

EXTENSIONS TO REAL-TIME
OBJECT-ORIENTED SOFTWARE
DESIGN METHODOLOGIES

TIMOTHY G. WOODCOCK

EXTENSIONS TO REAL-TIME OBJECT-ORIENTED SOFTWARE DESIGN METHODOLOGIES

by

Timothy G. Woodcock

A Dissertation Submitted to the Faculty of

The College of Engineering

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Computer Science

Florida Atlantic University

Boca Raton, FL

December 1996

© Copyright by Timothy G. Woodcock 1996

EXTENSIONS TO REAL-TIME OBJECT-ORIENTED SOFTWARE DESIGN METHODOLOGIES

by

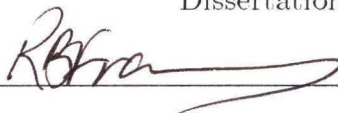
Timothy G. Woodcock

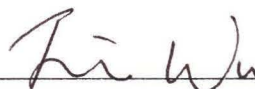
This dissertation was prepared under the direction of the candidate's dissertation advisor, Dr. Eduardo B. Fernandez, Department of Computer Science and Engineering, and has been approved by the members of his supervisory committee. It was submitted to the faculty of The College of Engineering and was accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

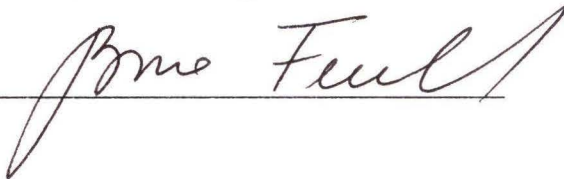
SUPERVISORY COMMITTEE:



Dissertation Advisor

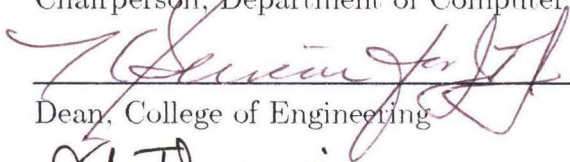




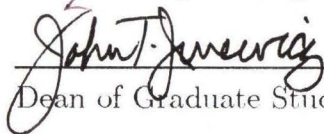




Chairperson, Department of Computer Science and Engineering



Dean, College of Engineering



Dean of Graduate Studies and Research

12/5/96

Date

ABSTRACT

Author:	Timothy G. Woodcock
Title:	Extensions to Real-Time Object-Oriented Software Design Methodologies
Institution:	Florida Atlantic University
Dissertation Advisor:	Dr. Eduardo B. Fernandez
Degree:	Doctor of Philosophy in Computer Science
Year:	1996

Real-time systems are systems where time is considered a system resource that needs to be managed. Time is usually represented in these systems as a deadline to complete a task. Unfortunately, by adding timing to even simple algorithms, it complicates them greatly. Real-time systems are by nature difficult and complex to understand.

Object-oriented methodologies have attributes that allow real-time systems to be designed and implemented with less error and some control over the resultant complexity. With object-oriented design, the system is modeled in the environment that it will be used in. Objects themselves, are partitions of the system, into logical, understandable units.

In this dissertation, we start by surveying the current real-time object-oriented design methodologies. By comparing these methodologies and developing a set of criteria for evaluating them, we discover that certain aspects of these methodologies

still need some work. The most important aspects of the methodologies are understanding the effects of deadlines on statechart behavioral models and understanding the effects of deadlines when object models are inherited or undergo aggregation.

The effects of deadlines on statecharts are then explored in detail. There are two basic ways that deadlines are added to statecharts. The first, and most popular, is adding timing as a condition on a state transition. The second is adding a count down timer to a state and forcing a transition if the timer reaches zero. We show that these are equivalent and can be used interchangeably to simplify designs.

Next, the effects of deadlines on behavior models when the corresponding object models undergo inheritance or aggregation are studied. We will first analyze the effects on the behavior model when object inheritance is encountered. We found eight ways that the behavior model can be modified and still maintain the properties of inheritance. Finally, deadlines are added and the analysis is repeated.

CONTENTS

ABSTRACT	iv
FIGURES	x
TABLES	xiii

Chapter

1 INTRODUCTION	1
1.1 Real-Time Systems	1
1.2 Object-Oriented Methodologies	3
2 REAL-TIME OBJECT-ORIENTED SOFTWARE DESIGN METHODOLOGIES	4
2.1 Introduction	4
2.2 Background	6
2.2.1 Real-Time Systems	6
2.2.2 Object-Oriented Programming	11
2.2.3 State Machines and Petri Nets	13
2.3 Classification Criteria	16
2.3.1 Support of Concurrent Processing	17
2.3.2 Controller Architecture	18
2.3.3 Deadline Management	18
2.3.4 First Design Decision	18
2.3.5 System Behavior	19
2.3.6 Use of Inheritance	19

2.3.7	Life Cycle	20
2.4	Design Methodologies	20
2.4.1	ARTS	21
2.4.2	COBRA	22
2.4.3	HOOD/PNO	24
2.4.4	HRT-HOOD	26
2.4.5	OCTOPUS	30
2.4.6	OMT	31
2.4.7	OPNets	32
2.4.8	ROOM	34
2.4.9	RTO	36
2.4.10	Transnet	38
2.5	Analysis	39
2.6	Chapter Summary	41
3	BEHAVIORAL MODELING WITH STATECHARTS	43
3.1	Introduction	43
3.2	About Time	43
3.3	Timed Automata	45
3.4	Statecharts	48
3.5	OMT Statecharts	52
3.6	Timed Statecharts	58
3.7	Evaluation and Analysis	61
3.8	Chapter Summary	67
4	TIMED STATE STATECHARTS	69
4.1	Introduction	69
4.2	Timed Transition Problems	69
4.3	Timed State Statecharts	71
4.4	Chapter Summary	79
5	OBJECT AND BEHAVIOR MODELS	81
5.1	Introduction	81
5.2	Object Models and Behavior Models	83

5.3	Object and Behavioral Inheritance	84
5.3.1	Inheritance Behavior	85
5.3.1.1	Addition of an Extra Transition	86
5.3.1.2	Retargeting and Splitting of a Transition	86
5.3.1.3	Weakening of a Precondition of a Transition	88
5.3.1.4	Strengthening of a Postcondition of a Transition	88
5.3.1.5	Strengthening of an Invariant Relationship	91
5.3.1.6	Refinement of a State into Two or More States	91
5.3.1.7	Addition of New Attributes	93
5.3.1.8	Modification of a State	94
5.3.2	A Student System Example	94
5.3.3	Program Specifications	99
5.3.4	Multiple Inheritance	103
5.4	Relationships And State Diagrams	104
5.5	Object and Behavioral Aggregation	106
5.6	Chapter Summary	113

6 THE EFFECT OF DEADLINES ON OBJECT AND BEHAVIOR MODELS 115

6.1	Introduction	115
6.2	Deadlines	117
6.3	Deadlines in Object Models and Behavior Models	118
6.4	Deadline Inheritance	119
6.4.1	Inheritance Behavior	121
6.4.1.1	Addition of an extra transition	121
6.4.1.2	Retargeting and splitting of a transition.	121
6.4.1.3	Weakening of a precondition of a transition	125
6.4.1.4	Strengthening of a postcondition of a transition	125
6.4.1.5	Strengthening of an invariant relationship	127
6.4.1.6	Addition of new attributes	127
6.4.1.7	Modification of a state	127
6.4.2	A Student System Example	129
6.4.3	Program Specifications	133

6.4.4 Multiple Inheritance 134

6.5 Relationships And State Diagrams 135

6.6 Object and Behavioral Aggregation 139

6.7 Chapter Summary 142

7 SCENARIOS AND EVENT TRACE DIAGRAMS 144

7.1 Introduction 144

7.2 Behavior Models with Deadlines 145

7.3 Background 147

7.4 Scenarios with Deadlines 149

7.5 Event Trace Diagrams with Deadlines 149

7.6 Chapter Summary 156

8 CONCLUSIONS 159

8.1 Contributions 164

8.2 Future Directions 164

REFERENCES 166

VITA 173

FIGURES

3.1	Example of a timed automata.	46
3.2	Statecharts Example	51
3.3	Example of Rumbaugh's OMT statecharts.	55
3.4	Example of Cook and Daniels Statecharts Formalism	57
3.5	Example of a timeout condition	59
3.6	Timed Statecharts+ example	61
3.7	Comparing OMT statecharts with regular statecharts	63
3.8	Example of broadcast communication paradox.	65
4.1	Gas Burner example, a) Statecharts+ and b) TSSC	75
4.2	Ambiguity resolution with timed state statecharts.	77
4.3	Example of timed and untimed transitions in TSSC	78
5.1	Object and Behavioral Models	84
5.2	Inheritance that adds a transition to the behavioral model. . . .	87
5.3	Inheritance where a transition is split and retargetted.	89
5.4	Inheritance weakening a precondition.	90
5.5	Inheritance where a state decomposes into two or more states. . .	92

5.6	Inheritance where an additional set of attributes is included in the subclass.	93
5.7	Object Model for a Student system.	95
5.8	Behavior model for the class student.	96
5.9	Behavior Model for class Foreign_Student.	97
5.10	Behavior Model for the class Undergrad.	98
5.11	Behavior Model for the state Grad_Student.	100
5.12	Behavior Model for class Thesis_Student.	101
5.13	Behavior Model for the class PhD_Student.	102
5.14	Example of Multiple Inheritance	105
5.15	Example of relationship in the object model and its temporal nature in the behavior model.	107
5.16	Object model for traffic light controller	109
5.17	Behavior model for traffic light controller	110
5.18	Coordinating Aggregation	112
6.1	Inheritance adding a transition to the behavior model.	122
6.2	Inheritance where a transition is split and retargeted.	124
6.3	Inheritance with weakening of a timed precondition.	126
6.4	Inheritance where an additional set of attributes is included. . . .	128
6.5	Object Model for the class Student.	130
6.6	Behavior model for the class student.	131
6.7	Behavior Model for class Foreign_Student.	132

6.8	Behavior Model for the class Grad_Student.	133
6.9	Example of Multiple Inheritance.	136
6.10	Example of relationship in the object model and relationship's temporal nature in the behavior model when deadlines are present.	138
6.11	Behavior model for traffic light controller.	141
7.1	Typical scanario for phone call.	150
7.2	Scenario with timing information.	150
7.3	Typical event trace diagram for phone call.	152
7.4	Event trace diagram with timing information.	153
7.5	Event trace with time interval requirement.	154
7.6	Event trace diagram with timing interval on event generation. . .	155
7.7	Statechart for typical telephone line.	157

TABLES

2.1 Summary of Methodologies 41

3.1 Comparison of Three Methodologies 66

Chapter 1

INTRODUCTION

1.1 Real-Time Systems

Real-Time software is increasingly important in today's world. As processors become less expensive, tiny embedded real-time systems are showing up in devices that would not have been imagined a few short years ago. Larger real-time systems, from military and space exploration, to complex medical lifesaving equipment, are becoming more common place. Often, real-time systems monitor life critical systems where failure means a loss of life. Many real-time systems have catastrophic effects if they fail to perform their functions correctly.

Real-time systems are by nature difficult to work with because there are many aspects of real-time software that make it unique and difficult. One of those aspects is that time is considered a critical system resource that must be managed. thus there is a requirement that some tasks complete in a specified amount of time. Not all tasks in a system will have deadlines, but the ones that do, often become critical to the correct performance of the system. Often, the system must know that

the resources a task needs to complete are available, before it will allow a task to be invoked on the system.

Real-time systems are also very periodic in nature, in that tasks are often need to be performed within a window of time, and may need to be performed again in some subsequent windows of time. The scheduling of these tasks is a complex, often NP hard, problem.

Many real-time systems also execute in distributed processors and thus have concurrent processing as another aspect of their systems. Distributed systems that are made up of several connected communicating processors must also take communication time into their deadline management calculations. Concurrency, even on a single processor, has implications for completing tasks by the deadlines.

Real-time software is also characterized by its nonportability. Software written for a specific application often can only work on that application. Changes to the environment, no matter how small, could render that whole system inoperative. This relationship to the environment becomes part of the software design.

These aspects of real-time software make it difficult to design and implement. Often the code has very long implementation and test cycles, and once finished is very difficult and expensive to change or correct. In the past, the software in these systems was composed of hand-tuned, assembly language functions that were very difficult to create, maintain, or improve.

1.2 Object-Oriented Methodologies

Object-oriented software methodologies have been shown to have advantages over classical software design techniques, such as structured design. Object-oriented designs are more extensible, and have better reuse than classical designs. Object-oriented methods may also have an advantage in productivity. By encapsulating the software and designing around objects that take the environment into account, object methods may have a distinct advantage when used to create real-time software.

Object-oriented methodologies, however, still need to be developed further before becoming the methodology of choice for real-time system environments. The goal of this dissertation is to explore some of the areas that need work and to identify other areas for future study.

First, we start by looking at several existing object-oriented real-time design methodologies and from this we compile a list of areas where further work needs to be done. Next, we examine behavior models and study how real-time deadlines affect them. Then, we explore the different ways that deadlines can be added to behavior models. A study of the relationship between object and behavior models when object models undergo the techniques of generalization, aggregation, and association is given next. Finally, we look at how the addition of deadlines affects the object and behavior model relationships.

Chapter 2

REAL-TIME OBJECT-ORIENTED SOFTWARE DESIGN METHODOLOGIES

2.1 Introduction

In real time systems, time is considered a limited resource that must be managed. Creating software where tasks and messages may have deadlines is difficult. Object-oriented methodologies have been shown to increase programmer productivity, software reuse, and software maintainability. It is of interest therefore to see if object-oriented techniques provide benefits for real time environments.

There are several objectives for this chapter. The first is to understand the issues that are unique to object-oriented real-time software development. Then we analyze how existing methods approach these issues. We also want to identify the deficiencies of the existing methodologies. From there, we want to develop a set of objectives for a more comprehensive methodology. Finally we want to develop a set of criteria for selection of a methodology for a specific type of application. This study could be of value because it allows the developer to look at the problem he is

trying to solve and then understand what methodology features would enhance the solution to the problem.

There are few studies have been published looking at both real time and object oriented programming. Kelly and Sherif did a similar analysis on real time software development methodologies [35]. Their analysis included only one object-oriented methodology and three other software development methodologies.

In this chapter, we will look at ten object oriented methodologies for developing real-time systems. These methodologies were chosen only to represent some of the available real-time object-oriented methodologies, and not as an all inclusive list of all such methodologies. The following methodologies will be examined:

- ARTS - Real-Time Object Model
- COBRA - Concurrent Object Based Real-Time Analysis
- HOOD/PNO - Hierarchical Object-Oriented Design by means of Petri-Net Objects
- HRT-HOOD - Hard Real Time Hierarchical Object-Oriented Design
- OCTOPUS - Object-Oriented design method for embedded real-time systems
- OMT - Object-Oriented design and analysis methodology
- OPNets - Object-Oriented high-level Petri Net model

- ROOM - Real-Time Object-Oriented Modeling
- RTO - Real Time Objects
- Transnet - Object-oriented technique using Petri nets

First through, some background on object-oriented methodologies, real-time systems, state machines and Petri nets is discussed in section 2.2. In section 2.3, we describe the criteria used to compare the different methodologies. In section 2.4, we examine each of these methodologies briefly. In section 2.5, each methodology is contrasted to the others in terms of the comparison criteria. Finally, section 2.6 states some conclusions and indicates possible future directions.

2.2 Background

2.2.1 Real-Time Systems

Real-time software systems manage time by including deadlines for tasks and messages. *All real-time systems must be considered in the context of their environment.* One way to interpret the deadlines of real-time software is to consider all the data as perishable. If the data is not used before the time expires, it becomes old and can not be used, thus, perishable.

Periodicity is another aspect of real-time systems, in that, some tasks must be performed at periodic intervals, called frames. Other tasks are evoked only when a certain event occurs. Some tasks need to wait for other tasks to complete before

they can start. Often, the tasks that need to be run in a time frame can be performed in a number of sequences.

Task scheduling is a problem in real time systems that continues to have a great deal of research devoted to it [6], [7], [8], [9], [21], and [64]. The scheduling algorithms are affected by the criticality of the task, task precedences, available resources, and task deadlines [60]. Also, tasks have to communicate and synchronize with other tasks. Scheduling algorithms in real time systems must ensure that task deadlines are met. This is different from nonreal time systems where the goal is faster response time. Stankovic points out that the dynamic environment often requires adaptive scheduling algorithms. Scheduling in this environment is an NP-Hard problem [62].

Shin and Ramanathan showed that all real time tasks, both periodic and aperiodic will have one of three types of deadlines [60].

- *Hard* - There are catastrophic consequences for missing a hard deadline.
- *Firm* - The consequences for missing a firm deadline are not severe, but the results of any task with a firm deadline are perishable; that is, they will cease to be useful when the deadline expires.
- *Soft* - All other deadlines are soft. The results of a soft deadline task will also decrease in usefulness over time after the deadline expires, but at a much slower rate than a firm deadline.

Periodic tasks often have hard deadlines and are characterized as time critical, where the system will fail (often catastrophically) if the deadline is not met.

Predictability is another important concept in real-time systems. Predictability means that at design time it can be shown that all the constraints of all the tasks can be met with 100% certainty. This requires prior knowledge of the exact characteristics and run time resource requirements for all the tasks in the system. Usually this can be done only in very small systems. In large systems the definition is relaxed for noncritical tasks.

For noncritical tasks, predictability is shown either probabilistically or run-time deterministically. For a probabilistic guarantee, a task can be shown to meet its constraints with a certain probability. In run-time determinism, the system looks at a task before accepting it and determines if it can meet the task's constraints without endangering any other running tasks constraints. If it can, the task is accepted; otherwise, the task is rejected.

Another concept that occurs often in real-time environments is that of system concurrency. Real-time software can have many tasks active at the same time. An example would be several active threads or processes in an operating system. Another example would be the active phone calls in a telephone switch. But, concurrency alone is not enough to make real-time software systems difficult. It is the asynchronous nature of the system that makes the concurrency more difficult to handle. For example, the activity of a robot cell in a distributed manufacturing

system, even if there were 100 robots, would not be so difficult if they all operated in lock step. It is their asynchronous activity that makes them difficult to control. Concurrent systems often must be proven to be deadlock free.

The dynamic nature of real-time systems can also contribute to programming difficulty. Some concurrent processes can be created and become active while others are becoming inactive and being destroyed. An example would be telephone calls in a telephone switch. At any instance in time, any number of calls could be in any number of states including dialing, connecting, talking, and disconnecting.

Another important issue in the design of real-time systems is at what level of abstraction one should introduce the concept of time and how to map time constraints defined at some level to the lower levels.

There is a tacit assumption that in a true real-time system, the requirement to interface with low level hardware while meeting stringent hard deadlines, prohibits the use of high level languages in the software. In other words, assembly language must be used. Stankovic [62] points out that this is a common misconception about real time software. He also points out that clever hand-coded optimized machine language software is labor intensive. Also, this code often contains timing bugs that are difficult to trace, debug, or modify. Daponte et al. [19] show that the real concern should be whether the target language allows us access to the low level hardware interfaces without adding a run-time support penalty that is unacceptably high.

The issue of reliability is also important in real-time systems. Software reliability criteria can be met through the use of n-version programming. Reliability can also be specified and quantified through the use of formal methods. However, both n-version programs and formal methods are difficult to use on large complex systems.

Real-time systems are very dependent on the environment of the system. The software for a jet airplane would not function outside of the airplane. Likewise, the software for an automobile engine computer would not be able to function without the engine sensors. The importance of the environment in real-time systems indicates that any methodology used must be able to represent the physical environment also; in this sense the object-oriented approach provides clear advantages.

Embedded systems are usually real-time systems. One of their main characteristics is that they require flexibility and extensibility (different environments and different applications). The object-oriented approach appears very promising to satisfy this requirement.

Real-time systems are usually multiprocessors. Obviously, satisfying deadlines will depend on the specific multiprocessor configuration at hand. A design methodology must consider this effect. There are some real-time operating systems that help accomplish this task, for example CHAOS [26], [56], and Clouds [14], [20], [50].

2.2.2 Object-Oriented Programming

Object-oriented programming techniques have been shown to have advantages over classical design techniques such as structured design. One advantage is their enhanced ability to provide extensible designs. Another would be the increased amount of reuse of design artifacts and code. Overall, object-oriented methods do appear to improve productivity [2]. Object-oriented techniques also have several advantages in a real-time environment. Some researchers believe that current (nonobject-oriented) real time software development techniques will not be adequate for meeting the challenges of the future generations of complex real time systems [59]. The object oriented paradigm offers a better way of creating and controlling the development of complex real time systems; for example, they might make it easier to prove predictability.

An important advantage of object-oriented methods is their modeling power, it is possible to build a model for some systems that reflects its semantics much more closely than with other methods. This is an important advantage for the design of real-time systems that have complex relationships between components and with the environment.

A significant aspect of real-time systems is the fact that they are composed of physical units and they can be modeled using a hierarchic approach [24]. The environment of the real-time systems are also important. Object-oriented methodologies support these views of the software better than other methodologies.

In the object-oriented paradigm, information hiding allows a designer to have a variety of implementations for each method in an object, each satisfying a different type of constraints. For example, an object might have three different methods for performing a task, each of which was guaranteed to complete in a set amount of time (slow, medium, fast, or 5 ms, 2 ms, 1 ms, etc.).

Object-oriented paradigms may also have some advantages in applying strategies such as n-version fault tolerance. The object view of the physical units of the system may allow an easier way of creating and using n-version software. Formal methods appear also to enhance object-oriented designs. This can result in an increased ability to prove system reliability, safety, and other properties. Currently, there are two main approaches to including formal methods in the OOA. The first is to place the axioms in the objects [18], [57], and the second method is to place the axioms in the state transitions [22], [25], [31], [41], [65], and [66].

On the other hand, there are some disadvantages to using an object-oriented methodology. There is a dichotomy between the underlying principles of object-oriented methodologies and software performance, which could affect its ability to satisfy deadlines or its predictability. For example, it is not clear what is the effect of inheritance of deadlines. Most general purpose object oriented methodologies have modeling support that is inadequate for real time software. It is not clear in most methodologies how to expand states to provide for details at lower levels of abstraction. The problem is exasperated by message deadlines and concurrency.

Another possible disadvantage of object-oriented systems for embedded applications is their requirement of efficient use of memory. In object-oriented implementations, a class may contain many operations that will not be used in a given application but will take memory space. It is necessary to consider ways for selecting only the needed operations in a given application.

The implementation language will also need to be considered. No matter what object-oriented language is used, C++, Ada, Eiffel, Smalltalk, or Java, extensions will need to be made to support the real-time environment. An extended language like RTC++ or Ada 9X may work, but further extensions are still likely.

2.2.3 State Machines and Petri Nets

The modeling of the behavior of objects in the object oriented paradigm, is most often done with either state machines or Petri nets. While both modeling techniques can describe the behavior of the objects, they also have limitations to what can be described with the techniques. A brief description of state machines and Petri nets follows.

State machines, or deterministic finite state automata, can be used to model the behavior of many kinds of systems [42]. State machines consist of a finite number of states, a set of transitions between states, and a list of the events or inputs that trigger the transition from a state. This is usually expressed as a 5-tuple $M = \{Q, \Sigma, \delta, q_0, F\}$ where:

Q is a finite set of internal states

Σ is the input alphabet

$\delta : Q \times \Sigma \rightarrow Q$ is the transition function

$q_0 \in Q$ is the initial state

$F \subseteq Q$ is a set of final states.

A state machine starts off in the initial state, q_0 . When an input (from Σ) arrives it causes one of the transitions from δ to be used to move to a new state (or sometimes stay in the same state) of Q . If the machine is in one of the final states, F , then the state machine can terminate.

To use state machines to analyze object-oriented software, one only needs to map the object-oriented concepts onto the state machine. For example, Σ , are the set of messages that an object could receive. The transition function, δ , and the state of the object determine what happens when a message is received. The object could accept the message and transition to a new state, ignore the message and stay in the same state, etc. The states of the objects determine what methods need to be invoked by the different objects.

One of the problems with state machines is that of state explosions. If the number of states involved are small, then the problem can be dealt with. Most problems of even low complexity have a large number of states. One way to deal with this problem is through the use of statecharts, which are a modification of state machines to allow the nesting of states [54]. Statecharts not only have a more concise way of representing complex states, but also allow substates to inherit and

evoke state transitions of the parent states. The states in a statechart are described in a hierarchical structure that allows a compact, yet precise representation of the dynamic behavior of the objects. Statecharts are usually more convenient than state machines.

Petri nets are similar in some ways to state machines [1]. They can be viewed as another type of automaton, or as a way of representing different kinds of systems. There are several kinds of Petri nets, but for this paper, we limit ourselves to *Marked Petri nets* only. A Marked Petri net is a 4-tuple $C = (T, P, A, M)$, where,

$T = \{t_1, t_2, \dots, t_n\}$ is a set of transitions.

$P = \{p_1, p_2, \dots, p_m\}$ is a set of places.

$A \subseteq \{T \times P\} \cup \{P \times T\}$ is a set of directed arcs.

$M = \{x_1, x_2, \dots, x_m\}$ where $x_i \geq 0$, assigning a number of tokens to each place in the net. This is called the marking of the Petri net.

In a Petri net, a transition can fire only if there is a token waiting in each place attached to it. The act of firing causes the tokens to be removed from the inputs places to the transition and placed in the output places. When two transitions are enabled and do not share an input place, they can fire concurrently. When two enabled transitions do share an input place, firing either would remove the token from the shared place, disabling the other transition. This is known as *conflict*. In a conflict, the choice of which transition will fire is arbitrary.

The concepts of reachability and p-invariants are also used with Petri nets.

A marking M is said to be reachable from M' if starting with M' , there exists some series of transitions that can be fired that results in M . A p-invariant is a set of places, I , that have the property that the sum of all the tokens of all the places in the set, for any marking of the set is a constant. That is

$$C = \sum_{p \in I} M(p)$$

Where M is a reachable marking and I does not have any proper subsets that are p-invariants. The reachability chart and the p-invariants of the network can be used to verify the behavior of the Petri net.

Petri nets also suffer from state explosion. Similarly to statecharts, Petri nets can be arranged in a hierarchical system that controls the state explosion. Petri nets can model very well aspects such as parallelism and nondeterminism. Petri nets can model concurrency, by explicitly showing the parallelism, and be used to prove that a system is deadlock free. Petri nets can also be modified into colored Petri nets or into timed Petri nets in order to increase their ability to model certain applications. For example, a timed Petri net can be introduced so that the deadlines of the real time software can be modeled. However, adding colors or timing detracts from the Petri nets ability to detect deadlocks or other conditions.

2.3 Classification Criteria

Based on the discussion in Sections 2.2.1 and 2.2.2, the following criteria will be used as guidelines for comparison of the design methodologies.

- Support of concurrent processing.
- Controller architecture.
- Deadline management.
- First design decision.
- Modeling of system behavior.
- Use of inheritance.
- Life Cycle.

A discussion of each criterion follows.

2.3.1 Support of Concurrent Processing

Real-time software is often characterized by a large amount of system concurrency. If, in general, tasks communicate, synchronize, or interact with a number of other independent asynchronous tasks and external events, then the system has a large amount of system concurrency. The concurrency can range from separate processors connected via a communications facility to asynchronous events in one process. Because the expression of concurrency is an important issue in real-time software, the design methodology should reflect it.

2.3.2 Controller Architecture

The scheduling algorithms for real time software are implemented by a controller. Thus the controller should be given consideration as part of the overall system. Each design methodology supports either a distributed controller mechanism or a single object controller mechanism. In a distributed controller, each object will contain the code necessary for synchronization and message passing and message handling. In a single object control design, one object is specifically designed for handling message traffic, synchronization, and the states of the other objects.

2.3.3 Deadline Management

One of the more important characteristics of real-time software is that every task and every message may have a deadline that must be considered. In some of the design methodologies the deadlines are considered in the design phase. In other systems, the deadlines are handled by the target programming language in the implementation phase. In any real time system with hard real time deadlines, this is a key issue to be decided. Furthermore, any system that does not have deadlines, or whose deadlines are never considered by the methodology are not truly real time systems. They may be relative time systems, but they are not real time systems.

2.3.4 First Design Decision

Kelly and Sherif [35] pointed out that the first decisions that are made in a design are often the hardest to change later in the software life cycle. Thus, these first

design decisions are problematic in that they are the most persistent decisions made, yet they are made at a time when little is known about the resultant system. For example, partitioning a large system into sections early on in a project could result in a poor design if later in the design it is discovered that message traffic between parts of the system will use a serious amount of system resources. Considering the first design decision as a part of the methodology allows the designer to make a better choice in the design of the system. This criterion is of the lesser importance when only object-oriented systems are being considered.

2.3.5 System Behavior

During the design phase of the life cycle, the behavior of the design artifacts must be analyzed. In the methodologies presented in this paper, only state machines and Petri nets are used to model the system behavior. Each technique has some advantages. State machines are easy to use and can be expanded in the more powerful statecharts when necessary. Petri nets are more powerful and have the ability to model the concept of concurrency and show that the system is deadlock free. Petri nets can be expanded into timed or colored Petri nets if desired.

