The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations

Kevin Jeffay

Technical Report # 89-09-15

Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195 USA
September, 1989

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
University of Washington
1989

The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations

Kevin Jeffay

Technical Report # 89-09-15

Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195 USA
September, 1989

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
University of Washington
1989

University of Washington

Abstract

The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations

by Kevin Jeffay

Chairperson of the Supervisory Committee: Professor Alan C. Shaw
Department of Computer Science and Engineering

Real-time computer systems are loosely defined as the class of computer systems that perform computations and I/O operations in a time frame defined by processes in the external environment. Within this class of systems are *hard-real-time* systems. Hard-real-time systems are characterized by the existence of timing constraints that must be strictly adhered to for the system to function correctly. In this dissertation we study the problem of designing and analyzing hard-real-time systems. A message passing discipline for designing hard-real-time systems is developed. The discipline consists of (1) a graphical notation for specifying the logical structure of systems, (2) an operational semantics for reasoning about the real-time behavior of the system, and (3) a formal implementation model.

A design is a directed graph in which vertices are program components and edges are communication channels. Program components communicate by sending messages on unidirectional channels. We develop a novel semantics of communication based on a paradigm of process interaction called the *real-time producer/consumer paradigm*. For each message sender, these semantics specify the rate at which messages are consumed. With these semantics the real-time behavior of the designed system is completely understood in terms of the rates at which messages are sent.

We present a formal implementation model for our discipline based on periodic and sporadic tasks. The model is developed to study complexity and optimality issues concerning the implementation of designs constructed using our disicpline. For this model we investigate several uniprocessor scheduling and synchronization problems. New results on the optimal, non-preemptive, scheduling of periodic and sporadic tasks are developed. We introduce the concept of an *implementation strategy* to deal with tasks that share resources. An implementation strategy consists of the integration of a scheduling and synchronization policy. For our formal model we demonstrate that optimal implementation strategies based on *earliest deadline first* scheduling and WAIT and BROADCAST synchronization exist. The analysis of these scheduling and synchronization problems yield a decision procedure that can efficiently determine if an arbitrary acyclic, and some cyclic, design graphs can be guaranteed to adhere to the specifications derived from the real-time producer/consumer paradigm.

Some paper designs of actual systems, and a prototype programming system based on our discipline, demonstrate the viability of our research. They show that our design discipline is expressive enough to describe the real-time behavior of actual systems and yet is simple enough to allow rigorous analysis of the real-time behavior of systems so designed.

Table of Contents

List of Figures

List of Tables

Acknowledgements

My thanks to Alan Shaw who, as my dissertation advisor, would unfailingly ask me the single question that would both serve to clarify my ideas and reaffirm my convictions. His cheerful and indefatigable attention to detail in his review of this dissertation has been nothing short of inspiring. Alan's stated goal of having this document be the best piece of work I have produced to date has surely been met.

I also thank Ed Lazowska and Richard Anderson for their service on my Reading Committee. Their timely and insightful comments have contributed greatly to the quality of this final draft. In addition, I thank Richard for his help in the development of the intractability results in Chapter 3. His stamina in wading through the early drafts of the scheduling proofs has also been appreciated.

The road to the Ph.D. was made less weary, and indeed downright enjoyable, by many warm souls. I thank Hank Levy for his good cheer and for his gentle prodding to distraction. I hope that gravity, and a little wax, remain our best friends. Carl Binding culturally enriched my tenure at Washington by expanding my perspective beyond the Atlantic and Pacific coasts. Rick Korry ensured that there existed the requisite amount of bread and wine in a graduate student's diet and was gracious enough to always be present at its consumption. I am indebted to Dave Wagner for making me appreciate the problems of being a moose in modern-day America and to Ewan Tempero and Dave Socha for their friendship. My thanks to Marc Donner for showing me through empirical studies that sushi *is* the breakfast of champions (and the occasional lunch of computer scientists).

I do not know how I can thank in words my wife Susan for her support and faith throughout our years in Seattle. Over a lifetime together I hope I can offer a reasonable substitute.

Lastly I thank my parents for encouraging me to pursue the Ph.D. It has been the rewarding and deeply satisfying experience that I am sure they always wished it would be.

The research described in this dissertation was supported in part by a Graduate Fellowship from the IBM Corporation and in part by grants from the National Science Foundation (CCR-8700435) and from the Office of Naval Research (N00014-89-J-1040).

Chapter 1

Introduction

1.1 The Domain of Real-Time Computing

In the early days of computing, everything within a computer's physical environment was subservient to the needs of the computer. Due to the large cost of early computers, the primary goal in operating a computer was efficient utilization of its central processor. The earliest machines were not multiprogrammed. Therefore, if a job completed and an output device for recording its results was not ready and waiting, valuable processing cycles were lost. The occurrence of such an event was considered a serious operational error. A similar error occurred if the computer finished a job and the next one was not ready. The pace of activity in the computer's physical environment was determined primarily by the needs of the computer's processor. There was a quantifiable monetary cost associated with failing to meet these needs.

As advances in technology greatly reduced the cost of processing hardware, a new field of computing arose. This is the area of *real-time* computing. Real-time computing represents the antithesis of early computing. A real-time computer system is subservient to the requirements of its physical environment. In real-time computing, it is now the processes in the environment that determine the pace at which computations must proceed. Failing to meet the processing requirements of the environment results in errors which can have a high monetary cost. In addition, such errors can also have a high cost in terms of destruction of the environment.

For these reasons, the design and analysis of real-time systems presents new challenges. In addition to determining *what* a real-time program or system does, it is imperative to determine *when in real-time* the system does it. This thesis is concerned with the development of software architectures for real-time computer systems. The goal is to develop programming methods, and analysis techniques, for constructing and reasoning about software systems that must satisfy the processing requirements of their external environment in real-time.

1.1.1 Real-Time Systems and a Spectrum of Real-Time Computing

Real-time computer systems have a number of characteristics which distinguish them from more traditional computer systems. Some of the more prominent characteristics include:

- the existence of event driven performance constraints,
- high cost of failure,
- a high degree of physical and logical concurrency,
- continuous operation, and
- availability, reliability, and fault tolerance requirements.

The first characteristic refers to the fact that the time frame within which real-time systems must perform operations is defined in terms of the occurrence of events. Events may be inputs from the external environment or they may be generated by one portion of the real-time system for use by a different portion of the system. For example, in a control system for a six legged walking robot, notable events include operator console inputs, the changing of the value of a force sensor on a leg, or the detection by the software of an unstable position of the legs [Donner 84].

The processing requirements of a real-time system are specified in terms of the occurrence of these events. These requirements form a set of constraints on the performance of the system. These constraints are commonly referred to as *timing constraints*. For example, the robot control program might be required to react to occurrence of an unstable leg position within 1 second. The reason why these requirements form constraints is because often there is a high cost of failure associated with not meeting these requirements. If the robot remains unstable it can tip over and injure the operator or destroy the machine.

Often there is an abundance of concurrency in the external environment and this is reflected in the structure of real-time systems. This suggests that real-time systems will be at least as hard to design and reason about as multiprogramming systems [Wirth 77c]. The additional requirements of continuous operation (in the sense of there not existing a well defined termination point), availability, reliability, and fault tolerance underscore this issue.

Real-time systems will exhibit these characteristics in varying number and degree. A useful view of real-time computing is as a spectrum of computing environments ordered by a measure of any one of the above features. For our purposes, we will focus on the issue of performance constraints. We define a spectrum of computing based on the nature of performance requirements that the environment imposes on the real-time system. At one end of the spectrum, where extreme performance constraints exist, is the domain of *hard-real-time* computing. Hard-real-time systems operate in an environment in which the cost of not satisfying the processing requirements of an application can result in a high cost failure of the system. Such a high cost failure could include the loss of human life. Due to the high cost of failure, the central issue in hard-real-time computing is assessing the compliance of a system to its timing constraints. Hard-real-time systems have a dual notion of correctness. In addition to being *logically correct*, that is, containing algorithms that will compute the correct values for a set of inputs, the system must be *temporally correct*. The system must compute the correct values at the correct times. The performance of hard-real-time systems fundamentally must be *predictable* in a deterministic sense. Hard-real-time systems must provide guarantees that every instance of every performance constraint will be satisfied.

At the other end of the spectrum we have *soft-real-time computing*. Soft-real-time systems must also be logically and temporally correct. However, for soft-real-time systems, temporal correctness takes on more of a stochastic flavor. Not adhering to every instance of every timing constraints does not automatically imply an error. The external environment is forgiving enough that an occasional violated constraint is tolerable. Simply meeting constraints on average is often sufficient for ensuring the correctness of soft-real-time systems.

1.1.2 Hard-Real-Time Systems

We are primarily interested in the study of hard-real-time systems. Given the rigid characterization of temporal correctness for hard-real-time systems, it is appropriate to question if such systems actually exist. That is, do systems which require *guarantees* of compliance with *each* instance of *each* performance constraint in fact exist? We will not attempt to provide an answer to this question in this dissertation. Our interest in studying the hard-real-time end of the spectrum is motivated by the following two observations.

First, it is our hypothesis that all real-time systems contain some hard-real-time component or subsystem. We believe that every real-time system either interacts with devices, or contains data, that must receive guarantees of performance. For example, real-time systems require guarantees of performance for their interactions with devices in order to ensure either that incoming or outgoing data is not lost or that accurate control of processes in the external environment is maintained. Performance constraints can also exist on the monitoring of data items. This is so that the system can detect and react to data with exceptional values.

Secondly, even if the external environment can tolerate the non-hard-real-time behavior of a system, there is clearly an advantage to treating the system as if it were a hard-real-time system. If guarantees of adherence to performance constraints could be provided, then the behavior of the system would be more predictable. This would aid in the construction and analysis of real-time systems.

1.2 Research Approach and Contributions

The goal of our research is to understand the problems in the development of hard-real-time systems.[1] Our approach to the study of real-time systems is to develop a discipline for designing real-time systems. This will involve the development of a means of specifying the real-time behavior desired of a system, and the development of a method for guaranteeing that this specification is adhered to. We will develop a discipline for

[1] From this point forward, where it causes no confusion, our use of the term *real-time* will be understood to mean *hard-real-time*.

describing systems that is expressive enough to capture interesting and important real-time behaviors, and yet is simple enough that the analysis of temporal correctness is tractable. In addition, the discipline and the analysis will be constructive. They can be used to directly implement a real-time system.

