling model along with its canonical scheduling models for single- and multi-threaded ROOM executables will be developed in Chapter 5.

3.1.3 Time and Space Complexity

The complexity of the scheduling models described in this section were derived in a similar way to [Kat94]. Let N be the number of application tasks, and L be the ratio of the longest period to the shortest period of all tasks. The worst case response time algorithms described above require that the response time of each task τ_i be found in an interval $(0, T_i]$. This is done by iterating a series of approximations for the worst case response time, $R_i^0, R_i^1, \ldots, R_i^K$, where $K(i)$ is an upper bound on the number of approximations generated for task τ_i, $1 \leq i \leq N$. Each approximation is computed from the summation:

$$\sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$$

for equation 1, which itself is of $O(N)$, or from the summation:

$$\sum_{j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil \cdot C_j$$

for equation 2, which is also $O(N)$. $K(i)$ is no greater than the worst case number of task releases in an interval $(0, T_i]$, so that $K(i)$ is of $O(N \cdot L)$ for both scheduling tests. Therefore, the time complexity for each task τ_i is of $O(N^2 \cdot L)$, and the overall time complexity for both scheduling models is of $O(N^3 \cdot L)$.

The space complexity for both scheduling models depends on the number of recursive calls necessary to calculate the worst case response time R_i for each task, and as we know it is no greater than the worst case number of task invocations in an interval $(0, T_i]$. Therefore, the space complexity for the two scheduling models is of $O(N \cdot L)$.

3.2 Overview of Symbolic Model Checking

Recently, a number of different methods have been proposed for verifying finite-state systems by examining state-graph models of system behavior. Most of these methods depend on algorithms that explicitly represent a transition relation, using a list or table which grows in proportion to the number of states. The symbolic model checking (SMC) technique, developed at Carnegie Mellon University by the Formal Methods - Model Checking group, tries to resolves the "state explosion problem" by representing the transition relation "symbolically" instead of explicitly. In their representation, they use *ordered binary decision diagrams* (OBDD) [Bry86], which allow many practical systems with extremely large state spaces to be verified. Using binary decision diagrams for transition relations, they were able to verify some examples that had more than 10^{20} states [BCM90]. Since then, various refinements of the OBDD-based techniques by other researchers at Carnegie Mellon have pushed the state space up to more than 10^{120} [BCL91].

3.2.1 Timing and Scheduling Analysis of Tasks

Symbolic Model Checking can also be used to verify the timing constraints of a set of tasks scheduled with a fixed priority algorithm. For the analysis to be described here, we used the Symbolic Model Verification (SMV) tool [McM92], which can check finite state systems against their specifications expressed in the Computational Tree Logic (CTL) [CCM96]. To test task schedulability, we create a Finite State Machine (FSM) model of a preemptive task, transform the FSM model into a SMV module for each task, and test the timing constraints of each SMV module against CTL and quantitative formulas within the framework of the chosen fixed priority scheduler, i.e. rate-monotonic or deadline-monotonic. Figure 5 is a schematic of the process representing SMC temporal verification of a set of real-time tasks.

Task FSM Model

The FSM model in Figure 6 represents all possible states, transitions and events (with their corresponding actions) of an independent and preemptive task with no resource sharing constraints. The states describe the task's behavior. At any given time, the task is in one of these states where a finite number of events may occur. The events result from internal processing of each task, the scheduler decisions and the timer associated with the task period. The actions associated with certain transitions keep track of how much CPU time

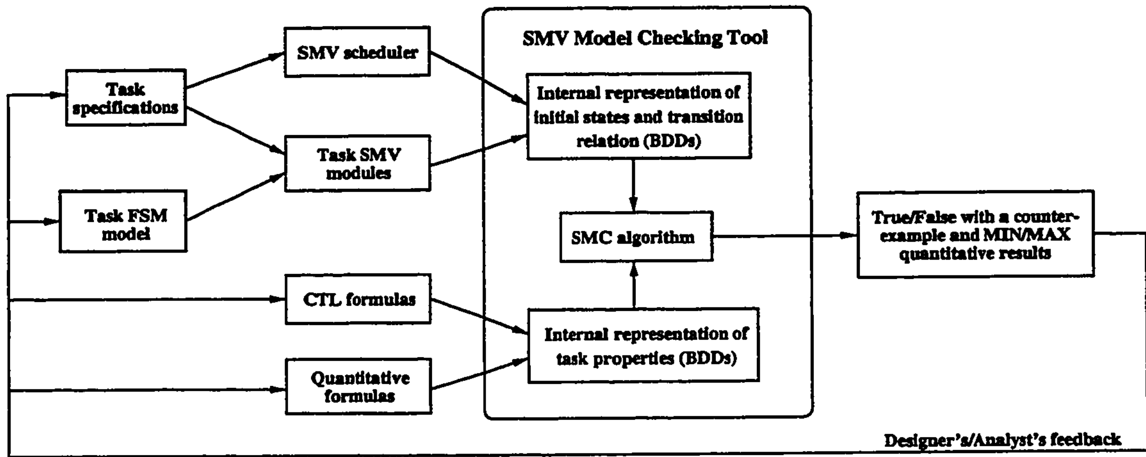

Figure 5: Schedulability Analysis with SMV toolset

(in clock units e.g. milliseconds) the task has used. The transitions indicate the actions to be performed. They are "selected" according to the current state of the task and the event type that is represented with values of boolean/scalar SMV defined symbols. In our task model, we have three symbols comprising an event type:

task scalar symbol, indicating which task was scheduled by the fixed priority scheduler.

timeout boolean symbol, set when the task is released by the scheduler.

finish boolean symbol, set when the task completes its execution.

SMV Scheduler

The SMV scheduler accepts requests for execution and chooses the highest priority task requesting the CPU. If a request arrives from a higher priority task after execution of another task has started, the scheduler preempts the executing task and starts the higher priority one. When the task finishes executing it resets its request, and the scheduler chooses another task. The scheduler implements a global timer that releases periodic tasks when their period arrives. It is the "heart" of the SMV program as it simulates the smallest time unit in our scheduling analysis. Another type of timer is also used, the local timer, which stores the elapsed task execution time. The task priority can be determined either with the deadline-monotonic or rate-monotonic algorithm.

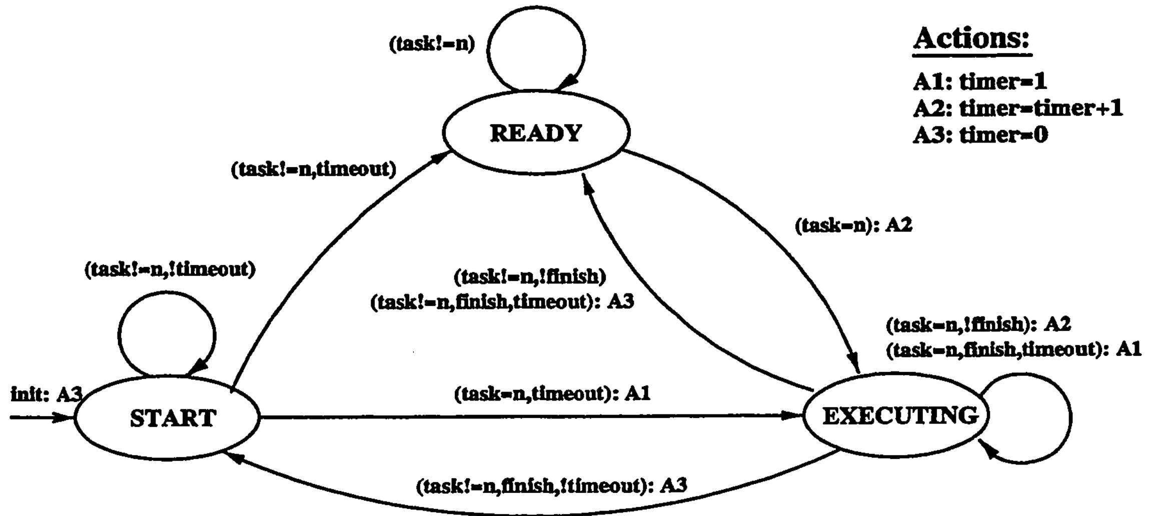

Figure 6: FSM Model of a Task

CTL Formulas

To test task schedulability, we create CTL formulas, imposing restrictions on our SMV model. According to the Completion Time Theorem [LSD87], each task has to finish before its deadline. For a set of three tasks, this notion can be expressed by the following CTL formula:

```
AG !(deadline1  & (!(T1.state = START) & !T1.finish) |
     deadline2  & (!(T2.state = START) & !T2.finish) |
     deadline3  & (!(T3.state = START) & !T3.finish))
```

where `deadline1`, `deadline2` and `deadline3` correspond to the deadlines of the first, second and third task. If the CTL formula is not satisfied, the SMV toolset will provide us with a counterexample, showing step-by-step execution of our SMV program leading to violation of the formula. By examining the counterexample, we can identify which task missed its deadline.

Quantitative Formulas

So far we have demonstrated how to test the schedulability of tasks using the SMC techniques, but it is also possible to obtain some numerical results about the worst and the best

case execution times of each task within a specified "time window", i.e. time interval defined by the global timer in which tasks are scheduled. This is accomplished by using the MIN and MAX quantitative algorithms [CCM96]. Because we are concerned here with timing and scheduling analysis of ROOM models which represent a family of *hard real-time* systems, we will concentrate on MAX quantitative results. Thus, the quantitative formulas calculating the maximum number of clock ticks a task needs to finish executing since its invocation (start) time are as follows:

```
MAX[T1.start, T1.finish]
MAX[T2.start, T2.finish]
MAX[T3.start, T3.finish]
```

where `start` boolean symbol is set when the task is released by the fixed priority scheduler.

3.3 Overview of Discrete Task Simulation

Besides investigating real-time scheduling and symbolic model checking techniques, we have also written a `Java` discrete task simulator, simulating a fixed priority scheduler for a set of N independent preemptive tasks. With this simulator, we have also verified the worst case response times of real-time tasks. If we examine the simulator complexity numbers, we notice that for an event-driven scheduler, the time complexity is identical to that of the RTST method, but the space complexity is equal to the size of the ready and sleep queues and the number of tasks. Therefore, it is of $O(R+S+N)$, where R and S are sizes of the ready and sleep queues accordingly. For a timer-driven scheduler, the time complexity is of $O(N^3 \cdot L \cdot (T_N)^{N-1})$, because to guarantee a necessary and sufficient condition from the simulator test we must construct all possible task phasings, whenever release jitter is introduced.

3.4 Empirical Complexity Results

In this section, we will examine task's worst case response time and empirical time/space complexity results obtained from real-time scheduling theory and symbolic model checking methods which were used to analyze the schedulability of independent and periodic tasks.

We will examine two task configurations, first one (see Table 1) representing a set of three independent tasks and the other one (see Table 3) representing a set of three tasks, each introducing release jitter. As mentioned before, the second task configuration implements a timer-driven scheduler.

3.4.1 Independent Tasks

Task	T(ms)	C(ms)	D(ms)
A	100	20	100
B	150	30	150
C	350	125	350

Table 1: Temporal Characteristics of Three Independent Tasks

To obtain more meaningful results for the first task set we will gradually increase task temporal characteristics, such as period, deadline and execution time by a scalar value. The empirical results presented in Table 2 were obtained from using real-time scheduling theory (RTST) and symbolic model checking (SMC) methods on the task set from Table 1.