2.3.6 Use of Inheritance

In object-oriented methodologies, inheritance is a valuable mechanism to reuse existing classes of objects. There are several issues with inheritance in real-time software beyond the general object-oriented inheritance issues. Specifically,

the issues of the inheritance of deadlines and behaviors are important in real-time software. How are class deadlines inherited? If a subclass overrides a soft deadline with a hard deadline, can the methodology ensure the predictability of the software? What is the relationship between deadlines at different levels of design abstraction, when more detailed state transitions are involved?

2.3.7 Life Cycle

Every methodology supports some phase of the software life cycle. Some only support the analysis or design phases by offering tools and techniques that only help in these phases. Some support a full life cycle by offering tools and techniques that start with the specifications and enforce completeness and consistency all the way to the code release. It could be desirable to even support the life cycle beyond code release and into code maintenance. In general, full life cycle methodologies are preferable to partial life cycle methodologies because the tools of each phase are integrated together. A bug fix in the code that results in a change to the design, should start in the design tool.

2.4 Design Methodologies

The ten methodologies that are examined here are all from recent publications and have been used on real projects. In this section, each methodology will be briefly discussed. The design steps, major characteristics, and strengths and weaknesses will be outlined.

2.4.1 ARTS

ARTS is an object-oriented methodology for designing real-time systems [45]. It is implemented in RTC++ which is an extension of C++ intended to support this methodology [34]. It provides not only data encapsulation but also timing encapsulation. This methodology is more concerned with the low level design and implementation issues (such as issues with RTC++) than with high level design issues (like behavior modeling). RTC++ is a language that could be used to implement any of the methodologies discussed in this paper.

Showing that a system is predictable is one of the key aspects of the ARTS methodology, for which it uses rate monotonic scheduling.

In ARTS the objects can be single threaded or multithreaded, but the multithreaded objects have better predictability. Both types of objects can have the problem of priority inversion, which happens when a task of high priority is blocked by a task of low priority. By using a property called priority inheritance, which is not the same as object-oriented inheritance, priority inversion can be overcome. In most cases, priority inheritance is used to change the priority of a low priority task that is blocking a higher priority task to the priority of the higher priority task. In multithreaded objects, a free thread can be used to run the higher priority task.

Their use of the term ‘inheritance’ in priority inheritance is confusing. It is also unclear in ARTS how the behavior of the objects is modeled. It is clear that ARTS supports predictability and provides a solution to priority inversion, but this

does not appear tied to object behavior.

2.4.2 COBRA

The Concurrent Object-Based Real-Time Analysis (COBRA) methodology was developed by Gomaa [27]. It is a blend of concepts from Real-Time Structured Analysis (RTSA), Object-Oriented Analysis (OOA), and Jackson System Development (JSD). COBRA uses the RTSA notation and state diagrams. It is similar to JSD in that its model uses concurrent processes for objects and functions. Like OOA, it uses object structuring criteria.

The main steps of the COBRA methodology are the following:

1. Decompose the system into independent distributed subsystems. Here distributed implies more than just concurrent, but processes that can actually reside on separate processors.
2. Identify the objects for each subsystem.
3. Identify the operations for each object.
4. Create a statechart model from the objects treating each as a concurrent task.
5. Analyze the behavior of each object with event sequencing scenarios.

COBRA's decomposition of each problem into subsystems with an emphasis on a distributed environment, is supported by a structuring criteria and a mapping

to the distributed nodes. COBRA's support for this decomposition approach gives this methodology an advantage in distributed environments such as a manufacturing cell with independent robots.

The criteria supported by COBRA considers the following five types of objects as the most relevant.

1. *External Device I/O* objects, which map every physical entity in the real world to a software object that models the device.
2. *Control* objects, which control all the other objects in the system.
3. *Data abstraction* objects, which encapsulate data that the system needs to remember.
4. *Algorithm* objects, which encapsulate algorithms used in the problem domain.
5. *User* objects, which are needed to model the role of users in the model. The user objects are different from the external device objects, but the difference is not clear in Gomaa's paper. Apparently, they are just user interfaces.

The operations for each object are characterized by their period. There are two types of operations: asynchronous and periodic. Asynchronous operations are activated by an object or event to perform an action. Periodic operations activate themselves at regular intervals. Both kinds of operations can be unregulated or

dependent upon the state of the object. Operations that are dependent upon the state of the object may perform different tasks depending upon the object state.

The system is then modeled using the objects and operations. Each object is treated as a concurrent task, so that the system model supports a great deal of external concurrency. The system behavior is modeled using event sequencing scenarios. Event sequencing scenarios use control objects that respond to incoming events from the external environment and control the system state transitions. This is the same as in Rumbaugh [54].

One disadvantage of COBRA is that deadlines are not considered. Another disadvantage is that the event sequencing scenarios are not able to prove the system is deadlock free, because they are just specific scenarios, not a complete representation such as statecharts or Petri nets.

2.4.3 HOOD/PNO

Hierarchical Object-Oriented Design (HOOD) is a design methodology for real time software defined by the European Space Agency. It has been extended with Petri net objects (PNO) to model the system behavior (HOOD/PNO) [49]. PNO is a method of describing the control structure and behavior of each object using Petri nets. The HOOD/PNO methodology covers the entire software life cycle including analysis, design, and implementation

HOOD/PNO uses a parallel recursive life-cycle process that takes a level of

abstraction, defines the behavior at that level, and decomposes it into the next lower level. The steps of this methodology are as follows:

1. Determine the relevant objects of the system from the physical system requirements.
2. For the current level of abstraction, collect the objects into object classes.
3. Describe the physical objects in terms of their external behavior, their internal structure and their relation to other objects.
4. Redefine physical objects and classes into software objects and classes. (The authors claim that generally the physical description does not take into account all the responsibilities required by the specifications.)
5. For a given level of abstraction, define the operations for each object.
6. For a given level of abstraction, define the operations of the object.
7. Describe the objects' behavior by Petri nets and verify the properties of boundedness, liveness, and safeness in the design. Then computing the p-invariants of the Petri net model, decompose the objects into next lower level of abstraction.

HOOD/PNO is an object oriented design methodology that includes object-oriented design analysis (OODA), HOOD, PNO, and implementation rules for translating detailed designs into specific target language code. The strength of this

methodology is that it covers the entire life cycle from requirements to code. This methodology can be applied with a top down approach or a bottom up approach. The steps above show the top down approach.

One disadvantage of HOOD/PNO is that it does not directly deal with the problems of concurrency. While concurrency can be modeled using Petri nets, it is not specifically designed into the software. Also, the issues of object deadlines are left to the implementation language and not dealt with as design issues.

2.4.4 HRT-HOOD

The Hard Real Time Hierarchical Object-Oriented Design (HRT-HOOD) is another adaptation of HOOD for real time environments [10]. In this case the emphasis is on supporting the abstractions that are typically needed by hard real-time system designers. This allows the designer better conceptual tools for specifying and analyzing the deadline requirements of the software.

HRT-HOOD was developed based on the belief that the design methodology must provide the following support:

- objects that recognize the kinds of activities and artifacts found in real-time systems.
- the appropriate scheduling paradigms.
- explicit definition of the timing requirements for each object.

- definition of the relative importance of each object to the overall successful functioning of the system.
- support for different modes of operation. (i.e. An airplane will have different modes of operation such as on ground and in flight. It is reasonable to expect the software to behave differently in these different modes of operation.)
- explicit definition and use of resource control objects, which are objects that interface to system resources (i.e. sensors, memory, etc.)..
- decomposition into a software architecture that facilitates processor allocation, scheduling paradigm analysis, and timing analysis.
- tools to perform worst case execution time and schedulability analysis.

HRT-HOOD separates the high level design activity into two parts: the logical design and the physical design. The logical design is concerned with satisfying the functional requirements that can be made independently of the constraints imposed by the execution environment. The physical design addresses the timing *and schedulability from the functional requirements and the other constraints. The physical design can be viewed as a refinement of the logical design, they are both iterative and concurrent processes.*

The result of the logical design is a set of objects that can not be further decomposed (terminal objects). HRT-HOOD supports five kinds of objects:

- **PASSIVE** - Objects that are invoked by other objects. They have no spontaneous control over their own or other object's operations.
- **ACTIVE** - The most general class of objects with the least restrictions placed on them. These objects can control when their own operations are executed and can call operations in other objects. Since the effect of these objects can not be analyzed, they are allowed for background activities only.
- **PROTECTED** - This is an extension of the basic HOOD object types. These objects can control when their operations are executed but can not call operations in other objects. These objects must be analyzed for the blocking times they impose on the objects that call them.
- **CYCLIC** - This is another extension of the basic HOOD object types. These are the periodic activities of the system. Their operations are demands for immediate attention. They can also spontaneously invoke operations in other objects.
- **SPORADIC** - This is another extension of the basic HOOD object types. These objects represent the sporadic activities of the system.

Every object has code to control its behavior and synchronization which is called the object control structure (OBCS). The concurrent activities inside the objects are called **THREADS**. An object can have one or more **THREADS** that operate

independently from the operations of the object and whose order of execution is controlled by the OBCS. At the highest level of design each system is represented by a single CYCLIC or SPORADIC object. These objects are decomposed into lower level objects at each iteration of the design cycle.

The physical design maps the logical design onto the physical resources of the system. The physical design does the following:

- allocate the objects in the logical design to the physical processors.
- schedule the communications network such that message delays are bounded.
- *schedule the processors so that all objects on all processors meet their deadlines.*

During the physical design, objects are assigned their timing attributes. Also the abstractions for handling timing errors are created. These can include stopping an object that uses more compute time than was requested, and stopping an object that executes past its deadline.

It is clear that HRT-HOOD has very strong deadline management and is a true real-time design methodology. It is not clear how well this methodology supports concurrency beyond assigning objects to physical processors and threads inside an object.

2.4.5 OCTOPUS

OCTOPUS is a methodology for applying object-oriented techniques to embedded real-time systems [69]. OCTOPUS contains extensions to OMT to handle specific real-time embedded system problems such as concurrency, synchronization, communication, handling of interrupts, hardware interfaces and end-to-end response time.

The steps of this methodology are as follows:

1. Create the system requirements specification from case scenarios.
2. Create a system architecture to partition the system into independent subsystems and specifying the subsystem interfaces.
3. Analyze the subsystem and create the OMT object and dynamic models necessary for the subsystem.
4. There are two required subsystems: a hardware wrapper and at least one other subsystem. The hardware wrapper isolates the software from the hardware. The wrapper translates any external stimuli (ie hardware inputs, buttons, etc.) to logical stimuli (events) for the software.
5. Analysis is first done in *implicit concurrency* mode, where each subsystem is designed and analyzed as if it had its own fast processor. Processing is considered to occur in zero time.

6. Each event in the object and dynamic models are assigned a significance value $(c, e, 0, 1)$. Here c represents a hard deadline, the e represents a soft deadline, the 0 represents no deadline, and the 1 represents a deadline determined by some other state.
7. The *explicit concurrence* mode is created by mapping the object model into event threads.

OCTOPUS has strong design and analysis support for concurrency. The controller design is implicit with an external hardware wrapper. Deadline management support is included. The first design decision is that of subsystems. The methodology uses statecharts for behavior modeling. Inheritance is supported as in OMT. The life cycle covers the design and implementation phases. OCTOPUS is well suited to embedded real-time development. Also, once the architecture decisions have been made, it would be difficult to change them, unlike a regular object-oriented system, where changes are expected and isolated from the system.

2.4.6 OMT

OMT is an object-oriented methodology that enjoys great popularity [54]. Unlike the other methods discussed, this is not a real-time methodology, but a general methodology. Several extensions and proposals exist to add real-time features to OMT such as OCTOPUS (above) and [15].

Originally OMT consisted of three complementary models: the object model, the dynamic model, and the functional model. The functional model has been eliminated in the recent OMT - Brooch unified model (UML). The object model describes the static relationship of the objects in the system. The dynamic model describes the behavior of the individual objects.

Real-time extensions such as that of Chonoloos [15] capture the timing information in the event trace diagrams, scenarios, and statecharts. Rumbaugh's recent additions to OMT [53] also support real-time software with deadlines in the event trace diagrams and statecharts. Concurrency is not supported in OMT.

2.4.7 OPNets

OPNets is an object oriented methodology that models the behavior of the objects as Petri nets [40]. One of the authors' motivations for developing OPNets was to correct a problem they saw in PNO. In PNO, an object's control structures and communications are not separated. Furthermore they saw the behavior of the object in the control structure of the whole system as only being implicitly defined. As a result, a modification of an objects' inner control structure could result in a change to the control structure for the whole system. To overcome this problem they proposed the OPNets methodology. OPNets identifies objects based on their concurrency relationships. The object's internal control structure and external structure are clearly separated. High level Petri nets are used to model the behavior of the

objects and the relationships between objects. This external structure then defines the message passing between objects. At the next lower hierarchical level the Petri net nodes are expanded to represent the internal control structure of the object. The internal structure is not visible to other objects in the model.

There are two types of objects defined in OPNets. The first are primitive objects which are the basic unit of behavior representation. Primitive objects define sequential behaviors and static properties. These objects can not have concurrency in them. The second type of object are the composite objects. Composite objects are made up of primitive objects and other composite objects. Composite objects have concurrency, and synchronize the sequential behaviors of primitive objects.

The steps of this methodology are as follows:

1. Define the system in terms of mutually communicating aggregate objects and their interconnection relations.
2. Define external message passing structure and internal control structure of each object.
3. Define static properties and behaviors for each primitive object.
4. Model the behavior of each object with a Petri net.
5. For each primitive object, analyze the local behavior, reachability, and firing sequences.

6. For each composite object, analyze its behavior in terms of its internal objects.

OPNets uses a hierarchical Petri net approach to model very complex real time systems. Like Transnet the methodology supports only part of the life cycle, the analysis and design phase. OPNets can model concurrent actions, both inside and outside of objects. OPNets do not include deadline management in the design cycle.

2.4.8 ROOM

The Real-time Object-Oriented Modeling (ROOM) methodology was created to go beyond creating and verifying a design, into automatically producing implementations of the design [58]. ROOM is based on several principles of how a design methodology should work. The key modeling concepts must be intuitive and domain specific. Each software domain has its own concepts that are well understood by the developers. The development process should not allow discontinuity. The authors describe this as a seamless formal relationship between the artifacts and activities of the analysis, design, implementation, testing, and documentation. Lastly, the methodology should support an iterative design process.

In ROOM systems are modeled using two paradigms, dimension and abstraction level. The dimension model partitions the system based on the problem's nature. The abstraction level partitions the system into three levels: the *system* level for modeling concepts at the highest level, the *concurrency* level focusing on

issues of parallelism, and the *detail* level which focuses on the implementation. The model technique is iterative, building a model at a level of abstraction and analyzing it, then refining the model at the next level of abstraction. The steps in the ROOM methodology are as follows:

1. Analyze the current level of refinement using ROOM modeling concepts and paradigms.
2. Design and implement current abstraction in a ROOM model.
3. Verify that the model meets requirements.
4. Move to the next level of abstraction and repeat.

The ROOM model uses active objects called actors. The actors have ports that accept messages, where messages are units of information that flow between actors. Actors can be decomposed into groups of actors and their messages, which are hidden from the higher levels of abstraction. The behavior of the actors is formalized using finite state machines and statecharts.

The advantages of ROOM are the iterative process and the abstraction levels. Another advantage is the formal support of the methodology for the entire life cycle from requirements to implementation and to verification.

2.4.9 RTO

Real Time Objects (RTO) are a methodology that has as its major goal the explicit programming of the real time scheduling [47], [48], [19]. The authors claim that this methodology is well suited for hard real time programming. RTO defines its objects such that internal concurrency is not allowed.

The following are the RTO mechanisms that are supported by the methodology:

- *Objects* - RTO objects are single threaded (atomic) objects. The object behavior is modeled by a state machine. RTO objects are reactive in that they are at rest until a message arrives, then depending on their state, they perform an action. Execution of an action can not be preempted.
- *Classes* - RTO uses a decentralized synchronization control; therefore, each object has code to synchronize concurrency.
- *Message Passing* - Objects communicate through asynchronous message passing.
- *Unexpected Messages* - When an object receives a message that it can not handle, it can decide to do one of the following:
 - discard the unexpected message.

- *defer the message until later, assuming that when the object goes into another state it will be able to deal with the message.*
 - pass the message to another object. This is different from the concept of inheritance in that the object can send the message to any other object not just its parent object.
- *Components and Controllers* - To handle scheduling the idea of a component is introduced. A component is a collection of objects with similar time requirements operating on a (physical or virtual) processor. Each component has a special object called a *controller*. The controller collects all the message traffic, reorders the messages, and dispatches messages. With this implementation any user programmable control strategy can be used, or a standard controller can be imported from a library.
 - *Standard Controllers* - While the controller can be programmed by the user to implement any control strategy, RTO also has a default standard controller that operates concurrently with the other objects inside its component and dispatches messages in FIFO order without concern for time (soft deadline).
 - *Driver Objects* - Driver objects encapsulate the physical system and external events into internal messages and vice versa.

The advantages of RTO are that it supports concurrency, it considers the object deadlines in the high level design, and that any scheduling algorithm can be

programmed into the controllers. RTO has a decentralized control strategy because there are multiple components (each containing objects with a similar deadlines), *and each component has its own controller, so the control code for synchronization* and communication is forced into each component.

2.4.10 Transnet

Transnet is similar to HOOD/PNO in that it also uses Petri nets to model and verify the behavior of the system [55]. Transnet is different from that method in that it is concerned not only about the functionality of the design but also with the deadlines of the software and message passing and with object concurrency.

The steps of this methodology are as follows:

1. Identify the objects and the calls between the objects.
2. Model the object behavior as high level Petri nets.
3. Define data types as primitive sets together with their operations.
4. Construct the Petri net reachability trees.
5. Analyze the trees for reachability, safeness, deadlock, and freedom from starvation.
6. Assign timing to Petri nets.
7. Validate timing and net execution.

One disadvantage of Transnet is that it only supports the specification and preliminary design steps of the life cycle. Its advantage is deadline management support and concurrence analysis.

2.5 Analysis

Table 2.5 shows a summary of the ten methodologies presented above. Each of the methodologies is compared according to the comparison criteria of Section 2.3.

Concurrency is often an important issue in real time software problems. The methodologies that have better support for concurrency are COBRA, RTO, Transnet, OCTOPUS, OMT and OPNets. Related to this is the control structure of the methodology. The methodologies that support a single object control are COBRA and Transnet. The other methodologies support distributed control mechanisms. A very concurrent system with hard or firm message passing deadlines would benefit from a distributed control structure system design.

Another important issue in real time systems is the handling of real time deadlines. Methodologies that do not deal with the deadline issues are not true real time methodologies. In this paper, ARTS, HRT-HOOD, ROOM, and OCTOPUS have the best support for real time deadlines. These all incorporate techniques for determining if the deadlines will be met and for showing the predictability of the systems. RTO and Transnet methodologies both have some deadline handling support

but they do not include any techniques for analyzing the predictability of the system. The remaining methodologies all support deadlines only in the implementation phase of the design cycle and may not be true real time design methodologies.

In all of the methodologies studied, except for COBRA and OCTOPUS, the first design decision is selecting the objects. In COBRA, the first design decision is selecting the concurrent processes. In OCTOPUS, the first decision is subsystems. This is not a surprising result in that all the methodologies are of the object oriented paradigm.

It is not clear what behavior modeling technique is incorporated in ARTS and HRT-HOOD. The behavior modeling for the COBRA methodology is event sequencing scenarios and statecharts. RTO, ROOM, OCTOPUS, and OMT also use scenarios and statecharts for behavior modeling. HOOD/PNO, Transnet, and OP-Nets use Petri nets. Selic point out that these modeling techniques are still limiting [59]. *None of these modeling techniques really allow the modeling of the deadlines of tasks and messages in the real time system.* The interaction of hard, firm, and soft deadlines in a system is not considered by any of the modeling techniques.

None of the methodologies studied in this paper discuss if, or how, deadlines and behaviors are inherited. If a class has a hard deadline and an object inherits from this class, is the deadline inherited? Say a subclass overrides a soft deadline with a hard deadline. How can the system ensure that this deadline will be met when an inherited operation is performed? Another problem is how is the object

behavior inherited? It is not at all clear if behavior is inherited, and if it is, how parts of the behavior can be overridden by the subclass.

	ARTS	COBRA	HOOD/PNO	HRT-HOOD
Concurrency Support	N	Y	N	N
Control	Distrib	Central	Distrib	Distrib
Deadline Management	Y	N	N	Y
First Design Decision	Objects	Concurrency	Objects	Objects
Behavior Modeling	Statechart	Statechart	Petri Nets	Statechart

	OCTOPUS	OMTs	OPNets	Room
Concurrency Support	Y	N	Y	Y
Control	Distrib	Central	Distrib	Distrib
Deadline Management	Y	N	N	N
First Design Decision	Subsystems	Objects	Objects	Objects
Behavior	Statechart	Statechart	Petri Nets	Statechart

	RTO	Transnet		
Concurrency Support	Y	Y		
Control	Distrib	Central		
Deadline Management	Y	Y		
First Design Decision	Objects	Objects		
Behavior Modeling	Statechart	Petri Nets		

Table 2.1: Summary of Methodologies

2.6 Chapter Summary

Object-oriented design methodologies have several advantages in real time software design. First, they have some general advantages such as isolating the impact of changes on the design, and encouraging the reuse of design artifacts and code. The object-oriented paradigm can model appropriately the environment in which the software will be used, which is very important for real-time systems. The

information hiding aspect can allow the real-time system to have several methods for performing each task. There may also be some fault tolerance advantages. Lastly, the formal methods necessary to show the predictability of the system can be easily and naturally incorporated into the system.

The disadvantage of object-oriented techniques is that most general purpose object oriented methodologies have inadequate modeling support for real time software. Few methodologies even attempt to deal with deadlines, which is arguably the most important feature that separates real time software from regular software.

In this chapter, we examined ten real-time object-oriented software design methodologies and compared their strengths and weaknesses. A design methodology for real time software should help the designer deal with the special problems of the real time environment. If the environment has hard real time deadlines, then the methodology should support the consideration of this in the design stage, not just at the implementation stage. Concurrency is another issue that can be handled in a number of ways. The design methodology should support concurrency at the level that is required by the problem at hand.

Chapter 3

BEHAVIORAL MODELING WITH STATECHARTS

3.1 Introduction

Most object-oriented systems behavior models are based either on statecharts or on Petri nets. After reviewing both modeling techniques, we elected to concentrate on statecharts. Among the reasons for this decision is the fact that there is a mapping between statecharts and Petri nets, that is, they have similar modeling power for most of the common software applications. Personal preference may be the overriding deciding factor for choosing between the two methods.

Our first goal was to evaluate statecharts and the problems associated with extending them to incorporate real-time deadlines. We first looked at the nature of time measurements in software. Next, we looked at how simple automata are changed when timing is added. After that we reviewed statecharts.

3.2 About Time

Before we look at adding timing to automata or statecharts, we should first look at the nature of time. There are several ways of thinking about time. Each

has different attributes and causes us to think about our problems differently.

The first way of thinking about time we would call absolute time. Absolute time is very specific. Ten o'clock, Tuesday, and July 4th are all expressed in absolute time. Absolute time is not what we normally think about when we deal with real time programming. However, most software today can read the system clock and use absolute time.

When we are in a real time environment we more often refer to time in a second way, relative time. With relative time, we are concerned with issues such as did one event happen before another. Time is expressed in terms of before, after, and within an interval. This can be much less specific than absolute time but more significant from the point of view of an application.

The third way of expressing time is in the sense of temporal logic. Here the view of time is expressed as eventually something happens, or as something will never happen. This temporal logic view of time is well suited for making arguments about properties such as safeness and liveness.

Clearly all three of these views of time have a place in real-time software. The first view is no different than exists in non-real-time software. That is we can call functions that read the system clock and compare it to a value. The second view of time is more of a real-time view. This is the kind of timing we are attempting to add into the object behavior models. Finally, the temporal logic view of time is an important view that we need for reasoning about our designs. Adding this timing

alone does not make the software real-time software.

3.3 Timed Automata

Any automata from simple state machines to Turing machines can be modified with timing. This can be done by simply restricting the state transitions so that they can only be taken during specified time intervals. This simple change has far reaching effects that include state explosion, intractability, and undecidability. However, real-time systems require these timing constraints for understanding and verifying the critical timing interactions.

One of the best discussions of timed automata is that presented by Alur and Dill [3]. Here the timed automata are explained using sets of resetable timers. These timers can be reset as an action on a state transition or can be used as a condition on a state transition. Figure 3.1 shows an example of a timed automata. In this example, the transition from state S_0 to state S_1 occurs when the symbol a is read by the automaton. This transition causes the timer x to be reset to 0. The transition from state S_1 back to state S_0 is constrained by the condition $(x < 2)?$ In this example the transition will be taken if the symbol b is read before 2 time units have past after reading the symbol a .

There are several problems and issues addressed in the current literature for timed automata. These include:

- Timed automata suffer from state explosion when applied to realistic problems.

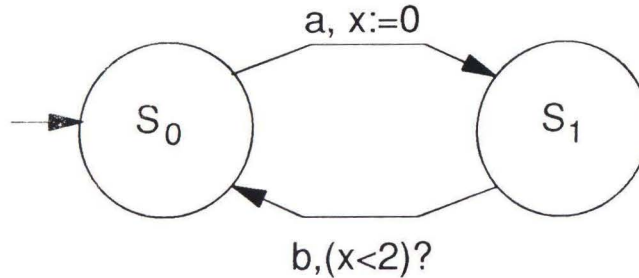


Figure 3.1: Example of a timed automata.

- Analysis of timed automata for properties such as reachability, safeness, liveness, and others, is often intractable.
- There is no agreement on the number and kind of clocks used in the automata.
- There is no agreement on discrete versus dense (continuous) time.

The state explosion issue is similar to the problem that led to the introduction of statecharts. The idea of a hierarchy and concurrency can be used to control the exponential state growth of most interesting problems. However, timing can cause even faster state growth, because each event can cause different transitions depending on the amount of time that has passed.

The tractability of timed automata is significantly more complex than that of regular automata. The issue of reachability, which can be fairly straightforward in a simple state machine, is many times more complex in timed state machines.

One example is the simple state machine that has an unreachable final state when an included timing constant is not an integer [37], [38]. Trying to show that a timed automata is empty (no strings reach the final state) is undecidable.

The number of clocks used in the automata is also an issue. Several authors recommend using multiple clocks [3], [5], [37], [38], and [46]. Others use the single clock model [30], [33], and [43]. The multiple clock models can be reset by an action or event. And some multiple clock models require that the clocks run synchronously and others do not. A model with multiple clocks that does not require synchronous clocks will definitely have more power when modeling a distributed system.

The last issue is that of discrete versus dense (continuous) time. If the clocks increase monotonically by an integer amount, then the clocks are discrete. Some argue that dense time is needed for most real-time situations. Several hybrid models that incorporate both times have been proposed [4]. It does appear that if the integer value of increase in a discrete clock is small compared to the values that were tested for, then we would not be able to observe a difference between discrete and dense time.

There are several solutions for getting around the problems of timed automata. Most involve restricting the automata in some way so that the resulting automata can be easily analyzed. One example of this restriction is the Alternating RQ timed automata [37]. Here the timed automata is restricted to one that has a finite number of clocks where each clock can be queried only once after it is reset.

Thus, on any path through the automata the clocks should alternate between resets and queries. Using this restriction it can be shown that the resulting automata are tractable. However, this also results in more states, as extra states are necessary just to keep the clocks understandable.

While some of this work on timed automata is theoretical in nature, the problems of intractability, undecidability, and reachability will also appear in timed statecharts. Thus we need to look closely at these solutions and see how we can incorporate them into future models. These solutions can be incorporated both directly (via a rule for the models) or indirectly in the way we structure the modeling technique. In any event, we should consider the above problems when looking at timed statecharts.

3.4 Statecharts

Harel's statecharts [29] are extensions to state machines that incorporate the concepts of hierarchy, concurrency, and communications. Statecharts, which are also known as Harel Diagrams, have become one of the most important tools for specifying and analyzing complex systems. They are now being used in a wide variety of tools.

Statecharts originated while examining the problems with specifying *reactive systems*. These systems are difficult to specify because they must react to a wide range of internal and external stimuli. This stimuli and the resulting actions are

complex and often have timing constraints. As a result, the traditional state machines are unmanageable, due to the exponential growth of the number of states needed to specify even moderately sized problems.

The exponential growth of state machines is a side effect of their limitations when describing concurrency. State machines consist of states and transitions. In a finite state machine, only one state can be active and all the others are inactive. Transitions are directed connections between two states. Each transition has a associated event and action. When an event arrives a transition is taken from the active state to another state and the associated action is performed. As the number of possible conditions of the system grows, the number of simple states necessary to represent each of these conditions grows and the number of connections between all the states grows. Every event that can happen must have a transition (fixed or implied) from every state. Events that are not explicitly defined for a state must have an implicit transition, such as transition to an error state or ignoring the event. Statecharts are an extension to the basic state machine that uses concurrency and hierarchy to eliminate the need for many states and transitions.