There are three distinct research efforts in the development of our design discipline:

- the development of a graphical and graph notation for the logical design of real-time systems,
- the development of an operational semantics for the notation for specifying and reasoning about performance constraints, and
- the development of a formal model of an implementation of a real-time system.

The first component of the design discipline is a notation for specifying the logical structure of a system. We view a real-time system as a directed graph where vertices are processes and edges are communication channels. Processes communicate via message passing.

We will develop an operational semantics for the message passing operations that implicitly specifies the real-time behavior of each process. The semantics are based on a paradigm of process interaction we call the *real-time producer/consumer paradigm*. For each message sender, these semantics specify the rate at which messages are consumed. With these semantics the real-time behavior of the designed system is completely understood in terms of the rates at which messages are sent.

Lastly, our design discipline will be constructed on top of a formal tasking model based on *cyclic* tasks. Cyclic tasks are an abstraction of the processes in the design discipline. We will formulate and solve several uniprocessor scheduling and synchronization problems for cyclic tasks. The solutions yield new results on optimal preemptive and non-preemptive execution of *sporadic* and *periodic* tasks. In addition, the scheduling analysis will enable us to develop a decision procedure for efficiently determining whether or not a processor has sufficient capacity to ensure that an implementation of a design based on the formal model will have the desired real-time behavior. With this underlying model, we will be able to demonstrate that the analysis of systems designed using our discipline will be quantitatively and qualitatively better than existing methods.

1.3 Related Work

Previous research in the area of hard-real time systems has typically followed one of three identifiable approaches:

- computational models of hard-real-time systems.
- system design methodologies, and
- the development of programming languages and systems,

Each of these approaches will be examined briefly below. We view our work as logically spanning the areas of computational models and design methodologies. We are interested in developing and extending existing formal models of real-time systems to include more realistic features of systems such as shared data, inter-process communication, precedence, and mutual exclusion. At the same time we are very much concerned with being to apply these results in a practical setting. The computational model should be realistic enough to serve as the basis for constructing real-time systems.

1.3.1 Formal Models of Real-Time Systems

In the area of formal models, two major avenues of research have been the development of *program logics* and *analytical models*. Program logics are used to reason about the real-time behavior of programs using the text of the program and the semantics of the language. They seek to extend methods for reasoning about the correctness of concurrent programs, to include real-time behaviors. Analytical models are more abstract models of computation. They are constructed primarily to study decision problems concerning the real-time performance of operating system policies such as scheduling and resource allocation.

Program Logics

Examples of program logics include an extension to temporal logic [Bernstein & Harter 81], an extension to Hoare's CSP [Zwarico & Lee 85], and an extension to Hoare logic

[Shaw 89]. Bernstein and Harter prove properties of the temporal relationships between predicates on system states. For predicates *P* and *Q*, they formulate and prove assertions of the form "whenever *P* is true, *Q* will be true within *n* time units." They assume that the processing time of processes is negligible and that processes are only ever interrupted at well defined points. Zwarico and Lee based their reasoning system on a similar set of assumptions and prove that programs adhere to specifications with predicates containing time values. Shaw considers an execution model wherein each process executes either on a dedicated processor, or in a restricted manner on a shared processor. Schemata for determining upper and lower bounds on the execution time of program statements are presented.

Related to program logics are logics for verifying system specifications. Jahanian and Mok have developed a new logic called RTL (Real-Time Logic) for proving safety properties of real-time systems [Jahanian & Mok 86]. Given a event-action specification of a real-time system, one can either determine if the specification satisfies safety assertions, or identify conditions under which the assertions will not hold.

The work in program logics does not directly relate to our research.

Analytical Models

The majority of analytical models for real-time systems have focused on the problem of scheduling tasks that must be completed before a specified deadline. A model which has received considerable attention is the model of periodic tasks presented in [Liu & Layland 73]. Liu and Layland considered a characterization of a real-time system as a set of independent tasks that make requests for execution at regular intervals. Tasks are assumed to be preemptable at arbitrary points.

This model has been extended along a number of dimensions by several researchers. Leung and Merrill added release times and deadlines different from the tasks' period [Leung & Merrill 80]. Mok considered tasks that were not purely periodic [Mok 83]. Lastly, Locke and Jensen generalized the concept of a deadline and considered the problem of scheduling to maximize the value of the computations performed by tasks [Jensen et al. 85]. The execution of periodic tasks on multiple processors has been considered by several researchers [Dhall & Liu 78, Leung & Whitehead 82, Bertossi & Bonuccelli 83].

A more traditional variant of the problem of scheduling tasks with deadlines concerns scheduling real-time tasks that make only a single request for execution. Results for this problem have been reported in [Garey et al. 81], [Frederickson 83], [Garey & Johnson 77], and [Mok & Dertouzos 78]. We believe that the models of repetitive task execution are better suited to the study of real-time systems.

The results of these studies have indicated that optimal uniprocessor scheduling policies exist for periodic tasks with deadlines. In most cases *earliest deadline first* scheduling has been shown to be optimal. When multiple processors are introduced, the scheduling problem is frequ tly intractable. In general, the analysis presented in these works assumes that ta are do not communicate, have precedence constraints, or share data (execute critica ctions). More pragmatic models of a real-time systems have been developed separately by Leinbaugh, Harter, and Zhao [Leinbaugh 80, Leinbaugh & Yamini 86, Harter 87, Zhao et al. 87a].

Leinbaugh proposed a very general model for the execution of tasks with resource and I/O requirements. He demonstrated sufficient conditions for tasks in his model to meet their real-time requirements. Extensions to this model for a specific high-level language kernel have been reported by Stoyenko [Stoyenko 87a]. Zhao et al. have developed heuristic methods for scheduling and resource allocation in distributed systems. For systems statically partitioned into levels according to process priority, Harter has developed a method for deriving response times for processes in each level.

These latter models are closer to actual systems but lack much of the rigor of the simpler models. As such, only sufficient conditions are given for determining if a model of a system has the desired real-time behavior. However, if these conditions are met often enough in practice then this is not a problem.

Our work is closely related to that of Liu and Layland, Leung and Merrill, and Mok. We adopt task models similar to theirs but our focus will primarily be on non-preemptive scheduling problems. We share some of more pragmatic concerns of Leinbaugh and Zhao et al., however our approach will be more formal. For the scheduling and resource allocation policies that we develop, we present proofs of correctness, optimality, and complexity.

1.3.2 Design Methodologies

Methodologies have been developed for the design and specification of real-time systems. Examples of research in this area include MASCOT [Simpson & Jackson 79], DARTS (Design Approach for Real-Time Systems) [Gomaa 84], and Statecharts [Harel 87].

MASCOT and DARTS both view a real-time system in terms of processes and communication channels. MASCOT is closer to a programming system than DARTS. MASCOT processes communicate via *pools* and *control queues*. Pools represent shared data and control queues are similar to Hoare style condition variables [Hoare 74]. DARTS is concerned more with the functional behavior of the system and the problem of structuring a system into a set of tasks and defining their interfaces. DARTS allows for time critical functions to be placed in special high priority tasks. In addition, DARTS tasks can have periodic executions. Both systems have a graphical notation for displaying designs. In this regard our notation for specifying designs will be similar.

Statecharts is a finite state machine based formalism for the specification and design of *reactive systems*. Reactive systems are loosely defined as event driven systems and include many real-time systems. Statecharts add concurrency, abstraction/refinement hierarchies, and communication to finite state machines. Time-outs and delays on actions and event occurrences can be specified.

None of these systems deal directly with timing issues. In contrast, the emphasis in our design discipline will be concentrated on the specification and analysis of the real-time behavior of the systems being designed.

1.3.3 Programming Languages

A common approach to the study of hard-real-time systems has been a language based approach. Early real-time languages such as TOMAL [Hennessy et al. 75], Modula [Wirth 77a, 77b], and Pearl [Martin 78], primarily addressed the shortcomings of existing languages for dealing with concurrency and physical I/O. In its time, each language was novel for the inclusion of processes, modules, and synchronization primitives tied to hardware interrupts. With these features, a programmer could implement an entire system maintaining a high level of abstraction. This greatly simplified the construction of real-time systems. What was missing from early real-time languages was a framework for reasoning about the real-time behavior of programs written in the language. A notable exception was the language Modula.

Modula has a special process called a *device process* that can be used to perform physical I/O. Wirth showed how the delay a device process can experience while waiting to respond to an interrupt could be bounded [Wirth 77c]. If this delay is less than the interarrival time of the I/O interrupt then, in principle, the interrupt can be serviced in real-time. However, this analysis was only approximate and was limited to device processes. One could show that data entered the system in real-time but could not show that the data was processed in real-time. The real-time behavior of ordinary processes was not addressed.

Contemporary languages for real-time computing, such as Real-Time Euclid [Kligerman & Stoyenko 86], and Ada [US DoD 83], have retained much of the flavor of the earlier real-time languages. In terms of increasing the ability of the programmer to understand the real-time behavior of a program, Real-Time Euclid represents a significant improvement over the earlier languages while Ada, at best, represents no improvement.

Real-Time Euclid is an extension of the language Concurrent Euclid for real-time processing. Concurrent Euclid programs are composed of modules, monitors, and processes [Holt 83]. Real-Time Euclid re-engineers these basic components to address real-time concerns and adds interrupt and exception handling. Processes in Real-Time Euclid have the ability to be activated by external events or to be activated periodically by a clock. Processes can also have deadlines that stipulate they will complete execution so many time units after they are initiated. With these features, the analysis possible for device processes in Modula can be extended to normal processes in Real-Time Euclid. Another notable feature of Real-Time Euclid is the requirement that all uses of sequential language constructs have a bounded execution time. These bounds are enforced at run time. The Real-Time Euclid compiler includes an elaborate simulator that will determine, for all possible interleavings of processes, whether or not processes will meet their deadlines [Stoyenko 87b].

Ada provides little help for reasoning about the real-time behavior of programs. Ada tasks can be activated by interrupts but there are no mechanisms for ensuring the performance of these tasks. Ada provides a facility for tasks to suspend (delay) execution but the semantics of this operation leave unspecified the maximum duration that a task will be suspended. Many Ada compilers provide PRAGMAs ("hints" to the run-time system) for real-time scheduling of tasks, however, these are not part of the Ada language.