Scalar	RTST/SMV			RTST		SMV		
	R_A (ms)	R_B (ms)	R_C (ms)	Time (sec)	Space (bytes)	Transition BDD Nodes	Time (sec/min)	Space (Mbytes)
1	20	50	245	≤ 1	670	1566	8.8sec	1.13
2	40	100	490	≤ 1	670	1841	1.36min	1.56
3	60	150	735	≤ 1	670	2028	6min	1.94
4	80	200	980	≤ 1	670	2096	21.6min	2.31
5	100	250	1225	≤ 1	670	2318	45.78min	2.44

Table 2: Empirical Results for Three Independent Tasks

In the SMV manual [McM92], the overall time complexity of an SMV program is derived from three factors: an increase in the transition relation BDD nodes, an increase in the state set BDD nodes and an increase in the number of iterations. From Figure 7, we can see that the overall empirical time complexity required to calculate the worst case response times for all the tasks within our task set is of $\Theta((T_N)^3)$, where T_N is the longest

task period. Also, by looking at the transition BDD nodes and space columns in Table 2 we can confirm this result, as the overall empirical time complexity of $\Theta((T_N)^3)$ results from the linear increase in the state set BDD nodes [4] and the linear increase in the transition relation BDD nodes. We can also deduce that the number of iterations required to obtain SMV output, i.e. true or false with a counterexample and MAX quantitative results, also increases linearly with respect to T_N.

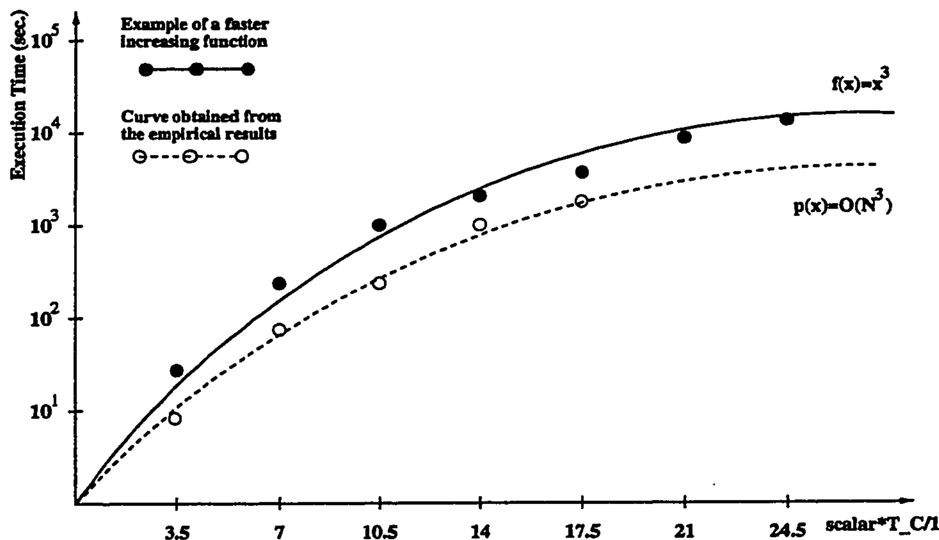

Figure 7: SMV Execution Time for Three Independent Tasks

Now, looking at the time and space results from the RTST columns, we notice that the execution time and memory utilization is constant during the scheduling analysis performed with real-time scheduling theory method on our task set. This result confirms our earlier time and space complexity observations from Section 3.1.3. We know that the space complexity depends on the number of recursive calls necessary to calculate the worst case response time R_i for each task, and is equal to $O(N \cdot L)$. Because N and L are constant, our memory utilization should be constant too. The overall time complexity is of $O(N^3 \cdot L)$ and, in a similar way, leads to the constant execution time for our task set.

We should also notice that both tests, real-time scheduling theory and symbolic model checking, provide us with the necessary and sufficient timing analysis test for a set of

[4]State set BDD nodes is calculated by subtracting transition BDD nodes from total number of bytes allocated divided by BDD size.

independent tasks (with no release jitter and no blocking effect) scheduled by the deadline-monotonic or rate-monotonic algorithm.

3.4.2 Tasks with Release Jitter

Task	T(ms)	C(ms)	D(ms)	J(ms)
A	50	10	50	0/5/10/15/20/25
B	75	15	75	0/5/10/15/20/25
C	175	60	175	0/5/10/15/20/25

Table 3: Temporal Characteristics of Tasks with Release Jitter

In a similar way as in the previous section, we gathered some empirical results (presented in Table 4) from using real-time scheduling theory and symbolic model checking methods with the task set described in Table 3. For the symbolic model checking method the assumption is made that all tasks are invoked at the time instant $t = 0$ and they run for the length of the longest task period. In contrast to tasks without release jitter, this initial invocation does not represent the worst case phasing for each task.

J	RTST			SMV	RTST		SMV		
	R_A (ms)	R_B (ms)	R_C (ms)	CTL formula holds	Time (sec)	Space (bytes)	Transition BDD (bytes)	Time (sec/min)	Space (Mbytes)
0	10	25	120	yes	≤ 1	720	1497	2sec	1.11
5	15	30	125	yes	≤ 1	720	2050	15sec	2.17
10	20	35	130	yes	≤ 1	720	3008	7.32min	4.52
15	25	40	135	yes	≤ 1	720	3366	35min	11.47
20	30	45	140	yes	≤ 1	720	4902	92.8min	15.1
25	35	50	145	yes	≤ 1	720	5538	209min	31.85

Table 4: Empirical Results for Tasks with Release Jitter

To examine the time complexity of the SMC method, we present the graph in Figure 8. According to the SMV empirical time complexity results obtained in the previous section, we can see that the overall empirical time complexity required to test the schedulability of tasks from Table 3 is of $\Theta((T_N)^3 \cdot J)$, where J is the task release jitter. Similarly by examining Table 4, we conclude that the SMV empirical space complexity is of $\Theta(T_N + J)$.

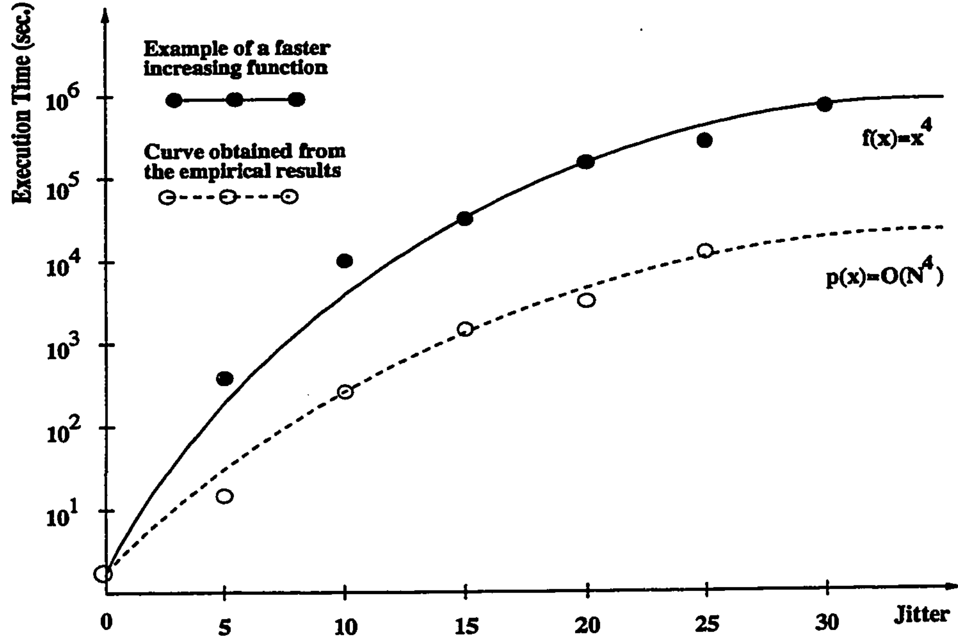

Figure 8: SMV Execution Time for Tasks with Release Jitter

The empirical time and space complexity results for real-time scheduling analysis are identical to those from the previous section, and they confirm our earlier observations from Section 3.1.3. Again, we should remind the reader that the real-time scheduling test is only a sufficient condition for task schedulability when release jitter is introduced. On the other hand, the symbolic model checking test with the assumptions made here (that all tasks are invoked at the time instant $t = 0$, they run for the length of the longest task period and they introduce release jitter), provides us with neither sufficient nor necessary conditions for task schedulability. To guarantee a necessary and sufficient condition from the symbolic model checking test, we must construct all possible task phasings whenever release jitter is introduced. For the set of of N tasks, we have $T_1 \cdot T_2 \dots T_{N-2} \cdot T_{N-1}$ unique task phasings, which gives rise to another complexity number $O((T_N)^{N-1})$. Therefore, constructing a necessary and sufficient scheduling test with symbolic model checking technique for our task set has the empirical time complexity of $\Theta((T_N)^{N+2} \cdot J)$.

3.5 Conclusions

In this chapter, we reviewed three scheduling analysis techniques: real-time scheduling theory, symbolic model checking and discrete task simulation. Because `TargetRTS` implements a timer-driven scheduler, our choice of a scheduling analysis technique is heavily based on the results obtained from the corresponding scheduling model. In Table 5 we present a summary of the relevant characteristics of the event-driven and timer-driven scheduling models which do not take scheduler implementation costs (e.g. timer blocking penalty, task context-switch overhead) into consideration.

		Event-Driven	Timer-Driven
RTST	Time	$O(N^3 \cdot L)$	$O(N^3 \cdot L)$
	Space	$O(N \cdot L)$	$O(N \cdot L)$
	Condition	N&S	S
SMV	Time	$\Theta((T_N)^3)$	$\Theta((T_N)^{N+2} \cdot J)$
	Space	$\Theta(T_N)$	$\Theta(T_N + J)$
	Condition	N&S	N&S
Simulator	Time	$O(N^3 \cdot L)$	$O((T_N)^{N-1} \cdot N^3 \cdot L)$
	Space	$O(R+S+N)$	$O(R+S+N)$
	Condition	N&S	N&S

Table 5: Scheduling Model Characteristics

Because of the timer-driven nature of the ROOM scheduler implemented by `TargetRTS`, the best choice of the timing and scheduling analysis technique is the real-time scheduling theory. Therefore, we will extend this theory to include implementation costs of ROOM models in Chapter 5, where we will validate it and use it to test the temporal behavior of a ROOM model of an automobile cruise control embedded system.

Chapter 4

ROOM Design Guidelines for Hard Real-Time Systems

In this chapter we develop guidelines to apply the real-time scheduling theory presented in the previous chapter to ROOM models. As we shall see, the biggest challenge comes from various sources of priority inversion that can result in large and possibly unbounded blocking times. In ROOM models, two kinds of priorities may be defined: *message priority* and *thread priority*. Recall that a ROOM model can be implemented as a single-threaded executable, where all the actors are assigned to one thread, or it can be implemented as a multi-threaded executable, where each thread implements one or more actors. In both cases, the underlying real-time operating system is responsible for scheduling threads. We assume that the operating system uses preemptive priority scheduling and allows application control over thread priorities. As mentioned in Chapter 2, the run-time system `TargetRTS` implementing the ROOM virtual machine, is responsible for processing messages within a thread according to the fixed-priority scheduling algorithm based on message priorities, and the underlying operating system is responsible for scheduling threads based on thread priorities. Thus, there are two-levels of priority scheduling:

- Within the context of a single thread, the processing of messages takes place in message priority order. This ordering is enforced by the message handling loop provided by `TargetRTS`.
- Across the whole system (i.e. across the complete ROOM model), the underlying operating system schedules threads in thread priority order.

There are four sources of priority inversion which contribute to transaction blocking times. In the following sections, we shall consider these sources of priority inversion and suggest design and implementation guidelines to minimize their adverse effects.

4.1 Assigning Actors to Threads

The first potential source of blocking comes from inappropriate assignment of actors to threads. In particular, the designer of a real-time system must assign actors to threads such that the system is schedulable under the worst case conditions. In the following sections, we will investigate two possible actor-thread assignments and we will elaborate on their relative strengths and weaknesses.