The concept of generalization hierarchy is incorporated into statecharts using superstates and substates. A substate is contained by a superstate. If the substate is active then the superstate is also active. Thus, more than one state is active at a time. The details of a superstate can be ignored by zooming out, and looking only at the external interactions of the state. In the same manner, we can zoom in and

look only at the internal of the superstate, thus allowing ourselves the ability to use hierarchy to concentrate only on the level of detail needed to solve the problem at hand.

A superstate can also allow more than one substate to become active at the same time. This allows concurrency in the system design. Concurrent states, shown together but separated by a dotted line in Harel diagrams, helps restrict the number of states necessary to build the system. The concurrent states can be synchronized by having transitions that cause state changes in all parts of the concurrent states. Otherwise the states are unsynchronized.

Communication between the states is based on the broadcast mechanism. There are several shorthand conventions necessary for keeping the diagrams uncluttered. A transition to a superstate implies that the marked default substate is the state entered. A transition from a superstate implies that, when the condition for the transition is encountered, the transition is taken no matter which substate is active at the time. Transitions from substates and to substates are also allowed.

Figure 3.2 is an example of a simple statechart. Entering state A automatically enters substates B and D simultaneously. If the event that triggers T3 occurs then state D would cease to be active and state F would become active. If state F is active and the event that triggers T7 occurs, then state A becomes inactive, regardless of which of states B and C were active. Likewise, if transition T8 is taken, state A becomes inactive no matter which substates were active.

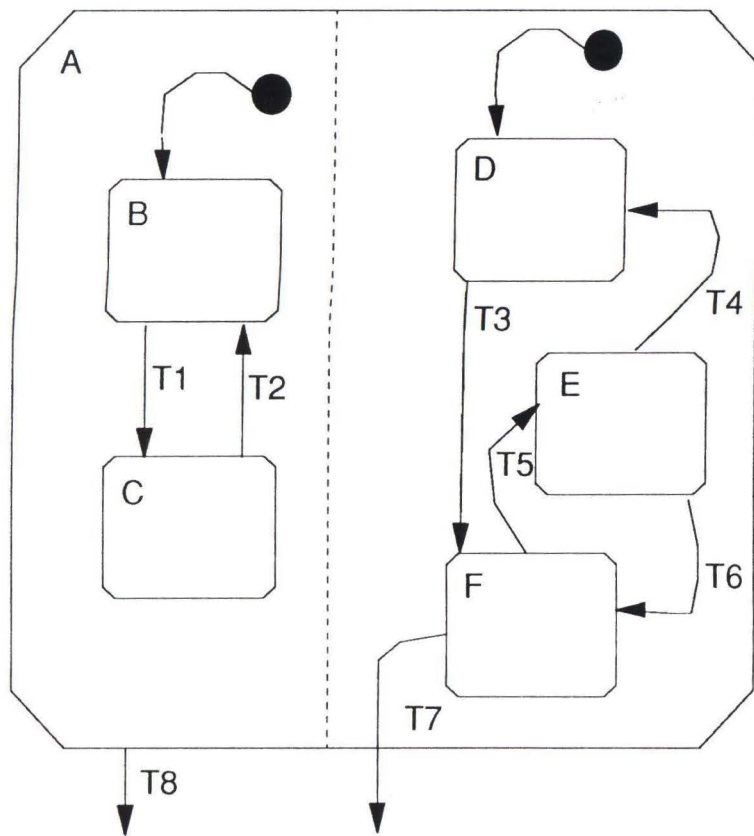


Figure 3.2: Statecharts Example

The result is that statecharts are a powerful tool used to visualize the states of a complex system. This tool allows us to concentrate on only the pieces of the problem that we need to at any point in time, yet result in a complete overall picture of the system.

Several extensions have been added to statecharts since they were introduced. One example of this is the extension of statecharts to be a graphical language for the programming of CNC machines [28]. In this example, the statechart is used as a graphical user interface (GUI) to create programs and to program numerical controllers of machine-tools (CNCs). The paper claims, but does not show, that the statecharts can be translated directly into CNC programs that have real-time considerations.

3.5 OMT Statecharts

An important variation of statecharts is the one used by Rumbaugh in OMT [54]. This extension is one of the most complete from a design consideration. It includes the concepts of conditional events that trigger actions, state activities, and lambda transitions.

One of the key features of OMT statecharts is the specification of the transitions. All the transitions are controlled by a criteria that includes an event, conditions, and actions. This is written as *event(attribute)[condition]/action* and attached to each transition as a label. The event is the event that triggers the transition and

the attributes are information passed along with event, but separate from it. For example, the event can be a signal from the keyboard controller signalling that a key was depressed. The attribute could be the information on which key was depressed. In many cases, the event is really an *event-expression* that is a boolean expression describing atomic events. The events can be single events (say event a), a combination of events ($a \vee b$, $a \wedge b$), or other special events (time-outs, λ events, negations, etc.).

The condition in an OMT statechart transition label is an expression that describes the set of conditions necessary to enable the transition. This is usually a description of other states that must be active (or inactive) for this transition to be enabled. Events that occur when the condition has the transition disabled do not cause a state change.

Actions are events created by the transition. There does not appear to be any limits on how many events can be generated or what kinds of events can be generated.

One feature of OMT statecharts that is very different from Harel's statecharts is the activities that occur inside the states. We can specify activities that will happen when a state is entered. Likewise, we can specify activities that will happen when a state is exited. We can also specify activities that will happen when certain events occur, even if that do not require, or cause a state change. These activities are specific, and have durations. As a result we can specify lambda transitions that

will occur, causing state changes, based on the activities in the state completing.

On one hand these state activities are convenient additions to statecharts. Many activities that will be performed in a state, can be abstracted to a high level entry, exit, or event activity. However, these activities may require further reduction into a statechart as we move into lower levels of design abstraction. It is important to note that these activities do not give OMT statecharts any more modeling power over any other statecharts method. Anything that can be modeled using activities can be modeled using substates in the statecharts.

One important benefit of using the OMT method is that a designer is strongly reminded that states do something and are not just parking places. Activities are performed inside states. That activity may be to perform functions, write to files, sleep, or wait for a key to be pressed, but it is still doing something.

There is one other difference between OMT statecharts and other methods. There is an underlying assumption in most statecharts that all transitions are instantaneous. When an event occurs, the transition does not take any time to change states. However, in OMT statecharts, a state can have exit activities, that take time to be performed. While this may not be a problem in non-real-time software, it adds a definite complication to real-time systems.

Figure 3.3 shows an example of Rumbaugh's OMT statecharts. As we can see, the states can have activities in them. These can be regular, entry, and exit activities. Further more there can also be activities that are triggered by external

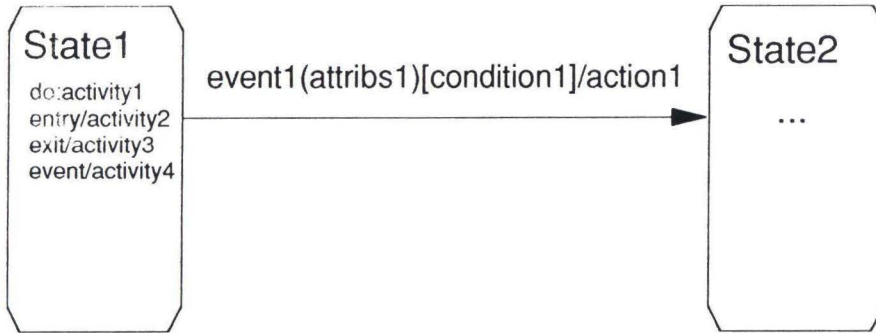


Figure 3.3: Example of Rumbaugh's OMT statecharts.

events. Transitions are triggered by events with conditions, which cause actions. Activities are similar to actions except that activities take a significant time to complete. Actions on the other hand are so quick to complete that they are considered instantaneous.

Hooman et.al. used an axiomatization formalism to make formal assertions about the properties of statecharts [31]. In their paper, the authors show how logical specifications can be added to the statecharts in order to make formal assertions about safeness and liveness properties.

Another example of formalism being added to statecharts is the Syntropy method of Cook and Daniels [18]. They argue that the graphical notations of statecharts, while powerful, often lack expressive power. To overcome this, written

words are often added to statecharts, but the words are themselves often ambiguous. Something more formal is needed.

The Syntropy method is based on OMT. This begins by associating each part of the OMT notation with a precise mathematical meaning. For example, a one to many association is interpreted precisely as a mathematical function mapping the objects of the first set into the objects of the second set. This can be a very simple function, $a \in b$, or a much more complex function.

In their use of statecharts only events, cause state changes. The design must specify which objects are affected by what events, when events can happen, and what are the consequences of the events. To do this, each state has a list of the events to which the state will respond. Transition guards control when states can respond to events (change states). Events that are on the list but can not trigger state changes are considered undefined. It is not clear why they allow events on the list that would be considered undefined, unless it was to enable them for substates.

Figure 3.4 is an example of a Cook and Daniels statechart. This example shows a simplistic traffic light controller that allows the traffic light to be in a reset state where all the yellow lights flash or in the normal functioning state (Running) where the lights cycle thorough their sequence. These states are switched between by the events Reset and NotReset. The rules of this kind of statecharts say that an event that is not on the list of events for the current active state, cannot be generated. The section at the bottom of the statechart labeled 'Allow' shows what

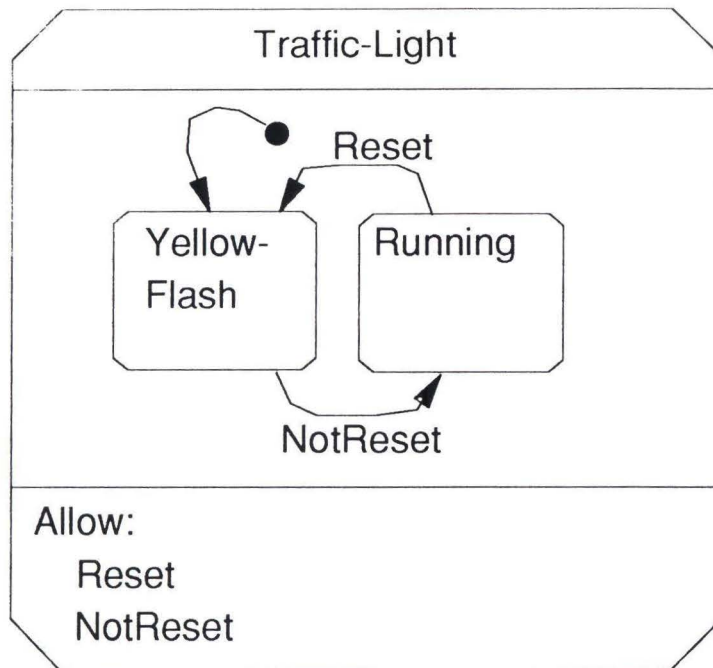


Figure 3.4: Example of Cook and Daniels Statecharts Formalism

events this statechart will accept. In this case it is the events Reset and NotReset. Cook and Daniels describe this as an implied contract that requires the design to ensure that all events are allowed before being generated. This is a somewhat limiting restriction.

Once generated, the events can cause one or more of several consequences. First they can cause a state change in the statechart. They can also cause a change to the object's properties, or they can cause a change to the membership associations of the object. Also, they can generate other events. And, lastly, they can cause the

termination of the object. In all cases, the consequences of the event can be clearly stated and precisely described.

3.6 Timed Statecharts

One of the significant shortcomings of statecharts for real-time systems is their inability to describe and model timing constraints. There are several ways to introduce timing into statecharts. One way was proposed by von der Beeck [65], who added the concept of timed transitions to statecharts. A timed transition has an upper and lower time bound for the transition; that is, the event must be active for a minimum amount of time (the lower bound) before the transition can be taken, but it must be taken before the maximum amount of time elapses (the upper bound). This allows the introduction of the concepts of delay and time-out into the model.

The formal syntax of the transitions are as follows:

$c[c]/a$ for untimed transitions

$([c] \text{ for } I)/a$ for timed transitions

where c is an event, c is a condition, a is a sequence of generated actions and events, and I is a time interval $\{l, u\}$, that specifies an upper bound u , and a lower bound l , for the duration of the time interval. Using the timed and untimed transitions, the real-time system characteristics of delay, time-out, and preemption can all be modeled. In this model, the transitions triggered by events are always untimed.

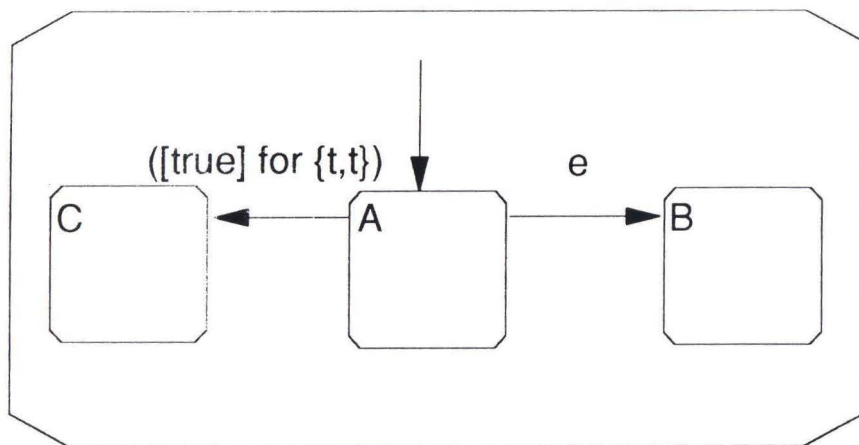


Figure 3.5: Example of a timeout condition

This timing technique does appear powerful in specifying delays and timeouts. An example of a time out condition is shown in figure 3.5. If state A is active and event e occurs before t time units pass, then state B becomes active. However, if t time units pass before event e occurs, then state C becomes active.

Another example is the extension of statecharts into objectcharts [16]. Here statecharts are extended with default states, global timing, and timed transitions. The timed transitions are the same as those above. Each state in an objectchart that has hierarchy, must have a defined start state. This is the same as Harel's default state.

Objectcharts require a global clock. The global clock needs to be available to every object. Also all the states in the objectchart need to have access to the clock. How this would be handled in distributed systems is not clear.

Lastly, `objectcharts` uses timed transitions for timing. Transitions can have specified minimum and maximum delays. Any `objectchart` transitions can use time as a firing consideration.

Another approach for introducing time into statecharts is `statecharts+`, which are based on the model of timed automata [66]. Like the timed statecharts, `statecharts+` have timed state transitions. In addition, `statecharts+` also have timed states. However, there is a difference in the behavior of these timed states and timed transitions compared to the above.

The timing constraints on the states contain upper and lower bounds. For example, a state may have a time specification like $[l, u]$ where l is the minimum amount of time the state must be active before a transition can be taken and u is the maximum amount of time that a state can be active before transition must be taken. It is not clear what happens if the state times out and no transitions are available.

The `statecharts+` timed transitions are the same as the previous timed transitions. The timing is used to restrict when a transition is enabled. If an event occurs when the transition is enabled it must be taken. To accomplish this, a set of clocks must be established and driven by a master clock. The clocks for a transition can be reset (as the result of an action) but the clocks for the states can not.

An example of this is shown in figure 3.6. Here state A has a timing constraint where it must stay in the state for l_s time units, and must leave the state by u_s time

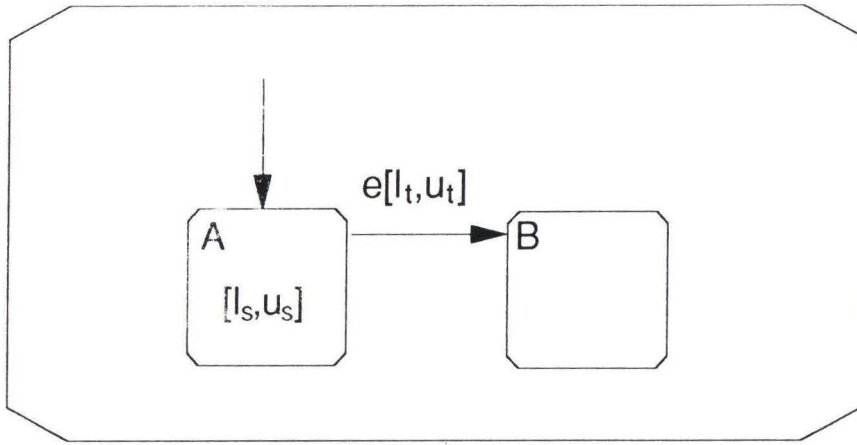


Figure 3.6: Timed Statecharts+ example

units. The transition taken when event e occurs between l_t and u_t time units leads to state B.

3.7 Evaluation and Analysis

To make a detailed analysis of the way statecharts are used to describe and analyze the behavior of object-oriented objects, we chose three methods to study. We chose OMT, Coleman, and Cook and Daniels. We chose Rumbaugh et.al. [54] OMT, because it is a popular method that is well documented. We chose Coleman et.al. [16] and Cook and Daniels [18] to compare the structure of their transition specifications.

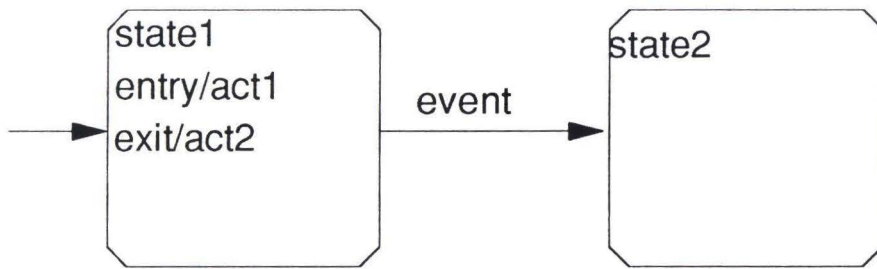
By analyzing the strengths and weaknesses of these three methods, we will have a better understanding of what features are desired in a statechart behavioral description.

The OMT statecharts can have very detailed states that specify what activities occur when the state is active. In this way, OMT statecharts are different from Harel's original statecharts. This does cause a problem with the assumption that state transitions occur instantaneously. In OMT, there are entry and exit actions that take time to be performed. In non-real-time software, this is not a problem. However, in real-time software these actions need to be carefully considered and the time to perform them accounted for.

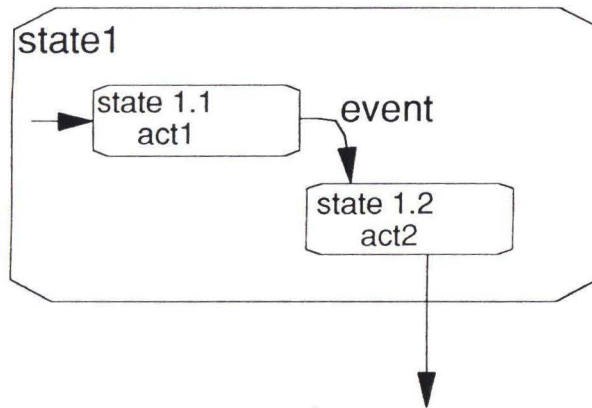
The activities in an OMT statechart could be considered a high level textual description of the lower level states in a statechart. For example, figure 3.7 a shows a detailed OMT statechart. In figure 3.7 b there is an equivalent mapping of this to a Harel statechart. Rumbaugh et.al. advises using entry and exit activities when all the transitions into or out of a state cause the same actions.

The Cook and Daniel method gathers the common information about the transition specifications into each state. Where it is necessary, formal mathematics are added to clarify the specifications. Since all the common information for the transitions are specified in one place, it is fairly easy to analyze the behavior of the statechart. The transitions only need to carry the information unique to each transition.

The Coleman method has all the information for the transitions in the transition preconditions and postconditions. This is not as easy to follow as the Cook and Daniels method. For instance, it is more difficult to understand when transitions are



a.



b.

Figure 3.7: Comparing OMT statecharts with regular statecharts

affected by a condition change. The use of transition conditions makes it easier to add new transitions and states, thus this methodology facilitates design iterations.

Two different ways of representing time appear in these three methods. OMT *uses absolute time only. Here there are no local clocks, but the system clock can be read.* Cook and Daniels also uses only absolute timing. In this method, there is only a global clock that every object can access and read. This is rudimentary timing and can only be used to restrict two processes from overlapping. To be really useful for real-time programming both methods need to include expanded timing information. The Coleman method uses relative timing. In this case, a local clock can be started when a state is entered. This clock can then be used as a condition for state changes and triggering events in the model.

Another way of comparing models is to look at how the objects and states communicate with one another. There are two basic models: the broadcast and the client server model. In the broadcast model, all states are aware of all events and conditions generated. If a state does not change when an event occurs, it is ignored. In contrast, the point to point or client server model sends the events only to specific states. Each state must know what events it expects and what transitions are associated with each event. Undefined events are not allowed. OMT and Cook and Daniels use the broadcast communication method. Events can occur in parallel, and occur instantly everywhere.

There are paradoxes that can occur when broadcast is used, but special rules

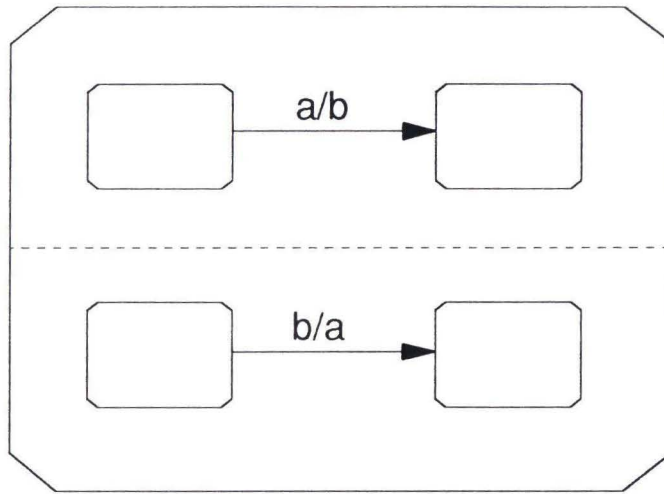


Figure 3.8: Example of broadcast communication paradox.

are put in to control them. Several examples are shown in [31]. These problems occur because of our assumption that all the transitions occur instantaneously and simultaneously. Consider the case in figure 3.8 where in concurrent parts of the statechart event a triggers a transition that generates action b . In the same statechart, the occurrence of event b triggers a transition that generates action a . At this point the occurrence of either event a or b will cause the other to occur instantaneously.

The Coleman method uses the client server form of communication. In this model, all events happen instantly but sequentially. Events are only sent to specific states. Event traces are used to map out the scenarios. The paradoxes of the broadcast method are avoided.

	OMT	Cook and Daniels	Coleman et.al.
Concurrency	implicit	implicit	implicit
Expression of time	Absolute	Absolute	Relative
Real-time	Yes	Yes	Yes
Communication	Broadcast	Broadcast	Client Server
Clocks	Global Clock	Global clock	Local clocks
Structure of timing	Transitions	Transitions and states	Transitions

Table 3.1: Comparison of Three Methodologies

Finally, we summarize the three methodologies examined above. OMT is less graphical than the other methods. Much of the information for the behavior is recorded textually inside the states. When this is used correctly it can be powerful, however, it does not increase the modeling power of the method.

The Cook and Daniels methodology is the easiest to analyze since information is gathered in a table in the states. But for the same reasons it is more difficult to design with. It is difficult to add new transitions and events must be carefully specified before being generated. Concurrency is implicit and communication is broadcast to leave more freedom to the designer.

The Coleman methodology is easier to design with, for the same reason that Cook and Daniels is not. That is, transitions can be added with relative ease. However, this method does not facilitate analysis. Each transition must be examined individually during analysis. It has fair expression of timing characteristics, but could be better with timers. Coleman has also clearly addressed the communication issue. It appears that this method and the Cook and Daniels method have the same

modeling power, and this suggests that it may be possible to transform between them to combine the easier design ability of this method and the easier analysis of the Cook method.

While some of the methods here discuss part of the relationship of the behavior model to the object model, none of the methods details how this should happen. There is no discussion of the relation of composition of objects or inheritance and the behavior of the statecharts. In OMT, Rumbaugh skirts the issue by advising that only objects with meaningful dynamic behavior should be modeled with a statechart. Rumbaugh also advises that events in the statecharts are the methods from the object model. Clearly, more work needs to be done exploring the link between the object and behavioral models.

3.8 Chapter Summary

The behavior models used in most object-oriented design methodologies are either statecharts or Petri nets. In this chapter we concentrated on statecharts, looking at the problems associated with incorporating real-time deadlines into the behavior models.

First we examined the difficulties in adding deadlines into simple automata. From this analysis we found that time constraints add a high level of complexity to simplest of automata. These time constraints often make it difficult to prove simple automata properties, such as showing that every state is reachable.

Next we examined three methods of adding formalism to statecharts. We examined the methods of Cook and Daniels, Coleman, and Rumbaugh's OMT. These three methodologies were similar to one another in most respects. All lacked support for inheriting deadlines and none supported only timed state automata.

Chapter 4

TIMED STATE STATECHARTS

4.1 Introduction

All the methods of adding timing to statecharts in the previous chapter used or included timed transitions. We wondered if timing could be specified in the statecharts using only timed states. In some cases, using only timed states results in a model that is easier to design with, is more extensible, and that better represents the deadlines we are trying to model.

In the timed state methodology the deadlines are modeled with count down timers. Upon entering a state with a deadline a timer is started. If the deadline expires before the state has been left, an exception or time-out event is created. This real-time behavior modeling methodology can be easier to evaluate than the timed transition methodology.

4.2 Timed Transition Problems

In the previous chapter, timing was added to statecharts by making time a condition on a state transition as follows:

$e[c]/a$ for untimed transitions

$([c] \text{ for } I)/a$ for timed transitions

where e is an event, c is a condition, a is a sequence of generated actions and events, and I is a time interval $\{l, u\}$, that specifies an upper bound u , and a lower bound l , for the duration of the time interval. Another approach to modeling time in the state transitions is to simply add a time interval to every transition. Transitions that we want to behave as untimed transitions would have a zero to infinity time interval. The formal syntax of this transition model would be:

$e([c] \text{ for } I)/a$ for all transitions

where e is an event, c is a condition, a is a sequence of generated actions and events, and I is a time interval $\{l, u\}$, that specifies an upper bound u , and a lower bound l , for the duration of the time interval. To model all the situations that can occur in real-time systems we will need some special events such as a λ event or a time out event to trigger a state change. This gives this model all the power of the previous model plus some as events can now trigger timed transitions.

There is one important ambiguity that is treated in different ways in the literature. How the upper bound of the time interval is interpreted is an important difference. In some cases, the upper bound is treated as a gate that simply turns off the ability of the transition to be taken. This is called *weak time semantics*.

For example, events that occur after the upper time bound has past are treated as any unexpected event would be. In the other case, the upper time bound is a requirement. Here the transition must be taken before the time limit expires. This is called *strong time semantics*. Part of the analysis of the system would be to show that the event was indeed generated before the time limit expired.

Like the upper time bound ambiguity there is a similar ambiguity concerning the lower time bound. One way to consider the lower bound is that it is the specification of the amount of time, since entering the state, that must pass before a transition can be considered. In other words, if event e occurs before time l , it is ignored, and if it occurs after time l , the transition is taken to a new state. A different way of thinking about the lower time bound is the amount of time the event must be true before the transition can be taken. Here the timing starts only when the event (with the conditions) becomes active and if the event stays true for the minimum time, the transition can then be taken to the next state. This nondeterministic form is not found often.

4.3 Timed State Statecharts

The timed statecharts above all have some disadvantages. First, there is ambiguity in the meaning of the common implementations. In most of the implementations, it is not clear if the model uses *weak* or *strong* time semantics. Second,

there can be obscure design flaws. To overcome these problems we look at moving the timing information from the transitions to the states. A further refinement where time intervals are changed to state timers, with strong time semantics, results in a new design method that results in better models. We call this Timed State Statecharts or TSSC.

This new timed statechart method is similar to normal state machines in that all transitions are freed from timing constraints. If an event occurs, and the conditions are true, the transition is taken. States can have timers, and substates are subservient to superstate timers. When a state timer counts down to zero a new event is generated called a state time-out. The time-out event is used to transition to a new state. In some cases, additional states are necessary to model the behavior of the system. In some cases these are dummy or place holder states. However, these extra states are far from state explosion, and in fact serve to clarify the design.

The following additions are necessary create TSSC from statecharts:

1. States can have a timer that is reset to its starting value whenever the state is entered. States without timers can be considered states with timers set to infinity. The notation used is $\text{State}[t]$.
2. Every timer in the system has the same period, but incorporates whatever granularity is needed by the state. This incorporates the idea of the master

clock, but allows us to use hour, minute, second, or even microsecond timers if that is what the design calls for.

3. When a state timer counts down to zero, a time-out event is created. A transition that uses this event should exist in the statechart.

4. All transitions are untimed and will be taken if the state allows the transition. This occurs regardless of the state timer.

As an example of how TSSC works, we use the gas burner example of [66].