Conceptually, Ada and Real-Time Euclid extend the earlier real-time languages by creating elaborate versions of existing program constructs. These constructs have necessitated aggressive compiler and run-time system implementations. An alternate approach has been to formulate new abstractions and semantics for real-time programming. Languages such as ESTEREL [Berry & Cosserat 85], and SIGNAL [Gautier et al. 87] have proposed a new paradigm for programming and reasoning about real-time systems. ESTEREL and SIGNAL are termed *synchronous* languages and present the abstraction that ordinary sequential programming constructs always execute in zero time. In the case of ESTEREL, the abstractions of synchrony and zero execution time result in a simple and elegant language. An ESTEREL program consists of a set of processes that communicate via message passing. Individual messages are sent and received in the same instant (synchronously and in zero time). With these semantics, the ESTEREL compiler uses a novel compilation strategy to transform a set of communicating processes into a large finite state machine. State transitions are performed on the arrival of external inputs. States correspond to the processing that all the processes in the original program would have performed had they been in a particular state when the input arrived. If the worst case time required to perform the computations in each state, is less than the minimum interarrival time between any two inputs, then the abstraction of zero execution time is valid.

While we are not developing a programming language, our work is influenced by several of the motivating principles behind the languages Modula and ESTEREL. These influences will be discussed in more detail in Chapter 2 when we present our design discipline.

1.3.4 Programming Systems

Rather than defining a new programming language for developing real-time systems, many researchers have chosen to develop *programming systems*. Programming systems differ from programming languages in that they are primarily concerned with the integration of program components developed using traditional sequential languages. A programming system can be viewed as the definition of a programming discipline for a particular language. Examples of such systems include the the Stream Machine [Barth et al. 85], RNet [Coulas et al. 87], the SARTOR environment [Mok et al. 87], and GEM [Schwan et al. 87]. A common focus of these systems is the development of process communication and synchronization mechanisms for real-time processes developed in a sequential language.

The Stream Machine is a data flow programming system developed for data acquisition and process control applications. The basic components of the Stream machine are *modules*, *streams*, and *piers*. Modules are sequential programs. Streams are time ordered sequences of values used for communication between modules. Piers are a module's tap onto a stream. Piers read values from and write values to a stream. Modules can be annotated to specify timing constraints on pier operations. The mechanisms for ensuring these constraints are met are not discussed.

RNet is a system for constructing distributed real-time systems. An RNet program consists of a set of program modules written in Concurrent Euclid, and a specification of real-time constraints on the execution of the modules. Processes in modules communicate via message passing operations. Processes receive messages on ports that can have a deadline. The deadline specifies an upper bound on the duration that a message can reside at a port without being processed. Some heuristics, and programming guidelines, are presented for ensuring that an RNet program will satisfy the timing constraints in its specification.

The SARTOR system is a CAD environment for synthesizing real-time software. A real-time system is represented as a directed acyclic graph of computation and I/O nodes. Nodes communicate along edges by exchanging messages. Given the processing costs of the nodes, the system can be statically analyzed to determine if each node satisfies a set of execution constraints. The constraints are oriented toward guaranteeing that no data is lost in the system.

GEM is an operating system kernel and set of programming primitives used to construct control systems for robots. GEM supports processes and micro-processes (threads of control within a single process). Deadlines can be associated with processes. Inter-process communication is via message passing. GEM supports three paradigms of interaction: *asynchronous execution with data loss*, *synchronous execution without data loss*, and *synchronous or asynchronous operation with possible loss of aged data.*

Although numerous references are made to the existence, and importance, of strict timing constraints, no method for determining if a program will satisfy these constraints is given.

Of each of these systems, the SARTOR environment is most closely related to the our design discipline. Both use a graphical and graph based notation and both use a similar underlying tasking model. A detailed comparison between the two will be postponed until Chapter 6.

1.3.5 Summary

In summary, there have been many approaches to the problem of describing, constructing, and analyzing hard-real-time systems. Our research will most closely resemble the work in design disciplines, programming systems, and analytical models. Our work departs from the majority of the cited references in that we will concentrate on methods for describing and analyzing the real-time behavior of systems. Our descriptions will be implementable and under a range of conditions, we will claim that our strategies for implementing real-time systems are nearly optimal and quantitatively better than existing methods.

1.4 Thesis Overview

In the following chapter we present our discipline for designing hard-real-time systems. The notation and semantics of process communication via message passing are defined. A method for reasoning about the real-time behavior of designs is also presented. The notation is a graphical and graph based notation. The semantics of message passing are understood in terms of the rates at which processes exchange messages. With these semantics one can determine best case upper and lower bounds on the propagation times of messages along paths in a graph corresponding to a design. In this manner the real-time behavior of the design is completely understood in terms of the rates at which messages are sent. A design for a digital stopwatch is used to illustrate these concepts.

Having defined the discipline, there are two problems to investigate. The first problem is to determine whether or not the semantics of process communication we define can be implemented. The second issue concerns the utility of the notation. The former issue is the subject of Chapters 3 through 5. Chapter 3 presents an abstract model of a design in terms of cyclic tasks. It investigates the problems of preemptive and non-preemptive scheduling of cyclic tasks on a uniprocessor. For sporadic and periodic workload characterizations, the *earliest deadline first* scheduling policy is shown to be an optimal policy whenever one exists. The complexity of determining whether a set of cyclic tasks can be scheduled is also examined. We show that one can always efficiently determine if a set of sporadic tasks can be scheduled on a uniprocessor. For periodic tasks the problem of determining if a set of tasks can be scheduled non-preemptively is equivalent to determining if P = NP.

Chapter 4 generalizes the scheduling problems of Chapter 3 to include a set of resources which are shared among the tasks. We develop the concept of an *implementation strategy* to solve scheduling and synchronization problems for sporadic tasks with resource requirements. For several characterizations of a task's resource requirements, the integration of an *earliest deadline first* scheduling policy with a WAIT and BROADCAST synchronization discipline is shown to lead to an optimal implementation strategy.

Chapter 5 examines the reduction to practice of the task models of the previous two chapters. It applies the results of the formal model to the implementation of designs constructed using the discipline. The goal of Chapter 5 is to determine when and how a design can be implemented such that the behavior specified by the semantics of message passing is guaranteed to occur. This problem has two components which we call *processor independent* and *processor dependent*. The processor independent component refers to the fact that certain cyclic designs can never be implemented. A characterization of the properties of cycles which render a design graph incapable of being implemented is developed. The processor dependent analysis consists of mapping processes in a design into an instance of the tasking model studied in the previous chapters.

The cumulative result of Chapters 3 through 5 show that we can efficiently and constructively determine when a design will adhere to its specification. This will demonstrate that the notation is simple enough to be realized.

Concerning the use of the notation, Chapter 6 presents the results of some paper and laboratory experiments with the notation. Our experiences with a prototype implementation of a programming system designed to support the abstractions of our discipline are discussed. We next examine and present designs for four actual real-time systems. In all cases we will argue that the design discipline is both expressive enough to succinctly describe the real-time behavior of systems, and that it allows the efficient analysis of interesting and important properties of the system. Chapter 7 presents our conclusions and reviews our contributions.

Chapter 2

A Message Passing Design Discipline for Hard-Real-Time Systems

2.1 Introduction

Hard-real-time computer systems have been defined as the class of systems that require strict adherence to timing constraints. In order to understand the problems of hard-real-time computing we therefore must understand the concept of a timing constraint. There are at least two broad classes of timing constraints: *performance constraints*, that define limits on the system's response time to events, and *behavioral constraints*, that define the rates at which external agents generate inputs to the system [Dasarathy 85]. It is our thesis that timing constraints can be understood in terms of producer/consumer relationships on system components. Whenever a system component *produces* an event, such as an interrupt or a message, the event must be responded to, or *consumed*, within a well-defined period of time. Since system components may be software or hardware (or even human) objects, producer/consumer relationships provide a uniform mechanism for specifying both behavioral and performance constraints.

In this chapter we propose a model of hard-real-time computing based upon a paradigm of process interaction called the *real-time producer/consumer paradigm* (RT/PC). This paradigm of interaction serves as the semantic basis for a message passing design discipline for real-time systems. We begin with a more detailed discussion of an origin of timing constraints. Section 2.2 motivates our use of producer/consumer relationships to express timing constraints and formally defines the RT/PC paradigm. Sections 2.3 and 2.4

describe the syntax of our discipline and the rules for constructing designs. Section 2.5 presents our semantic model. The semantics are based on the RT/PC paradigm and are expressed in terms of the rates at which messages are exchanged. Section 2.6 discusses the derivation of these rates.

To illustrate this discipline, we present a prototypical design for a digital stop-watch in Section 2.7. Section 2.8 shows how one can reason about real-time properties of systems constructed using our discipline.

2.2 An Origin of Timing Constraints

We have argued that one of the salient features of a hard-real-time system is the existence of a set of timing constraints that must be adhered to if the system is to be considered correct. In this section we demonstrate how interactions with processes in the external world naturally motivate a class of timing constraints. We also show how these constraints can be specified in terms of producer/consumer subsystems.

2.2.1 Producer/Consumer Systems

In a traditional multiprogramming system, a standard paradigm of process interaction is the *producer/consumer* paradigm. A producer/consumer system consists of two agents a *producer* and a *consumer*. The *producer* produces data objects that are consumed by the *consumer*. The key problem in a producer/consumer system is to synchronize the *producer* and the *consumer* so that all objects produced by the *producer* are ultimately consumed by the *consumer*. We will refer to producer/consumer systems that have this property as *correct producer/consumer systems*.

Consider the canonical implementation of a producer/consumer system shown in Figure 2.2.1 below. Here the *producer* and *consumer* are software processes that are synchronized by a monitor [Hoare 74]. The *producer* process sends data to the *consumer* process through an intermediary monitor. The monitor implements a set of data buffers, and provides routines for synchronizing access to the buffers. The monitor is provided so that the processes need not proceed in lock step.