4.1.1 Single-Threaded Executable

The simplest approach to implementing a real-time system designed in ROOM is to assign all user actors to a single user thread. Thus, ROOM processing will take place in message priority order and the only source of priority inversion is due to ROOM run-to-completion semantics (as explained in Chapter 2). If we assume that all messages associated with a given transaction are assigned the same priority, then the maximum blocking time of a high-priority transaction is limited by the most "time-consuming" transition within lower priority transactions. In many systems, this is perfectly tolerable or even preferable to multi-threaded approaches, and leads to a simple, low overhead approach to implementation and low CPU utilization. Because of its low implementation costs, i.e. no inter-thread message passing and no thread context-switch latencies, this simple approach is preferable when all transactions can finish their processing before their respective deadlines. It should be noticed that if the lowest-priority transaction is not schedulable under the single-threaded executable, it will not be schedulable under a multi-threaded executable. The single-threaded approach, however, ceases to be effective when a high-priority transaction misses its deadline, due to "time-consuming" transitions within lower priority transactions. To avoid these significant priority inversions for higher priority transactions, multi-threaded configurations are used. In single-threaded ROOM executables, there is only one user thread [1] with all the actors assigned to it. Because there is only one user thread, we can

[1] In practice, single-threaded ROOM models also contain system threads such as a timer and external layer thread.

assign it a priority lower or higher than the priority of system threads. If we assign a higher priority, the currently executing transaction within the user thread will block any system thread including the timer thread, preventing any higher priority transaction from executing. However, if we assign a lower priority to the user thread than to any system thread, a pending higher priority transaction will begin executing after the current transition runs to completions. Accordingly, our first guideline suggests a possible thread priority assignment for single-threaded ROOM models.

Guideline 1 *In a single-threaded ROOM executable, the user thread (to which all actors are assigned) should have a static priority which is lower than the priority of any system thread.*

Following this guideline implies that message send operations should never change the priority of the receiving thread, which is always constant. Also, within the context of a single thread, the processing of messages takes place in message priority order under the sole control of `TargetRTS`.

4.1.2 Multi-Threaded Executable

In a multi-threaded configuration, each actor is assigned to one of the application-defined threads. Each thread then acts as a message handler for the associated actors. A thread may then be preempted by another thread depending on thread priorities and the scheduling algorithm of the underlying operating system. Threads are useful if a transition within a lower priority transaction causes a higher priority transaction to miss its deadline e.g.. by making a blocking call to external functions. Note however that a multi-threaded executable has higher implementation costs and in some cases might be inferior to single-threaded approaches. In particular, inter-thread message passing is an order of magnitude more expensive than intra-thread message passing. Also, one must take into account the costs of thread context-switch when considering multi-threaded executable performance. Because of its higher implementation costs, it also leads to higher CPU utilization.

We mentioned in the introduction to this chapter that there are two kinds of priority-based scheduling in ROOM models. This two-level priority scheduling can result in priority inversion or unnecessary preemption if thread priorities are not adequately defined. As noted earlier, the transactions associated with external messages represent tasks in the system and accordingly, external message priorities are the "real" application priorities.

Ideally (when there is no implementation cost), processing across the whole system should be driven by the message priorities. Thread priorities, on the other hand, would then be the artifacts of the implementation.

Static Thread Priorities

`TargetRTS` supplied by ObjecTime allows an application designer to define static thread priorities (in addition to message priorities). This can lead to priority inversion or unnecessary preemption, since (in general) each actor, and hence each thread, may be processing messages of different priorities. Still, many systems may be designed successfully, and indeed are, with this capability. One way to use it effectively would be to design a system such that the functionality of the system is partitioned into "control" actors (where most message processing is non compute-intensive), and "data-processing" actors (where message processing may be compute-intensive, but state machines are simple, and the actor implements only one transaction) parts. All control actors can then be mapped to a single control thread, while other data processing actors may be mapped to separate threads to allow preemption. Each data processing thread can be assigned a priority depending on the time criticality of the function it implements. This implies that all messages associated with an actor in the data processing thread have the same priority. The control thread can be assigned the priority of the highest or lowest message processed by the thread. If it is assigned the priority of the highest priority message, then processing of a lower priority message, m_i with priority $p(m_i)$, by the thread may lead to priority inversion for transactions with priorities $p(m_i) < P < p(thread)$. On the other hand, if it is assigned the priority of the lowest message, then processing of a higher priority message m_j with $p(m_j)$ by the thread may lead to unnecessary preemption by transactions with priorities $p(thread) < P < p(m_j)$. Clearly, assigning the highest priority messages to the control thread is the wisest choice and it may be acceptable if the aggregate low-priority workload in the control thread is not large enough to cause missed transaction deadlines initiated by data processing actors. The following guideline formalizes the above suggestions:

Guideline 2 *In a multi-threaded ROOM executable with static thread priorities, each data processing thread should be assigned the priority of the messages that it accepts, and the control thread should be assigned the priority of the highest priority message to be processed by this thread.*

This guideline also implies that the message send operation should never change the priority of the receiving thread. Because static thread priorities means no thread priority changes, its implementation costs and consequently CPU utilization is lower in comparison to dynamic thread priority assignment to be described in the next section.

Dynamic Thread Priorities

Lets begin by describing a system where a static thread priority approach fails to work. The train tilting system presented in [SPFR98] is one such example. It may not be possible to nicely separate the control and data processing functionalities into separate actors. Even when it is possible, there may be sufficient low priority workload in the control thread that can lead to missed transaction deadlines. This problem was identified in [SFR97], and it was suggested that `TargetRTS` should automatically manage thread priorities to reflect the priority of the messages to be processed by the thread. Thus, each thread can have a dynamic priority which is immediately incremented whenever a higher priority message arrives for one of the actors managed by the thread. Likewise, the priority of a thread can be automatically adjusted to that of the highest priority queued message when the thread finishes processing a message. Hence, we suggest that the thread priority should be a dynamic attribute determined by the messages waiting to be handled, which is stated in the following guideline:

Guideline 3 *In a multi-threaded ROOM executable with dynamic thread priorities, a thread priority should be equal to the highest priority pending message, including the message being currently processed, if any.*

With a sufficient number of thread priorities, this scheme gives the advantage of preemptability offered by multiple threads, without the problems of priority inversion as described in the previous section. In fact, the application designer does not need to worry about the thread priorities, as it is automatically taken care of by the run-time system. However, we should note that there is an overhead associated with dynamic priority changes, and that this overhead must be accounted for during the timing and scheduling analysis of ROOM multi-threaded executables with dynamic priority thread assignment.

4.2 Assigning Message Priorities

The previous discussion has presented guidelines based on the assumption that message priorities are known. In reality, message priorities are themselves artifacts of ROOM and more generally, real-time system design and should be derived in a systematic manner from system requirements. Since system behavior and the timing constraints are described in terms of transactions, we first assign priorities to transactions and then derive message priorities.

If we trace the sequence of transition actions triggered by an external message, we will form a transaction tree. An example of the transaction tree is given in Figure 9 with each directed path (branch) representing a ROOM transaction.

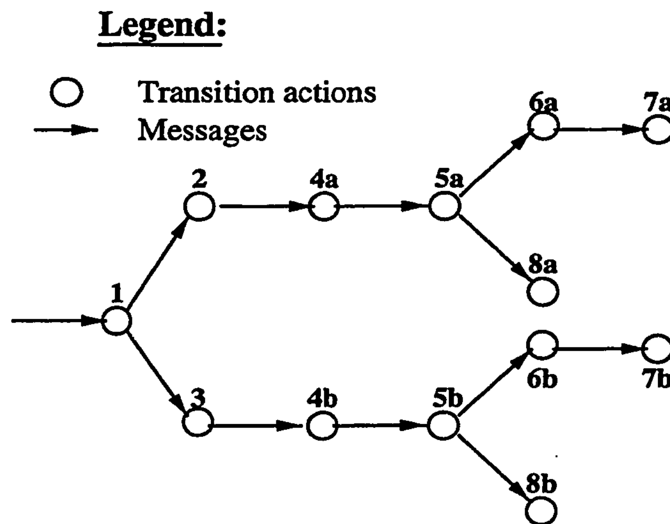

Figure 9: ROOM Transaction Tree

Now we assign a priority to each transaction [2] according to its importance as defined in the system requirements, i.e. an emergency transaction will get a higher priority than a data logging transaction. Finally, each message within a transaction tree is assigned the *dynamic* priority of the highest priority transaction to which this message belongs. Notice that this dynamic message priority assignment allows the same message to be sent with different priorities, according to different transactions the message belongs to. Also, looking at our transaction tree in Figure 9, we can see that the messages triggered by transition actions 4a and 4b belong to two different transactions, yet physically, i.e. within a ROOM model,

[2] A transaction priority must fall within the priority range of its transaction graph.

these messages may represent the same ROOM message. To assign message priorities across the entire ROOM model, we must examine all transaction trees.

A number of factors must be considered in determining transaction priorities. First, since scheduling occurs based on transaction priorities, they have a direct impact on response times, and hence the respect of transaction deadlines. Thus, any assignment of transaction priorities must take transaction deadlines or activation periods into consideration. Since all the transactions within the same transaction tree have the same activation period, we chose to assign rate-monotonic priorities to each transaction tree and corresponding deadline-monotonic priorities to transactions within these transaction trees. Second, transaction priorities should be assigned such that any unnecessary processing is avoided. For example, in automobile cruise control, if the driver turns the cruise control off, the closed loop control execution must be aborted. By assigning higher priority to the "CruiseOff" transaction (even though it has a "looser" deadline) we can abort the control loop faster, and reduce the response time for the CruiseOff transaction.

4.3 Blocking due to Message Passing

The third source of blocking comes from priority inversion associated with access to shared data structures. In the case of multi-threaded ROOM models, actors communicate via inter-thread message passing mechanisms. The message passing itself requires the sharing of message queues, and therefore, the processing associated with sending and receiving messages can incur unbounded blocking. Immediate Inheritance and Priority Ceiling protocols [SRL90] can be used to bound the blocking time associated with such priority inversion. We propose the use of a simpler Immediate Inheritance protocol and show that using this protocol, such blocking may be bounded.

The protocol works as follows: let the *ceiling priority* of a message queue be the priority of the maximum priority transaction, i.e. of the highest priority message, that can access it. When a thread accesses a message queue, its priority is raised to the level of the ceiling priority. The priority returns to its previous value when the access is completed. In addition to ensuring mutual exclusion (no explicit locks are needed), the protocol ensures that a thread accessing its message queue can only be blocked once. In particular, when a thread tries to access its message queue, a lower priority thread (having a static priority assigned according to Guideline 2 or inheriting its priority according to Guideline 3) may be

accessing the same message queue and executing with a ceiling priority of equal or higher priority than the first thread. The thread will then have to wait for the lower priority thread to finish before it may execute. Based on the above discussion, we suggest the following implementation guideline for message passing using shared message queues.

Guideline 4 *Each message queue should have a ceiling priority which is equal to the highest priority of transactions than may send messages to this queue. The* send *and* receive *operations should be performed at the ceiling priority of the queue. No explicit locks are needed in either the send or receive operation.*

This guideline overrides Guidelines 2 and 3 in determining the priority at which the send and receive operations are performed in a thread. Thus, the sender thread raises its priority during a send operation when it accesses a queue with a higher ceiling priority. Likewise, if a thread receives messages then it will execute at the ceiling priority of the queue, and not at the static priority for the static thread priority assignment nor at the priority of the highest priority pending message for the dynamic thread priority assignment. Note that this priority change takes place when the actual message receive operation begins and lasts until the receive operation completes. It is important to ensure that blocking due to message passing is bounded, and this is presented as an explicit guideline.