The specification of the problem can be stated as follows:

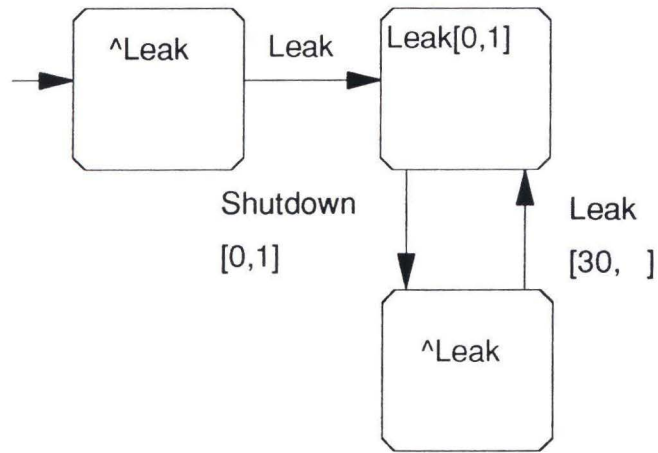
- A leak should be detected within one second.
- When a leak is detected, the gas should be turned off.
- After 30 seconds the gas can be turned on to see if the leak still exists.

In figure 4.1 the Statecharts+ solution of Wang and Chen is shown along with our TSSC solution. As we can see the Statechart+ solution uses both timed states and timed transitions. The default state is `!Leak` (not leak). If a leak occurs the event `Leak` changes to state `Leak[0,1]`. In this state the leak must be detected within one second, so the state `Leak` has a time limit of one second represented by the `[0,1]`. The shutdown transition must be taken within one second. This changes to a new state which Wang and Chen unfortunately gave the same name as the

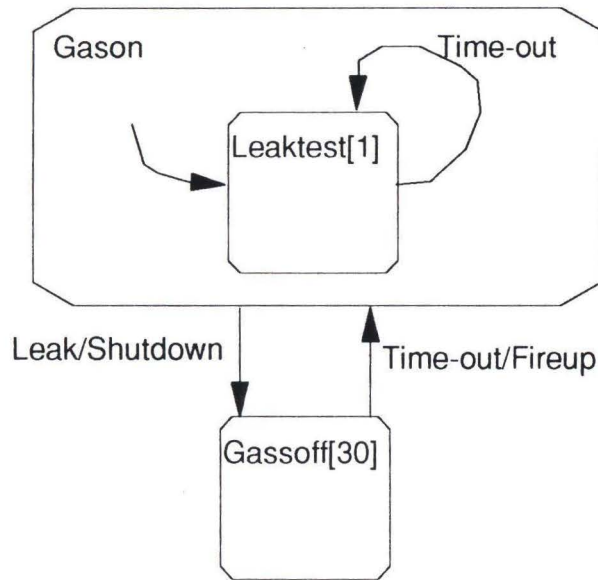
default state `Leak`. After 30 seconds the gas can be turned on to see if the leak still exists. This is purportedly shown by the transition `Leak[30,]` back to the state `Leak`. In this case the timed state (`Leak[0,1]`) may be redundant and if eliminated would not alter the behavior. Also the unfortunate use names in this example makes it difficult to follow.

The TSSC solution uses two timed states and one untimed state. State `Gason` is a super state of timed state `Leaktest`. `Leaktest` cycles performing a leak test every minute. If a leak is detected event `Leak` causes a state change to state `Gassoff` and causes action `shutdown` to occur. When state `Gassoff` times out the transition to state `Gason` occurs and action `Fireup` is started.

One advantage of the timed state statecharts is that the time bound ambiguities are handled explicitly. An example of this is shown in figure 4.2. In figure 4.2a, the timed transition statechart is shown. In this example if state `S1` is active and event `e` occurs between time `x` and time `y`, then state `S2` will become active. As discussed earlier the weak/strong time semantic ambiguity exists in both the upper and lower time bounds. In figure 4.2b, the timed state statechart is shown in the case where event `e` must occur at or before the upper time bound. In part c we show the case where the transition is disabled if the upper time bound is past before event `e` occurs. In this example, we only consider the more common lower bound where the transition is not enabled until the lower time bound has past. A similar expansion can be done to incorporate the other lower time bound ambiguity. As we



a.



b.

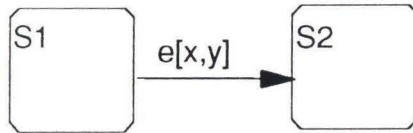
Figure 4.1: Gas Burner example, a) Statecharts+ and b) TSSC

can see, these ambiguities are explicitly defined when using timed state statecharts.

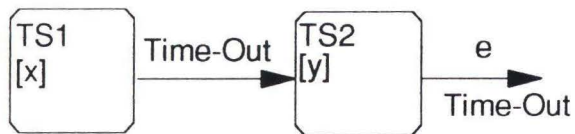
The ability of timed state statecharts to explicitly resolve ambiguities does not diminish their ability to model any desired behavior. As shown in figure 4.3, we see how to model a set of states where there is a combination of timed and untimed transitions. In this case figure 4.3 *a* shows an example where when state S2 is active, a timed transition would make state S1 active and an untimed transition would make state S3 active. Thus if event *e* happens between times *x* and *y* then state S1 becomes active. If event *e1* happens at any time then state S3 becomes active. Figure 4.3 *b* shows how TSSC would preform the same tasks. State S2 is shown as using nested states. The default state is TS1. If event *e1* occurs at any time state S3 (not shown in figure 4.3 *b*) would become active. If state TS1 is active for *x* time, a time out occurs that makes state TS2 active. Now if event *e* occurs state S1 (not shown in figure 4.3 *b*) would become active. If state TS2 is active for time *y*, then a timeout makes state TS3 active, disabling event *e* but not event *e1*. Thus by using the hierarchical power of statecharts and the simplicity of timed states, it is always clear what behavior is desired.

There are several advantages to timed state statecharts. They are a closer representation to the real-time systems that we are trying to model. Real-time systems are often concerned with tasks completing inside a time interval. Timed states is a closer model of this requirement.

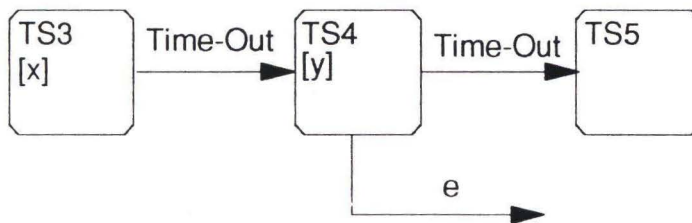
Timed state statecharts also do not have any ambiguity in their notation.



a.

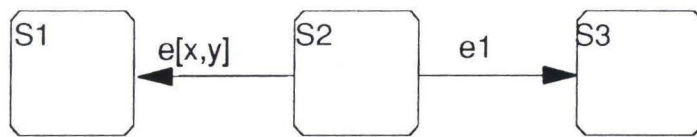


b.

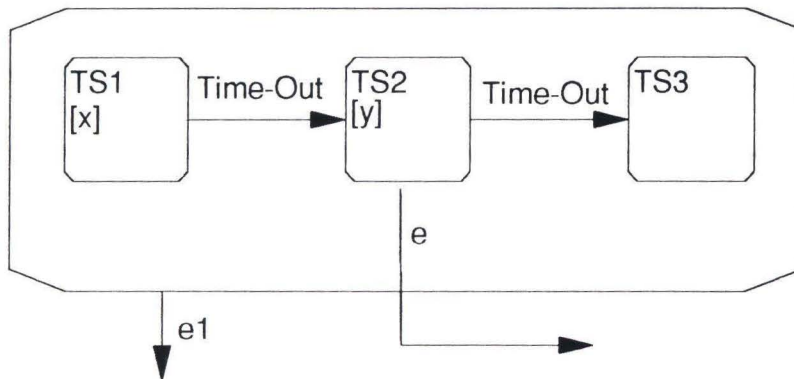


c.

Figure 4.2: Ambiguity resolution with timed state statecharts.



a.



b.

Figure 4.3: Example of timed and untimed transitions in TSSC

This sometimes results in a few more states in the statechart. However, these extra states will more often than not be swept up by the hierarchy of the statecharts. Therefore, these extra states are not a problem.

Timed state statecharts will expose poor designs quicker than timed transition statecharts. One reason for this is that it is easy to examine the timer of the outer most timed state and then see that the cumulative time of all strings of timer inside this state add up to less than the outer most timer. We can easily examine all the timed states inside a timed state and ensure that they are reachable before the state expires. Further more this can be repeated from the lowest to the highest level in the statechart hierarchy.

While TSSC have several real advantages, almost all of the important real-time object-oriented methodologies use timed transitions to model the timing in their behavioral models. In order to make the next chapters understandable to the main stream object-oriented researchers, timed transition models will be used almost exclusively.

4.4 Chapter Summary

In this chapter we examined the modeling power of statecharts with real-time deadlines, where the deadlines are modeled by timed states. We defined how timed state statecharts work and used them to model real-time deadlines. These timed state statecharts were then compared to timed transition statecharts.

We further identified the and discussed the strong time - weak time semantic problem. We showed where the timed state statecharts eliminated this semantic problem in the real-time environment.

Timed state statecharts have the advantage of better representing real-time deadlines. Furthermore, they do not have ambiguity in their notation. They can result in more states being defined, but this is controllable with the statechart hierarchy. Timed state statecharts also can expose design problems quicker than timed transition statechart models can.

Chapter 5

OBJECT AND BEHAVIOR MODELS

5.1 Introduction

This chapter deals with the relationship between object and behavior models. The effects of generalization inheritance, aggregation, and relationship associations on the object models are well known. These associations are a key part of the difference between object-oriented and other design methodologies. In some object methodologies the behavior model is not considered until after all the object relations have been settled.

Object models have a well defined inheritance notation with clear meanings associated with it. Behavior models are less defined with no clear notation and no clear understanding of how to reuse the behavior models. Most methodologies rely on complete respecification of the statecharts [63].

First, we will examine how the behavior model changes during object model inheritance. The term inheritance has two meanings in object-oriented terminology. The first is called subtyping and the second is called redefining of methods. This

chapter concentrates on subtyping as it is the more desired form of inheritance as well as the more restrictive.

There are eight ways that the behavior model can be changed after object model inheritance, and still maintain subtyping. Each of these is examined in detail. These all fall into three major categories, refinement of transitions, refinement of states, or refinement of attributes. Examples are used to illustrate the methodology.

Next, we examine how the object model association of relationship affects the behavior model. An object model relationship connection shows the communication paths between objects. There is a corresponding behavior model connection that defines the temporal nature of the communication path. This allows a clearer and fuller description of the object relationships. We introduce a new model notation for clearly showing this relationship.

Finally, we examine the object model aggregation and how this affects the corresponding behavior model. Object model aggregation causes some form of concurrency in the behavior model. Through the use of examples we show the resultant concurrency.

We also show how to incorporate coordinating aggregation in the behavior model. By introducing another new notation we show how a difficult concept can be easily and clearly explained in the behavior model.

In this chapter, we look at how object models and behavior models are related in Section 5.2. Second, we look at object and behavioral inheritance, including

multiple inheritance, in Section 5.3. Next, we look at object and behavioral association in Section 5.4, while we explore how these models are affected by aggregation in Section 5.5. Lastly, we present some conclusions in Section 5.6.

5.2 Object Models and Behavior Models

Object oriented methodology is the definition of a problem and the environment of its solution in terms of objects. These objects have a name, a list of attributes, and a list of actions (also called operations or methods) that they perform. Using the techniques of aggregation, generalization, and relationship these objects can be combined to create software to control or simulate the system.

Each object has a behavioral model associated with it. In this paper, we will use the graphics shown in Figure 5.1 to represent the object and behavioral models. The behavioral model is usually represented as some variety of statechart [29]. Exploring the relationship of the object and behavior models by considering the effects of inheritance and aggregation is of prime importance to develop strategies to make better use of object-oriented technology and these aspects have not yet been explored much.

The statecharts used in this paper for the behavior models are Harel's statecharts. Specifically, we use the basic statechart constructs of hierarchy, broadcast communication, and concurrency. Hierarchy is used to simplify the models and reduce the number of transitions that must be shown. Broadcast communication is

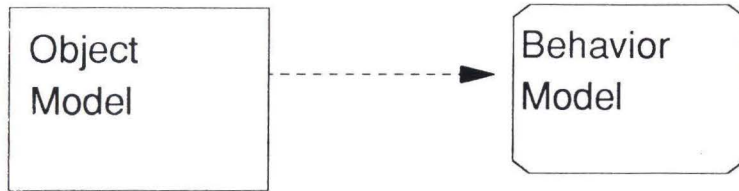


Figure 5.1: Object and Behavioral Models

assumed to make the models easier to construct. For this work we do not need to use any of the advanced features of Harel's statecharts, such as the history entry point or states that have multiple modes of concurrency.

5.3 Object and Behavioral Inheritance

The concept that objects are related by inheritance and generalization is important in object-oriented technology. A subclass inherits when it takes the properties of the superclass and specializes by incorporating features that make it unique. Inheritance is valuable when a class is being reused from a previous problem or from a software library. In generalization, objects have their similarities factored out into a superclass. This leaves each subclass object describing only what is different from the common properties of the superclass.

Coleman point out that there are two types of inheritance *subtyping* and *redefining of methods* [16]. Subtyping requires that the child class can be substituted

for the parent class anywhere. Furthermore, any event trace that would have been accepted by the parent class must also be accepted by the new class. The set of event traces are also referred to as the set of value vectors [44]. For subtyping to hold the set of traces of the parent should be a proper subset of the set of traces of the child. Note that this set of event traces is not the same as the event trace diagram that will be discussed later.

Redefining of methods can be seen in ROOMcharts [57]. In ROOMcharts all generalization is considered redefining of methods for pragmatic reasons. Selic believes that, even if overriding was disallowed, that it would not be possible to ensure behavioral equivalence. Both Coleman et al. and Selic point out examples where subtyping inheritance could not be achieved, even when following a rigid criteria. The issue of subtyping vs redefining of methods remains an open issue where further research is warranted.

5.3.1 Inheritance Behavior

We have identified the following eight different ways that the behavior model can be affected during inheritance [16, 17, 44]. They are also illustrated on the following pages.

1. Addition of an extra transition.
2. Retargeting and Splitting of a transition.

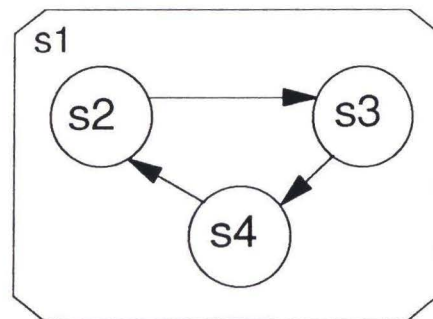
3. Weakening of a precondition of a transition.
4. Strengthening of a postcondition of a transition.
5. Strengthening of an invariant relationship.
6. Refinement of a state into two or more states.
7. Addition of new attributes resulting in additional independent states.
8. Modification of a state to change its interpretation but resulting in an unchanged diagram.

5.3.1.1 Addition of an Extra Transition

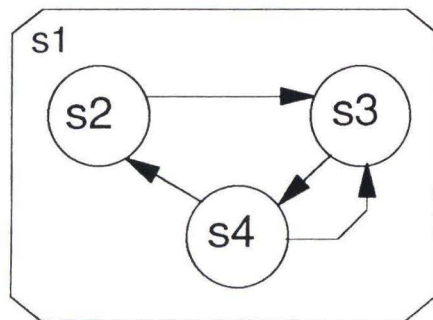
The addition of a transition is fairly straightforward. When a transition is added to the behavior model of the child class it still models the behavior of the parent class, but the extra transition adds new behaviors. An example of this type of inheritance is shown in Figure 5.2.

5.3.1.2 Retargeting and Splitting of a Transition

Retargeting a transition changes the transition to a new internal substate of the original state. This is often used in conjunction with the refinement of a state into two or more states below. Here we modify the transition to point to a new specific internal state of the original state.



a.



b.

Figure 5.2: Inheritance that adds a transition to the behavioral model.

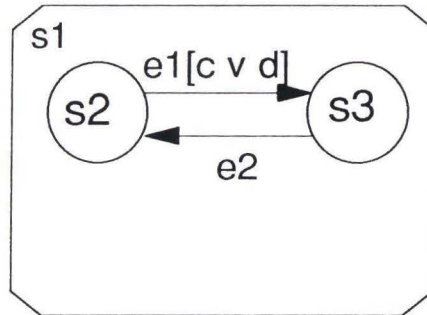
At the same time, a transition can be split into two or more transitions. These new transitions can be controlled by different conditions, but the combination of conditions needs to logically OR'ed into the original conditions. There are two ways splitting is used: to go to different internal states based on conditions, or to emanate from different internal states, and generating different events when triggered. For example, in Figure 5.3 event *e1* is split depending on condition *c* or *d* and retargeted to state 4 and state 5. Similarly event *e2* can trigger different events depending on the substate of state 3 that was active when the event occurs.

5.3.1.3 Weakening of a Precondition of a Transition

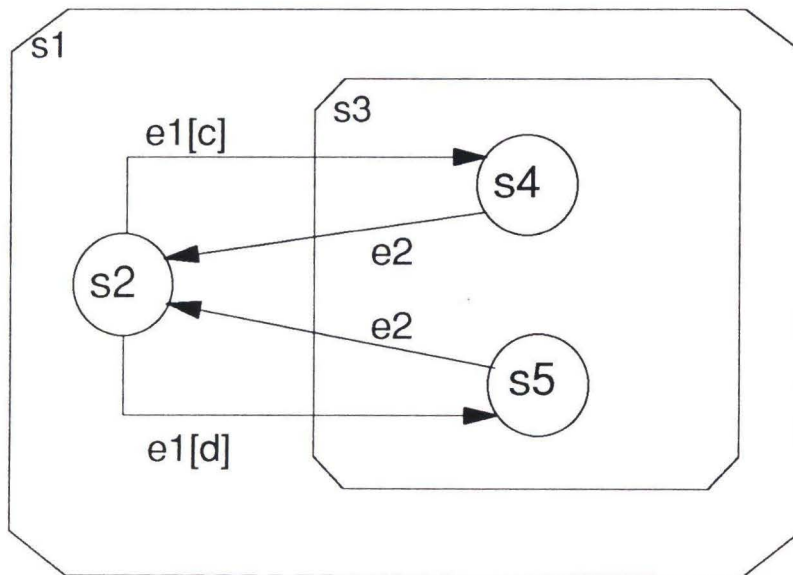
A child class can also weaken a precondition of a transition. In this case, the child class allows the transition to occur more often. For example, in Figure 5.4a, in the parent class the event *e1* triggers a transition from state *s2* to state *s3* when condition *c1* is true. In part *b* of the figure the precondition has been weakened so that event *e1* causes the transition from state *s2* to state *s3* whenever conditions *c1* or *c2* are true. Care must be taken here however, as Coleman [16] pointed out, because it is possible to create cases where changing a precondition that affects internal class choices can result in inheritance that is not subtyping.

5.3.1.4 Strengthening of a Postcondition of a Transition

Strengthening a postcondition is similar to weakening a precondition. In this case, Coleman means setting fewer conditions when the transition is taken in

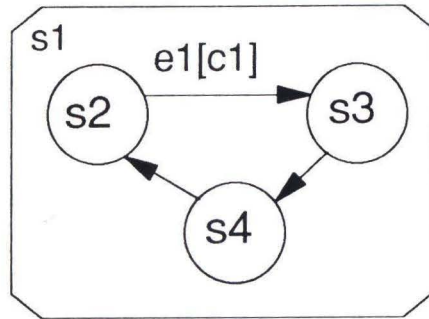


a.

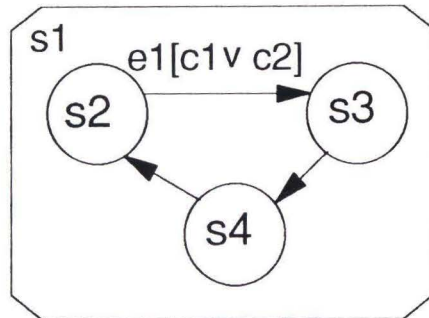


b.

Figure 5.3: Inheritance where a transition is split and retargetted.



a.



b.

Figure 5.4: Inheritance weakening a precondition.

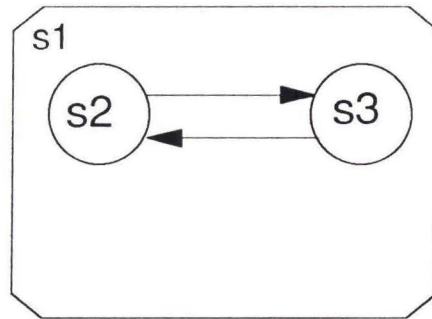
the subclass than the parent class sets. This also has the problem with internal class choices pointed out in [16]. Coleman clearly use weakening a precondition to mean the union of additional terms in the condition statement. Likewise they use strengthening a postcondition to mean the intersecting of additional terms in the postcondition. It is not clear that this is universal as Lecoecuche and Sourrouille [39] appear to allow behavior that contradicts this. Both may be correct.

5.3.1.5 Strengthening of an Invariant Relationship

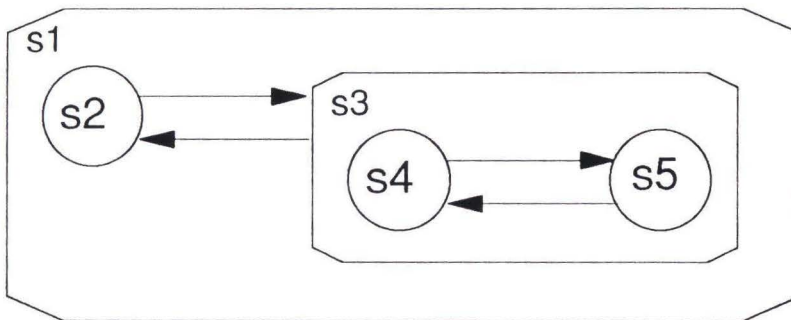
An invariant relationship is an assertion that is always true for a class. This could be something as straightforward as the fact that the attribute '*miles*' in a class *Truck* can never decrease. Again, care must be taken when using specialization that uses invariant strengthening in order to maintain strict subtyping. The invariants are usually handled in the functional model. Since this paper will not deal with the functional model, this relationship will be explored in a future work.

5.3.1.6 Refinement of a State into Two or More States

A state can be refined into two states. An example of this is shown in Figure 5.5. In this case, the behavior will be the same as that of the parent except for some specific conditions where it will have a finer definition. This may be the most common form of inheritance.

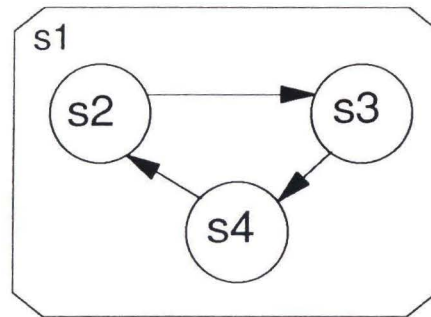


a.

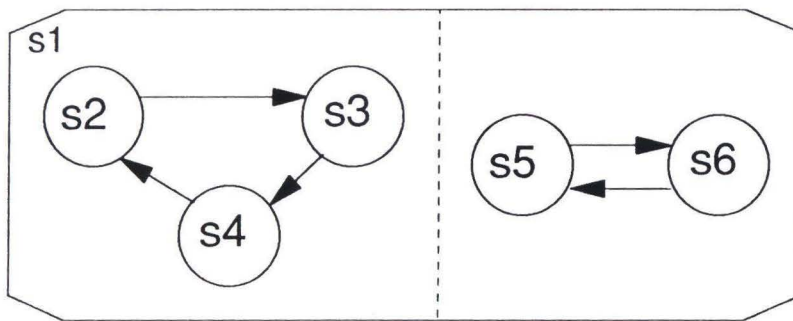


b.

Figure 5.5: Inheritance where a state decomposes into two or more states.



a.



b.

Figure 5.6: Inheritance where an additional set of attributes is included in the subclass.

5.3.1.7 Addition of New Attributes

Additional attributes also could be included in the subclass, as shown in Figure 5.6, which could result in a subclass having additional states that are concurrent with the original states. This is also a very common form of inheritance.

5.3.1.8 Modification of a State

A state can be modified (usually by overriding) for better performance. In this case, the behavioral model does not change. Since the behavior model does not change, subtyping will be maintained.

5.3.2 A Student System Example

To better illustrate how these inheritance cases apply to a real model consider Figure 5.7, an object model for a Student system. Here we see that the class Student has three subclasses that inherit its properties, the classes Foreign_Student, Undergrad, and Grad_Student. Foreign_Student is specialization of the class Student with the additional attribute of Visa_Status. In this model the class Grad_Student is further specialized into two classes, Thesis_Student and MS_Nonthesis. Thesis_Student is further specialized into PhD_Student and MS_Student.

The class Student has the behavioral model shown in Figure 5.8, where the operation *admit* creates a student in the Admitted state. If the student's GPA falls below some limit, the student is placed on academic probation and enters state On-Probation. When the student's GPA rises above the threshold, he is placed back into the admitted state. When the condition program-completed becomes true, the event get-degree occurs and the final state is entered. At any time the student may withdraw. This is the basic behavior of all objects of the class Student.

Next, let's look at the class of Foreign_Student. In Figure 5.9 we see that

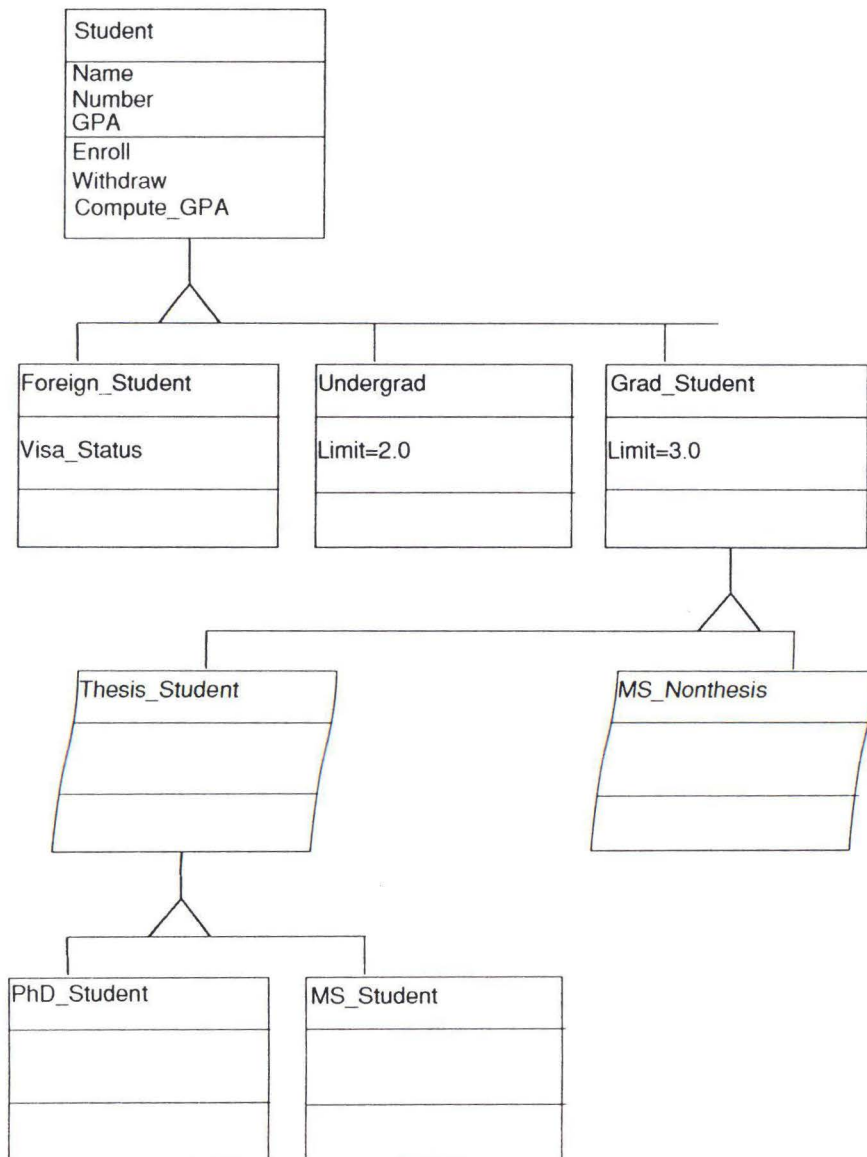


Figure 5.7: Object Model for a Student system.

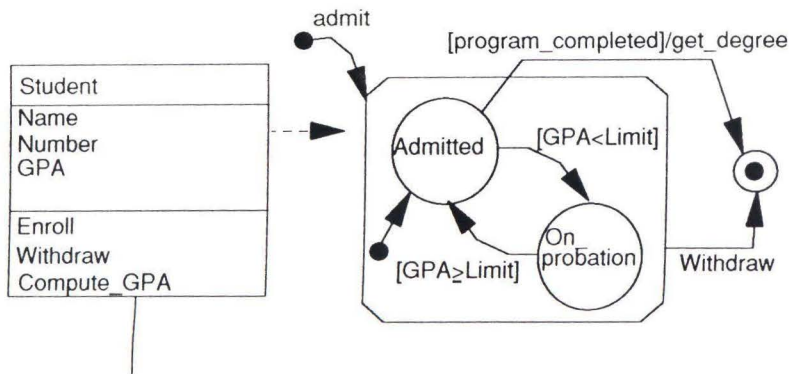


Figure 5.8: Behavior model for the class student.

the object model for the class `Foreign_Student` is specialized by the addition of some new attributes. The behavior model gets the addition of the concurrent state containing the substates of `Visa-OK` and `Residence-Problem`. This is an example of inheritance that causes model concurrency.