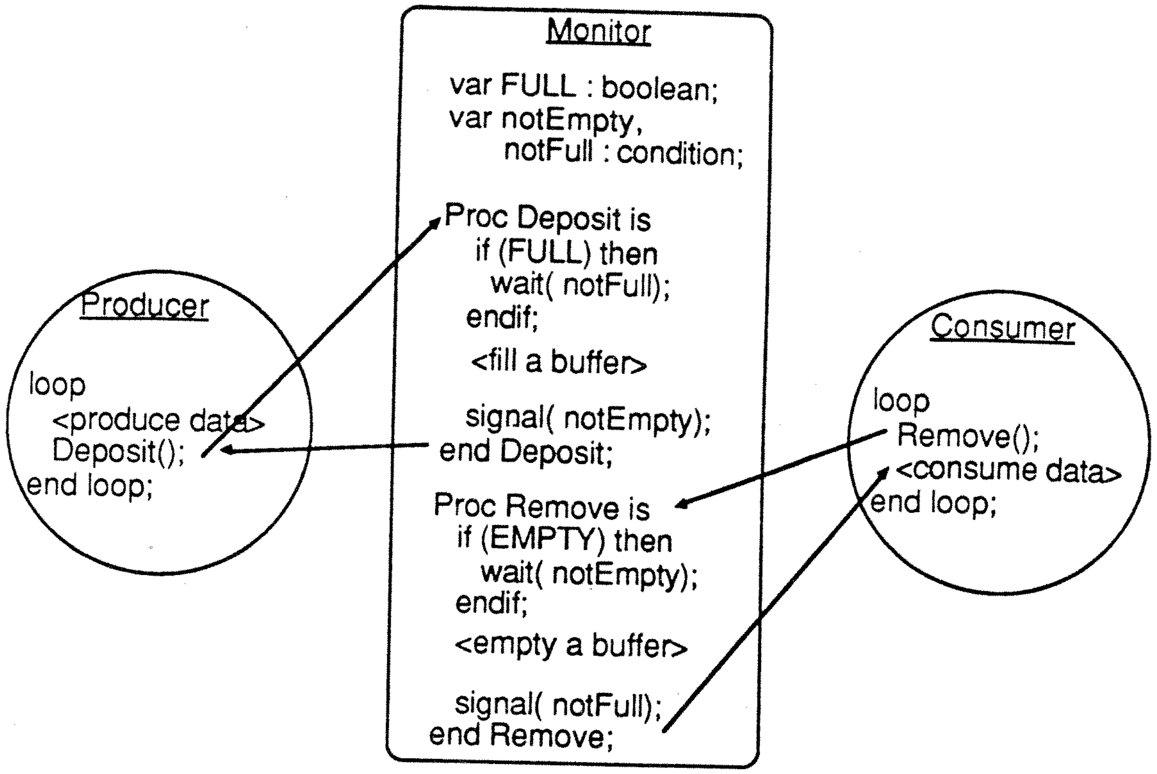


Figure 2.2.1: A producer/consumer system. Producer and consumer are software processes. Assume all buffers are initially empty.

For our purposes, we can think of the *producer* executing an infinite loop, and creating an infinitely long sequence of data items. When the *producer* has produced a data item, it calls the Deposit monitor entry to buffer the data item in the monitor. If the *producer* produces data items faster than the *consumer* consumes, then the buffers in the monitor will eventually fill up and the *producer* will be forced to wait until an empty buffer becomes available. After the *producer* has deposited its data item, it will execute the signal operation to alert the *consumer* that there is at least one full buffer ready to be consumed. If the *consumer* was waiting on the condition variable notEmpty, then the *consumer* will wake up and resume execution. If the *consumer* was not waiting then the signal has no effect.

Conceptually, the *consumer* is also executing an infinite loop, consuming the *producer*'s infinite sequence of data items. The *consumer* acquires data items by calling the Remove monitor entry. If ·re are no full buffers rea·v for the *consumer*, then the *consumer* will wait until it is sigr ·ed by the *producer* that · iffer has been filled. When the *consumer* does obtain a full buffer, it will signal the p· ·ucer to inform it that there is at least one empty buffer available. If the *producer* was waiting on the condition variable notFull, then the *producer* will wake up and resume execution. Otherwise the signal has no effect.

In principle, since the *producer* will wait for the *consumer* when all the buffers are full, and since the *consumer* executes an infinite number of remove operations, all data items produced by the *producer* are eventually consumed by the *consumer*. Therefore, this monitor-based implementation of a producer/consumer system will be correct. In this implementation, the semantics of the synchronization primitives, and a description of the algorithms used in the deposit and remove routines, are sufficient for reaching this conclusion. In particular, the correctness of the monitor-based implementation will not be dependent on the number of data items that *producer* produces, the number of buffers implemented in the monitor, or on the relative speeds at which the processes make progress.

2.2.2 Real-Time Producer/Consumer Systems

Processes often interact with the external world. Abstractly, we can model these interactions using the same producer/consumer scenario. For example, the *producer* could be a process in the external environment that sends inputs to a software *consumer* process. To reason about the correctness of the above system, we relied on the fact that the *producer* was forced to wait for the *consumer* if the *consumer* ever lagged too far behind. If the *producer* is a process external to the computer, there is no reason to believe that this external process can be forced to wait for the *consumer*. The external process may not be under the control of our computer. For example, the *producer* may be a device that samples and digitizes an analog signal. The digitized samples are then input to the *consumer*. For many applications such as speech processing, it is reasonable to expect that the sampling rate of the device is fixed in the hardware of the device and cannot be altered by the computer. Therefore, if the *consumer* is not fast enough, data from this external *producer* will be lost.

If the *producer* can never be made to wait for the *consumer* then the producer/consumer system above reduces to the one shown in Figure 2.2.2 below [Wirth 77c]. This is similar to the previous scenario except that now the wait operation has been removed from the deposit routine. In addition, since the *producer* never waits, there is no need for the *consumer* to ever to signal it. Hence the condition variable notFull can also be eliminated.

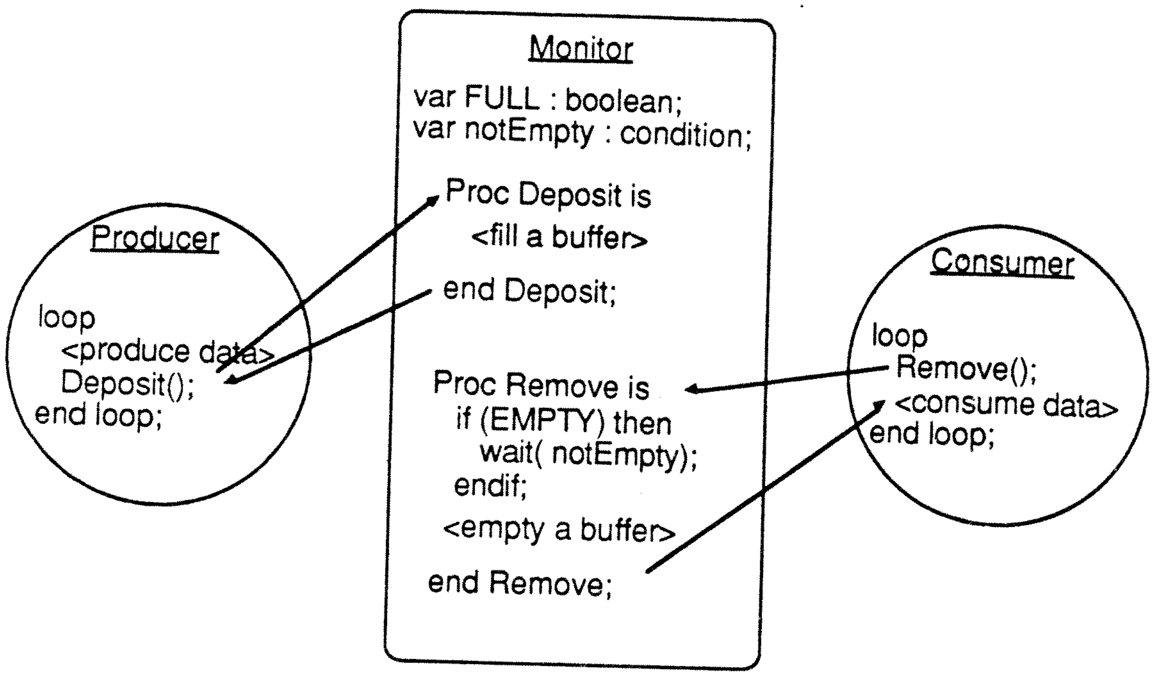


Figure 2.2.2: A real-time producer/consumer subsystem. The *producer* is a hardware process and the *consumer* is a software processes.

In this case, knowledge of the semantics of synchronization, and the buffering algorithms of the monitor, are not sufficient to determine if all data produced will be consumed. In order to determine the correctness of this system, we will need information on the implementation of the *consumer* and on the behavior of the environment. At a minimum we will need to know the rate at which the *producer* produces data items and how much time, in the worst case, it takes the *consumer* to process a data item.

The producer/consumer system in Figure 2.2.2 is called a *real-time producer/consumer* (RT/PC) *system* since the correctness of this system now has a temporal component. Since the behavior of the external processes is fixed, there is a timing constraint on this system which specifies that the processing of the *consumer* never lag behind the *producer* for more than *n* data items, where *n* is the number of buffers in the monitor. For the monitor-based implementation, satisfying this constraint is a necessary and sufficient condition (assuming the hardware does not fail) for ensuring no loss of data.

2.2.3 The Real-Time Producer/Consumer Paradigm

There is a more general statement of this timing constraint that is valid for any implementation scheme. If we ignore the specifics of the monitor implementation, we can identify the following salient features of the RT/PC system of Figure 2.2.2. Abstractly, we have two processes, a *producer* and a *consumer* that are connected via a unidirectional communication channel as shown in Figure 2.2.3 below. When the *producer* produces a data item, the *producer* sends the data item to the *consumer* on the channel.

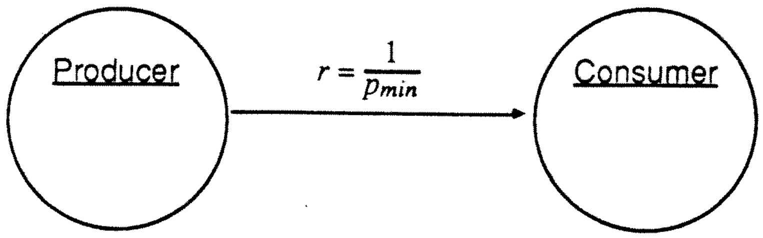


Figure 2.2.3: An abstract real-time producer/consumer system.