Guideline 5 *The execution time for the message send and receive operations must be less than a specified bound determined by the time-scales of the timing constraints of the system.*

4.4 Blocking due to Run-to-Completion Semantics

A final source of blocking is due to the run-to-completion semantics of ROOM models, where the processing of a transition may be delayed if a previous transition within the same thread has not finished. This blocking is inherent in ROOM execution semantics, and therefore cannot be avoided. However, the scope of this blocking is limited to within the actors mapped to the same thread, and equals the maximum of the processing times associated with all of actor transitions over all of the actors. This is given as follows:

$$B_i^{RTC} = max\ c_j : Thread(j) = Thread(i)$$

where B_i^{RTC} denotes the blocking time due to run-to-completion for transition i and c_j denotes the execution time of transition j.

While it is impossible to completely eliminate run-to-completion blocking, it is possible to minimize its adverse effects. Our next guideline makes explicit the rule with which new functionality may be added to an actor, without affecting the schedulability of higher priority transactions to which this actor is associated.

Guideline 6 *For all of the actors within a thread, an upper bound on tolerable blocking time must be specified and enforced so that the execution times of any transition within these actors must not exceed this bound. If necessary, a transition may be divided into multiple transitions to meet this bound.*

Complementing the above guideline is the guideline about grouping actors into threads. It is advantageous to have as few threads as possible since many computational resources (e.g. message queues at the level of ROOM, thread execution stack) are allocated on a per-thread basis and scheduler implementation costs (inter-thread message passing, thread context-switch) are increased.

Guideline 7 *Two actors may be safely assigned to the same thread if the transitions in each actor associated with the lower priority transactions satisfy the blocking time bound of the higher priority transactions in the other actor.*

Chapter 5

Developing Canonical Scheduling Models for ROOM

In Chapter 3, we presented an idealized scheduling equation for timer-driven schedulers with the timer release jitter but without specifying other scheduler implementation costs, such as the timer blocking penalty, task context-switch overhead or priority inversion for the active task when lower priority tasks are being scheduled. In Chapter 2, we showed that `TargetRTS` implements a timer-driven scheduler with the asynchronous `TargetRTS` and OS timing services. In this chapter, we will present a generic resource scheduling model for the scheduling analysis of single-threaded and multi-threaded ROOM executables that includes the common overhead and blocking costs of `TargetRTS` and the underlying OS scheduler. Then we will develop a specific scheduling model for each executable. The generic and canonical scheduling models are based on observations presented in [SPFR98].

5.1 Generic Resource Scheduling Model

We let each transaction Γ_i with n transitions has the generic overhead cost O_i, generic blocking penalty B_i, specific overhead for single- or multi-threaded ROOM executables $O_i^{specific}$, specific blocking penalty for single- or multi-threaded ROOM executables $B_i^{specific}$ and total transaction execution time $C_i = \sum_{k=0}^{k=n-1} c_k$, where c_k is the worst case execution time of the kth transition within the Γ_i transaction. Then, our generic resource scheduling model for the worst case response time of transaction Γ_i scheduled by `TargetRTS` and

the underlying OS scheduler, is given by:

$$R_i^{n+1} = C_i + O_i + O_i^{specific} + B_i + B_i^{specific} + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot (C_j + O_j + O_j^{specific}) \quad (4)$$

where $hp(i)$ denotes the set of transactions of higher priority than transaction Γ_i. The above equation is a simple extension of the scheduling equation [ABR+93] for a hybrid event- and timer-driven scheduler. Notice that by replacing the execution time of each transaction C_j by $C'_j = C_j + O_j + O_j^{specific}$, by ignoring the blocking penalty B_i and $B_i^{specific}$ and the overhead costs O_i and $O_i^{specific}$ for transaction Γ_i, the above equation reduces to the timer-driven scheduling model presented in Chapter 3.

We mentioned in Chapter 4 that even a single-threaded ROOM executable contains two additional threads: the timer thread and the external (interface) layer thread. In our generic resource scheduling model, we will only include the implementation costs of the timer thread, although implementation costs of the external layer thread can be easily added as part of the blocking term. Before we can make precise the blocking and overhead terms of equation 4, we must describe how the timer thread is implemented. In particular, it controls a timer actor which implements the timing service provided by the ROOM virtual machine. The timer actor receives a request for timing services sent through the timing `SAP` to manage the timers set up by other actors. It blocks for the length of the shortest timeout period or until a new timeout request is received. Upon return, it sends expired `timeout` messages to the appropriate destination actors.

Thus, periodic activities can be implemented within any actor by setting a periodic timer through the timing service. The system then delivers a `timeout` message once every specified time period. For each periodic timeout, there are two messages: a `timeout` message sent from the timer actor, and an `informAt` message sent to the timer actor. Each `informAt` message contains the next timer expiration time, which is calculated based on the previous expiration time and the period of the timer. In the remaining part of this chapter, we will refer to the sending of the `informAt` message to the timer thread, the context-switch to the timer thread, the processing of the `informAt` message by the timer thread, and the context-switch back to the timeout requesting actor as timer *re-arming*. Before re-arming, a check is made for missed timeouts (i.e. those timeouts whose expire time has passed), and no messages are sent for the missed timeout intervals. A count of these missed timeouts can be requested by the application.

Now let us define the following common implementation cost parameters of the single-

and multi-threaded ROOM executables.

1. $c_{inter-send}$: It is an overhead associated with sending a message to an actor located in a different thread and involves inserting a message into the destination actor's thread inbox queue.

2. c_{cs}: This parameter is used to account for the context-switch overheads. We define the context-switch overhead to be the time it takes to change the priority of a thread, suspend and save the hardware state (registers, stack-pointer, instruction-pointer, page table pointers) of the running thread, select a new thread to run, load the new thread's saved hardware state, and then begin executing.

3. c_{rearm}: This is a more coarse-grained parameter to associate with the entire overhead of re-arming the timer. This overhead includes:

   (a) The overhead of sending the `informAt` message to the timer thread=$c_{inter-send}$.

   (b) The context-switch overhead to the timer thread=c_{cs}.

   (c) The processing associated with the `informAt` message in the timer thread=$c_{informAt}$.

   (d) The context-switch overhead back to the timeout requesting actor=c_{cs}.

Because the timer actor runs in a separate thread, the `timeout` and `informAt` message passing between the timer actor and any other actor, within a user-defined thread, involves inter-thread sends. Also, to minimize the release jitter of the `timeout` messages, we run the timer thread at the highest priority and all of the `informAt` messages are sent at the highest priority as well. On the other hand, the `timeout` messages are sent at the priority requested by the actor, which is in turn determined by the priority of its current transaction.

5.1.1 Generic Overhead for Higher Priority Transactions

In this section, we specify the overhead term of equation 4 representing the generic transaction implementation costs within single- and multi-threaded ROOM executables for higher priority transactions. In particular, every time the timer expires for a higher priority transaction, a context-switch from the currently executing thread to the timer thread occurs, the

timer actor sends the `timeout` message to the appropriate actor, context-switch to this actor (i.e. to the thread where the actor is located) occurs and finally the actor rearms the timer. Thus, the generic implementation costs (O_j) for each higher priority transaction within a single- and multi-threaded ROOM executable can be calculated as follows:

$$O_j = c_{inter-send} + c_{rearm} + 2 \cdot c_{cs} = 2 \cdot c_{inter-send} + 4 \cdot c_{cs} + c_{informAt}$$

5.1.2 Generic Blocking

Now let us specify the blocking term of equation 4 representing the generic transaction blocking penalty within single- and multi-threaded ROOM executables. The first source of generic blocking for a transaction Γ_i comes from the ROOM run-to-completion semantics and is given as follows:

$$B^{RTC}_{init(i)} = max\ c_j : Thread(j) = Thread(init)$$

where $B^{RTC}_{init(i)}$ denotes the blocking time due to run-to-completion semantics for the initial transition within transaction Γ_i and c_j denotes the worst case execution time of transition j.

The next source of generic blocking comes from the processing of timer expiration for timers set for lower priority transactions. (This is because the timer expire activates the timer thread, which runs at the highest priority in the system.) In particular, when the timer expires for a lower priority transaction, a context-switch from the currently executing thread to the timer thread occurs, the timer actor sends the `timeout` message to the appropriate actor and then there occurs another context-switch back to the original thread. We do not take the re-arming of the timer into account, since the control must pass to the lower priority transaction for that to occur. The timer re-arming will be accounted for as part of the overhead for the currently executing transaction. Therefore, this generic blocking term for a transaction Γ_i is given by the following equation:

$$B^{timeout}_i = \sum_{j \in lp(i)} (c_{inter-send} + 2 \cdot c_{cs})$$

where $lp(i)$ denotes the set of transactions of lower priority than transaction Γ_i.

Finally, we must include the effect of timer blocking (B_{timer}) [1], since the transaction's release may be delayed by the operating system due to the coarse granularity of its system

[1]The transaction release jitter overhead (J_{timer}) is a jitter effect caused by the timer thread processing associated with the `informAt` message, and is included in our equations as the $c_{informAt}$ term.

clock and the asynchronous nature of the `TargetRTS` timing service and the system clock. Thus, the generic blocking penalty (B_i) can be calculated as follows:

$$B_i = B^{RTC}_{init(i)} + B^{timeout}_i + B_{timer}$$

5.1.3 Generic Overhead

Finally, we specify the overhead term of equation 4 representing the generic implementation costs for currently executing transactions. This overhead includes sending a `timeout` message from the timer thread and, as we mentioned before, timer re-arming.

$$O_i = c_{inter-send} + c_{rearm} = 2 \cdot c_{inter-send} + 2 \cdot c_{cs} + c_{informAt}$$

5.2 Single-Threaded Executable

As we know the simplest way to implement a real-time system designed in ROOM is to assign all user defined application actors to a single user thread. Before we define the specific overhead and blocking terms for this model, we will present the reader with the additional implementation cost occurring in the single-threaded ROOM executable.

1. $c_{intra-send}$: It is an overhead associated with sending a message to an actor within the same thread and involves inserting the message into the thread message queues according to the message priority without changing the thread's priority.

5.2.1 Specific Overhead for Higher Priority Transactions

In this section we specify the overhead term of equation 4 representing the specific implementation costs, within a single-threaded ROOM executable, for higher priority transactions. Because all user defined actors are placed in one user defined thread, when a higher priority transaction completes, there is no context-switching between threads. However, we have to account for delays associated with message passing between actors within each transaction. Thus, the specific implementation costs ($O^{specific}_j$) for each higher priority transaction Γ_j within a single-threaded ROOM executable is given by the following equation:

$$O^{specific}_j = (size(M_j) - 1) \cdot c_{intra-send}$$

where $M_j = \{m_0, m_1, \ldots, m_{n-1}\}$ is the set of messages associated with each higher priority transaction Γ_j.

5.2.2 Specific Blocking

Here, we define the specific transaction blocking term within a single-threaded ROOM executable. Because there is only one user thread, a transaction is blocked only once due to run-to-completion semantics. In particular, when the transaction is released it can be only preempted by a higher priority transaction and only after the currently executing transition runs to completion. It is also possible that when a higher priority transaction is invoked, a lower priority transaction is in the process of re-arming its timer. Therefore, the specific blocking penalty ($B_i^{specific}$) for single-threaded ROOM executables is given by:

$$B_i^{specific} = c_{rearm} = c_{inter-send} + 2 \cdot c_{cs} + c_{informAt}$$

5.2.3 Specific Overhead

When we consider the specific overhead for currently executing transactions within a single-threaded ROOM model, we only have to account for intra-thread message passing delays between actors, as previously described for higher priority transactions.

$$O_i^{specific} = (size(M_i) - 1) \cdot c_{intra-send}$$

where $M_i = \{m_0, m_1, \ldots, m_{n-1}\}$ is the set of messages associated with the current transaction Γ_i.