For subclass `Undergrad` the behavior model is almost the same as for `Student`. As we can see in Figure 5.10 the only change is that the `Limit` has been strengthened by replacing the variable `Limit` with a specific value. This indicates that a GPA of 2.0 is necessary to keep the student off probation.

In Figure 5.11 we see the behavior model for the class `Grad_Student`. Here the changes include the change of the GPA limit and the state `Admitted` has been decomposed into two states, that reflect the additional need for a graduate student to apply for candidacy. Note that this example, because of the transition out of `Candidate` state, slightly violates subtyping. We can accept this when the semantics

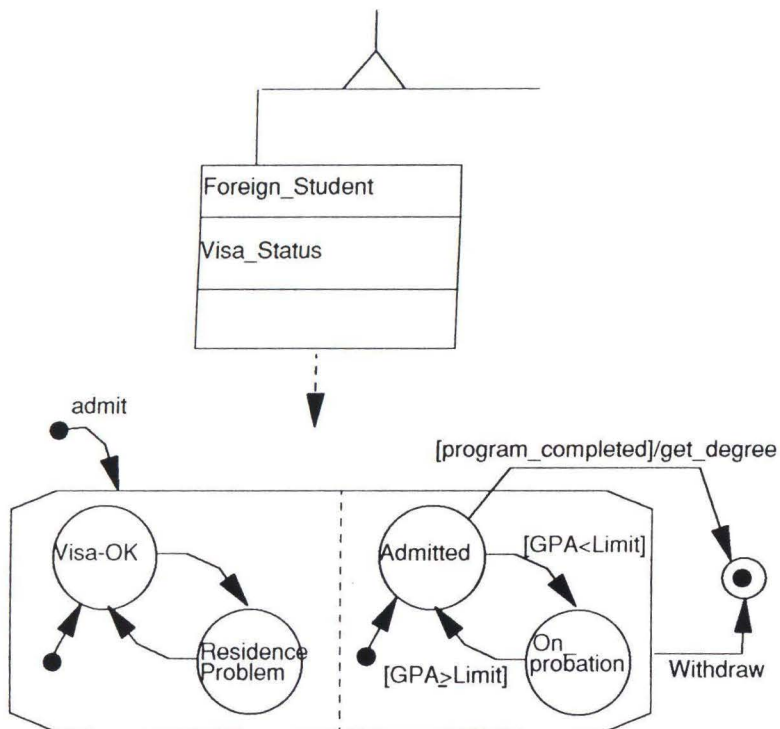


Figure 5.9: Behavior Model for class `Foreign_Student`.

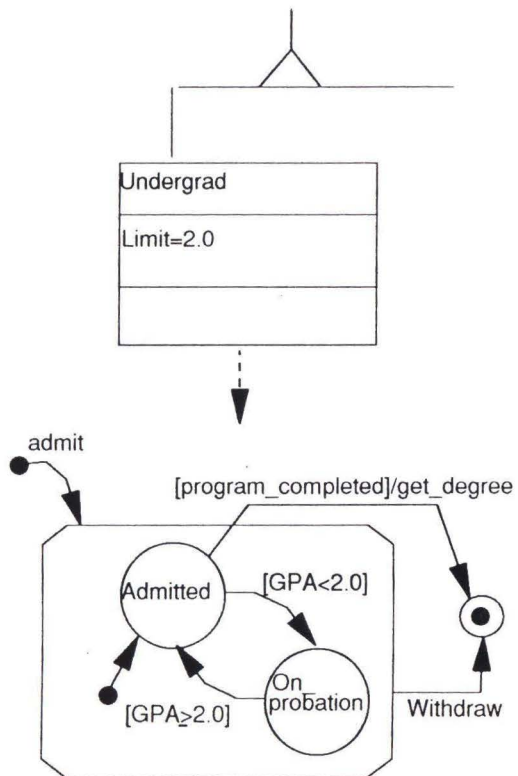


Figure 5.10: Behavior Model for the class Undergrad.

of the specification clearly requires it.

In Figure 5.12 we have the behavior model for the class Thesis_Student. In this case, the state Candidate has been decomposed into two states Cand1 and Preparing_Thesis. Likewise in Figure 5.13, for PhD_Student the state Admit1 has been decomposed into two states. These examples demonstrate the most common ways of inheritance specialization for the behavior model.

5.3.3 Program Specifications

All the models are related through the program specification. It is the program specification that prescribes how the objects are related including their behavioral differences. As an illustration of this point, we refer to our student system example from above. The program specification will include a definition of a student. The behavior of the student also will be described as the actions that can be performed on the object student. For example, the specification might include statements like, "The student can withdraw at any time.", or "If the students GPA falls below the limit, the student will be placed on probation.", or even "The student can not complete the program and get a degree if the student is on probation." From these and other statements, the behavior model of the student in Figure 5.8 was created.

The specification also will describe each specialization of the student object, such as the Grad_Student object. Here the specification describes the graduate

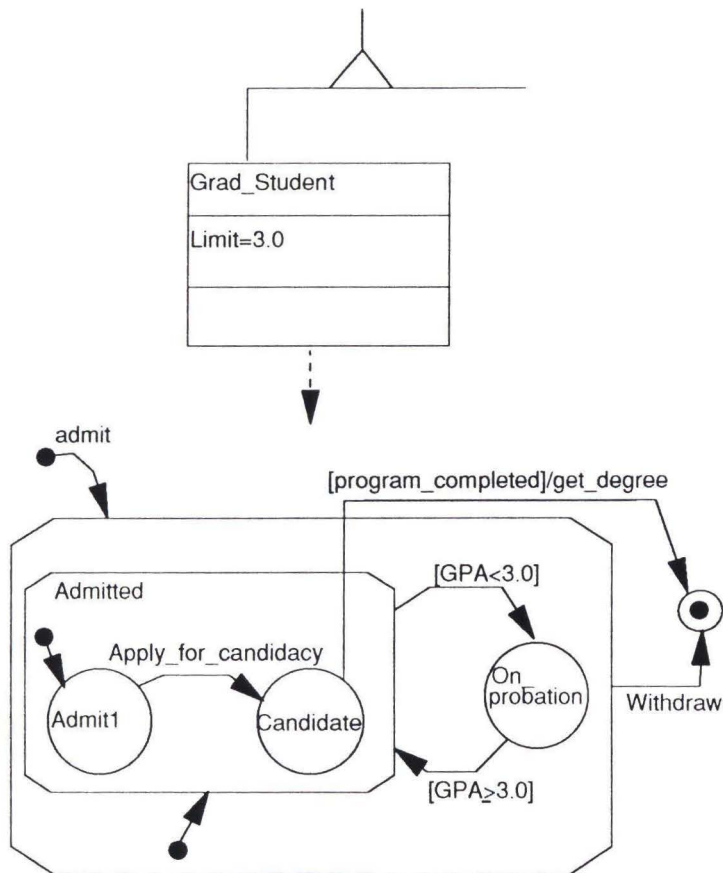


Figure 5.11: Behavior Model for the state **Grad_Student**.

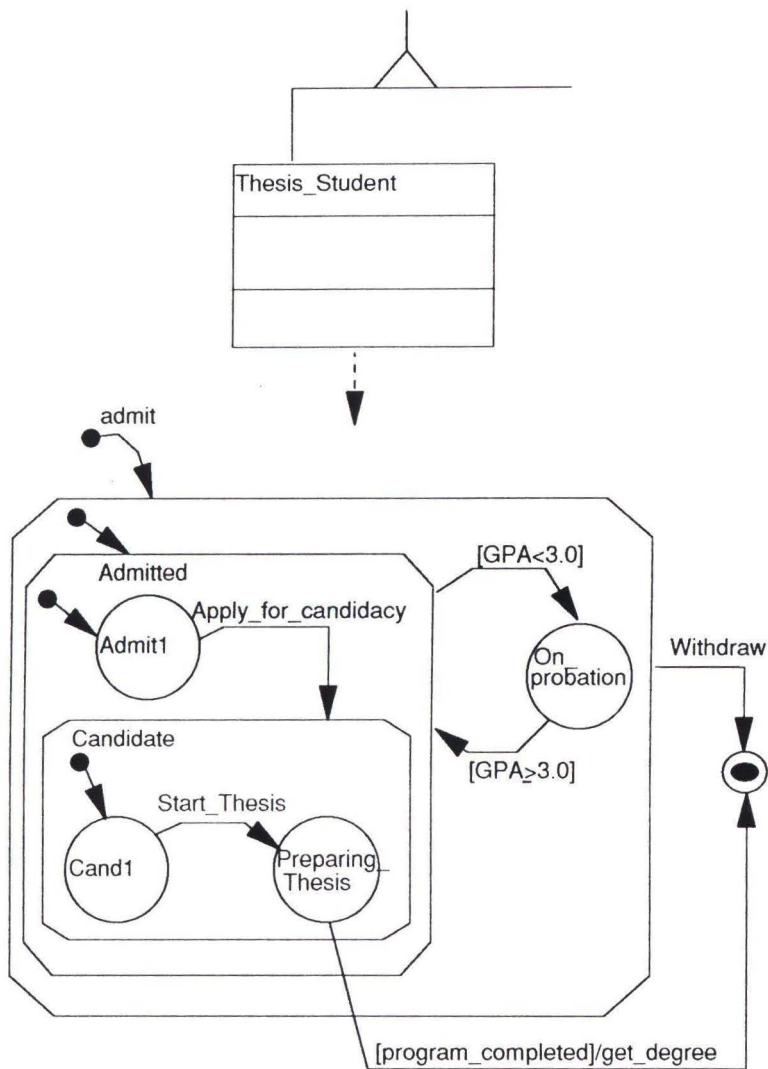


Figure 5.12: Behavior Model for class Thesis_Student.

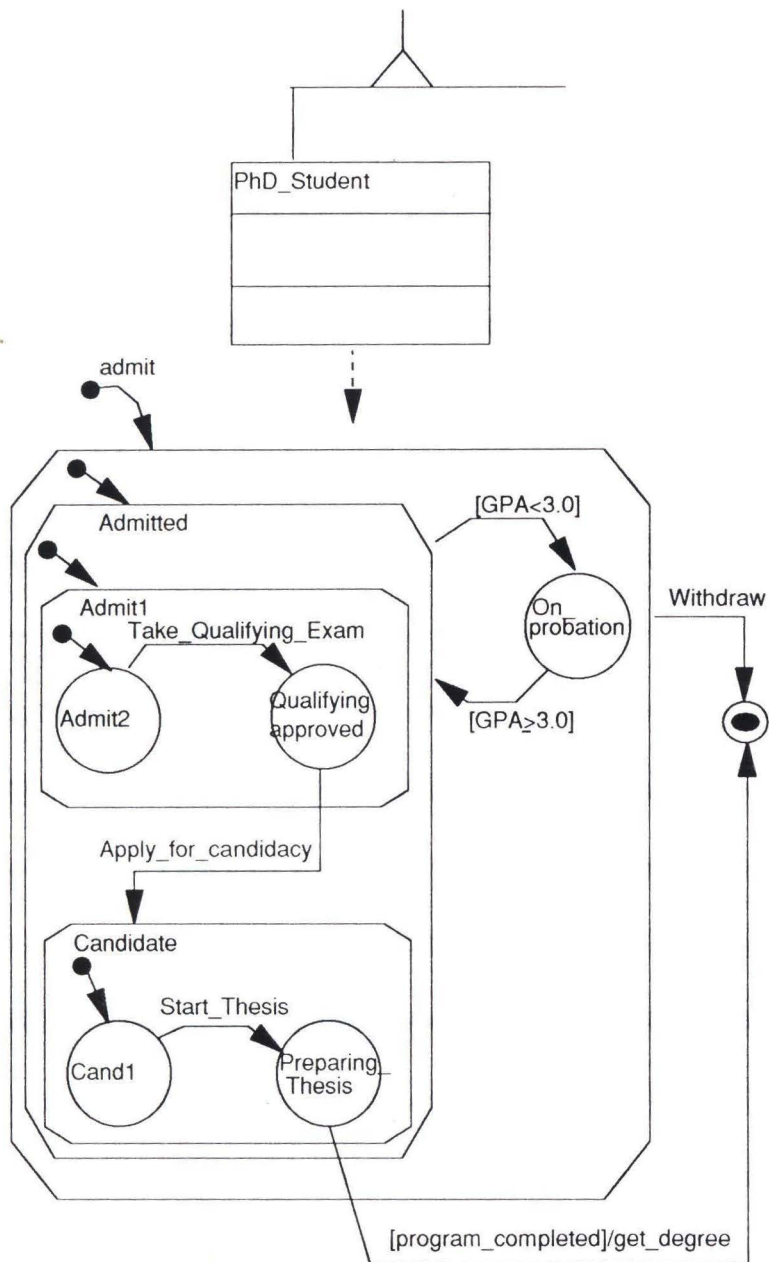


Figure 5.13: Behavior Model for the class PhD_Student.

student's additional behavior of having to apply for candidacy before being allowed to complete the program and get the degree. It is from these descriptions in the specifications that the specialized object and behavior models are obtained. Of course, in many cases, the program specification is ambiguous or incomplete, which may make the decision about substates more complex.

5.3.4 Multiple Inheritance

One area that is not dealt with much is that of multiple inheritance [44]. One reason for this is that multiple inheritance can become complicated. While multiple inheritance can be avoided most of the time, there are occasions where its use can simplify the design greatly.

When multiple inheritance is used, the object model is relatively straightforward. The program specification will show what parts of the each object will be inherited in the new object. However, the behavioral model is not so clear. To demonstrate this we use another example in our student system. Shown in Figure 5.14, our specification states that we can combine the `Foreign_Student` and the `Grad_Student` to create a new class of `Foreign_Grad_Student`.

In multiple inheritance, the behavior model will inherit the parts of the parent model that contain the most specialized aspects. For example, `Foreign_Student` and `Grad_Student` are both subclasses with differing semantic aspects, and, we should inherit the most specialized aspects of each branch. In the case of the

Foreign_Grad_Student, the details that make the Foreign_Student different from Student are inherited. Likewise, the details that made Grad_Student different from Student are inherited. Thus, the new model has both the concurrent residency status states and the states that represent the requirement for applying for candidacy.

5.4 Relationships And State Diagrams

Perhaps the most common object model association is that of relationship. With relationship objects are associated to one another with links that describe how they cooperate together in a meaningful way. For example, the object person can be linked to the object company by the association "*works for*". The multiplicity of an association specifies how many instances of one class are related to a single instance of another.

In general, a relationship implies a coordinated action between two classes and is a path for object communication. While the object model shows the static aspects of a relationship, the behavior model shows the dynamic and temporal aspects of the relationship. Usually two objects in different classes can only communicate when they are in specific states. This is best exposed in the behavior model.

Cook and Daniels [18] showed some aspects of the connection between the object model associations and the behavior model. They observed that associations in the object model may result in conditions on the behavior model transitions. Furthermore they showed how these associations could be given precise mathematical

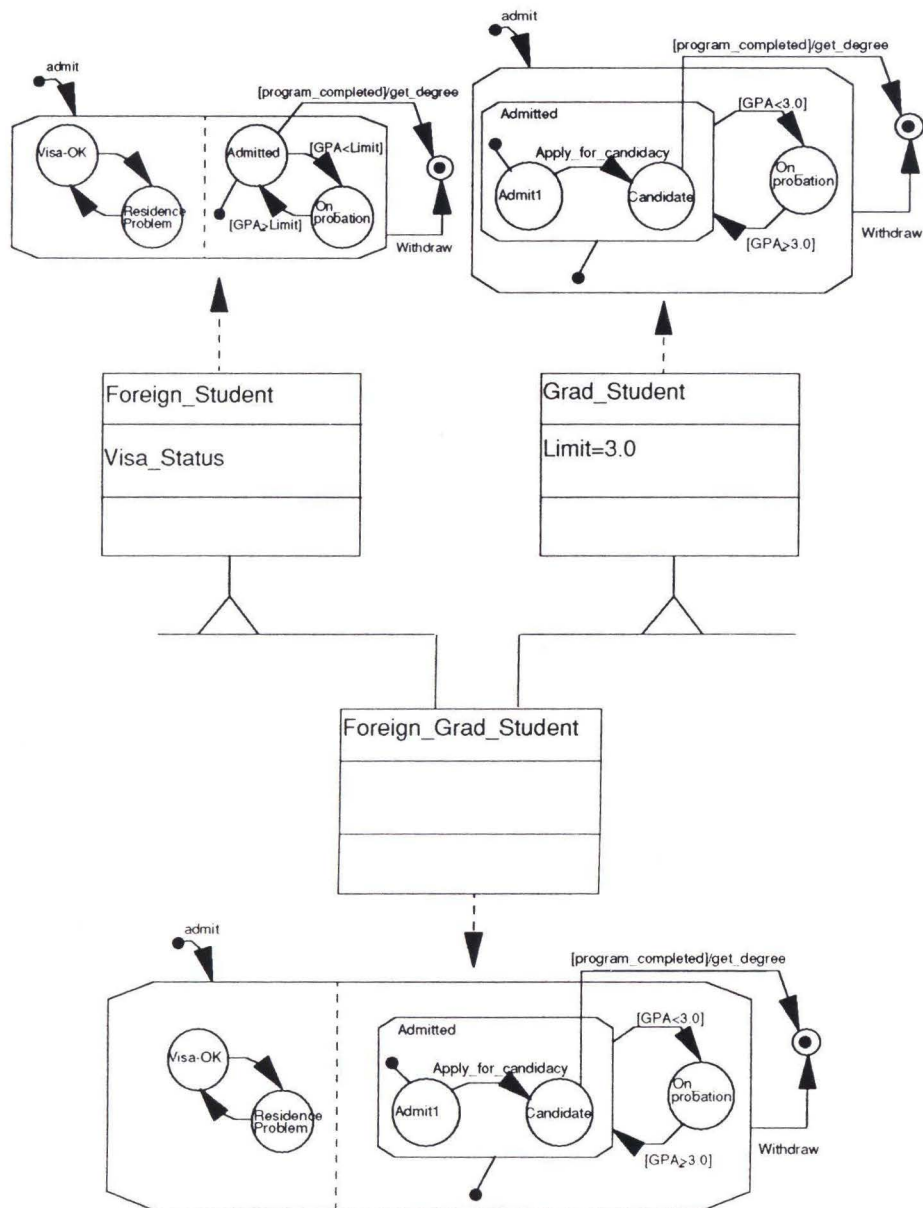


Figure 5.14: Example of Multiple Inheritance

expressions that resulted in accurate behavior models.

Our approach to expressing the temporal aspects of relationship is to introduce a graphical connection in the statechart. (In our statecharts we show these temporal relationships with dashed lines.) These do not represent allowable state changes, just allowable communication paths. This method of showing the relationships can easily augment, and can be augmented by, the mathematical expressions used by Cook and Daniels.

For example, in the object model in Figure 5.15 we see that the class Faculty is related to the class Thesis_Student by the relationship Advises. Furthermore, each Thesis_Student is advised by only one Faculty, but a Faculty can advise several Thesis_Students. However, this is only half of the relationship, because there is still a temporal aspect of this relationship not apparent in the object model. In the behavior model in Figure 5.15, we see that the Thesis_Student can only engage in the communication with the Faculty when Thesis_Student is in the state of Preparing_Thesis. Likewise, the Faculty can only communicate with the Thesis_Student class when the Faculty is in the Research_Activity state. At other times this communication path is not valid and no communication can take place.

5.5 Object and Behavioral Aggregation

Another important concept in the object-oriented methodology is that of aggregation. Aggregation is a form of association where a class is composed of

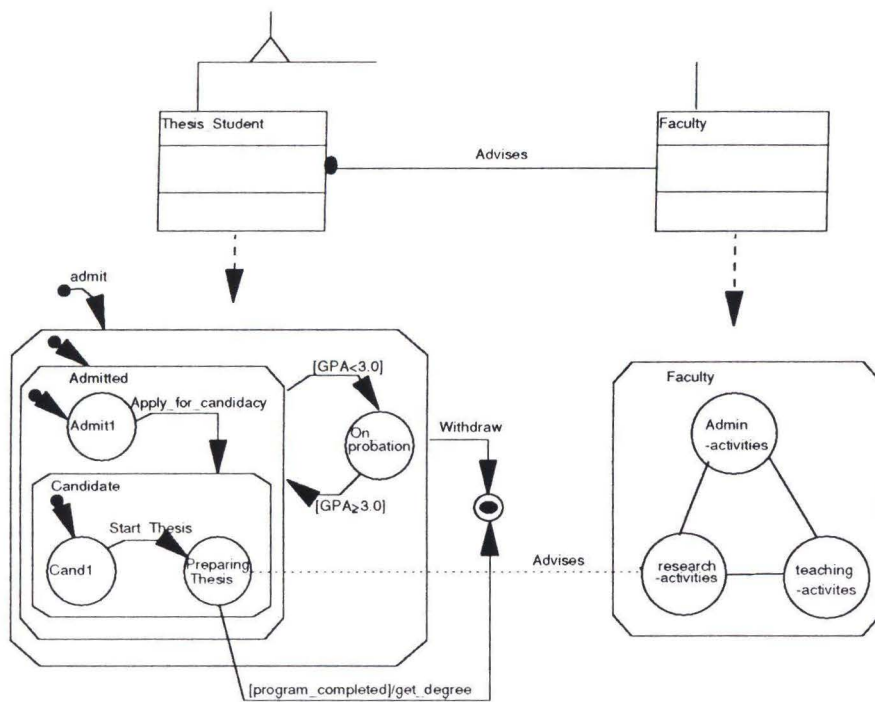


Figure 5.15: Example of relationship in the object model and its temporal nature in the behavior model.

distinct objects in a part of relationship. We think of the subclass as being one of several dissimilar parts of the superclass. We think of the superclass as being an entire assembly composed of the subclass components. When the object model uses aggregation, the corresponding behavioral model is represented by concurrent states. Concurrency can also happen within a given state diagram (intraobject concurrency). This kind of concurrency is not related to aggregation.

A simple example of this can be found in the traffic light controller from Drusinsky [23]. The specification for Drusinsky's traffic light controller is as follows:

- There are two directions, *Main* and *Secondary*, with alternating lights.
- Lights alternate based on a *Timeout* signal read from the *Timeout* variable.
- The initial state (All-y) is for all lights to flash yellow. *Reset* occurring in All-y state starts On-going. *Reset* in On-going returns to All-y.
- A counter counts the cars waiting in the main direction. The counter can sense the difference between cars and trucks.
- If main is red and four or more cars or one or more cars followed by a truck are waiting in the main direction, a hidden camera shoots the intersection.

Part of the object model for this traffic light controller is shown in Figure 5.16 and the behavior model is shown in Figure 5.17. In these figures, we see that the aggregation causes the behavior model to have a more complex nature. Still the

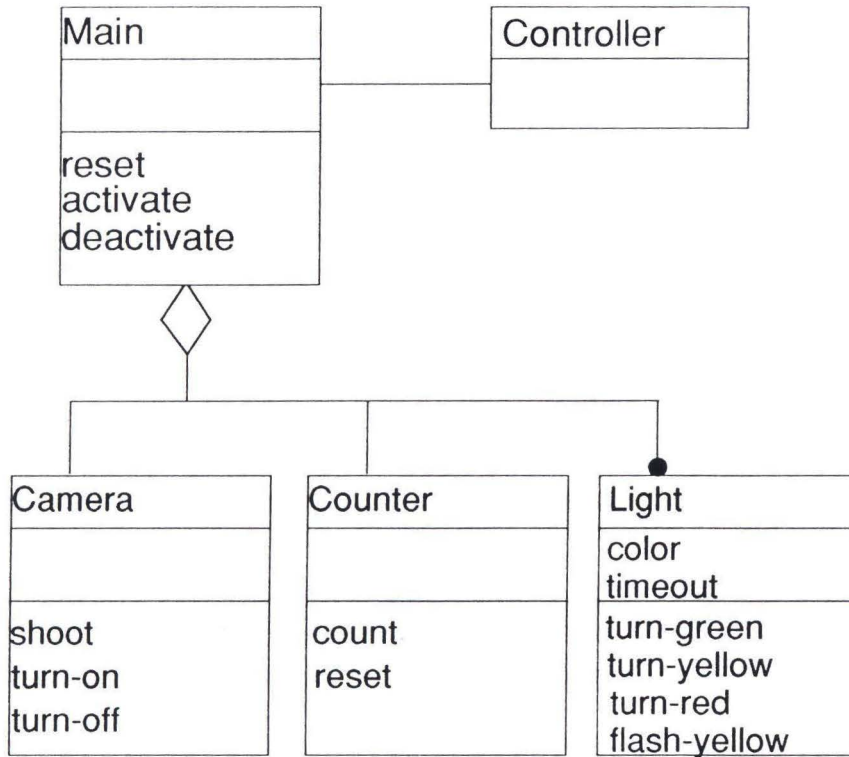


Figure 5.16: Object model for traffic light controller

behavior model exhibits concurrent behavior in the camera and counter components, although in this case their states only make sense as nested states within specific states of the aggregate (camera and counter only work when the main light is red).

Many times, operations in aggregations must be coordinated. The example in Figure 5.18 shows a document that is composed of sections and each section is composed of subsections. For legal reasons the document is created in the original state. However, if any modification is made to the document, this must be considered

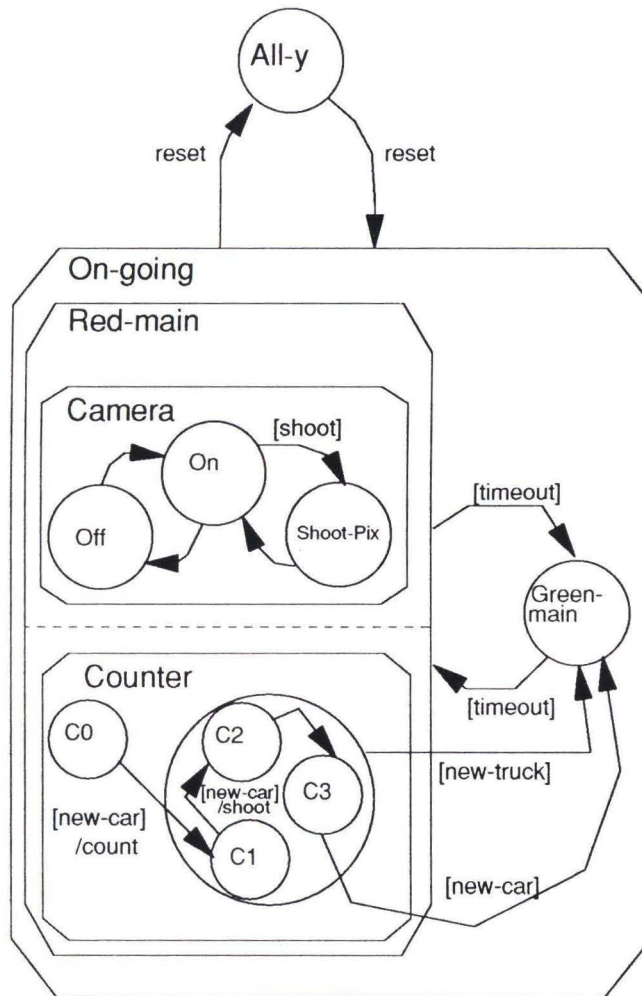


Figure 5.17: Behavior model for traffic light controller

as modified.

There are two interesting effects that we wish to explore for this example. The first is that of modifying a section or subsection. If a section or subsection is modified, its state must change from original to modified. If it was a subsection that was modified then the corresponding section must also change state. Likewise, when any section changes state the entire document must change into modified state. This reflects the requirement that any change to any section or subsection will result in the document being marked as changed.

The second effect is where the action of deleting a document or section results in the deleting of all the lower level parts. Deleting a section causes all the subsections to be deleted. It would not make sense for a document to be deleted but for the sections to remain intact. This effect is related to propagation of operations in aggregations [51].

The modeling of this feature is shown in Figure 5.18 as dotted arrows connecting the state transitions. This is similar to the way we connected states for the temporal part of associations earlier. In this case, the models could show this relation by setting conditions and using the statechart communication mechanism to perform the changes at the other levels. However, it is clear that these activities are needed when the new connections are added to the behavioral model.