The channel is unidirectional, from the *producer* to the *consumer*, since the *consumer* cannot communicate with, or control the *producer*. We will assume that this communication channel imposes no delay on the flow of information from the *producer* to the *consumer*. That is, we will assume that data items arrive at the *consumer* immediately after they are sent by the *producer*. The rate r at which data items travel on the channel from the *producer* to the *consumer* can be measured in terms of the worst case minimum inter-arrival time, p_{min}, of data items at the *consumer*. For this abstract producer/consumer system we make the following observation:

> Assume the worst case rate r is realizable over an *arbitrarily long* interval of time. If the consumer does not consume data items at rate r, then there is *no* amount of buffering that we can impose between the *producer* and the *consumer* to ensure that every data item produced by the *producer* is consumed by the *consumer*. To guarantee the correctness of the system, it is necessary for the *consumer* to consume data items *at the rate at which they are produced.*

Based on this observation we define a paradigm of process interaction called the *real-time producer/consumer paradigm*. The RT/PC paradigm states that in a producer/consumer system:

> the i^{th} output of the *producer* must be consumed by the *consumer before* the $(i+1)^{st}$ output is produced.

The RT/PC paradigm stipulates that the *consumer* must execute in time frame defined by the *producer*. Within this time frame, the processing of data items takes *no time*. That is, if the outputs of the *producer* are "ticks" of a discrete, virtual time clock, then the time required for the processing of data items by the *consumer* cannot be measured with this virtual time clock. Relative to this clock, the processing of a producer's outputs always takes zero time. Therefore, from the perspective of the *producer*, the *consumer* appears to be infinitely fast. The *producer* cannot tell the difference between the *consumer* and a process that *is* infinitely fast.

The RT/PC paradigm requires that the abstract producer/consumer system of Figure 2.2.3 must function correctly with zero buffers if it is going to function correctly at all. This requirement, however, is not meant to imply that buffers are not useful. For example, the *producer* and *consumer* may be out of phase with each other and therefore a single buffer may be used *by an implementation* to synchronize their actions. In addition, if the worst case rate *r* is not realizable over on an arbitrary interval then buffers may be used by an implementation to temporally overcome a burst in input arrivals. However, in all cases, the fundamental abstraction required to ensure the correctness of a RT/PC system is that data is consumed at the *rate* at which it is produced.

2.3 System Components and Graphical Notation

It is our thesis that the real-time behavior and requirements of systems can be easily understood, and succinctly specified, by formulating process interactions in terms of the RT/PC paradigm. In this section we present the syntax of a message passing design discipline in which the RT/PC paradigm serves as an operational semantics for communication among a set of processes. With this discipline one can specify timing constraints and derive temporal properties of a design without requiring a knowledge of an implementation.

In our discipline, designs are expressed using a graphical and graph notation. A real-time system is a directed graph $G = (V,E)$ where V is the set of vertices and E is the set of edges. Vertices are program components or external devices and edges are unidirectional channels of communication between these components. A vertex is either a *process, data repository, input device* or *output device.* Program components communicate by sending and receiving messages on channels. Edges can either be *synchronous channels* or *asynchronous channels.* Each type of vertex and edge is represented with a unique graphical icon. A sample design graph is shown in Figure 2.3.1. Vertices and edges are typically labeled with text names.

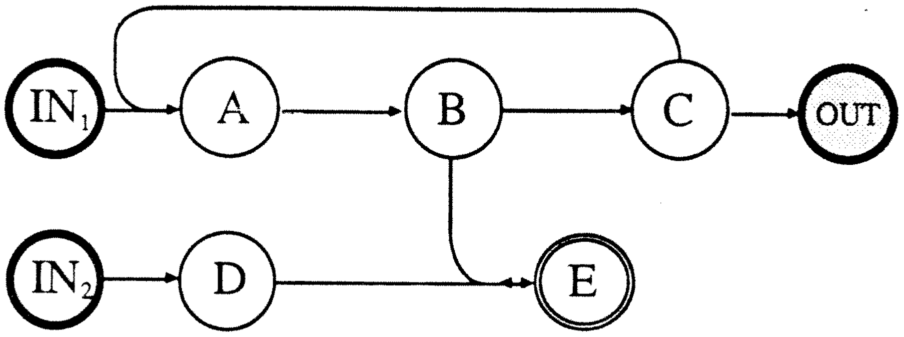


Figure 2.3.1: A sample design graph for an imaginary real-time system.

2.3.1 Processes

Graphically, a *process* is represented with a circle. The structure of a *process* consists of a single message input port, a set of message output ports and a body of sequential program code. A schema for a *process* is shown in Figure 2.3.2. Conceptually, a *process* is always ready to process a message. A *process* accepts a message from its input port using the ACCEPT statement, processes the message and then waits at the ACCEPT statement to receive its next message. In the course of processing a message, a *process* may emit messages to other processes, devices, or data repositories. The A_EMIT statement is used to send a message on an *asynchronous channel,* and the S_EMIT statement is used to send a message on a *synchronous channel.*

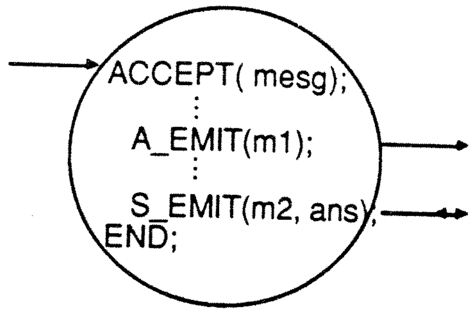


Figure 2.3.2: A schema for a *process.*

The input and output ports of a *process* are statically bound to communication channels. Processes send and receive messages from other processes and devices on asynchronous channels. Asynchronous channels are represented graphically with a single headed arrow. When a *process* sends a message on an *asynchronous channel,* it neither waits for, nor receives, a response. Processes send messages to data repositories on synchronous channels. Synchronous channels are represented graphically with a double-headed arrow.[1] When a *process* sends a message on an *synchronous channel* it will wait for a reply from the receiver. All channels have the property that all messages sent are received. This will be explained in more detail in Section 2.5.

There are two structural restrictions on the interconnections of processes. These restrictions are:

- A *process* may have any number of synchronous or asynchronous output channels; however, no two asynchronous output channels of a process may have the same receiver.[2]

- There must exist a path from an *input device* to each process in the system.

The first restriction is imposed to simplify the implementation of the discipline. The second restriction will simplify the analysis of designs that follows. Since processes only emit messages when they receive a message, a *process* that is not on a path originating at an

[1] The double-headed arrow is simply the icon for a synchronous channel. In terms of the graph, a double-headed arrow represents a *single* edge whose direction is *from* the process *to* the data repository.

[2] A process may have zero output channels, however, this process will not perform any useful function.

input device will never do anything. Designs that meet both criteria are said to be *well-formed.*

2.3.2 Data Repositories

A *data repository* encapsulates persistent data or data that is shared between processes. By persistent data we mean data whose values are required by a *process* to be retained between the consumption of two or more messages. Graphically, a *data repository* is represented with two concentric circles. The structure of a *data repository* consists of a single input port and a body of sequential program code. For simplicity, we do not allow a *data repository* to emit messages. A schema for a *data repository* is shown in Figure 2.3.3. Like a *process*, a *data repository* is conceptually always ready to process a message. A *data repository* accepts a message from its input port, processes the message, sends a reply to the sender, and then waits to receive its next message. *Data repositories* receive messages from processes on synchronous channels.

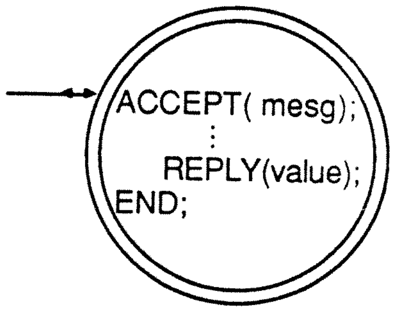


Figure 2.3.3: A schema for a *data repository*.

2.3.3 Input Devices

An *input device* is a specification of the behavior of a physical input device. Input devices are the ultimate sources of all messages in the system. Graphically an *input device* is represented with darkened circle. Conceptually, we view an *input device* as an external process that periodically emits messages to a *process* on an asynchronous channel. An *input device* has only a single output port. A schema for an *input device* is shown in Figure 2.3.4.

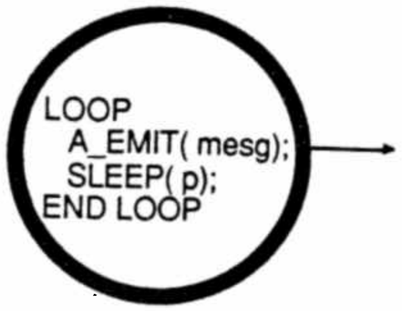


Figure 2.3.4: A schema for an *input device*.

We will assume that input devices pause for some non-zero length duration between emitting messages to the system. The duration of this pause need not be a constant, however, we assume that its minimum length is known. This minimum duration is specified by the parameter *p* in the SLEEP meta-operation above. The semantics of the SLEEP operation are that the *input device* will pause for a interval of length at least *p* time units. Requiring devices to pause ensures that no two messages can be emitted simultaneously. There will always exist a minimum separation of *p* time units between any two message emissions.

For certain devices it is quite possible that the worst case minimum inter-arrival time of messages is not known. In this case we have two alternatives. One choice is to treat the inter-arrival time as a variable and then when analyzing an implementation of the design, derive bounds on the inter-arrival time that the implementation can support. A second choice is to specify a value for the inter-arrival time corresponding to a rate of service known to be acceptable for the application. Both of these alternatives will be treated in greater detail in Chapters 5 and 6.

2.3.4 Output Devices

The final type of vertex is called an *output device*. An *output device* is a specification of the behavior of a physical output device. Conceptually, an *output device* is a *process* that is always ready to receive a message. Like a *process*, an *output device* also has a single input port and receives messages on an *asynchronous channel*. Graphically an output device is represented with a darkened and shaded circle as shown in Figure 2.3.5.