5.2.4 Canonical Scheduling Model

By combining all of the above specific implementation costs along with the generic implementation costs presented in the previous section to the generic resource scheduling model defined in equation 4, we obtain the following canonical scheduling model for the single-threaded ROOM executable:

$$\begin{aligned} R_i^{n+1} &= C_i + 3 \cdot c_{inter-send} + 4 \cdot c_{cs} + 2 \cdot c_{informAt} + (size(M_i) - 1) \cdot c_{intra-send} \\ &+ B_{init(i)}^{RTC} + \sum_{j \in lp(i)} (c_{inter-send} + 2 \cdot c_{cs}) + B_{timer} \\ &+ \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot (C_j + 2 \cdot c_{inter-send} + 4 \cdot c_{cs} + c_{informAt} + (size(M_j) - 1) \cdot c_{intra-send}) \end{aligned} \quad (5)$$

5.3 Multi-Threaded Executable

Based on our results for single-threaded ROOM executables, we now define the implementation costs specific to a multi-threaded ROOM model with dynamic thread priority assignment, where according to guideline 3, thread priority will change so as to be equal to the highest priority pending message.

5.3.1 Specific Overhead for Higher Priority Transactions

Let us specify the overhead term of equation 4 representing the specific implementation costs within a multi-threaded ROOM executable with dynamic thread priority assignment for higher priority transactions. In the worst case scenario, it is required that each user defined actor is placed into a separate thread. In this configuration, each higher priority transaction will undergo a thread context-switch when its processing is completed. Also, we have to account for inter-thread message passing delays between actors within each transaction. Thus, the specific implementation costs ($O_j^{specific}$) for each higher priority transaction Γ_j within a multi-threaded ROOM executable is given by the following equation:

$$O_j^{specific} = c_{cs} + (size(M_j) - 1) \cdot c_{inter-send}$$

where $M_j = \{m_0, m_1, \ldots, m_{n-1}\}$ is the set of messages associated with each higher priority transaction Γ_j.

5.3.2 Specific Blocking

Let us also define the specific transaction blocking term within a multi-threaded ROOM executable. Because in the worst case a ROOM model may consist of n threads corresponding to n user defined actors, a higher priority transaction may be blocked by a lower priority transaction each time a message is sent to another actor.

It is also possible that when a transaction is invoked, a lower priority transaction is in the process of re-arming its timer. This blocking term is bounded by the processing time (c_{rearm}) associated with re-arming the timer thread as previously described.

Finally, there is also blocking due to shared access to message queues. The message queues of each thread are accessed by other threads during inter-thread send operations. Thus, blocking is incurred if a transaction is invoked when a lower priority transaction is

sending a message to the invoked transaction's thread. We will refer to this blocking time as B_i^{MQ}. Using the Immediate Inheritance Protocol [BW96], we access shared message queues at the ceiling priority of the queue. This ensures that a higher priority transaction can be blocked by either the timer re-arming or message queue access, but not both. Therefore, the specific blocking penalty ($B_i^{specific}$) for multi-threaded ROOM executables is given by:

$$B_i^{specific} = max(c_{rearm}, B_i^{MQ}) + \sum_{j \in all(i) \wedge j \notin init(i)} B_j^{RTC}$$

where B_j^{RTC} denotes the blocking due to run-to-completion semantics for each transition Γ_j, except the initial transition triggered by the external (real world) message, of the current transaction Γ_i.

5.3.3 Specific Overhead

When we consider the specific overhead for currently executing transactions within a multi-threaded ROOM model, we only have to account for inter-thread message passing delays between actors.

$$O_i^{specific} = (size(M_i) - 1) \cdot c_{inter-send}$$

where $M_i = \{m_0, m_1, \ldots, m_{n-1}\}$ is the set of messages associated with the current transaction Γ_i.

5.3.4 Canonical Scheduling Model

Thus, the canonical scheduling model for multi-threaded ROOM executables with dynamic thread priority assignment is given by the following equation:

$$\begin{aligned} R_i^{n+1} &= C_i + (size(M_i) + 1) \cdot c_{inter-send} + 2 \cdot c_{cs} + c_{informAt} \\ &+ \sum_{j \in lp(i)} (c_{inter-send} + 2 \cdot c_{cs}) + B_{timer} + \sum_{j \in all(i)} B_j^{RTC} + max(c_{rearm}, B_i^{MQ}) \\ &+ \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot (C_j + (size(M_j) + 1) \cdot c_{inter-send} + 5 \cdot c_{cs} + c_{informAt}) \end{aligned} \quad (6)$$

5.4 Validating Canonical Scheduling Models

In order to validate our canonical models, we have to estimate the implementation cost parameters defined in the two previous sections. We then have to use these canonical scheduling models along with their estimated worst-case (maximum) implementation overheads to

predict the transaction response times of our sample ROOM models. We then compare the predicted response times to the empirically measured results. All of our experiments were performed on a Sun UltraSPARC-I work station, with a 167Hz clock and 128 MBytes of main memory, running Solaris 2.5. All of our timing measurements were made using the `clock_gettime` call provided by the real-time library of Solaris (the same call was used in `TargetRTS` for the timing service). The overhead for the `clock_gettime` call was measured to be between 3 and 4 microseconds [2] The clock resolution, as provided by the `clock_getres` function call, is 1 microsecond. The timer blocking term (B_{timer}) was estimated to be as high as 10 milliseconds, according to the granularity of the system clock. The measurements of other implementation cost parameters were as follows:

Overhead	Min (μsec)	Max (μsec)	Mean (μsec)	Std. Deviation (μsec)
$c_{intra-send}$	19.848	25.708	20.368	0.844 (4.14%)
$c_{inter-send}$	106	141.368	120.619	7.301 (6.05%)
c_{cs}	78.123	179.346	104.834	6.183 (5.90%)
$c_{informAt}$	17.032	86.698	62.489	12.602 (20.17%)

Table 6: Implementation Overheads

Transition execution times of each transaction included in our sample ROOM models were implemented with a spin loop, as given below, with the value of the `loopCount` variable determined according to the desired transition execution time.

```
while (i<loopCount) {
    j=&i;
    i++;
}
```

Using measurements, it was estimated that 10.3 loops were required for each microsecond of execution time. Thus, for a task of $10ms$ or higher execution time, this resulted in less than 1% difference in the actual measurement.

[2]The `clock_gettime` overhead was measured by inserting a time-stamp before and after this function call.

5.4.1 First ROOM Model

In this section, we validate the canonical scheduling models using the set of periodic transactions, as shown in Table 7, under the single- and multi-threaded ROOM executables. We will assume that each periodic transaction is implemented using the ROOM timing service and a periodic actor. The state machine of such a periodic actor is trivial and only responds to timeout messages. When the timeout message arrives, the computation associated with the periodic actor is performed on a single transition with the execution time equal to the execution time of the corresponding transaction. In the multi-threaded configuration, each periodic actor was identified as a data processing actor and was assigned to a separate thread. We will further assume that the computation neither blocks nor sends a message to another actor.

			Single-Threaded		Multi-Threaded	
Transaction	C_i (ms)	T_i (ms)	$R_i^{predicted}$ (ms)	$R_i^{measured}$ (ms)	$R_i^{predicted}$ (ms)	$R_i^{measured}$ (ms)
Γ_1	5	50	44.402	21.687	19.581	9.744
Γ_2	8	50	46.902	28.583	28.347	12.940
Γ_3	10	100	48.402	24.762	39.113	33.693
Γ_4	15	100	52.902	39.218	70.411	53.278
Γ_5	25	200	52.402	61.635	96.177	85.808

Table 7: Transaction Specifications for the First ROOM Model

The above table shows that the measured multi-threaded worst-case response times fall below the predicted multi-threaded worst-case response times as desired. The single-threaded measurements are also consistent with the predicted single-threaded worst-case response times, except the measurement for the Γ_5 transaction being approximately $9.2ms$ larger than expected. This error was introduced by the `TargetRTS` timer jitter (approximately $10ms$), caused by missed "timeouts" due to the hardware interrupts such as network, keyboard and mouse interrupts. The numbers also reveal the conservative nature of the canonical scheduling models. The overestimation was found largely due to (a) overestimation of the timer blocking, and (b) overestimated implementation overheads [3]. The

[3] When two transactions are released at the same time, the full overheads of the *informAt* message processing are not applicable. This is because the timer thread processes all of the expired timeouts when it wakes up.

measured and predicted results also reveal smaller processor utilization and longer response times of higher priority transactions within the single-threaded ROOM model than within the multi-threaded ROOM model.

5.4.2 Second ROOM Model

The transition and transaction specifications for the second validating model are shown in Tables 8 and 9. Here, we will assume that each periodic transaction is implemented using the ROOM timing service and two actors. The first actor responds to five timeout messages (each with a different period), it executes transition τ_1 and sends a message to the appropriate second actor which executes its single transition. In the multi-threaded configuration, $actor_1$ was identified as a control actor, while others were identified as data processing actors; each actor was then assigned to a separate thread. The following ROOM model also assumes no transition blocking.

Transition	$c_i(ms)$	$Actor_i$
τ_1	4	1
τ_2	1	2
τ_3	4	3
τ_4	6	4
τ_5	11	5
τ_6	21	6

Table 8: Transition Specifications for the Second ROOM Model

			Single-Threaded		Multi-Threaded	
Transaction	C_i (ms)	T_i (ms)	$R_i^{predicted}$ (ms)	$R_i^{measured}$ (ms)	$R_i^{predicted}$ (ms)	$R_i^{measured}$ (ms)
Γ_1	$5(c_1+c_2)$	50	23.428	21.255	23.863	25.247
Γ_2	$8(c_1+c_3)$	93	32.015	37.256	32.770	32.435
Γ_3	$10(c_1+c_4)$	100	42.602	25.338	43.677	32.610
Γ_4	$15(c_1+c_5)$	100	64.276	39.801	65.991	47.621
Γ_5	$25(c_1+c_6)$	200	85.863	64.256	87.898	78.363

Table 9: Transaction Specifications for the Second ROOM Model

The results obtained from the second ROOM model also show that the measured single-threaded and multi-threaded worst-case transaction response times fall below the predicted worst-case response times, as desired. As previously, some measured response times are larger than expected due to the `TargetRTS` timer jitter of approximately 10*ms* not being included in our canonical scheduling models. The overestimation of other predicted results was found largely due to (a) overestimation of the timer blocking, (b) overestimated implementation overheads, and (c) overestimated preemption of lower priority transactions within the single-threaded ROOM executable [4]. Because of the overestimated preemption of lower priority transactions within single-threaded ROOM executables, the predicted response times for single- and multi-threaded executables vary largely due to the time differences between intra- and inter-message passing mechanisms. However, the measured response times for single- and multi-threaded ROOM executables still reveal smaller processor utilization and longer response times of higher priority transactions within the single-threaded ROOM model than within the multi-threaded ROOM model.

[4]The single-threaded canonical model assumes that lower priority transactions can be preempted at any time during their execution, but in reality they are only preempted during their message passing time intervals.

Chapter 6

Case Study of Automobile Cruise Control

To illustrate the concepts developed earlier in this thesis, we will use a variant of an automobile cruise control system, presented in [SFR97]. Here, we assume that the current automobile speed is readily available within the `Speedometer` actor. In [SFR97], the current automobile speed was calculated from an external interrupt event generated by the engine drive shaft. We also modified the behavior of the system under driver control. In particular, our `ManualControl` state contains the `Accelerating`, `Braking` and `ReadytoCruise` states; also the automobile cruise control can switch to state `AutomaticControl` whenever the driver shifts the cruise control lever to either the *cruise* or *resume* position while neither the brake nor the accelerator pedals are pressed. In [SFR97], the `ManualControl` state contained the `Initial`, `Accelerating`, `NotBraking` and `Braking` states and the cruise control was switched to `AutomaticControl` when the driver shifted the cruise control lever to either the *cruise* or *resume* position when the brake pedal was not pressed, or the driver shifted the cruise control level to the *cruise* position when the car was started, or he released the accelerator pedal in the `Accelerating` state.