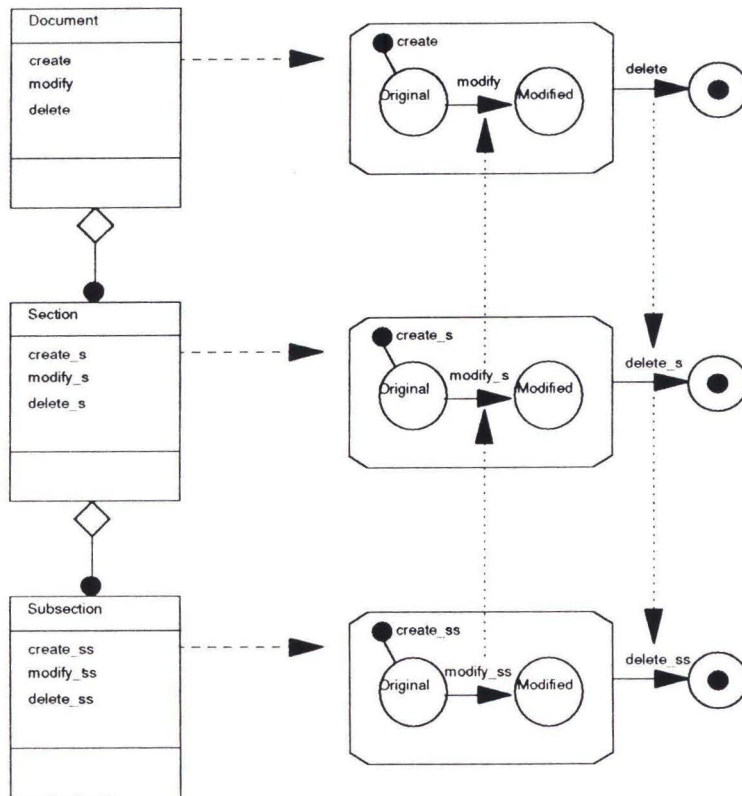


Figure 5.18: Coordinating Aggregation

5.6 Chapter Summary

As the demands for complex software increases, so will the need for tools and methodologies to support this software. The object-oriented methodology is one of the most promising for increasing the ability to create, maintain, and understand complex software. However, the relationship of the object model and the behavior model in object oriented methodologies must be examined.

Having an object model implies that there is a corresponding behavior model. When the object model is related to other objects by the use of association, aggregation, and generalization there are corresponding relationships with the behavioral model. The simplest object association is that of relationship. The resulting behavioral relationship is implemented by the use of conditions on the state transitions. Association implies a communication path between objects. The behavioral model further strengthens this concept by exposing any temporal relationships in this communication path. Thus, we can clearly show when objects must be in specific states to communicate.

The aggregation relationship is a more complex form of association. The simplest behavioral model associated with object model aggregation is that of concurrency. In this case, object model aggregation results in behavior model concurrency. However, behavior model concurrency is not always the result of object model aggregation. In other cases, the behavioral model is more complex, but concurrency is always involved. Also, at times the need for coordinated behavior is necessary.

Object model generalization results in the most complicated behavior model. When the object model of a superclass is inherited, the behavior model of a subclass can be one where a new state or a new transition is added. Also, it could result in a model where a state is decomposed into two or more new states or the model could have a transition changed where a precondition is weakened, or where a post-condition or invariant is strengthened. Finally, a state can be modified for better performance.

Chapter 6

THE EFFECT OF DEADLINES ON OBJECT AND BEHAVIOR MODELS

6.1 Introduction

This chapter closely mirrors the previous chapter. In it we will reexamine all the issues discussed previously, except that now we will consider how real-time deadlines would be reflected in the object and behavior models.

First, we will consider how deadlines are introduced into the models. The deadlines are represented as attributes or invariants in the object model and as time constraints on the transitions or states in the behavioral model. Since the behavioral model describes the temporal nature of the system and deadlines are concerned with the temporal aspects of the system, it is apparent that the behavioral model needs to be examined closely in a real-time system.

We then examine how the behavior model with deadlines can be changed during object model inheritance. We will again look at subtyping inheritance since it is the more desired form of inheritance. But this time our concern is how deadlines affect our models.

In the previous chapter we saw that there are eight ways that the behavior model can be changed after object model inheritance, and still maintain subtyping. Each of these is examined in detail. These all fall into three major categories, refinement of transitions, refinement of states, or refinement of attributes. We concentrate on the differences necessary for incorporating deadlines.

After that, we examine how deadlines are incorporated into the object model association of relationship and how this affects the behavior model. Deadlines can affect the periods when the communication paths between objects are valid. This can be easily shown in the behavior model connection that defines the temporal nature of the communication path.

Finally, we examine how deadlines affect the object model aggregation and the corresponding behavior models. Deadlines can be incorporated into any part of an aggregation. Since the resultant behavior model is concurrent by nature, overlapping periods may result.

Because even simple real-time systems with soft deadlines are highly complex when compared to non-real-time software, the object-oriented techniques are appealing for these systems. However, many problems need to be addressed before object-oriented methodologies are routinely used for real-time systems. There is little agreement on how the deadlines should be introduced into the object-oriented models.

In this chapter, we look at real-time deadlines in Section 6.2. Second, we look

at object and behavior models with deadlines in Section 6.3. Then, we look at object and behavioral inheritance, including multiple inheritance, with deadlines in Section 6.4. Then, we examine object and behavioral association with deadlines in Section 6.5, followed by an exploration of how these models are affected by aggregation in Section 6.6. Finally, we present some conclusions in Section 6.7.

6.2 Deadlines

To use the object-oriented methodology in real-time environments, we must introduce the concept of deadlines into the methodology. Deadlines are time factors that constrain the system during the performance of certain functions and are classified as hard, soft, or firm. Hard deadlines are deadlines which, if not met, will result in catastrophic failure of the system. Hard deadlines are often used in life critical systems where a catastrophic failure results in loss of life. Hard deadlines must be met.

Soft deadlines are deadlines where not meeting the deadline simply means that the results are invalid and the procedure will need to be repeated before going on. This could be as simple as a timeout in a GUI waiting for user input.

Firm deadlines are similar to hard deadlines except failure to meet a firm deadline is only serious, not catastrophic. In this charter, we will not make a distinction between hard and soft deadlines as the distinction is not germane to the problems we will be discussing. Our examples only show generic deadlines to keep

them simple.

Deadlines can manifest themselves in two ways. First, they can put constraints on a complete path through a use case. Second, they can put constraints on the execution time of an operation. We will consider only the later in this chapter. The first case will be examined later.

Deadlines can also be represented as conditions on transitions. When viewed this way they are considered as controls that determine when an event must occur for a transition to be taken. Equivalently deadlines can be represented as a timer on a state. When it is more convenient to illustrate a point we will use timed states.

6.3 Deadlines in Object Models and Behavior Models

In the object-oriented methodology, the object model contains the conceptual description of an object while the behavioral model contains the dynamic and temporal description of an object. This distinction between the two models allows us to create complete descriptions of encapsulated objects.

The behavior models used in this chapter are statecharts [29]. Specifically, we use the basic statechart constructs of *hierarchy*, *broadcast communication*, and *concurrency*. Hierarchy is used to simplify the models and reduce the number of transitions that must be shown. Broadcast communication is assumed to make the models easier to construct. Concurrency allows the construction of real world models where multiple things happen at the same time. For this work, we do not

need to use any of the advanced features of Harel's statecharts, such as the history entry point or states that have multiple modes of concurrency. Most of the time deadlines will be shown as conditions on the state transitions, where the conditions appear as a minimum and maximum time in the style of [16] and [65]. Occasionally, we will use the equivalent style of a timed state, where the deadline will appear as a single timeout value for the state, similar to [11]. In all cases, we will omit the units (year, day, hour, minute, etc.) to keep our examples generic.

6.4 Deadline Inheritance

The concept that objects are related by inheritance and generalization is important in object-oriented technology. A subclass inherits when it takes the properties, including the deadlines, of the superclass and specializes by incorporating features that make it unique. Inheritance is valuable when a class is being reused from a previous problem or from a software library. In generalization, objects have their similarities, which may include deadlines, factored out into a superclass. This leaves each subclass object describing only what is different from the common properties of the superclass.

In the previous chapter, inheritance was viewed as *subtyping* or *redefining of methods*. Now a more detailed evaluation of the relationship of classes and subclasses is called for. The most prevalent type of inheritance is that of subtyping. Rumbaugh defines subtyping with the '*is a*' relationship [54, 52]. More precisely, B is a subclass

of A, if B is an A, and B could be used anywhere an A is expected. Here the behavior model of a subclass will always be a subtype if it is only extended with new attributes or methods. Furthermore, he argues that creating a subtype by restriction (adding a set of restrictions to the base class to limit the subclass) is difficult to implement. The 'is a' relationship is fairly easy to follow, and as Rumbaugh points out, if followed carefully will keep you out of trouble. Coleman et al. also uses the 'is a' definition of subtyping [16]. They present some rules that would allow subtyping by restriction, but shows examples where after following the rules the behavior model breaks down and violates subtyping.

Selic argues differently, that behavioral equivalence can not be guaranteed between parent classes and subclasses [57]. This appears to be a claim for *implementation inheritance*, which Coleman treats lightly and Rumbaugh warns sternly against using. We will not consider this type of inheritance again in this chapter.

Sourrouille argues that subtyping is neither necessary nor sufficient to ensure that the behavior of a subclass can be substituted for a parent class [61]. He defines behavior substitutability and shows that if an object's behavior is behavior substitutable, then any sequence of events requested accepted by an object will be accepted by the substitute object.

6.4.1 Inheritance Behavior

We consider the effects of deadlines on the behavior model (timed transitions and timed states) for maintaining the subtyping and the behavior substitutability relationships. We consider the same eight different ways that the behavior model can be affected during inheritance shown in the previous chapter.

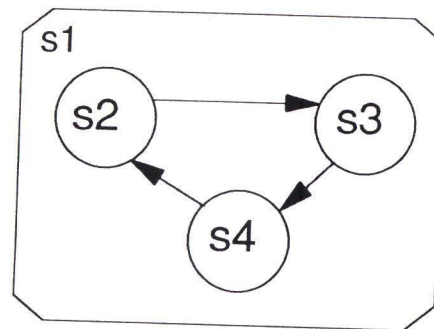
6.4.1.1 Addition of an extra transition

The addition of a transition is fairly straightforward. It is easy to see that when a transition is added to the behavior model of the child class, that it still models the behavior of the parent class, the extra transition just adds new behaviors. An example of this type of inheritance is shown in Figure 6.1. This change maintains both the properties of subtyping and of behavior substitutability. Furthermore, it would not matter which transitions in the diagram were timed or untimed, or if the added transitions are timed or untimed. In either case, both the subtyping and behavior substitutability properties would still hold.

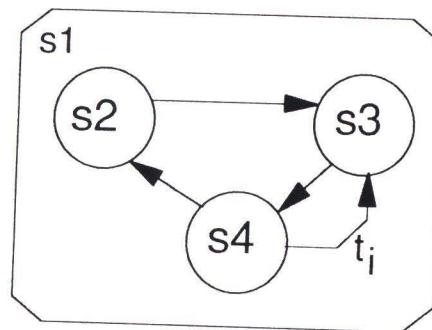
There is one exception, if lambda transitions are allowed. An added transition fired by a timed lambda condition could violate subtyping. This type of firing condition would have to be added carefully during inheritance.

6.4.1.2 Retargeting and splitting of a transition.

Retargeting a transition changes the transition to a new internal substate of the original state. This is often used in conjunction with the refinement of a state



a. Parent Object



b. Child Object

Figure 6.1: Inheritance adding a transition to the behavior model.

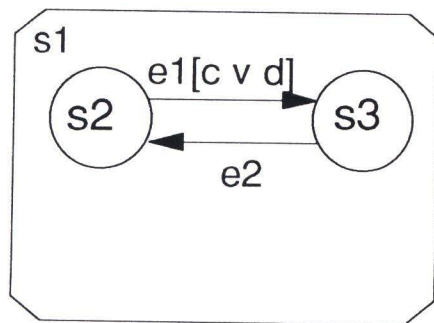
into two or more states below. Here we modify the transition to point to a new specific internal state of the original state.

At the same time, a transition can be split into two or more transitions.

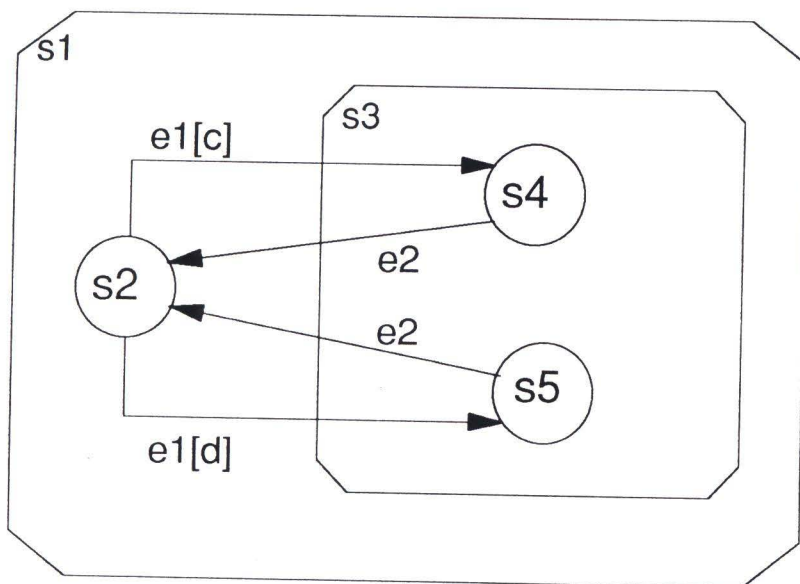
These new transitions can be controlled by different conditions, but the combination of conditions needs to logically OR'ed into the original conditions. There are two ways splitting is used: 1) to go to different internal states based on conditions, or 2) to emanate from different internal states, and generating different events when triggered. For example, in Figure 6.2 event *e1* is split depending on conditions *c* or *d* and retargeted to state 4 and state 5. Similarly event *e2* can trigger different actions depending on the substate of state 3 that was active when the event occurs.

In this case, deadlines can be added as conditions on the transitions. It is easy to see that an untimed transition can be inherited and split so that if the event occurs in the first 5 seconds one path is taken, but after the 5 seconds another path is taken. In our example this would mean that *c* is the time interval $[0,5]$ and *d* is the interval $[5,\infty]$. For splitting to work with timing an further requirement is necessary. The new requirement is that the new conditions not only logically OR into the original condition, but must also logically AND into the nothing. Overlapping conditions are not allowed.

Therefore, even with deadlines, subtyping will be maintained by the inherited behavior. Behavior substitutability will also be maintained as any sequence of timed event traces accepted in Figure 6.2a, will be accepted in Figure 6.2b.



a.



b.

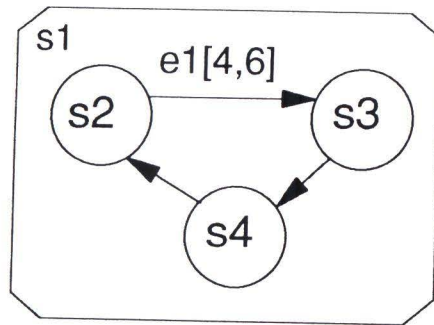
Figure 6.2: Inheritance where a transition is split and retargeted.

6.4.1.3 Weakening of a precondition of a transition

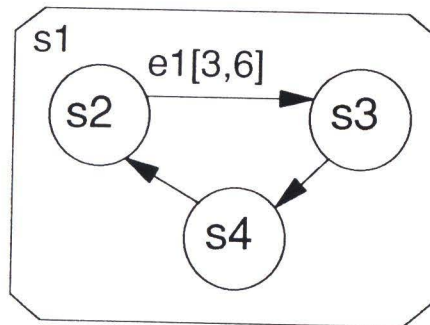
Preconditions are the conditions that the caller agrees to satisfy before invoking a method. To maintain the subtyping requirement, a child class can only weaken a precondition of a transition. In this case, the child class allows the transition to occur more often (Figure 6.3). Since timing can be considered a condition on the transition, then weakening the precondition would mean to change the timing to allow the transition to fire over a larger time range. This can be done by either decreasing the lower time bound, increasing the upper time bound, or both. Subtyping then holds because if a transition time bound of, for example, $[4,6]$ is weakened to $[3,7]$ then an event trace (5, GetValue) would still result in the same state transition being taken. Weakening a precondition that contains timing will not violate subtyping. Behavior substitutability will also not be violated by weakening a precondition.

6.4.1.4 Strengthening of a postcondition of a transition

Postconditions are the conditions that the transition activity agrees to satisfy before completing. A deadline used as a postcondition, would be the time limits that the activity is completed by. For example, a postcondition of $[3,7]$ means that the activity will complete in between three and seven seconds. This can be strengthened to $[4,6]$ in the inherited behavior model and subtyping will be preserved. Behavior substitutability will also be preserved when deadlines are strengthened in the



a.



b.

Figure 6.3: Inheritance with weakening of a timed precondition.

postcondition.

6.4.1.5 Strengthening of an invariant relationship

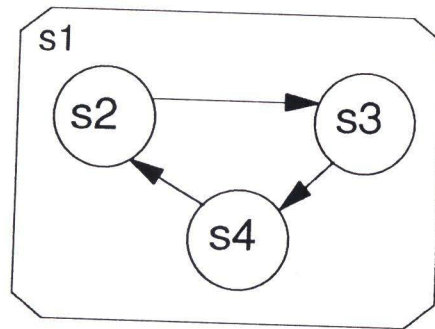
An invariant relationship can be strengthened and the subtyping requirements will still hold. An invariant relationship is an assertion that is always true for a class. This could be something as straightforward as the fact that the attribute *milage-driven* in a class *Truck* can never decrease. Invariants do not seem to have any interpretation as deadlines.

6.4.1.6 Addition of new attributes

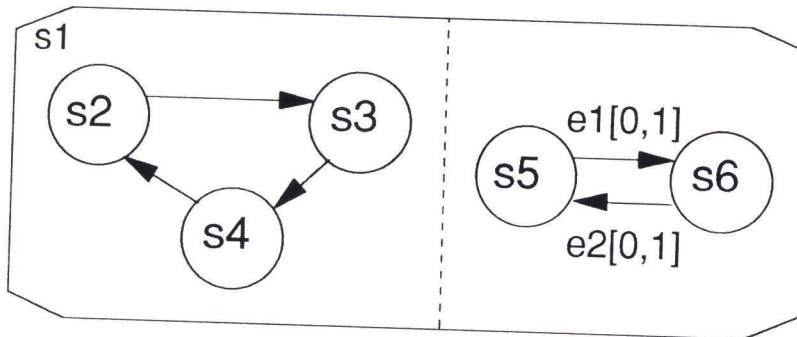
Additional attributes can be included in the subclass which could result in a subclass having additional states that are concurrent with the original states, as shown in Figure 6.4. Here timing can be included as a condition on any of the transitions of the new states. Any deadlines on the pre-existing states remain unchanged. By definition, this satisfies the subtyping requirements and the behavior substitutability requirements.

6.4.1.7 Modification of a state

A state can be modified for better performance. In this case the behavioral model does not change. Overriding some of the attributes or operations of the parent class implies replacing them with different implementations for the subclass. Modifying a state is a way to override the deadlines of an object, resulting in new



a.



b.

Figure 6.4: Inheritance where an additional set of attributes is included.

objects with the same behavior but different deadlines. This may be convenient in real-time systems, where we may override an operation with different versions of the operation having different execution times.

6.4.2 A Student System Example

To better illustrate how these inheritance cases apply to a real model with deadlines, let us reconsider the student system example of the previous chapter, but now with deadlines added (Figure 6.5).

In this example, specific deadlines are present in the class `Grad_Student` and in the class `Foreign_Student`. The `Grad_Student` has a deadline of seven years overall to finish and a deadline of six years to apply for candidacy. The `Foreign_Student` class has a deadline of two years residency before the visa expires and the state Residence Problem is entered. While these are almost trivial deadlines, they will serve to illustrate the ideas that we want to express.

The class `Student` has the behavioral model shown in Figure 6.6, where the operation `admit` creates a student in the `Admitted` state. If the student's GPA falls below some limit the student is placed on academic probation and enters state `On-Probation`. When the student's GPA rises above the threshold he is placed back into the `admitted` state. When the condition `program-completed` becomes true, the event `get-degree` occurs and the final state is entered. At any time the student may withdraw. This is the behavior of all objects of the class `Student`.

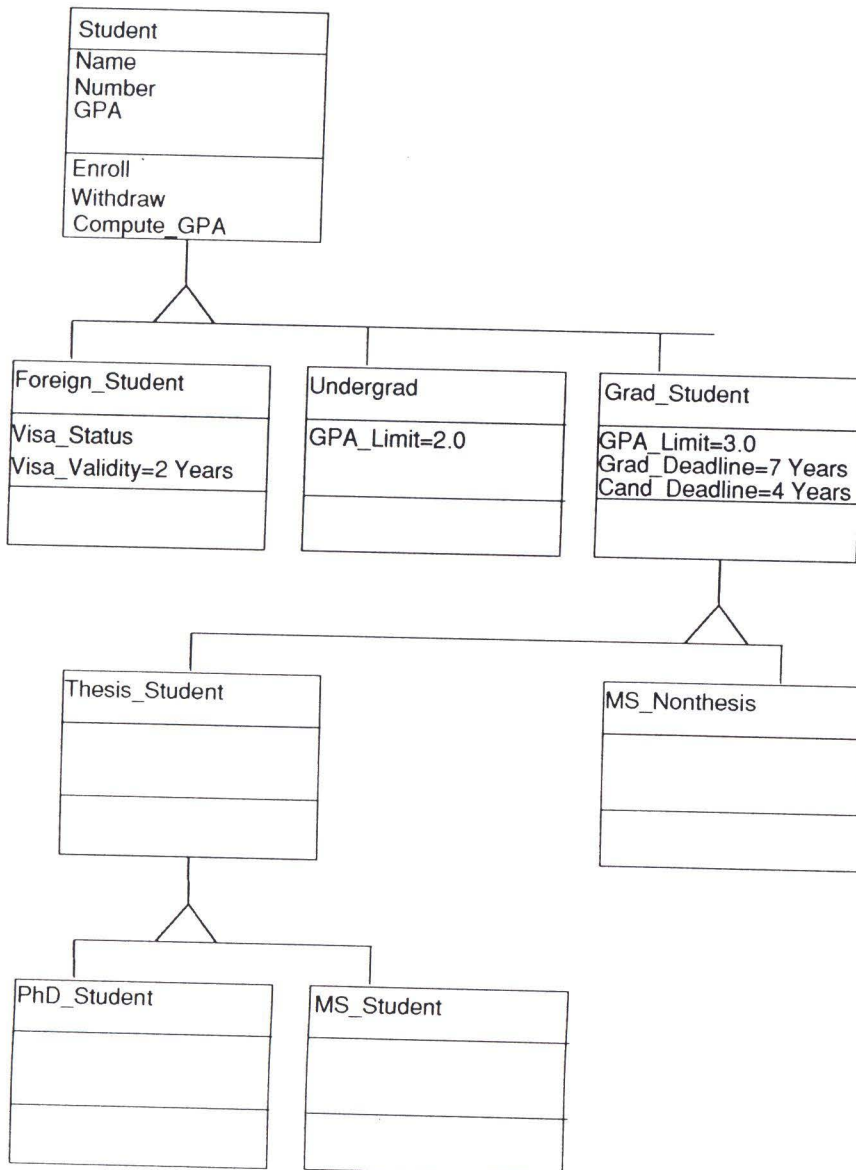


Figure 6.5: Object Model for the class Student.

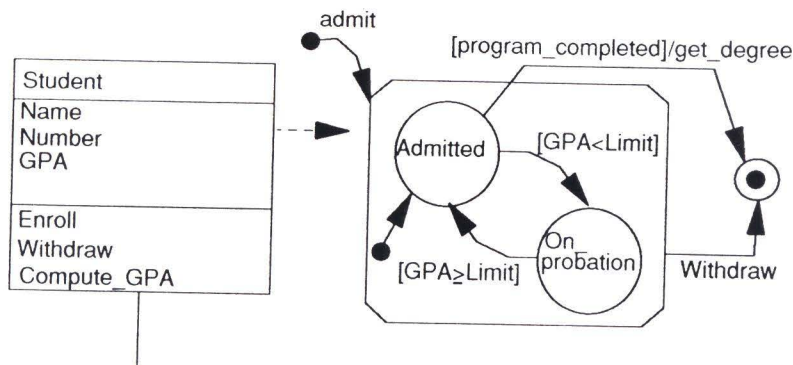


Figure 6.6: Behavior model for the class student.

Next let's look at the class `Foreign_Student`. In Figure 6.7 we see that the object model for the class `Foreign_Student` is specialized by the addition of some new attributes which contain another type of deadline. The behavior model then gets the addition of the concurrent state containing the substates of `Visa-OK` and `Residence-Problem`. `Visa-OK` is shown as a timed state (the 2 in brackets). In this case, after two years, the `Visa-OK` state will timeout and force a transition to the `Residence_Problem` state. When the `VisaRenew` event occurs, the transition back to `Visa-OK` state resets the timer for another two years.

In Figure 6.8 we see the behavior model for the class `Grad_Student`. Here the changes include the change of the GPA limit and the state `Admitted` has been decomposed into two states, that reflect the additional need for a graduate student to apply for candidacy. Also, deadlines have been added to the `Apply_for_candidacy` transition and the `program_completed` transition. These are both timed transitions

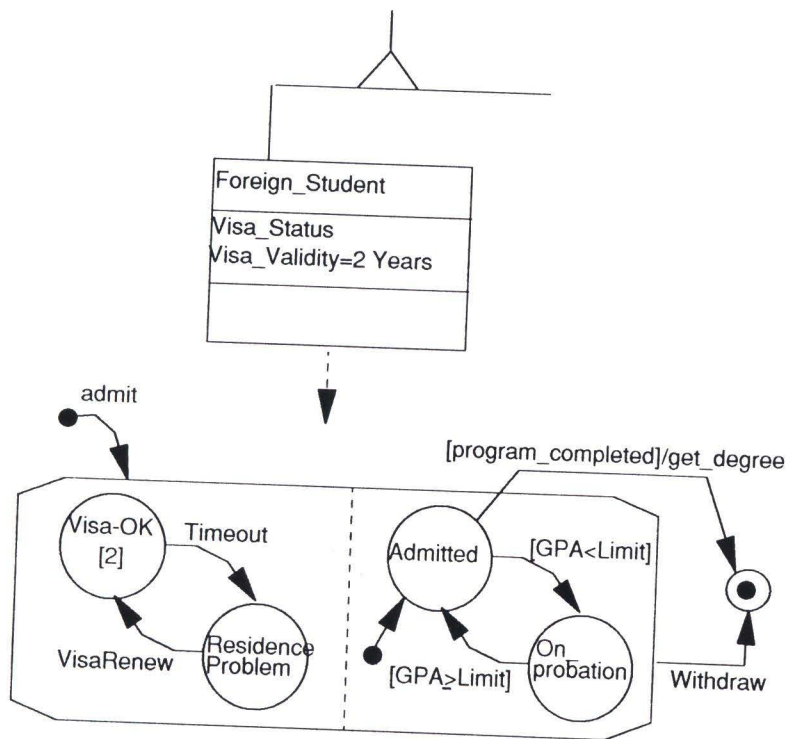


Figure 6.7: Behavior Model for class `Foreign_Student`.

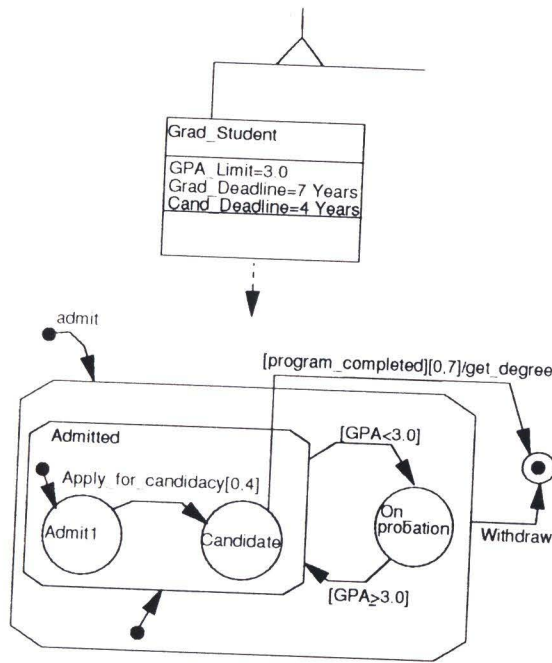


Figure 6.8: Behavior Model for the class Grad_Student.

that can only be taken during the specified intervals.

6.4.3 Program Specifications

All the models are related through the program specification, that prescribes how the objects are related, including their behavioral differences and any required task deadlines. As an illustration of this point, we refer to the student system example above. Its program specification will include a definition of a student. The behavior of the student will be described as the actions that can be performed on the object student. For example, the specification might include statements like, “The student can withdraw at any time”, or “If the student’s GPA falls below

the limit the student will be placed on probation”, or even “The student can not complete the program and get a degree if the student is on probation” From these and other statements the behavior model of the student in Figure 6.6 was created. The specification should explicitly express the deadlines for the system actions.

The specification also describes each specialization of the student object, for example the Grad_Student object. Here the specification describes the graduate student’s additional behavior of having to apply for candidacy before being allowed to complete the program and get the degree. The specification will also describe the deadlines for the tasks of the Grad_Student object.

6.4.4 Multiple Inheritance

In our previous work, we showed how multiple inheritance could be handled in the behavioral model [68]. We concluded that there are times that multiple inheritance will greatly simplify a design. Now, however, we are interested in how deadlines affect multiple inheritance. Let’s observe how this works by looking at an example of multiple inheritance where each object’s behavioral model contains deadlines. Our specification states that we can combine the Foreign_Student and the Grad_Student to create a new class of Foreign_Grad_Student. This is shown in Figure 6.9. Both Foreign_Student and Grad_Student contain deadlines.