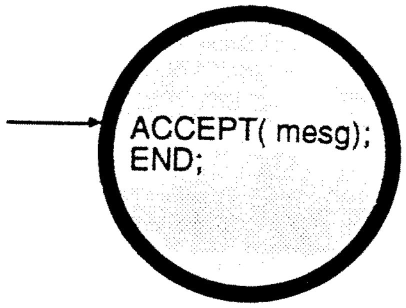


Figure 2.3.5: A schema for an *output device.*

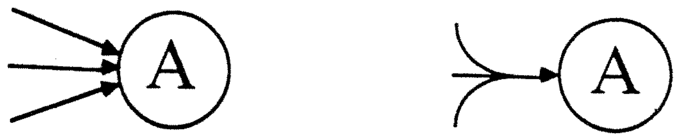
We will require that only one process send messages to an *output device.* If more than one *process* wishes to share an *output device,* then all processes should route their message to the device via a central "driver" process. This restriction is imposed for technical reasons that will become apparent in later chapters. In essence, it is a reflection that access to output devices often must be serialized.

We will frequently use the generic term *asynchronous process* to refer to both devices and processes. These components are called asynchronous processes since they are logically processes and since they are the only components that emit and receive asynchronous messages.

2.3.5 Channels

A *channel* is a connection between the output port of a sender and the input port of a receiver. For a given design, the bindings between output ports and input ports are permanent. While it is possible for a *process* or *data repository* to receive messages from several physical channels, conceptually, each process and data repository receives messages from a single logical input channel. If a process receives messages from multiple sources, then when the process accepts a message, the process does not know from which source the message came. (If the identity of the message sender is important then it can always be encoded into the text of the message.) When representing a node that receives messages from multiple sources, we will use one of two forms of notation shown in Figure 2.3.6. The two styles of notation are equivalent but will be used to emphasize certain points. The form in part (a) of the figure represents the point of view of the message senders: that each message sender has a dedicated channel. The form in part (b) represents the view of the message receiver: that each message receiver has only a single input port.

In both cases the notations represent three separate asynchronous channels connected to a single process.



a) Message senders' view. b) Message receiver's view.

Figure 2.3.6: Equivalent notations for a process that receives messages from multiple sources.

By maintaining a communication abstraction of a single sender and dedicated receiver for each channel, and a single logical input channel for each process and data repository, the behavior of each system component will not be dependent on how it is used (connected) in a design. This will aid in both the construction and analysis of designs.

There is one subtlety in these abstractions for data repositories. Since data repositories must reply to each message, a *data repository* must know the identity of the sender. In the case of a data repository which receives messages from multiple sources, we will assume that an implementation of our message passing discipline always replies to the correct party.

2.3.6 Mutual Exclusion Regions

The final aspect of our design discipline deals with mutual exclusion. We distinguish between two origins of mutual exclusion problems: *explicit* and *implicit* mutual exclusion. In our notation, mutual exclusion may be specified explicitly by grouping together a set of of processes or data repositories with a box as shown in Figure 2.3.7. The two processes and data repositories in Figure 2.3.7 are each said to be in a *mutual exclusion region*.

The box signifies that the processing of messages at these nodes constitutes a critical section. When a node in a mutual exclusion box commences processing a message, no message may be processed at any other node in the box until the first message has been nsumed. Processes and data repositories may not be combined in a mutual exclusion and each may appear inside at most one box.

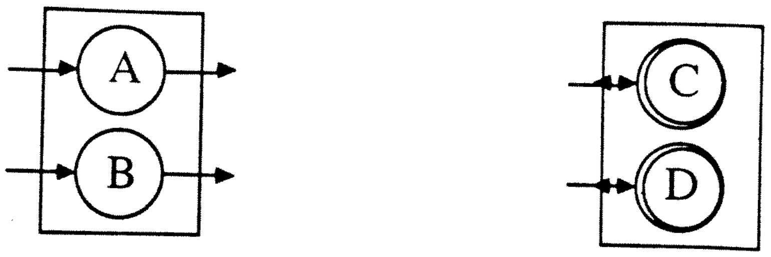


Figure 2.3.7: Explicit mutual exclusion regions.

Mutual exclusion is implicitly specified in the case of processes or data repositories that receive messages on multiple physical channels.

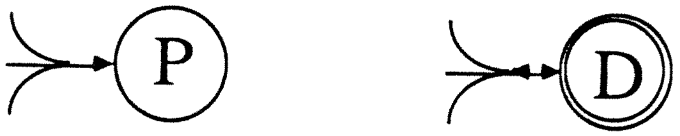


Figure 2.3.8: Implicit mutual exclusion regions.

Each process and data repository logically performs a single function. To ensure the consistency of these operations, we require that the processing of messages from different sources not proceed simultaneously at these nodes.

2.4 Constructing Processes and Data Repositories

Processes and data repositories are programmed using a sequential programming language extended to include the message passing operations ACCEPT, S_EMIT, and A_EMIT. We will impose several constraints on the use and implementation of the sequential language as well as on the use of message passing operations. These constraints are primarily oriented towards ensuring that the time required in the worst case to process a message at a process or data repository can be computed statically. This information will be needed in Chapter 5 when we discuss implementation strategies for our discipline. Simply put, in order to determine how a design performs in real-time, we must know how long it takes to perform basic operations such as consume a message.

We will require that the sequential programming language be used and implemented in such a manner that the execution time of each instance of each language construct can be statically determined. We define the execution time of an instance of a construct to be the time required to execute the construct to completion on a dedicated processor. In the case of constructs with data dependent execution times, the worst case execution time of each use of these constructs must be specified. For constructs, such as loops, with potentially infinite execution times, mechanisms to ensure that a specified worst case execution time is not violated must be employed. For example, a traditional *while loop* must be augmented so that it contains a loop counter and so that the loop termination condition is a conjunction with a clause specifying an upper bound on the number of loop iterations. This upper bound should be a constant or an expression whose value can be bounded without executing the system.[3] With such a programming language one can determine the time required in the worst case to execute the sequential code required to process a message.

The time to process a message will also be dependent on the time required to send and receive messages. In general, the cost of these operations will be dependent on the size of the message being sent. We assume that messages are typed objects and that their sizes are bounded. For the present, we will not concern ourselves with the transmission time of a message. We will simply assume that messages arrive immediately after they are sent. This is a reasonable assumption for implementations on a uniprocessor as well as on tightly coupled architectures.

A final restriction deals with the dynamic use of asynchronous channels. During the processing of a message, a *process* may emit at most one message on each asynchronous output channel. This restriction, combined with those in Section 2.3.1, ensure that a process will send at most one message to an other process, or device, at a time. Given the semantics of message passing defined in the next section, this restriction should not be too severe. For example, if during the course of processing a message, a *process* desires to send multiple messages to another *process*, it will be able to achieve this effect by combining the messages into a single message and sending this larger message.

2.5 Message Passing Semantics

The unique feature of our design discipline is the temporal semantics of message passing. Every asynchronous channel has a producer and a consumer of messages. *In our design*

[3] For a description of a language with similar statically determinable execution times see [Kligerman & Stoyenko 86].

discipline, all interconnected pairs of asynchronous processes (both internal processes and external devices) obey the RT/PC paradigm. Whenever a message is produced on a channel, the receiving process will consume the message *before* the next message is produced on that channel. All messages in the system are therefore consumed in a time frame defined by the message producers.

With the process construction rules from the previous section, we will be able to quantify this time frame for each message producer. For each asynchronous channel, we will specify an upper bound on the time allowed to consume a message sent on that channel. These bounds will be expressed in terms of the rate at which messages are emitted on a channel. In this manner, the temporal behavior of a design can be characterized completely by the rates at which messages are sent on channels. In this section we define the behavior of processes symbolically in terms of rates. In the following section we show how these rates are derived.

2.5.1 Message Transmission Rates and Inter-arrival Times

For each asynchronous channel in a design graph we will define a *message transmission rate* in terms of the worst case minimum message inter-arrival time at the channel's receiver. Let A and B be two inter-connected processes in a design graph. If p_B is the worst case minimum inter-arrival time of messages at process B from process A, then the message transmission rate, r, on the asynchronous channel connecting them is simply

$$r = \frac{1}{p_B}.$$

This message transmission rate will represent the worst case maximum arrival rate of messages on the channel.

We will assume that p_B is an integer. Conceptually, our design discipline is built upon a discrete time foundation. We choose discrete time because in a computer system, all time references appear as a sequence of discrete events. A fundamental assumption of our discipline is that there exists a basic indivisible time unit that is small enough that the values of all time related constants and variables, can be either be directly expressed as, or reasonably approximated by, integer multiples of this unit. Therefore, without loss of generality, we will assume throughout this dissertation that all time variables and constants take on integer values. Hence, in our discrete time domain, message inter-arrival times such as p_B, are assumed to be integers. Likewise, all message transmission rates are considered to be the reciprocal of an integer.

2.5.2 Asynchronous Channel Semantics

Under the RT/PC paradigm, a producer is presented with the abstraction that its message consumers are infinitely fast. While an infinitely fast consumer is sufficient for satisfying the RT/PC paradigm, it is not necessary. For the producer and consumer of messages on an asynchronous channel, the semantics of message passing are defined in terms of the message transmission rate of the channel. When a message is sent on a channel whose transmission rate is r, the producer of the message is *guaranteed* that its message will be consumed within $1/r$ time units of being sent. This is a necessary condition for satisfaction of the RT/PC paradigm. If a consumer can make this guarantee, then a producer will be just as content with this consumer as it would have been with an infinitely fast consumer.

In general, this guarantee is a stronger statement than simply stating that the RT/PC paradigm hold on all processes. However, for our design discipline, these semantics are a direct result of requiring adherence to the RT/PC paradigm. To adhere to the RT/PC paradigm, a process must consume every message it receives before the next message is sent on that channel. Since a process in our discipline has no guarantee that there will exist a delay of more than $1/r$ time units between the arrival of any two messages, the processing of messages within $1/r$ time units of their arrival is a necessary condition. Any process that waits for more than $1/r$ time units before consuming a message can be tricked by a malicious message producing process into not consuming a message before the next one arrives.

These semantics constrain the behavior of message consumers so that message producers are provided with guarantees of service. Since message consumers can be external processes (output devices), these semantics constrain the behavior of the external world as well. If an asynchronous channel with a transmission rate of r is connected to an output device, then the output device is required to have a worst case maximum service time of $1/r$ time units.