Automobile cruise control is a well studied example which has been used to illustrate real-time designed methods such as Octopus [AKZ96], ADARTS and CODARTS [Gom93]. In order to keep the example manageable, we have selected only a subset of its functionality. The cruise control system presented here includes simple closed-loop control behavior (maintaining automobile speed at a desired cruising speed), and responds to external events

triggered by the driver (e.g. engaging the cruise control, pressing the brake, etc.). Thus, it includes both time- and event-driven behaviors.

The primary function of the cruise control system is to perform automated speed control, which is achieved through a closed-loop feedback control system. The loop is initiated whenever the driver enables the cruise control and it used to maintain the speed of the car at the desired cruising speed. The closed-loop is also used to accelerate or decelerate to a memorized cruising speed and it is initiated whenever the driver engages the cruise control system. In either case, the closed-loop control is triggered by a periodic `timeout` message. The closed-loop processing involves determining the current speed of the car and updating the engine throttle value based on the current and desired speed. A subsidiary function keeps track of the current speed of the car, and continually updates the speedometer display. In addition to maintaining the speed of the car, the system must respond to real-world events triggered by the driver.

6.1 Structural Model

We have created a ROOM model by first developing actors that serve as hardware wrappers for the input and output devices. The main functionality of the system is embodied in the `CruiseControl` actor. The structural model of the ROOM automobile cruise control system is shown in Figure 10[1]. As can be seen in the figure, the `CruiseControl` actor interacts with all of the other actors through its interfaces. We have not shown the interaction of the system with with the external world, which is done through the timer `SAP` using the timing service provided by the ROOM virtual machine. Although the interaction of actors with the external world does not differ depending on the time- or event-driven behavior, the way in which these services are set up is different. In particular, the event-driven behavior is specified through an interrupt file [2] and the time-driven behavior is simply initiated by a time-driven actor. Also, we will assume that the ROOM virtual machine will handle all interrupts correctly and send them as messages to the hardware wrappers.

[1] `CruiseControl` actor encapsulates all of the other automobile cruise control actors.
[2] The interrupt file specifies time stamps of interrupts for each event-driven actor.

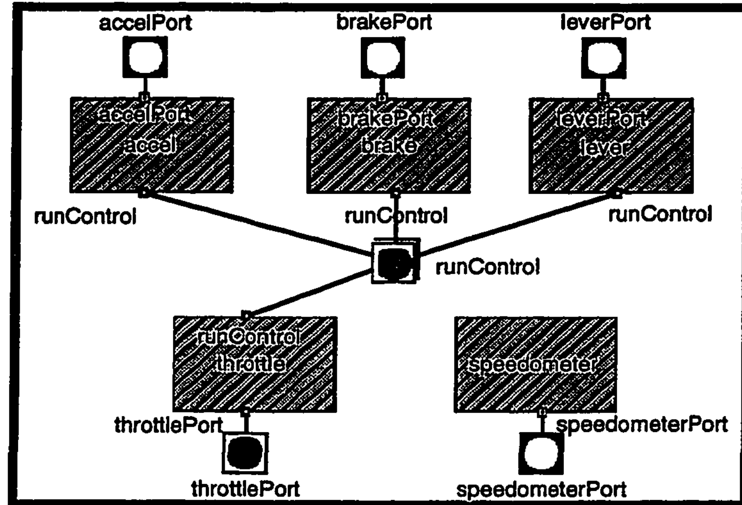

Figure 10: Cruise Control Structural Model

6.2 Behavioral Model

Because the response of the system depends on its internal state, i.e. when the system receives a "break" event it could be in the process of maintaining the speed of the car at the desired cruising speed or accelerating/decelerating to the memorized cruising speed, the system behavior can be most easily specified with a finite state machine model. We divided the behavior of the ROOM automobile cruise control actors into two groups: data-processing and control actors. As we know from Chapter 4, the data processing actors have compute-intensive message processing and simple state machines (each actor usually accepts only one type of transaction), while control actors have non compute-intensive message processing with more complex state machines. The `Accelerator`, `Brake`, `Lever`, `Speedometer` and `Throttle` actors belong to the first group, while `CruiseControl` belongs to the second group.

6.2.1 Description of Data Processing Actors

The behavior of these actors is very simple as shown in Figure 11. The brake, accelerator, and the lever actors have only a single state. They receive a `timeout` message from the underlying ROOM virtual machine when an interrupt is generated for the corresponding

device. For example, the brake actor sends a `brakePressed` or `brakeReleased` message to the cruise control actor, depending on the event which occurred. These actors are responsible for passing the appropriate message to the `CruiseControl` actor and they are redundant in our system since their role is to simply relay the messages received from the underlying ROOM virtual machine. However, in the context of a larger system which may include an anti-lock brake system, they may be necessary and may incorporate additional functionality.

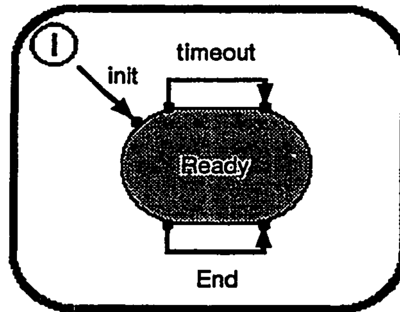

Figure 11: Accelerator, Brake and Lever Actor Behavior

As shown in Figure 12, the `Throttle` actor has a single input port on which it receives a `throttleValue` message from the cruise control actor, and sends an update throttle command to the throttle device.

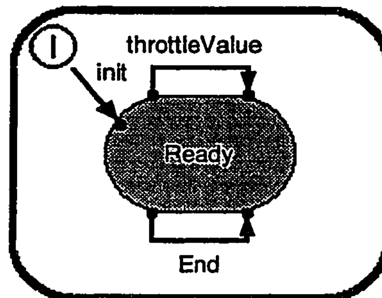

Figure 12: Throttle Actor Behavior

The `Speedometer` actor is responsible for providing the current speed of the automobile to the closed-loop feedback control; see Figure 13. It receives a `speedRequest`

message from the cruise control actor, and returns the current speed of the automobile in a `speedValue` message. Its behavior is not state dependent, and it includes a single state. For clarity we assume that the current automobile speed is always readily available within the `Speedometer` actor. In reality, we have to calculate the current speed from an external interrupt event generated by the engine drive shaft. This new speed is then sent to the external speedometer device.

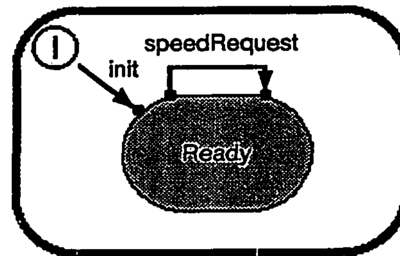

Figure 13: Speedometer Actor Behavior

6.2.2 Description of the Cruise Control Actor

The cruise control actor synchronizes cruise control system activities which are initially triggered by the arrival of an external event, i.e. brake is pressed, accelerator is released, cruise control is engaged, etc. The behavior of the cruise control actor is specified with a hierarchical state machine.

Top Level Behavior of the Cruise Control actor

At the top level, the cruise control behavior is characterized by a hierarchical state machine with two composite states: the `ManualControl` state specifying the manual (driver) automobile control, and the `AutomaticControl` state specifying the automatic control of the car. Figure 14 depicts the transitions between the two states. As can be seen, the system initially starts in the `ManualControl` state, and the speed of the car is under the driver's control. The cruise control system is switched to the `AutomaticControl` state whenever the driver shifts the cruise control lever to either the *cruise* or *resume* position while neither the brake nor the accelerator pedals are pressed. On the other hand, the

automatic control of the vehicle can be disabled by the driver, whenever he presses the brake or accelerator pedal, or explicitly turns the cruise control off.

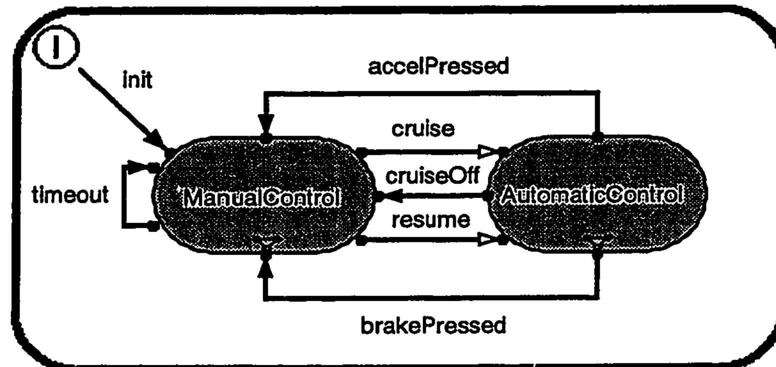

Figure 14: Top-Level Behavior of Cruise Control Actor

Automatic Control Behavior

Figure 15 shows the top level of the hierarchical state machine for the behavior of the car under automatic control. In particular, the `Automatic` state encompasses two other composite states: the `Resuming` and `Cruising` state.

In the `Resuming` state, depicted in Figure 16, the car automatically accelerates or decelerates to the last memorized cruising speed, and then switches to the `Cruising` state, in which the desired cruising speed is maintained. The decomposition of these states is similar and performs the operation of the closed-loop feedback control, triggered by a periodic `timeout` message.

Manual Control Behavior

In the `ManualControl` state, the speed of the car is under driver control, and the closed-loop for the cruise control is inactive. We assume that the periodic timers necessary for the closed-loop control operation are turned off when the cruise control leaves the `AutomaticControl` state. The `ManualControl` state is decomposed into a number of sub-states, such as `Accelerating`, `Braking` and `ReadytoCruise`, to keep track of whether the driver is accelerating, braking or the cruise control is ready to enter the

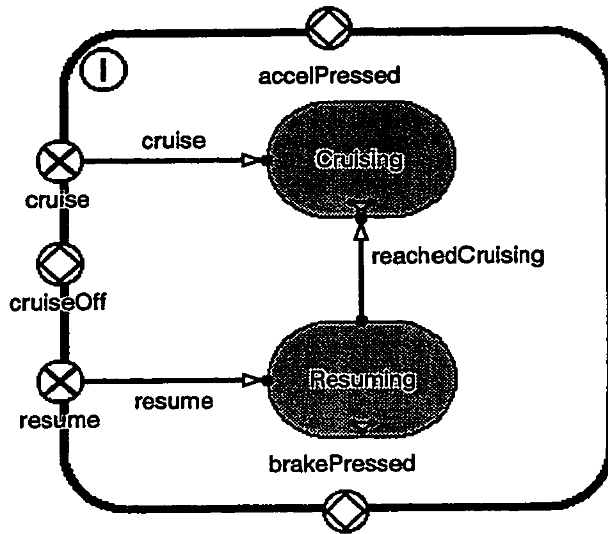

Figure 15: Automatic Control Behavior of Cruise Control Actor

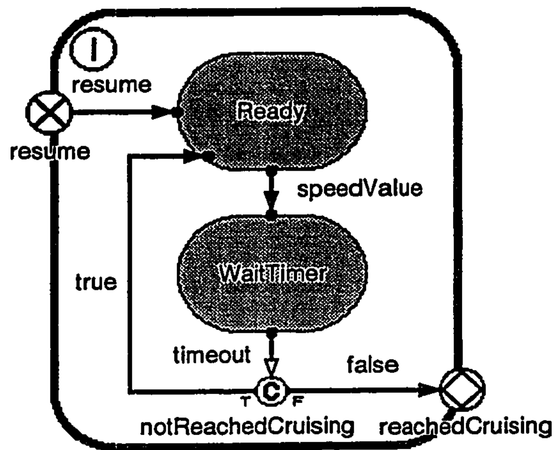

Figure 16: Resume Cruising Control Behavior

AutomaticControl state. Figure 17 shows the state machine describing the behavior of the system under driver control.