In multiple inheritance, the behavior model will inherit the parts of the parent model that contain the most specialized aspects. For example, Foreign_Student

and Grad_Student are both subclasses with differing semantic aspects. As such, we should inherit the most specialized aspects of each branch. Thus, in the case of the Foreign_Grad_Student, the details that make the Foreign_Student different from Student, especially the deadlines, are inherited. Likewise, the details that made Grad_Student different from Student are inherited. Thus, the new model has both the concurrent residency status states with its associated deadlines and the states that represent the requirement for applying for candidacy with its associated deadlines. In this case, there is no conflict in the deadlines so subtyping and request substitutability are both preserved.

Through careful design, deadline clashes can be avoided when multiple inheritance is used. In particular, good design practices should be employed whenever the multiple inheritance comes from classes which are different types of specializations.

6.5 Relationships And State Diagrams

Perhaps the most common object model association is that of relationship. In general, a relationship implies a coordinated action between two classes and while the object model shows the static aspects of a relationship, the behavior model shows the dynamic and temporal aspects of the relationship. Usually two objects in different classes can only communicate when they are in specific states. This is best exposed in the behavior model.

In our previous work, we examined the temporal aspects of relationship in

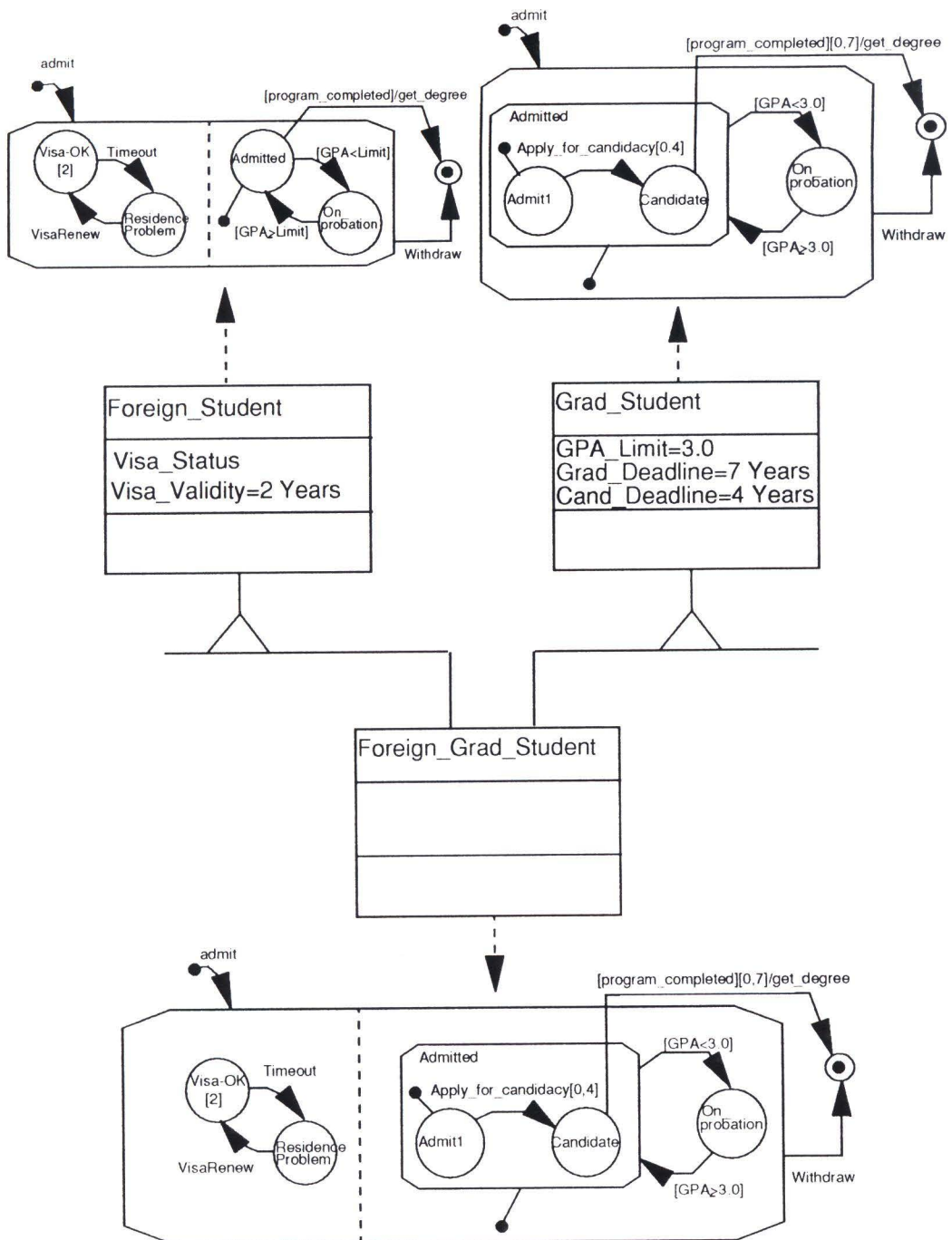


Figure 6.9: Example of Multiple Inheritance..

the behavioral model [68]. We introduced a graphical notation into the statechart to show the temporal behavior. Now we want to look at how deadlines affect this relation.

We need to consider the effects when one or more of the communicating states has a deadline. Lets consider the example of a Thesis_Student being advised by a faculty member. An example of the object and behavior models is shown below in Figure 6.10. In the object model of this example, we can see that the class Faculty is related to the class Thesis_Student by the relation advises. Furthermore, each Thesis_Student is advised by only one Faculty, but a Faculty can advise several Thesis_Students. This represents the static half of the relationship. The temporal aspect of this relationship is apparent in the behavioral model. Here we see that the Thesis_Student can only engage in the communication with the Faculty when Thesis_Student is in the state of Preparing_Thesis. Likewise, the Faculty can only communicate with the Thesis_Student class when the Faculty is in the Research_Activity state. At other times this communication path is not valid and no communication can take place.

There are several ways deadlines could be introduced into this model. First, either communicating state could have a deadline. In this case, these deadlines would represent the times that the states were available for communicating, or the length of time a communication path could valid. For instance, in this example, the student could have a time interval that causes him to be unavailable during exam

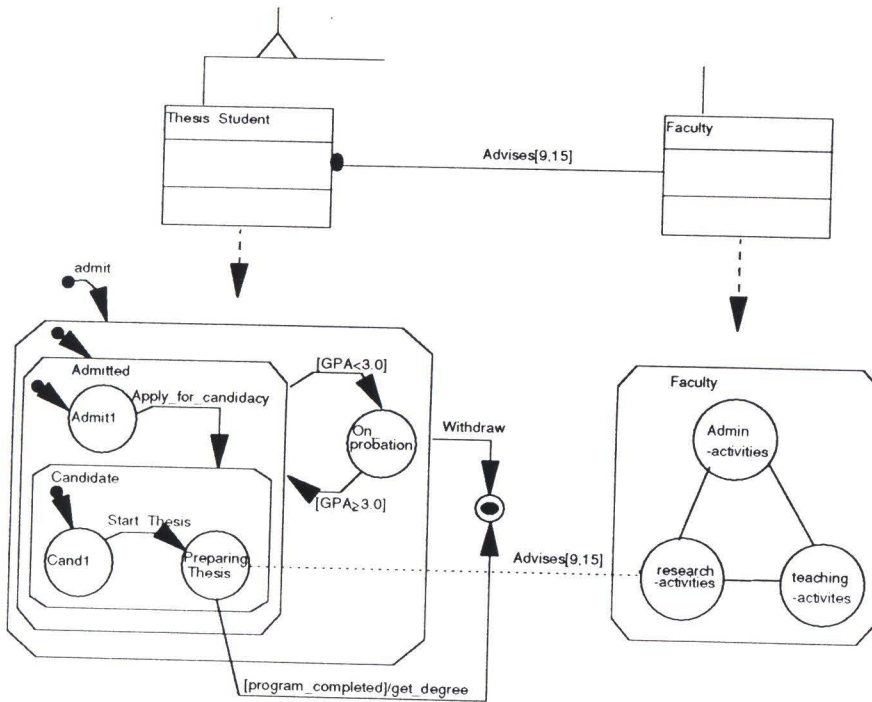


Figure 6.10: Example of relationship in the object model and relationship's temporal nature in the behavior model when deadlines are present.

week. Or the faculty could be restricted to advising the student only during preset office hours. Another case would be that the advisor and student could only meet for an hour at a time.

All these deadlines can be mapped into a timing restriction on the communication path as shown in Figure 6.10. Even if the communicating states have their own deadlines, these will be represented in the communication path timing. Thus the communicating path timings are a summary of other timings in the system that affect the communication path.

6.6 Object and Behavioral Aggregation

To examine aggregation with deadlines, we will again use the traffic light controller from Drusinsky [23] only modified to explicitly include the deadlines. A modified specification for Drusinsky's traffic light controller with deadlines is as follows:

- There are two directions, *Main* and *Secondary*, with alternating lights.
- Lights alternate based on a *Timeout* signal controlled by a timer in a timed state with a deadline.
- Deadlines are as follows: Light can be red for 120 seconds or less. Light must be yellow for 30 seconds, and Light can be green for 120 seconds or less.

- The initial state (All-y) is for all lights to flash yellow. *Reset* occurring in All-y state starts On-going. *Reset* in On-going returns to All-y.
- A counter counts the cars waiting in the main direction. The counter can sense the difference between cars and trucks.
- If main is red and four or more cars or one or more cars followed by a truck are waiting in the main direction, a hidden camera shoots the intersection.

The object model for the traffic light controller is shown in Figure 5.16 and the behavior model is shown in Figure 6.11. In these figures, we see that the aggregation causes the behavior model to have a more complex nature. Still the behavior model exhibits concurrent behavior with the camera and counter components, although in this case their states only make sense as nested states within specific states of the aggregate (camera and counter only work when the main light is red).

In the behavior model, we use the timed states to represent the deadlines. When the timer in the timed states counts down to zero the timeout transition is taken changing the state (and color) of the traffic light. Notice that if enough cars or cars and trucks are detected by the counter, the transitions are taken to change the light color before the timer times out.

In general, deadlines in aggregated components will become concurrent deadlines. Aggregation always implies concurrency and thus the deadlines for any one part of the aggregation will be concurrent with the deadlines of the other parts.

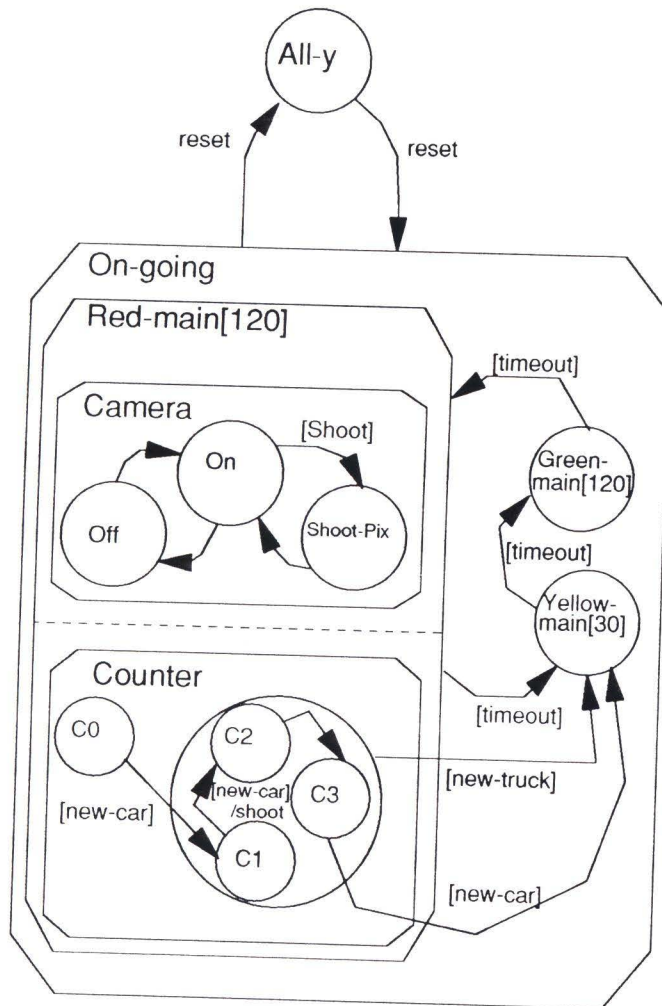


Figure 6.11: Behavior model for traffic light controller.

6.7 Chapter Summary

Real-time deadlines add complications to software systems. The adding of one simple deadline can turn simple solutions into difficult solutions. As real-time systems find their way into more applications, our ability to understand these applications becomes stretched. We support the use of object-oriented methodologies as the most promising technique for increasing our ability to create, maintain, and understand complex software. However, first we must understand how deadlines affect the relationship of the object and the behavior models in object oriented methodologies.

When an object in the object model is related to other objects by the use of association, aggregation, and generalization there are corresponding relationships in the behavioral model. When timing is added into these models in the form of a deadline, the problem of understanding becomes harder. The simplest object association is that of relationship. The resulting behavioral relationship is implemented by the use of conditions on the state transitions. Relations imply communication paths between objects. The behavioral model further strengthens this concept by exposing any temporal relationships in this communication path. When timing is added to any state or transition in the behavioral model, we can highlight this by annotating the connection between communicating objects with a timing notation. Thus we can clearly show when deadlines are affecting object communication.

Aggregation is a more complex form of association. The simplest behavioral

model associated with object model aggregation is that of concurrency. In this case, object model aggregation results in behavior model concurrency. However, behavior model concurrency is not always the result of object model aggregation. In other cases, the behavioral model is more complex, but concurrency is always involved. Adding timing to aggregations is fairly simple.

Object model generalization results in the most complicated behavior model. When the object model of a superclass is inherited, the behavior model of a subclass can be one where a new state or a new transition is added, or it could possibly result in a model where a state is decomposed into two or more new states. Also, the model could have a transition changed where a precondition is weakened, or where a postcondition or invariant is strengthened. Finally, a state can be modified for better performance.

The inheritance behaviors above can all have timed transitions or timed states associated with them, however, it is the specification of the program that will, in the end, dictate how the deadlines are modified or inherited in the system.

Chapter 7

SCENARIOS AND EVENT TRACE DIAGRAMS

7.1 Introduction

This chapter looks at the process for bringing real time deadlines into a behavioral model. The OMT dynamic model is made up of the statechart behavior model, scenarios, event trace diagrams, and possibly a set of event traces. We examine these pieces and look at how they can be extended for incorporating real-time deadlines into the models.

Some object-oriented real-time software design methodologies call for the design and analysis of the behavior models as if the system were conventional software, and introduces the deadlines into the models afterwards. We examine a different approach where the deadlines are incorporated in the dynamic models from the beginning in the scenario models. To facilitate this methodology, we introduce some new notations into the event trace diagrams, that help couple the scenarios to the statechart behavioral models.

In this chapter we look at how behavior models with deadlines are typically created in Section 7.2. Next we will review some background of scenarios and event

trace diagrams in Section 7.3. Next we look at how deadlines can be represented in scenarios in Section 7.4. Next we look at how deadlines can be represented in event trace diagrams and at some new notations that help make the concepts easier to understand in Section 7.5. Lastly we will present some conclusions in Section 7.6.

7.2 Behavior Models with Deadlines

In this paper we will briefly explore some real-time object-oriented methodologies, based on OMT object models [54]. Deadlines are included as object attributes, which is more general than either [15] or [53].

There are several characteristics of real-time software that result in greater difficulty in every phase of the life cycle. The most prominent characteristic is the concept of deadlines, where tasks have to be completed in a set amount of time. Another important characteristic is the concept of predictability, knowing exactly what the system response to an input will be. These characteristics are often complicated by resource restrictions. The object-oriented methodologies help a great deal with the problem of predictability. Also a real-time operating system can handle the problem of scarce resources. We deal here with the problem of deadlines, and how to ensure that they are being met in our environment.

Deadlines are time factors that constrain the system during the performance of certain functions. Sometimes the deadlines are hard deadlines, which if not met, will result in catastrophic failure of the system. Other times the deadlines are soft,

which if not met, simply mean the results are invalid and the procedure will need to be repeated before going on. In this paper we will not make a distinction between hard and soft deadlines as the distinction is not germane to the problems we will be discussing. Our examples will only show generic deadlines to keep them simple.

Normally deadlines are modeled as system constraints. Since, in object-oriented methodology the object model contains the informational description of an object while the behavioral model contains the temporal description of an object, these constraints are normally noted in the object model and defined in the behavioral model. The deadlines can be modeled as conditions on transitions or time-out values on states. Often the process is to develop the models ignoring the deadlines, as if it were a conventional system, to ensure the system behavior is correct. Then deadlines are added back into the model and an attempt is made to reconcile the system behavior.

When deadlines are represented as conditions on state transitions, the behavioral model then has several drawbacks. The most serious of these is the resultant level of model complexity. It becomes difficult to understand if the design meets that specification and if the implementation meets the design. The reason for this is that the deadlines depend on lower level abstractions than the object and behavior models. Therefore this level of complexity is counterproductive.

Another problem relates to the basic theory of timed state machines [3]. Once timing is added to even the most basic state machine, it becomes difficult

to impossible to determine some properties such as reachability, without adding constraints onto the models. Reachability is the ability to show that all the states can be reached from the start state given some set of events. Much of this has to do with the nature of time and how we measure time.

All the drawbacks are related to the complexity that timing adds to the behavior model. In order to simplify the behavior model we look for a better way to create the deadlines. By introducing the deadlines early in the event scenarios and event trace diagrams, the resulting behavioral models become simpler and easier to understand.

7.3 Background

Scenarios can be used for creating system requirements and for translating requirements into designs. One method for doing this is shown in [32]. Here the scenarios can be used to describe the user's view of the system's behavior and to validate the system requirements. Another example is shown in the SDOOSDT methodology of [67], where scenarios are used through out the system life cycle to develop and analyze the requirements, design, and implementation of the real-time systems.

Why are scenarios a powerful tool for software development? This may have to do with the way humans think as explained by cognitive science [12]. Carroll et al. detail how scenarios can be traced from the shared narratives of myth and

legends to modern technical expertise exchanges in the form of "war stories". This foundation from psychology suggests that scenarios are an important tool in the software developer's toolbox.

While the scenario methods of Hsia et al. and Carroll et al., differ in details, they are essentially the same. Both methods elicit a scenario from the user on how the system is expected to behave. The scenarios are then formalized (Hsia et al. uses regular grammars and Carroll et al. uses propositions that are then transformed into object-oriented artifacts). Next the scenarios are refined thorough the use of questions and formal verification methods. This is repeated iteratively until the design can no longer be refined.

In an OMT-based methodology the statecharts are created from the event trace diagrams. In other methodologies, such as the OOAD methodology proposed by Chen and Hung [13], this is not the case. Chen and Hung propose creating the statecharts early on in the development process and later developing scenarios and event trace diagrams for the purpose of identifying class operations and attributes. In a real-time environment, Chen's and Hung's methodology will have the same problems that OMT has when the deadlines are introduced late in the process. Chen and Hung also propose a modification to the event trace diagram, but their modification is only for identifying implicit operations in the event trace diagram.

We should also note the work being done to automate the generation of statecharts from event trace diagrams [36]. This automation has great potential

for automating design methodologies. However, for this automation to work in a real-time object-oriented environment, the notion of which object is responsible for the timing will be important.

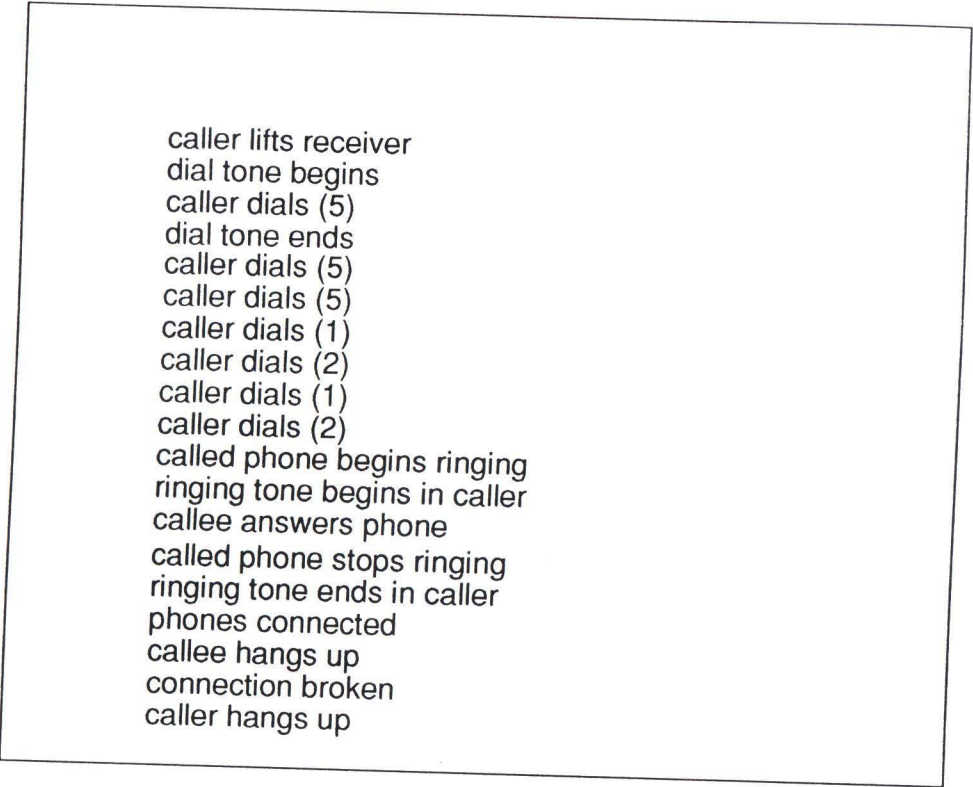
7.4 Scenarios with Deadlines

Our work in this chapter builds upon the work of Chonoles and Gilliam, who extended OMT with real-time constructs [15]. In OMT a scenario is written showing a specific set of events and actions that the model has to respond to and what actions will result. Figure 7.1 shows an example of an OMT scenario from Rumbaugh [54]. Often a model will need many scenarios. Some complicated models can have hundreds of scenarios.

When deadlines are present in the system, they should be represented in the written scenarios. For example, lets say the phone company has a system requirement that after the first number is dialed, the next number must be dialed within 5 seconds. This would result in a second scenario for our phone system showing the timeout behavior. This would result in the scenario that is shown in Figure 7.2.

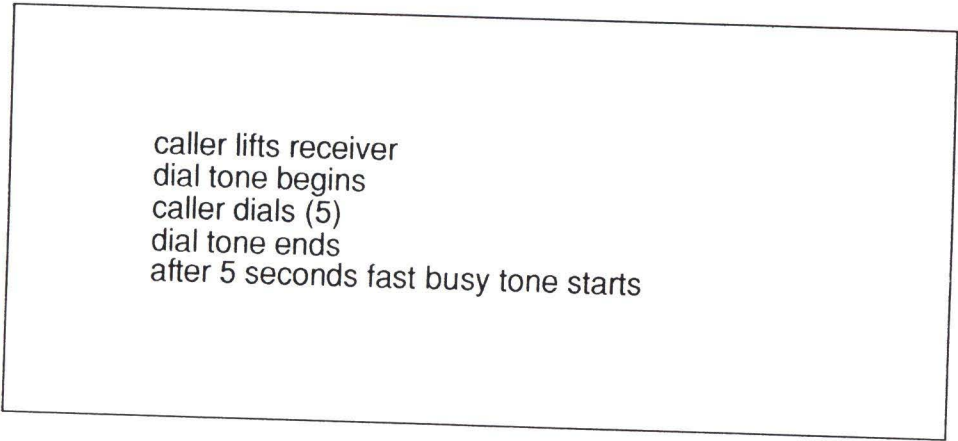
7.5 Event Trace Diagrams with Deadlines

Figure 7.3 shows an example of an OMT event trace diagram. Event trace diagrams are graphical representations that may combine some scenarios into one diagram. Note that the event trace diagram starts at the top and flows down the page. In this case an event that appears on the page under another event indicates



caller lifts receiver
dial tone begins
caller dials (5)
dial tone ends
caller dials (5)
caller dials (5)
caller dials (1)
caller dials (2)
caller dials (1)
caller dials (2)
called phone begins ringing
ringing tone begins in caller
callee answers phone
called phone stops ringing
ringing tone ends in caller
phones connected
callee hangs up
connection broken
caller hangs up

Figure 7.1: Typical scanario for phone call.



caller lifts receiver
dial tone begins
caller dials (5)
dial tone ends
after 5 seconds fast busy tone starts

Figure 7.2: Scenario with timing information.

that the event occurs after the first event. This only implies relative timing. For instance, if event 2 is one centimeter below event 1 in the event trace diagram we can not say that this implies that one minute or one unit of time has passed between the events. All this diagram does is show the events that occur and their relative order of occurrence. For normal nonreal-time software, this is sufficient for going to the next step, which is to create the behavioral model. However, for real-time programs this is not sufficient information.

For real-time systems the event trace diagrams need some additional notations. First the amount of time between events needs to be noted on the event trace diagram. Rumbaugh used timing marks associated with transitions to represent the time between the events [53]. For example in Figure 7.4, we see that five seconds after the dial tone ends with no input, the fast busy tone starts. This is done by connecting the events that have a timing relationship with a dotted line that has the time noted on it.

This is consistent with Chonoles' and Gilliam's observation that we should be suspect of any timing requirements that can not be represented as a time interval between an incoming and outgoing event, or between two outgoing events [15]. Any other timing requirements, such as timing between two incoming events, can violate object-oriented principles such as information hiding, and can be difficult to test. When the timing requirement is not a constant but an interval we have to add another notation. Say the requirement in the previous example is that the time

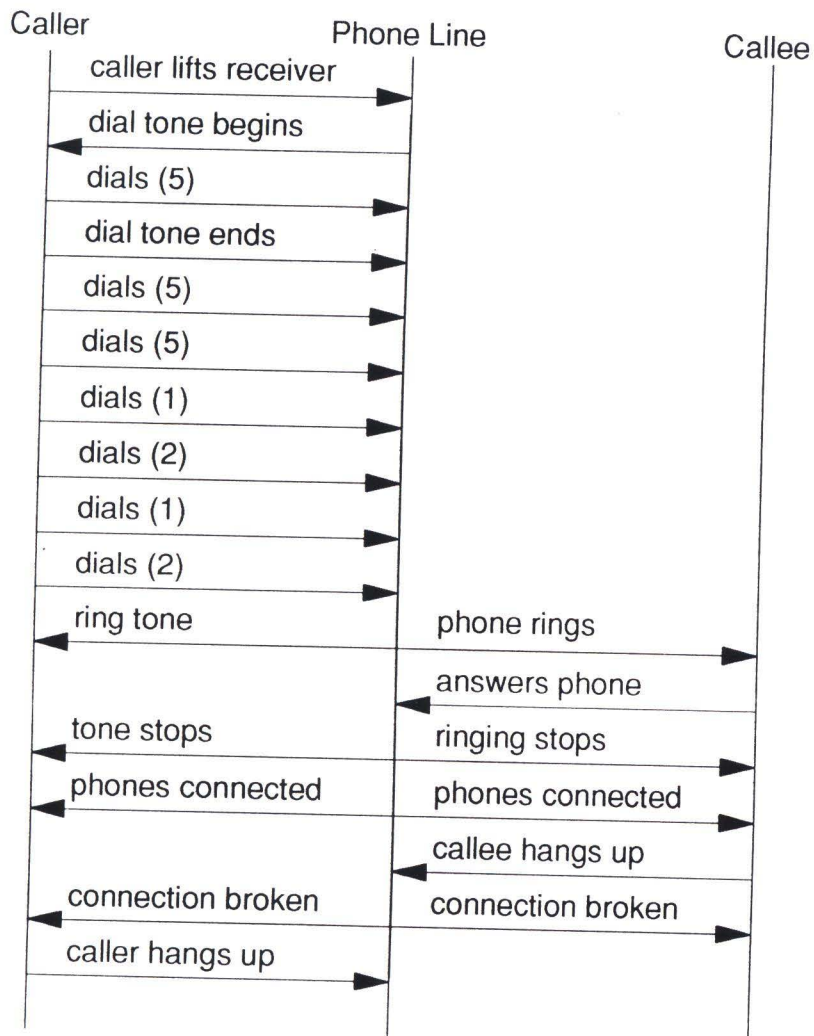


Figure 7.3: Typical event trace diagram for phone call.