2.5.3 Synchronous Channel Semantics

The semantics of synchronous message passing follows immediately from the semantics of asynchronous message passing. Before a process can complete its processing of a message, it must receive a response from every data repository it sends a message to. Therefore, if a process must consume a message within $1/r$ time units of its arrival, then all messages sent on synchronous channels must be responded to within $1/r$ time units of their emission. Therefore, unlike processes, the rate at which data repositories must consume messages is not equal to the rate at which messages are sent to data repositories. Data repositories must often consume messages at a faster rate than the rate at which they arrive.

For example, consider the simple design graph shown in Figure 2.5.1. In the worst case process A receives messages every 5 time units and therefore must consume each message it receives within 5 time units of its arrival. Data repository B, on the other hand, only receives messages every 20 units, however, it also must respond to each message it receives within 5 time units. If data repository B took longer than 5 time units to respond then process A could never consume its message in 5 time units.

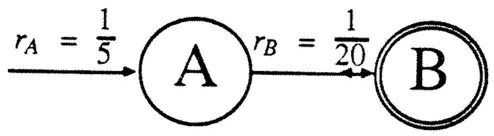


Figure 2.5.1: Message passing semantics for synchronous channels.

2.5.4 Mutual Exclusion

It is important to note that the concept of mutual exclusion is orthogonal to the message passing semantics we have presented. In a design graph, all interconnected asynchronous processes will adhere to the RT/PC paradigm regardless of whether or not their processing of messages must also conform to a mutual exclusion constraint.

2.6 Computing Message Transmission Rates

In order to determine the precise semantics of message passing, the rates at which messages are sent and received on asynchronous channels must be known. In this section we present a method for computing these rates.

2.6.1 Determining Message Output Rates

A channel's transmission rate is simply the rate at which a process emits messages on the channel. The rate at which a process emits messages on a channel is in turn determined by the rate at which the process receives messages. Both rates are defined in terms of the worst case minimum message inter-arrival time at the channel's receiver. For each process in the system, we define a *transmission rate function* for each asynchronous channel on which the process performs output. For a given process's output channel, the transmission rate function is a function f mapping the rate at which messages arrive at the process, to the rate at which the process emits messages on that channel. This is illustrated in Figure 2.6.1. For the purposes of defining the transmission rate, we assume that the processing of each message takes zero time.

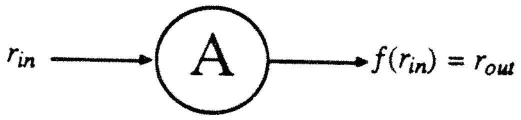


Figure 2.6.1: Message transmission rate function.

The following theorem shows that for our programming model, the transmission rate functions are simple linear functions. It shows that messages cannot be sent at a faster rate than they are received.

Theorem 2.6.1: If transmission rates are defined in terms of the worst case minimum message inter-arrival time, then for the programming model of Section 2.4, a transmission rate function is a linear function through the origin whose slope is a harmonic number. That is, each transmission rate function is of the form:

$$f(r) = \frac{1}{x}r, \quad x \in \mathbf{Z}^{+}.[4] \qquad (2.6.1)$$

Proof: Let A and B be two processes that are directly connected by an asynchronous channel. Let r_A be the worst case rate at which node A receives messages. This rate defines a discrete time frame for process A. In the worst case, A will receive a message every $1/r_A$ time units. Since a process only emits messages when it has received a

[4] $\mathbf{Z}^{+}$ represents the set of positive integers.

message, and since message emissions are discrete events (they either occur or they do not occur), the worst case minimum separation between messages on the channel from process A to process B will be an integer constant, x, times $1/r_A$. Therefore, the worst case transmission rate is $\frac{1}{x} r_A$, where x is a positive integer. Δ[5]

The transmission rate function is determined by examining the logic in each process. Under restricted models of programming, this determination could be automated. For example, if each process is a finite state machine that makes state transitions only upon the arrival of messages, then the denominator of the coefficient in the transmission rate function is simply the minimum number of state transitions required to go from two states in which messages are emitted. If the finite state machine is represented as a directed graph, then for every message emitting state, we can compute the closest state that emits a message on the same channel using standard algorithms for searching directed graphs. The minimum of these distances will be the denominator of the slope (i.e., the x in (2.6.1)).

2.6.2 Determining Message Input Rates

The message input rate for a process that receives message from a single physical channel, will simply be the rate at which messages are transmitted on this physical input channel. For processes that receive messages from multiple physical channels we need to refine the definition of a transmission rate.

As the actions of a process are governed by the receipt of messages, it is useful to view a process and the channels on which it receives messages together. In this view a process that receives messages from a single physical channel is called a *single producer/single consumer* (SP/SC) *process*. A process that receives messages from multiple physical channels is called a *multiple producer/single consumer* (MP/SC) *process*.[6] An SP/SC process has only a single producer of input messages while an MP/SC process has multiple producers of input messages. Both types of processes are illustrated in Figure 2.6.2.

[5] Throughout this thesis, the symbol Δ will be used to signify the end of a proof.

[6] The qualifiers MP/SC and SP/SC can be applied to data repositories as well.

A
a) SP/SC process
B
b) MP/SC process

Figure 2.6.2: Two characterizations of processes based on the number of physical input channels.

Recall from Section 2.3.5 that we distinguish between *physical* and *logical* input channels for MP/SC processes. An MP/SC process conceptually receives all its messages on a single logical input channel. We define the message transmission rate on this logical input channel in terms of the rates at which messages are sent on the physical input channels. Specifically, the logical input rate for MP/SC processes is defined as the sum of the rates on each physical input channel that is bound to the process. This is illustrated in Figure 2.6.3.

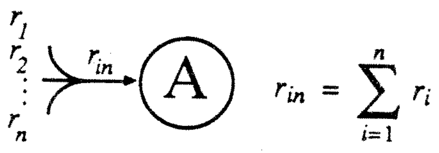


Figure 2.6.3: Input rate definition for MP/SC processes.

The input rate for MP/SC processes is the worst case *aggregate* rate at which the MP/SC process will receive messages. It should be noted that this is a different notion of rate from that used in the previous sections. The logical message input rate at an MP/SC process does not correspond to the physical input rate. That is, it is not the reciprocal of the worst case minimum inter-arrival time of messages. Since we make no assumptions about the relationships between the message producers on each channel, it is conceivable that messages may arrive simultaneously at MP/SC processes. Therefore, an MP/SC process's logical input channel theoretically has an infinite transmission rate. However, this is not an appropriate rate since an MP/SC process is not required to adhere to the RT/PC paradigm on its *logical* input channel. An MP/SC process is only required to adhere to the RT/PC paradigm on each *physical* channel from which it receives messages. That is, when a process P_1 sends a message to process A in Figure 2.6.3, P_1 is guaranteed that the message it sends will be consumed before it sends its next message to A. While process A is consuming process P_1's message, an other message from process P_2 may arrive. So long as this new arrival is consumed before process P_2 sends its next message to A, there is no problem. By consuming messages at the worst case aggregate rate, an MP/SC process will adhere to the RT/PC paradigm on all its physical input channels.

One implication of this definition of an MP/SC processes input rate is that an implementation of our discipline will be required to buffer messages at MP/SC processes. However, a single buffer for each channel will suffice. This is not a concern since we will see in Chapter 5 that most reasonable implementations of our discipline will require a single buffer per asynchronous channels for other reasons.

A second issue arising from this definition is that in general, the logical input rate of an MP/SC process will not be the reciprocal of an integer. More precisely, it is not guaranteed to be the reciprocal an integer multiple of our basic time unit. However, if our basic time unit is sufficiently small, then the reciprocal of this rate can be reasonably approximated by the next smallest integer multiple. This will have no effect on our ability to create and reason about the real-time behavior of designs. When we discuss the implementation of our discipline in Chapter 5, we will see that there is a small penalty to pay for adopting this definition.

Note that the output rates of a MP/SC process will conform to the original concept of a transmission rate. That is, there will exist a minimum separation of messages emitted on an MP/SC process's asynchronous output channels and these output rates will be a function of the input rate. Our definition of logical input rate maintains the abstraction that each process receives messages on a single logical input channel and that there exists a minimum separation between message arrivals on every asynchronous channel. From the perspective of a MP/SC process, messages arrive and are processed at a single rate. From the perspective of a process that sends messages to an MP/SC process, every message sent to an MP/SC process is consumed according to the RT/PC paradigm. In this sense we lose nothing by adopting the aggregate rate as the input rate.

2.6.3 Computing Channel Transmission Rates

Channels connecting input devices to processes implicitly have transmission rate functions equal to the identity function. The actual worst case rate at which messages travel on channels bound to devices is simply $1/p$, where p is the minimum message separation time from the specification of the input device. Given these external input rates, we can compute the rates at which messages flow on each asynchronous channel in a design. The transmission rate functions may be solved either symbolically or numerically depending on whether or not the output rates of the input devices are known. If a design is acyclic and well-formed, then the following theorem shows that we can always solve the transmission rate functions and label each asynchronous channel in the design with a worst case transmission rate.

Theorem 2.6.2: Let $G = (V,E)$ be a design graph whose edges corresponding to asynchronous channels are labeled with transmission rate functions of the form given by (2.6.1). If G is a well-formed, acyclic design graph, then the transmission rate functions can always be solved.

Proof: Let G' be the directed graph obtained from G by deleting all nodes corresponding to data repositories and all edges corresponding to synchronous channels. Since data repositories perform no output they have no edges leaving them. Therefore, if G was well-formed and acyclic then G' will be well-formed and acyclic. Since the directed graph G' is acyclic we can topologically sort the nodes in G' [Knuth 73]. Since the graph is well-formed, every node lies on a path from a node corresponding to an input device. Since the rate at which input devices emit messages is assumed to be known, the message transmission rates can be computed by traversing the graph according to the topological order of the nodes, and solving the transmission rate functions for each node's output edges.

Δ

In principle, designs may contain cycles of asynchronous channels. Cycles will be quite useful, in fact, for representing feedback loops. However, a transmission rate cannot always be determined for channels in a cycle with arbitrary transmission rate functions. For example, for the simple cycle shown in Figure 2.6.4, if the output channels from processes A and B have identity transmission rate functions, then the actual transmission rate cannot be determined for the channel from process A to process B.