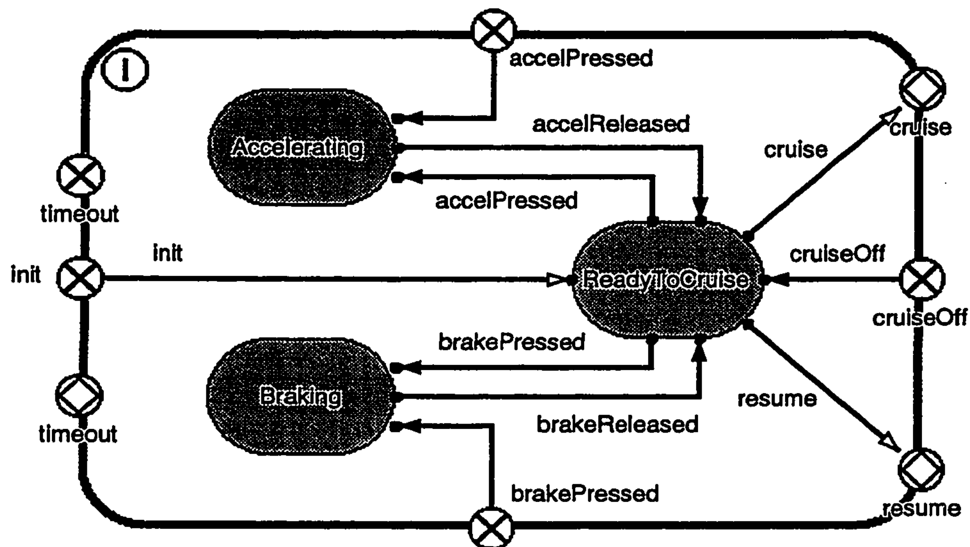

Figure 17: Manual Control Behavior of Cruise Control Actor

6.3 Transactions and Timing Constraints

During the cruise control operation, a number of transactions which we listed in Table 10 along with their timing constraints, take place. A description of these transactions is given below.

6.3.1 Closed-Loop Feedback Control

There are two transactions (i.e. the CruiseControl actor is in the Cruising state or it is in the Resuming state) that perform the closed-loop feedback control operation. Because their behavior and requirements are identical, we grouped them together under a single ControlLoop transaction. This transaction has an activation period of 100*ms*, and a deadline of 100*ms*. It takes place when the system is in AutomaticControl state. Figure 18 illustrates the message passing sequence for such a transaction. The transaction is triggered by a periodic timeout message sent to the CruiseControl actor from

Transaction	Stimulus	Period	Deadline
ControlLoop (CL)	`timeout`	100ms	100ms
EnterCruise (EC)	`cruise`	-	200ms
ResumeCruise (RC)	`resume`	-	200ms
CruiseReached (CR)	`speedReached`	-	200ms
BrakePressed (BP)	`brakePressed`	-	50ms
AccelPressed (AP)	`accelPressed`	-	50ms
CruiseOff (CO)	`cruiseOff`	-	150ms

Table 10: Description of Automobile Cruise Control Transactions

the timing service. The `CruiseControl` actor sends a `speedRequest` message to the `Speedometer` actor, which returns the current speed in a `speedValue` message. The `CruiseControl` actor then computes the throttle value by applying the control law, and sends a `throttleValue` message to the `Throttle` actor, which outputs the updated value to the external device.

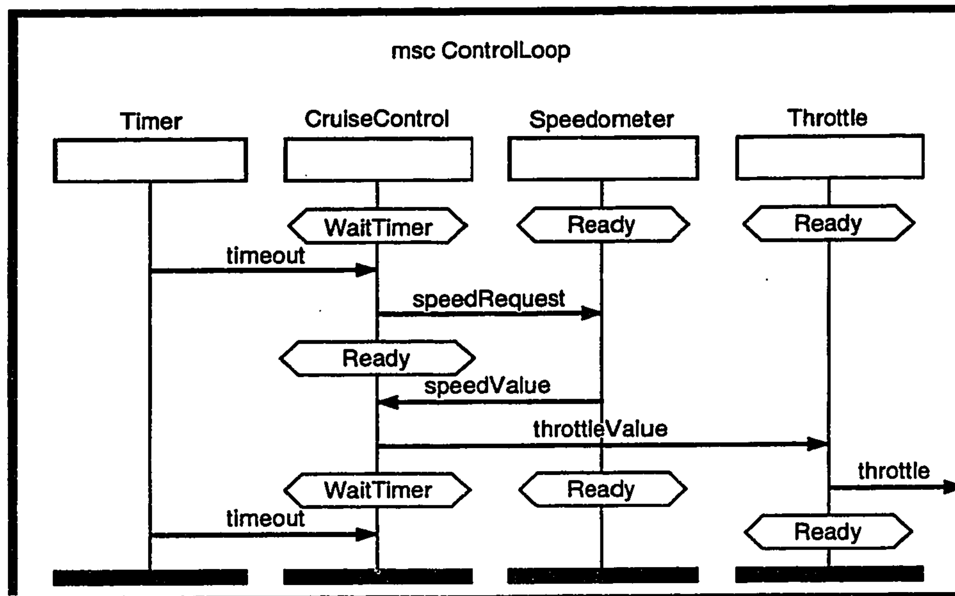

Figure 18: Message Sequence Chart for ControlLoop Transaction

6.3.2 Entering and Preparing for Automatic Cruise Control

As in the previous section, there are two transactions that mark a state change from the `ManualControl` to `AutomaticControl` state, and they have similar behavior and characteristics. In particular, when the lever is set to the cruise position and neither brake nor accelerator pedals are pressed, a `cruise` message is sent to the `CruiseControl` actor that triggers a transition to the `AutomaticControl.Cruising.Ready` state and sends a message to the `Speedometer` actor to get the current speed. When the speed value is returned, the `CruiseControl` actor updates the throttle device, the transaction is completed and we say that the cruise control is active. On the other hand, when the lever is set to the resume position and neither brake nor accelerator pedals are pressed, a `resume` message is set to the `CruiseControl` actor that triggers a transition to the `AutomaticControl.Resuming.Ready` state. Then, the following sequence of events is identical to the previous transaction until a cruising speed is reached. When the cruising speed is reached, the automobile cruise control changes its state to `AutomaticControl.Cruising` where it continues to execute the closed-loop control. All of these transactions are aperiodic, and therefore no activation rate is specified. The deadline for these transactions is $200ms$. Figure 19 depicts the message sequence chart for the EnterCruise (EC) transaction.

6.3.3 Exiting Automatic Cruise Control

Finally, there are several transactions associated with leaving the `AutomaticControl` state and entering the `ManualControl` state. These transactions are triggered by pressing the brake or accelerator pedal, or by switching the cruise control lever to the off position. For example, the BrakePressed transaction, triggered by a `brakePressed` message sent to the `Brake` actor, takes the `CruiseControl` actor out of the `AutomaticControl` and into `ManualControl` state which marks the end of the transaction, since the automobile speed is no longer under automatic control. All of these transactions are aperiodic and have no specified arrival time. The deadline for the BrakePressed transaction is very small and equals $50ms$, since it should reflect an urgent end to the cruise control. The deadline for the transaction arising from the accelerator being pressed is $100ms$, since it is less urgent than the previous one. The CruiseOff transaction

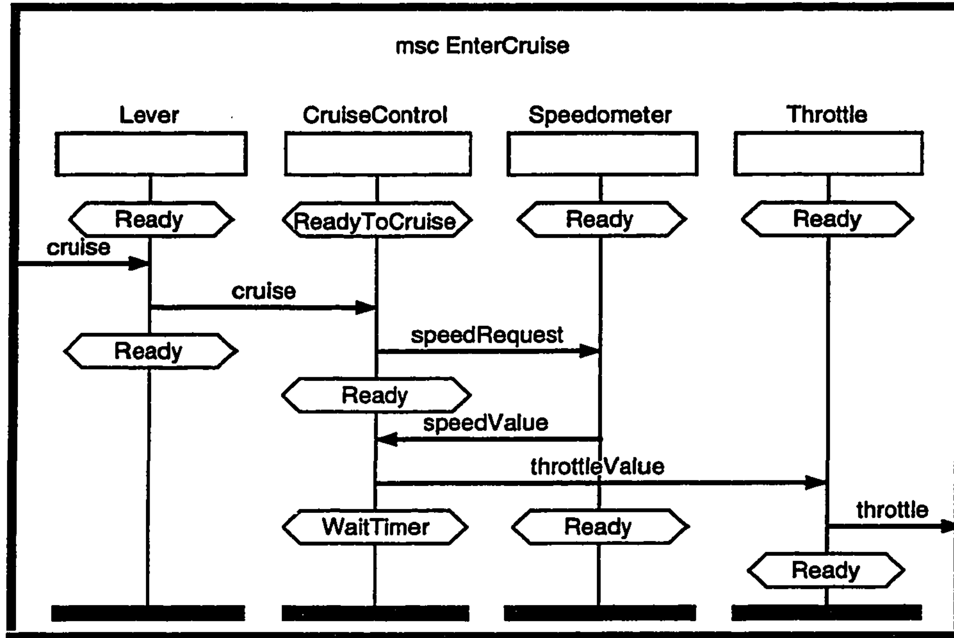

Figure 19: Message Sequence Chart for EnterCruise Transaction

deadline was set at $150ms$. Figure 20 depicts the message sequence chart for the Brake-Pressed (BP) transaction.

6.4 Schedulability Analysis

In this section, we will further validate the ROOM canonical models developed in Chapter 5 on a real-life system using our automobile cruise control example. To perform the timing and scheduling analysis, we will use the worst-case computation times of each transition as presented in Table 11.

CruiseControl	Speedometer	Other Actors
$c_{timeout} = 2ms$	$c_{speedRequest} = 3ms$	$c_* = 2ms$
$c_{speedValue} = 10ms$		
$c_* = 5ms$		

Table 11: Transition Computation Times

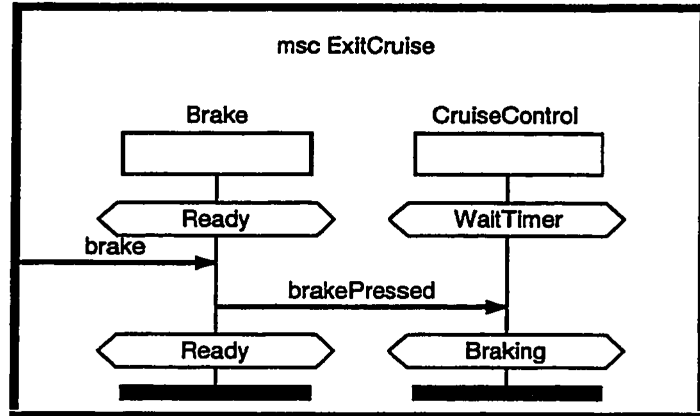

Figure 20: Message Sequence Chart for BrakePressed Transaction

Since the processing of the `speedValue` transition requires control law calculation, it is assigned a high execution time of 10*ms*. All other processing in the `CruiseControl` actor is given 5*ms* computation time, since it may involve either starting or revoking the cruise control's periodic timer. The execution time of the `speedRequest` transition in the `Speedometer` actor is assigned 3*ms*, since it involves speed calculation. An upper bound of 2*ms* is placed on all other transitions.

The transaction graph priorities are initially assigned in a rate-monotonic manner. Also, since there are no disjunctive transitions in our ROOM model of the automobile cruise control (each external message initiates a unique transaction), the transactions are assigned the priorities of their transaction graphs. Then, since the transactions that exit the `AutomaticControl` state (BP, AP and CO) terminate the closed-loop control transactions (CL), we raise their priorities above the priority of the CL transaction. Message priorities are assigned in accordance with the discussion in Section 4.2. Final transaction priorities are shown in Table 12.

In the multi-threaded ROOM configuration, the `CruiseControl` actor was identified as a control actor and was assigned to a control thread; the `Accelerator`, `Brake`, `Lever`, `Speedometer` and `Throttle` actors were identified as data processing actors and assigned to separate data processing threads.