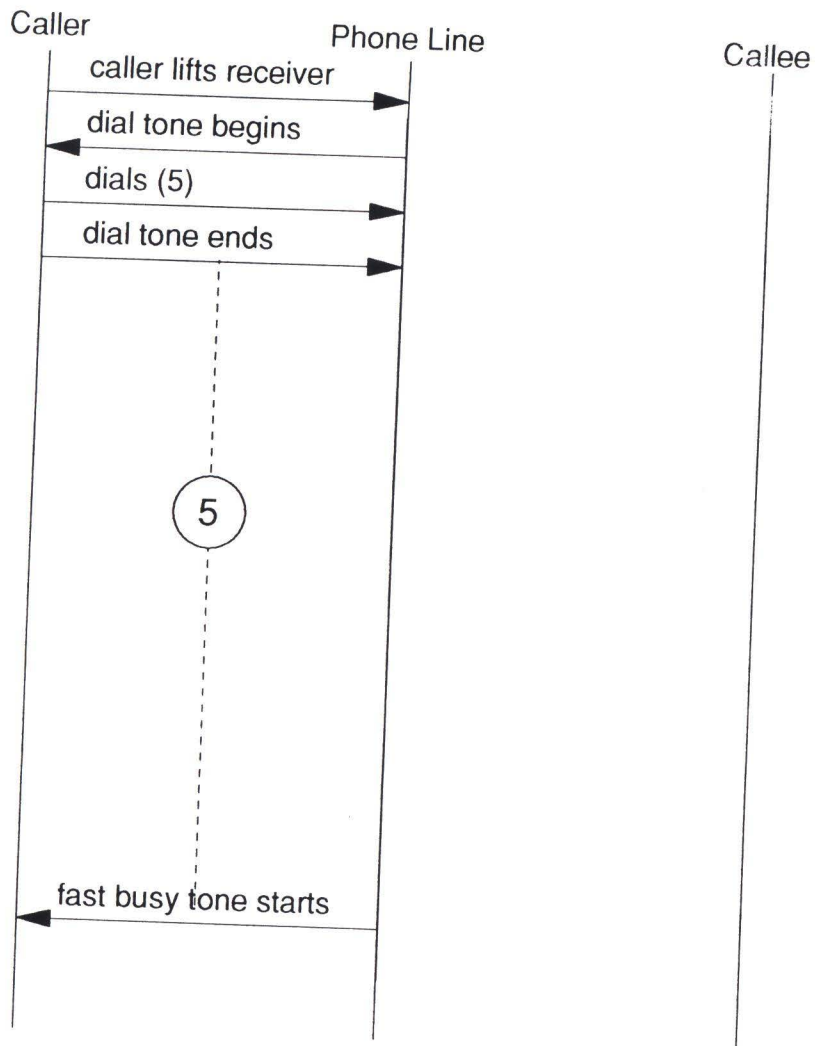


Figure 7.4: Event trace diagram with timing information.

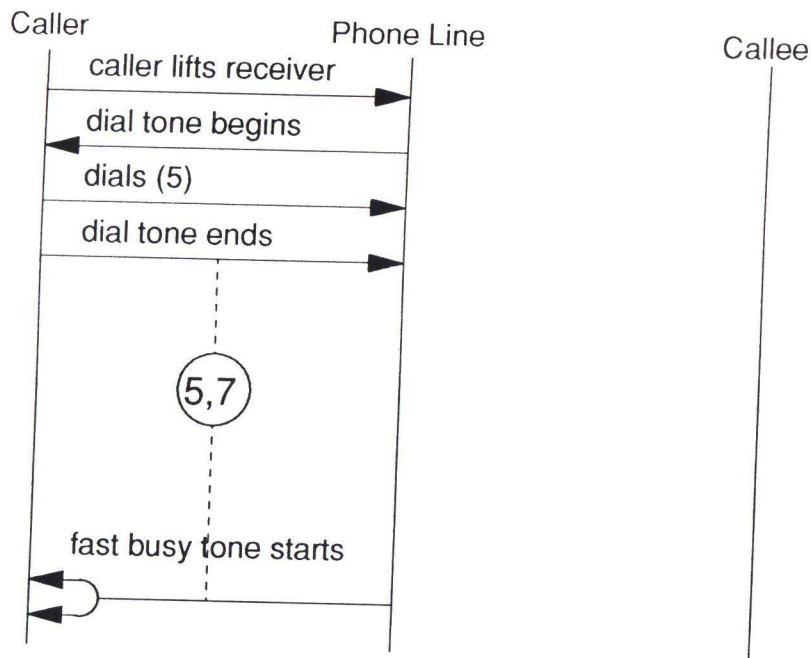


Figure 7.5: Event trace with time interval requirement.

between the first digit and the time-out is the interval 5 to 7 seconds. We would show this as the event with the split head as shown in Figure 7.5.

It is important to note that Figure 7.5 represents the fact that the caller sees the fast busy tone within a time interval. There is another way to view this, where the phone line generates the fast busy tone within a time interval (Figure 7.6). Here the event has a split tail implying that the phone line has the timing requirement, not the caller. This will make a difference in the behavioral model later in the design cycle so it is important to correctly decide where this timing will go.

The next step in the behavioral model design is to transform the event trace

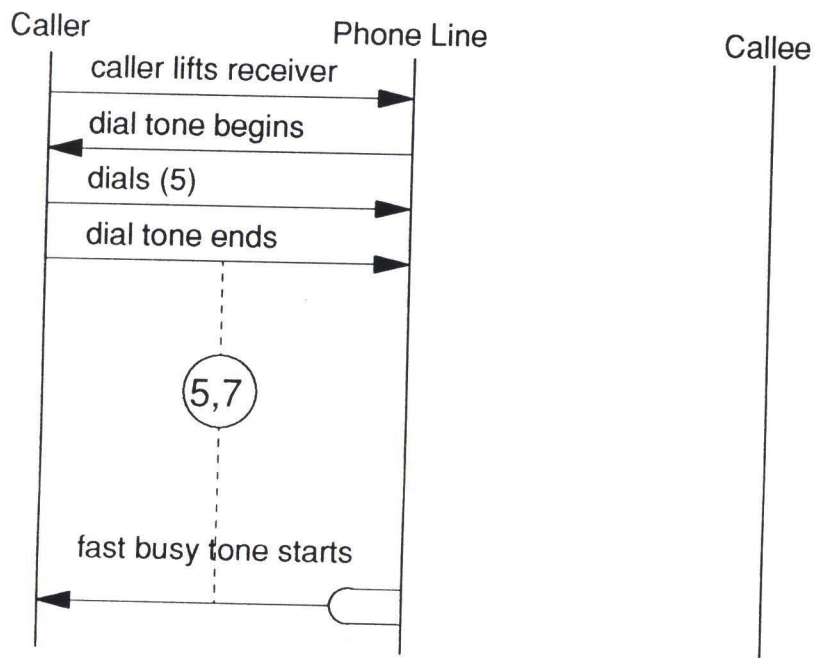


Figure 7.6: Event trace diagram with timing interval on event generation.

diagrams into a statechart behavioral model. Our original event trace diagrams can be used to produce the statechart shown in Figure 7.7.

7.6 Chapter Summary

Real-time deadlines add complications to software systems. The adding of one simple deadline can turn simple solutions into difficult solutions. However as real-time systems find their way into more applications, our ability to understand these applications becomes stretched. We support the use of object-oriented methodologies as the most promising technique for increasing our ability to create, maintain, and understand complex software. In the object-oriented paradigm we find that the behavioral model is where the real-time deadlines are manifested. It is important to understand how the timing specification and requirements get correctly introduced into the behavioral model.

The behavioral model starts with the narrative scenarios. This is one of the most important steps in creating our understanding of what a system will do and how it will work. Scenarios are very effective tools for communicating system design requirements because they are part of our human nature.

Scenarios are used to create event trace diagrams. The standard event trace diagrams used by Rumbaugh are good for standard object-oriented design, but lack the ability to express the timing information necessary for real-time applications. To overcome this we propose two additions to the event trace diagrams. The first is

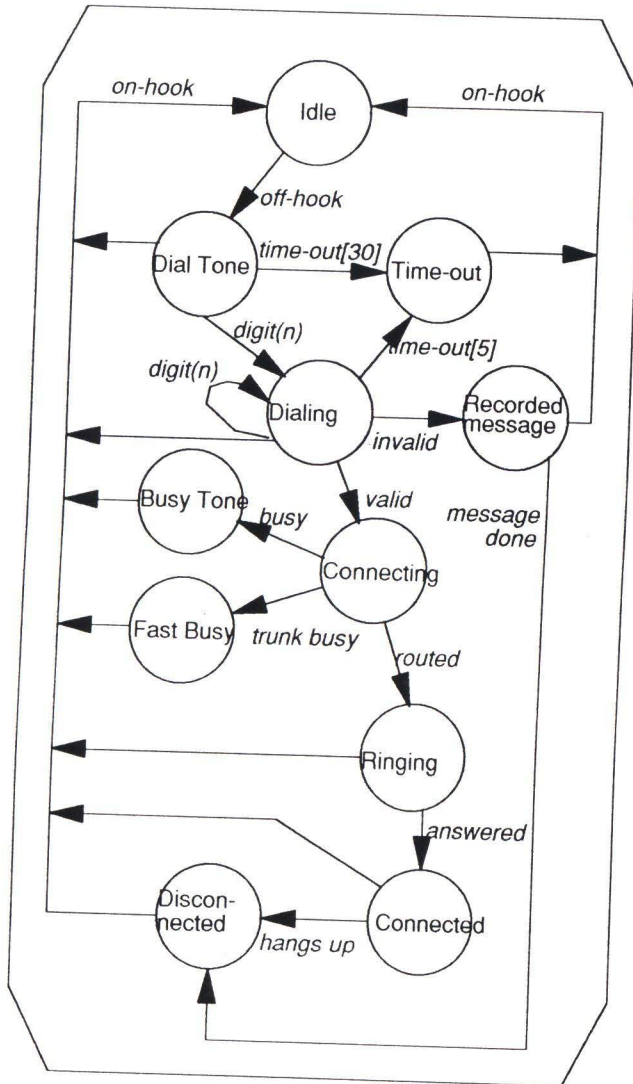


Figure 7.7: Statechart for typical telephone line.

the notation that time be represented as a dotted line between events. This allows us to graphically document what events are related in time to other events. This also allows to be clear with the cause and effect relationship between timed events. This helps us avoid the trap of designing systems that are difficult to analyze and test. The second additional notation is the that of event arrows that have split heads or split tails. These are used to graphically document the time intervals. Also by correctly using the split events, we can use this as a tool to ensure that the timing requirements are implemented in the correct behavior model.

Lastly the event traces are used to create the statecharts behavior models. By starting to look at the real-time deadlines in the scenarios, when the behavioral models are created they will be simpler and easier to verify than if the timing is introduced after the static behavior model has been created.

Chapter 8

CONCLUSIONS

Real-time systems are characterized by several unique aspects. One aspect is that there are deadlines associated with tasks. These deadlines are present in their specifications and can be of several types.

Another aspect of real-time systems is that tasks are often periodic in nature. Tasks will need to be run one or more times in a window of time. The scheduling of these tasks is often accomplished by a complex real-time operating system.

In this dissertation we analyzed ten object-oriented real-time design methodologies, using a criteria we developed.

Object-oriented methodologies have several advantages in real-time software design. This includes the general object-oriented advantages of isolating the impact of changes and encouraging reuse. The object-oriented methodology also models the operating environment which is an advantage since real-time software can not be looked at outside of the context of its environment. Object-oriented also allows the use of formal methods that can be used to show the predictability of real-time software.

Concurrency is important to consider in real-time environments. COBRA, RTO, Transnet, OCTOPUS, OMT, and OPNets all have support for concurrent environments. Concurrent systems with hard or firm message passing deadlines benefit from methodologies with a distributed control structure system design. All the methodologies except COBRA and Transnet have distributed control mechanisms.

Deadlines are also an important issue in real-time systems. ARTS, HRT-HOOD, and OCTOPUS have support for ensuring deadlines will be met and for showing the predictability of the system. OMT, RTO, and Transnet have some deadline handling support. The remaining methodologies support only the implementation phase of the design cycle.

The behavior modeling of ARTS and HRT-HOOD is not clear, but we assume that it is done with statecharts. COBRA uses event sequencing scenarios and statecharts for behavior modeling. RTO, ROOM, OCTOPUS, and OMT also use scenarios and statecharts. HOOD/PNO, Transnet, and OPNets use Petri nets. None of these methodologies support modeling of the deadlines of tasks and messages. The interaction of hard, firm, and soft deadlines together in a system is also not considered.

None of these methodologies support inheriting the deadlines and behavior models when the object models are inherited. In fact few of the methodologies include any support for inheritance.

As a result of this analysis we found that most of the methodologies were weak in the areas of modeling of system behavior, use of inheritance, and deadline management. All of the methodologies could use further development in these areas. Specifically we decided that an analysis of deadlines in behavioral models and a study of the effects of deadlines on inheritance and aggregation were necessary.

While some of the methodologies used Petri Nets and others used statecharts, we felt that since there is a mapping from Petri Nets to statecharts and back (although not a one-to-one mapping) that statecharts would be used in this dissertation for discussion and evaluation.

The issue of time becomes important as a starting point. An examination of how to measure time and how to represent it in a system was performed. Also, we examined how time affects simple automata. Since adding timing to simple automata makes the automata very complex, adding timing to statecharts (a complex automata) will also make them more complex.

Next, we examined how timing is added to statecharts. The most popular method is to add timing as a condition on a state transition. We explored how this is done for three current methodologies and how this is analyzed to understand, model, and document system performance.

Besides timing on the state transitions, timing can be added to the states themselves. This was relatively unexplored. We show that using timed states can often result in a clearer model than using timed transitions. However, the models

are equivalent and can be used interchangeably.

After that, we examined the relationship between the object and behavior models when the object model uses inheritance to generalize or particularize some classes. This is very important for several reasons. First, inheritance is one of the mechanisms that makes the object-oriented methodology so powerful. Second, reuse of the design artifacts of the object-oriented methodology is one of the desired results of object-oriented methodology. Finally, there is a relationship between the resultant behavior models after object model inheritance.

The resultant behavior models, after object model inheritance, should have the property of subtyping. That means that any place that the parent object is used, it can be replaced with the child object. The behavior of the objects will be the same for the same inputs. There are several ways that the behavior model can be modified during inheritance and have subtyping be maintained and we studied these in detail.

Of the behavior model modifications, weakening the precondition, strengthening the post condition or invariant relation, can be used in such a way that subtyping is not maintained. The rest will maintain subtyping in the child object's behavior model. Also, when using multiple inheritance it is important to always inherit the most specialized parts of the parent models.

The object model association of relationship is also very important. Relationship implies an object communication path. However, the object model only

shows the static communication path. The behavioral model can show dynamic aspects of the communication paths.

The behavioral model associated with object model aggregation almost always results in some degree of behavioral model concurrency. In most cases, aggregation will result in separate behavior models that work independently. Sometimes, for clarity these can be combined into a single model.

Next, deadlines were added to the object relationship model. Here the deadline added a subtle complication to the design analysis, but did not change the nature of the communications paths represented by object relationships. After that, deadlines were added to aggregation. Since aggregation results in concurrent behavior, the deadlines remain in the concurrent parts of the behavior model.

Finally, the relationship between the object and behavior models when real-time deadlines are present was analyzed. All of the allowable behavioral model modifications from inheritance were shown to still work when deadlines are present. We showed that the addition of additional transitions or states, refining a state into two or more states, splitting and retargeting of transitions, and the addition of attributes have no effect on deadlines. Weakening a precondition, strengthening a postcondition or an invariant relationship, or modifying a state can affect the deadlines, sometimes in a desirable way.

8.1 Contributions

The main contributions of this dissertation are the following:

- The work specifying the relation between the object and behavior models when the object model is used in relations, aggregation, or is inherited.
- The work examining how real-time deadlines affect the object and behavior model relations.
- The work on timed state statecharts.
- The survey of real-time object-oriented software design methodologies.

8.2 Future Directions

Here are some of the future work we would like to see come as a continuation of this dissertation:

- Investigation of the effects of real-time operating system priority algorithms on deadline analysis at design time.
- A detailed analysis of timed Petri nets showing all the different ways of modeling the timing, and how these representations help or hinder the modeling of real-time deadlines.

- A methodology for mapping timed statecharts to timed Petri nets. This would allow the Petri Net model to be used with automated tools to test for conditions such as deadlock, races, and liveness, while allowing statecharts to be used for the human design interface.
- A further analysis of concurrency in object-oriented methodologies including an analysis of what level of abstraction would be best for the concurrency to be introduced and analyzed.
- Analysis of the formal methods added to class structure diagrams and statecharts for proving that the design meets its specifications. Some of this is done in [18] but more work is needed.
- Specification of language constructs that must be added to standard object-oriented languages to support a real-time design methodology.
- Ways to verify the specified deadlines. OCTOPUS uses object interaction diagrams for this purpose. This can be combined with prototype execution.

REFERENCES

- [1] T. Agerwala, "Putting Petri Nets to Work," *Computer*, vol. 12, no. 12, pp. 85–94, December 1979.
- [2] M. Aksit and L. Bergmans, "Obstacles in Object Oriented Software Development," in *Proceedings of OOPSLA '92*, pp. 341–358, Vancouver, British Columbia, October 1992.
- [3] R. Alur and D. Dill, "The Theory of Timed Automata," in *Real Time: Theory in Practice. REX Workshop Proceedings 1992*, pp. 47–73, 1992.
- [4] R. Alur, T.A. Henzinger, and P.H. Ho, "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems," in *Hybrid Systems 1993*, pp. 209–229, 1993.
- [5] R. Alur, T.A. Henzinger, and M.Y. Vardi, "Parametric Real-Time Reasoning," in *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, pp. 592–601, 1993.
- [6] N.C. Audsley, A. Burns, and M.F. Richardson, "STRESS: A Simulator for Hard Real-Time Systems," *Software - Practice and Experience*, vol. 24, no. 6, pp. 543–564, June 1994.
- [7] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings, "Incorporating Unbounded Algorithms Into Predictable Real-Time Systems," *Computer Systems Science and Engineering*, vol. 8, no. 2, pp. 80–89, April 1993.
- [8] N.C. Audsley, R.I. Davis, and A. Burns, "Appropriate Mechanisms for the Support of Optional Processing in Hard Real-Time Systems," in *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software 1994*, pp. 23–27, 1994.

- [9] R.J.A. Buhr and R.S. Casselman, "Timethread-Role Maps for Object-Oriented Design of Real-Time and Distributed Systems," in *Proceedings of OOPSLA 94*, pp. 301-316, October 1994.
- [10] A. Burns and A.J. Wellings, "HRT-HOOD: A Structured Design Method For Hard Real-Time Systems," *Real-Time Systems*, vol. 6, no. 1, pp. 73-114, January 1994.
- [11] J.P. Calvez and O. Pasquier, "Implementation of Statecharts With Transputers," *Microprocessing and Microprogramming*, vol. 35, no. 1-5, pp. 133-140, September 1992.
- [12] J.M. Carroll, R.L. Mack, S.P. Robertson, and M.B. Rosson, "Binding Objects to Scenarios of Use," *International Journal of Human Computer Studies*, vol. 41, no. 1-2, pp. 243-276, July-August 1994.
- [13] J. Chen and Y. Hung, "An Integrated Object-Oriented Analysis and Design Method Emphasizing Entity/Class Relationship and Operation Finding," *Journal of Systems and Software*, vol. 24, no. 1, pp. 31-47, January 1994.
- [14] R.C. Chen and P. Dasgupta, "Implementing Consistency Control Mechanisms in the Clouds Distributed Operating System," in *Proceedings of the International Conference on Distributed Computing Systems*, pp. 10-17, 1991.
- [15] M.J. Chonoles and C.C. Gilliam, "Real-Time Object-oriented System Design Using the Object Modeling Technique (OMT)," *Journal of Object Oriented Programming*, vol. 8, no. 3, pp. 16-24, June 1995.
- [16] D. Coleman, F. Hayes, and S. Bear, "Introducing Objectcharts or How to Use Statecharts in Object-oriented Design," *IEEE Transactions on Software Engineering*, vol. 18, no. 1, pp. 8-18, January 1992.
- [17] S. Cook and J. Daniels. *Designing Object Systems*. chapter Sub-types, Inheritance and Conformance. Prentice-Hall, 1994.
- [18] S. Cook and J. Daniels, "Lets Get Formal," *Journal of Object Oriented Programming*, vol. 7, no. 4, pp. 22-66, July-August 1994.

- [19] P. Daponte, L. Nigro, and F. Tisato, "Object-Oriented Design of Measurement Systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 41, no. 6, pp. 874–880, December 1992.
- [20] P. Dasgupta, R. Ananthanarayanan, and S. Menon, "Distributed Programming with Objects and Threads in the Clouds System," *Computing Systems*, vol. 4, no. 3, pp. 243–275, Summer 1991.
- [21] R.I. Davis, K.W. Tindell, and A. Burns, "Scheduling Slack Time in Fixed Priority Pre-emptive Systems," in *Proceedings - Real-Time Systems Symposium 1993*, pp. 222–231, 1993.
- [22] N. Day, "A Comparison Between Statecharts and State Transition Assertions," in *Proceedings of IFIP International Workshop on Higher Order Logic Theorem Proving and its Applications - HOL'92*, pp. 247–262, 1993.
- [23] D. Drusinsky, "Visually Designing Embedded-Systems Applications," *Dr. Dobb's Journal*, vol. 20, pp. 62–68, June 1995.
- [24] E.B. Fernandez, T.B. Horton, and B. Abou-Haidar. "An Object-oriented Methodology for the Design of Control Software for Flexible Manufacturing Systems". Technical Report TR-CSE-93-67, Dept of Computer Science and Engineering, Florida Atlantic University, December 1993.
- [25] U. Furbach, "Formal Specification Methods for Reactive Systems," *Journal of Systems and Software*, vol. 21, no. 2, pp. 129–139, May 1993.
- [26] A. Gheith and K. Schwan, "CHAOS**(arc): Kernel Support for Multiweight Objects, Invocations, and Atomicity in Real-Time Multiprocessor Applications," *ACM Transactions on Computer Systems*, vol. 11, no. 1, pp. 33–72, February 1993.
- [27] H. Gomaa, "A Behavioral Analysis Method for Real Time Control Systems," *Control Engineering Practice*, vol. 1, no. 1, pp. 115–120, February 1993.
- [28] J. Gregoire, J.A. Makowsky, and S.E. Levin, "Programming Reactive Systems with Statecharts," in *5th Israel Conference on Computer Systems and Software Engineering*, pp. 87–103, 1991.

- [29] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Journal of Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.
- [30] C. Heitmeyer and N. Lynch, "Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems," in *Proceedings of the IEEE Real-Time Systems Symposium 1994*, pp. 120–131, 1994.
- [31] J.J.M. Hooman, S. Ramesh, and W.P. de Roever, "A Compositional Axiomatization of Statecharts," *Theoretical Computer Science*, vol. 101, no. 2, pp. 289–335, July 1992.
- [32] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen, "Formal Approach to Scenario Analysis," *IEEE Software*, vol. 11, no. 2, pp. 33–41, March 1994.
- [33] K. Inan, "On a Class of Timer Hybrid Systems Reducible to Finite State Automata," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 5, no. 1, pp. 83–96, January 1995.
- [34] Y. Ishikawa, H. Tokuda, and C.W. Mercer, "An Object-Oriented Real-Time Programming Language," *Computer*, vol. 25, no. 10, pp. 66–73, October 1992.
- [35] J. C. Kelly and Y. S. Sherif, "Comparison of Four Design Methods for Real-Time Software Development," *Information and Software Technology*, vol. 34, no. 2, pp. 74–82, February 1992.
- [36] K. Koskimies and E. Makinen, "Automatic Sythesis of State Machines from Trace Diagrams," *Software - Practice and Experience*, vol. 24, no. 7, pp. 643–658, July 1994.
- [37] W.K.C. Lam and R.K. Brayton, "Alternating RQ Timed Automata," in *Computer Aided Verification. 5th International Conference, CAV '93*, pp. 237–252, 1993.
- [38] W.K.C. Lam and R.K. Brayton, "Criteria For The Simple Path Property in Timed Automata," in *Computer Aided Verification. 6th International Conference, CAV '94*, pp. 27–40, 1994.

- [39] H. Lecoenche and J.L. Sourrouille, "A Dynamic Model Extension to Integrate State Inheritance and Objects," *ROAD*, vol. 2, pp. 37–43, July-August 1995.
- [40] Y. K. Lee and S. J. Park, "OPNets: An Object-Oriented High-Level Petri Net Model for Real-Time System Modeling," *Journal of Systems and Software*, vol. 20, no. 1, pp. 69–86, January 1993.
- [41] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements Specification for Process-Control Systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 9, pp. 684–707, September 1994.
- [42] P. Linz. *An Introduction to Formal Languages and Automata*. D.C. Heath and Company, 1990.
- [43] N. Lynch and F. Vaandrager, "Forward and Backward Simulations for Timing Based Systems," in *Real Time: Theory in Practice. REX Workshop Proceedings 1992*, pp. 397–446, 1992.
- [44] J.D. McGregor and D.M. Dyer, "A Note on Inheritance and State Machines," *ACM SIGSOFT*, vol. 18, no. 4, pp. 61–69, October 1993.
- [45] C.W. Mercer and H. Tokuda, "The ARTS Real-Time Object Model," in *Proceedings of 11th IEEE Real-Time System Symposium*, pp. 2–10, 1990.
- [46] X. Nicollin, J. Sifakis, and S. Yovine, "Compiling Real-Time Specifications into Extended Automata," *IEEE Transactions on Software Engineering*, vol. 18, no. 9, pp. 794–804, September 1992.
- [47] L. Nigro, "Real Time Objects in C++," in *C++ at Work '90*, pp. 123–135, Secaucus, NJ, September 1990.
- [48] L. Nigro and F. Tisato. *Advances in Object-Oriented Software Engineering*, chapter An Object-Based Architecture for Real-Time Applications. Prentice-Hall, 1992.
- [49] M. Paludetto and S. Raymond, "A Methodology Based on Objects and Petri Nets for Development of Real-Time Software," in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pp. 705–710, 1993.

- [50] M.P. Pearson and P. Dasgupta, "CLIDE: A Distributed, Symbolic Programming System Based on Large-grained Persistent Objects," in *Proceedings of the International Conference on Distributed Computing Systems*, pp. 626–633, 1991.
- [51] J. Rumbaugh, "Controlling Propagation of Operations Using Attributes on Relations," in *Proceedings of OOPSLA 1988*, pp. 285–296, 1988.
- [52] J. Rumbaugh, "Disinherited! Examples of Misuse of Inheritance," *Journal of Object Oriented Programming*, vol. 6, no. 1, pp. 22–24, February 1993.
- [53] J. Rumbaugh, "OMT: The Dynamic Model," *Journal of Object Oriented Programming*, vol. , pp. 6–12, February 1995.
- [54] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy. and W. Lorensen. *Object-oriented Modeling and Design*. Prentice Hall, 1991.
- [55] K. Sacha, "Real-Time Specification Using Petri Nets," *Microprocessing and Microprogramming*, vol. 38, no. 1-5, pp. 607–614, September 1993.
- [56] K. Schwan, A. Gheith, and H. Zhou, "Building Families of Object-based Multiprocessor Kernels," in *Proceedings of the International Conference on Parallel Processing*, pp. 366–369, 1992.
- [57] B. Selic, "An Efficient Object-oriented Variation of the Statecharts Formalism for Distributed Real-time Systems," in *Proceedings of the 11th IFIP International Conference on Computer Hardware Description Languages and their Applications - CHDL'93*, pp. 335–344, 1993.
- [58] B. Selic, G. Gullekson, J. McGee, and I. Engelberg, "ROOM: An Object-Oriented Methodology for Developing Real-Time Systems," in *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, pp. 230–240, 1992.
- [59] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, Inc., 1994.

- [60] K. G. Shin and P. Ramanathan, "Real-Time Computing: A New Discipline of Computer Science and Engineering," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, January 1994.
- [61] J.L. Sourrouille, "A Framework for the Definition of Behavior Inheritance," *Journal of Object Oriented Programming*, vol. 9, no. 2, pp. 17–20, March 1996.
- [62] J. A. Stankovic, "Misconceptions About Real-Time Computing," *Computer*, vol. 21, no. 10, pp. 10–19, October 1988.
- [63] R. Stevenson, "Whatever Happened to Finite State Machines?," *Object Magazine*, vol. 5, no. 7, pp. 55–61, November-December 1995.
- [64] K.W. Tindell, A. Burns, and A.J. Wellings, "An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks," *Real-Time Systems*, vol. 6, no. 2, pp. 133–151, March 1994.
- [65] M von der Beeck, "Enhancing Structured Analysis by Timed Statecharts for Real-Time and Concurrency Specification," in *Proceedings of IFIP International Conference on Decentralized and Distributed Systems*, pp. 369–381, 1993.
- [66] J. Wang and H. Chen, "A Formal Technique to Analyze Real-time Systems," in *Proceedings of International Computer Software and Applications Conference*, pp. 180–185. IEEE Computer Society, 1993.
- [67] W. Wang, S. Hufnagel, P. Hsia, and S.M. Yang, "Scenario Driven Requirements Analysis Method," in *Proceedings - Second International Conference on Systems Integration*, pp. 127–136, 1992.
- [68] T.G. Woodcock and E.B. Fernandez, "The Relationship Between the Object and Behavior Models," in *Proceedings - TOOLS USA '96*, June 1996.
- [69] J. Ziegler, M. Awad, and J. Kuusela, "Applying Object-Oriented Technology in Real-Time Systems with the OCTOPUS Method," in *Proceedings of First IEEE International Conference on Engineering of Complex Computer Systems*, pp. 306–309, November 1995.

VITA

Timothy G. Woodcock received his B.A. degree in physics from Florida Atlantic University, Boca Raton, in 1980, and his M.S. degree in computer science from Florida Atlantic University, Boca Raton, in 1989. He is currently a Ph.D. candidate in the Department of Computer Science and Engineering at Florida Atlantic University, Boca Raton.

His research interests include software engineering, object-oriented design and analysis, modeling, and project management.