6.4.1 Response Time Analysis

We now proceed with the response time analysis of all of the transactions within our automobile cruise control system. In the computing of the blocking and interference terms from other transactions, we take into account the enabling conditions for a transaction. For example, although the EnterCruise (EC) and ResumeCruise (RC) transactions which mark entry into the `AutomaticControl` state are lower priority than the ControlLoop transaction (CL), they can not be preempted by CL. Since the CL transaction is initiated by either EC or RC, it is only active in the `AutomaticControl` state and it can not be invoked while the EC or RC transaction executes.

`TargetRTS` and the underlying operating system (OS) implementation overheads were measured in Chapter 5 and will be directly used in the timing and scheduling analysis of the ROOM model of our cruise control system.

Closed-Loop Feedback Control. Let us turn our attention to the ControlLoop (CL) transaction. Although the BrakePressed (BP), AccelPressed (AP), and CruiseOff (CO) transactions have higher priority than CL, we do not need to consider their preemption effect, since any of those transactions terminates the CL transaction. There are four transitions in the CL transaction as shown in Figure 18 with the worst-case execution times (specified in Table 12) of 2ms, 3ms, 10ms and 2ms respectively.

Entering and Preparing for Automatic Cruise Control. The EnterCruise (EC) and ResumeCruise (RC) transactions have identical behavior. They contain five transitions as shown in Figure 19 with the bounded execution times of 2ms, 5ms, 3ms, 10ms and 2ms respectively. The CruiseReached (CR) transaction contains five transitions of which the first transition cancels the cruise control's periodic timer in the `Resuming` state, and the second transition starts the cruise control's periodic timer in the `Cruising` state. The remaining three transitions are identical to those of EC and RC. All of the above transactions are not preempted by any other transaction within our ROOM model.

Exiting Automatic Cruise Control. Finally, we look at the transactions (BP, AP and CO) that exit the `AutomaticControl` state. Each of them contains two transitions as shown in Figure 20 with the worst-case execution times of 2ms and 5ms. As before, the BP, AP and CO transactions are not preempted by other transactions within the ROOM model of the automobile cruise control system.

Transaction	C_i (ms)	T_i (ms)	P_i	Single-Threaded $R_i^{predicted}$ (ms)	Single-Threaded $R_i^{measured}$ (ms)	Multi-Threaded $R_i^{predicted}$ (ms)	Multi-Threaded $R_i^{measured}$ (ms)
ControlLoop	17 (2+3+10+2)	100	3	28.392	20.912	28.739	20.864
EnterCruise	22 (2+5+3+10+2)	200	4	33.418	24.511	33.880	31.815
ResumeCruise	22 (2+5+3+10+2)	200	4	33.418	33.128	33.880	30.984
CruiseReached	25 (5+5+3+10+2)	200	4	26.418	25.234	26.880	26.462
BrakePressed	7 (2+5)	50	1	29.840	17.013	29.956	16.654
AccelPressed	7 (2+5)	50	1	29.840	10.506	29.956	12.581
CruiseOff	7 (2+5)	150	2	28.840	9.482	28.956	17.584

Table 12: Specifications of Automobile Cruise Control Transactions

Since all of the measured worst-case transaction response times fall below the predicted response times, the response time analysis of our automobile cruise control example further validates the canonical scheduling models developed in Section 5. Note that the cruise control transactions are not preempted by higher priority transactions, and they are subjected to the same blocking intervals within a single- and multi-threaded executable. Consequently, the predicted response times for the single- and multi-threaded executables only vary due to the time differences between intra- and inter-message passing delays.

The most accurate response was measured for the `CruiseReached` transaction, since the response time analysis for this transaction does not introduce the `TargetRTS` timer blocking term (B_{timer}) [3]. To calculate other predicted response times the timer blocking term of $10ms$ was used. The least accurate response was measured for the `BrakePressed`, `AccelPressed` and `CruiseOff` transactions due to their possible run-to-completion blocking of $10ms$ within the `CruiseControl` actor. When the measurements were taken none of these transactions was blocked by the `speedValue` transition of the `CruiseControl` actor.

6.4.2 Discussion

As the cruise control example illustrates, once the priorities are assigned to transactions, it is relatively easy to apply ROOM canonical scheduling models to determine temporal behavior of any single- and multi-threaded ROOM executable. The biggest difficulty we

[3] The `CruiseReached` transaction is invoked as a consequence of the `ResumeCruise` transaction without introducing its own release timer.

encountered during the timing and scheduling analysis of the cruise control system, was to determine the time critical transactions that should be considered. Another challenge was to accurately compute the blocking and interference terms from other transactions such that transaction state dependent behavior would be taken into consideration. This process can get tedious in a complex system with large numbers of transactions. Also, we believe that it is virtually impossible to fully automate this process without an external feedback from a designer. One solution is to use a conservative approach where all higher priority transactions always preempt a lower priority transaction and where the run-to-completion blocking always occurs for a higher priority transaction, but this may yield very pessimistic results. We believe that a good tool support is needed to assign appropriate priorities to transactions and to accurately compute the blocking and interference terms for each transaction.

Chapter 7

Conclusions

In this thesis, we developed guidelines for single- and multi-threaded ROOM executables along with generic and canonical scheduling models for them, which were first validated and then used to perform timing and scheduling analyses of a real-time ROOM model of an automobile cruise control system characterized by both time-driven (periodic) and event-driven (reactive) behaviors.

7.1 Thesis Summary

In Chapter 1, we provided motivation for the development of the timing and scheduling analysis of real-time object oriented models. In particular, we identified problems associated with the development of real-time embedded control applications and presented a ROOM methodology which addresses these problems, so long as the appropriated schedulability techniques are available.

In Chapter 2, we provided an overview of the ROOM methodology and the ObjecTime toolset and listed their relative strengths for the development of real-time embedded control systems. The chapter also defines ROOM transactions.

Chapter 3 presented an overview of three timing and scheduling analysis methods, followed by a comparison of their time and space complexity results. We then selected a suitable scheduling analysis method for ROOM models.

In Chapter 4, we developed guidelines for applying the real-time scheduling theory presented in the previous chapter to ROOM models. We described single- and multi-threaded

implementations of ROOM models and we compared their relative strengths and weaknesses. For each implementation, a guideline was formulated which specified a suitable thread priority assignment. We also defined a guideline for desirable message priority assignment and two guidelines for bounding priority inversions of ROOM transactions.

In Chapter 5, we developed generic and canonical scheduling models for single- and multi-threaded ROOM executables that included the the overhead and blocking costs of `TargetRTS` and the underlying real-time operating system (RTOS) scheduler. Then we validated our canonical models through measurements.

Chapter 6 illustrated the concepts developed in earlier chapters using a ROOM model of an automobile cruise control system. The chapter provided an overview of the model, specified its time-critical transactions, and presented schedulability analysis results for these transactions.

7.2 Future Work

With the help of the design and implementation guidelines presented here, along with accurate ROOM canonical resource scheduling models, it is possible to easily determine temporal behavior of embedded control applications developed with the ROOM methodology and ObjecTime's target run-time system (`TargetRTS`). To extend the results presented in this thesis, several other areas deserve further investigation. They can be divided into theory issues and CASE tool extensions and are listed as follows:

1. Theory issues

   (a) Extending canonical scheduling models to handle distributed ROOM models, where the message passing mechanism is implemented using a real-time network and dedicated message-passing protocols.

   (b) Investigation of dynamic priority algorithms for ROOM transactions. In this thesis, we only examined fixed priority scheduling of transactions, thus further investigation of other priority algorithms, e.g. earliest deadline scheduling, would require a comparison between the fixed priority and dynamic priority algorithms, development of new guidelines and new canonical scheduling models for dynamic priority algorithms.

(c) The assignment of actors to threads in multi-threaded executables merits more careful investigation. Here, we specified just one possible actor-thread assignment for multi-threaded executables with easily distinguished control and data processing actors, where all the control actors are mapped to a single control thread and other data processing actors are mapped to separate threads. However, other multi-threaded ROOM models with different actor-thread configurations might offer better performance.

2. CASE tool extensions

   (a) Embedding support for the timing and scheduling analysis within the ObjecTime toolset.

   (b) Features to help the designer identify time-critical transactions to be considered in the response time analysis, assign appropriate priorities to these transactions, and accurately compute the blocking and interference terms for each transaction.

   (c) Changes to the `TargetRTS` system service to eliminate re-arming by properly implementing a periodic timer service.

Bibliography

[ABR+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[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.

[BAL97] H. Ben-Abdallah and S. Leue. Expressing and analyzing timing constraints in message sequence chart specifications. Technical report, University of Waterloo, Dept. of Electrical and Computer Engineering, April 1997.

[BCL91] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relation. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, August 1991. Winner of the Sidney Michaelson Best Paper Award.

[BCM90] J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. In *IEEE Symposium on Logic in Computer Science*, Philadelphia, June 1990.

[Boa93] M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, July 1993.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[BS89] T.P. Baker and A. Shaw. The cyclic executive model and ada. *Journal of Real-Time Systems*, 1(1):7–25, June 1989.

[BW96] A. Burns and A. Wellings. Advanced fixed priority scheduling. In *Real-time Systems: Specification, Verification and Analysis*, pages 32–65. Prentice Hall, 1996.

[CCM96] S. Campos, E.M. Clarke, and M. Minea. Analysis of real-time systems using symbolic techniques. In *Formal Methods for Real-Time Computing*. John Wiley & Sons, 1996.

[Fre98] P. Freedman. Investigating the suitability of ObjecTime for the software development of embedded control systems. In *ObjecTime Workshop on Research in Real-Time Object-Oriented Modeling*. ObjecTime Limited, January 1998.

[GF96] D. Gaudreau and P. Freedman. Temporal analysis and object-oriented real-time software development: a case study with ROOM/ObjecTime. In *IEEE Real-Time Technology and Applications Symposium*, Brookline, Massachusetts, June 1996.

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

[Har87] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8, August 1987.

[HG96] D. Harel and E. Gery. Executable object modeling with statecharts. In *ACM/IEEE 18th International Conference on Software Engineering*, 1996.

[Int94] International Telecommunication Union. *Message sequence charts standard (Z.120) reference*, 1994.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *Computer Journal (British Computer Society)*, 29(5):390–395, 1986.

[Kat94] D. Katcher. *Engineering and Analysis of Real-Time Operating Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 1994.

[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.

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

[Loc92] C. Douglas Locke. Software architectures for hard real-time applications: Cyclic executives vs. fixed priority executives. *Journal of Real-Time Systems*, 4(1):37–53, March 1992.

[LSD87] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, San Jose, CA, December 1987.

[McM92] K.L. McMillan. *The SMV system - DRAFT*. Carnegie-Mellon University, February 1992.

[Obj97] ObjecTime Limited, Kanata, Ontario. *MicroRTS Guide - Product Release 5.0*, March 1997.

[Sel95] B. Selic. Periodic tasks in ROOM. In *Workshop on Object-Oriented Real-Time Systems*, October 1995.

[Sel96] B. Selic. Tutorial: Real-time object-oriented modeling (ROOM). In *IEEE Real-Time Technology and Applications Symposium*, Brookline, Massachusetts, June 1996.

[SFR97] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for automated implementation of executable object oriented models for real-time embedded control systems. In *IEEE Real-Time Systems Symposium*, San Francisco, December 1997.

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

[SPFR98] M. Saksena, A. Ptak, P. Freedman, and P. Rodziewicz. Schedulability analysis for automated implementations of real-time object-oriented models. In *IEEE Real-Time Systems Symposium*, Madrid, December 1998.

[SR98] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Available from www.objectime.com/uml/index.html, March 1998.

[SRL90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[SW96] B. Selic and P. Ward. The challenges of real-time software design. *Embedded Systems Programming*, pages 66–82, October 1996.