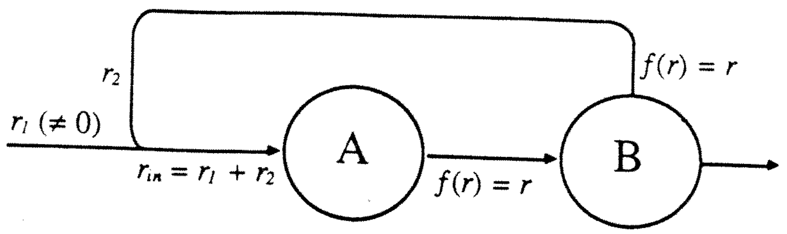


Figure 2.6.4: A system design with a cycle.

In this example the problem is not the existence of a cycle per se, but rather with the specific transmission rate functions. With these functions there is an unstable feedback of messages in this cycle. A more detailed discussion concerning the determination of transmission rates in graphs with cycles, will be postponed until Chapter 5.

2.7 An Example: A Digital Stopwatch

To illustrate our design discipline we present a design for a digital stopwatch. (The adjective *digital* refers to the type of display our stopwatch has.) Our choice of the stopwatch example was motivated by the work of Harel [Harel 79].

The stopwatch measures the length of intervals of time. The stopwatch has a 14 character wide display for output and has two buttons for input as shown in Figure 2.7.1. The two buttons are referred to as the *on/off* and *split* buttons.

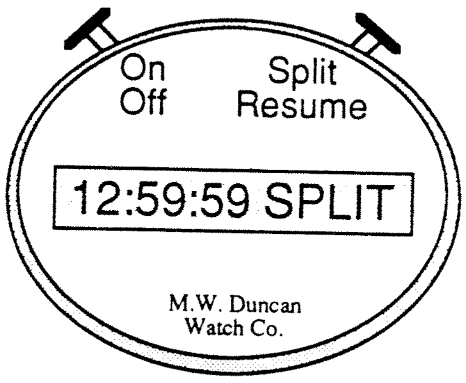


Figure 2.7.1: The digital stopwatch. (Shown in SPLIT mode.)

2.7.1 Functional Description

The watch can display two forms of time called *elapsed time* and *split time*. The elapsed time is the total amount of time that the watch has been on since the display was last reset. The split time is the value of elapsed time when the split button was pressed. Normally the watch displays the elapsed time. When the watch is *on*, the display continuously updates the *elapsed time*. If the watch is *on* and the split button is pressed, the current value of the display is frozen on the display (the updating process is suspended). In this case the displayed time is called the *split time*. Since the watch is *on*, the elapsed time is still being accumulated internally. Pressing the split button again resumes the display of elapsed time. If the watch is *off*, then pressing the split button resets (zeros) the display. The finite state machine shown in Figure 2.7.2 describes the operation of the watch in more detail. State transitions occur when buttons are pressed.

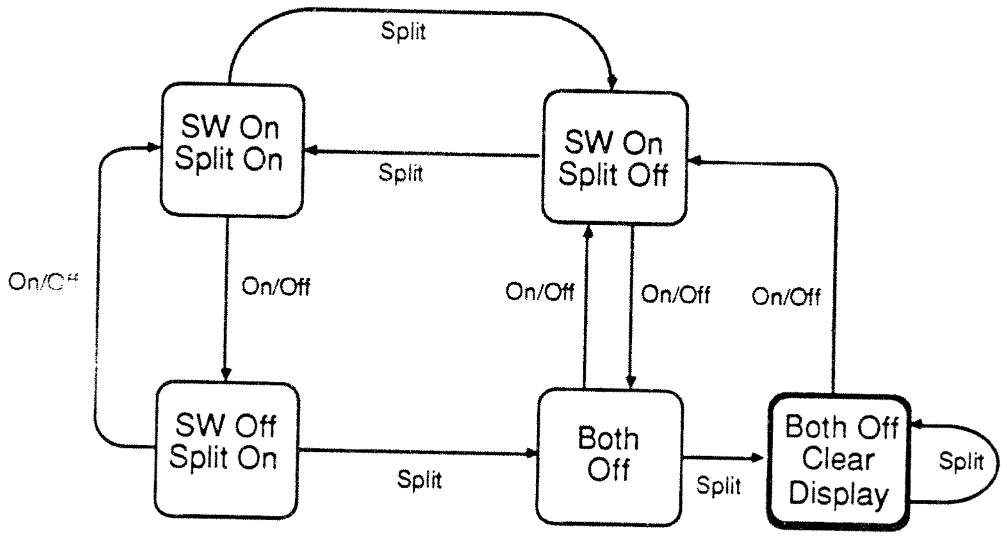


Figure 2.7.2: Stopwatch finite state machine.

The behavior within each state is given below.

Both Off/Clear Display: Both split and stop watch functions are off and the display is cleared (all zeros). This is the initial state of the watch.

SW On/Split Off: The watch is running. The display shows the continuously updating value of elapsed time.

SW On/Split On: The watch is running. The display shows the value of elapsed time at the time the split button was pressed. The word "SPLIT" appears next to the display to denote that the displayed time is the split time. In addition, the word SPLIT flashes on and off to indicate that the watch is still running. The value of elapsed time is still updated internally.

SW OFF/Split On: The watch is no longer running. The display is the same as the state above except that the word "SPLIT" does not flash.

Both Off: The watch is not running. The display shows the value of elapsed time at the time the on/off button was pressed.

2.7.2 Proposed Design

We have a designed a real-time system to implement this stopwatch. The design is presented in Figure 2.7.3 below. It consists of 2 input devices, 5 processes, and 2 data repositories. There are three implicit mutual exclusion constraints in the design. The channels are labeled with the name of the type of messages they transmit. In this example a message has an operation name field and a data field. For example, the Button Process process sends messages of type "Read/Write" to the Watch State data repository. An actual message from the Button Process to the Watch State data repository will specify the Write operation and contain the value of the current status of the watch buttons.

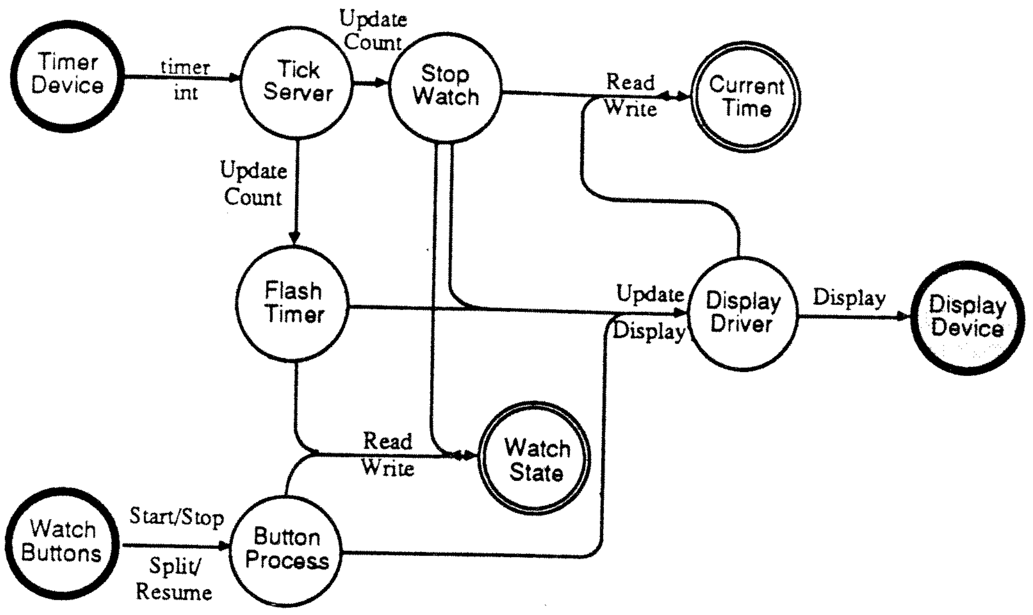


Figure 2.7.3: Stopwatch design.

A high-level description of the behavior of each component is given below.

Timer Device: An input device. The timer is the hardware clock. It provides a (continuous) reference stream of real-time ticks from which elapsed time is measured.

Tick Server: A process. The Tick Server process distributes streams of real-time "ticks", of varying frequencies, to two other processes in the system.

Stop Watch: A process. The Stop Watch process receives messages from the Tick Server. Upon receipt of a message it checks to see if the watch is running by sending a synchronous message to the Watch State data repository. If the watch is running it updates the elapsed time by sending a synchronous message to the Current Time data repository. If the watch was running the Stop Watch process will also send an asynchronous message to the Display Driver process to update the display.

Current Time: A data repository. Current Time maintains the watch's internal representation of elapsed time. It responds to requests to read (get) or write (set or update) the current value of time.

Flash Timer: A process. Also receives messages from the Tick Server. Upon receipt of a message it checks to see if the watch is running and in split mode by sending a synchronous message to the Watch State data repository. If the watch is in this state the Flash Timer process tells the display that it is time to flash (either turn on or off) the word "SPLIT".

Display Driver: A process. Receives messages telling it to display something. It determines what is to be output and performs the appropriate action. Depending on this action it may send a synchronous message to the Current Time data repository to get the current value of elapsed time.

Watch Buttons: An input device. The watch has two physical buttons; however, the watch's interface to these buttons is through a "button controller" that sends messages whenever a button is depressed or released. The message identifies the button (SPLIT or ON/OFF) and the operation (UP or DOWN). Simultaneous presses of both buttons are not recognized. While a button is depressed, actions on the other button are ignored.

Button Process: A process. Receives and decodes messages concerning operations on watch buttons. It takes two actions on a button (a press and release of the same button) to cause the watch to change state. Therefore, the Button Process process only emits messages when a button has been released (upon receipt of an UP message). When a button has been released, a synchronous message is sent to the Watch State data repository to update the state of the watch. Depending on the state value returned, an asynchronous message is sent to update the display.

Watch State: A data repository. It maintains the state of the watch. It responds to requests to read (get) or write (update) the current state.

Display Device: An output device. The watch a single 14 character wide display. The Display Device takes a pointer to a character string of this length and displays it.

2.7.3 Derivation of Message Transmission Rates

The asynchronous channels in our stopwatch design can be labelled with the transmission rate functions shown in Figure 2.7.4.

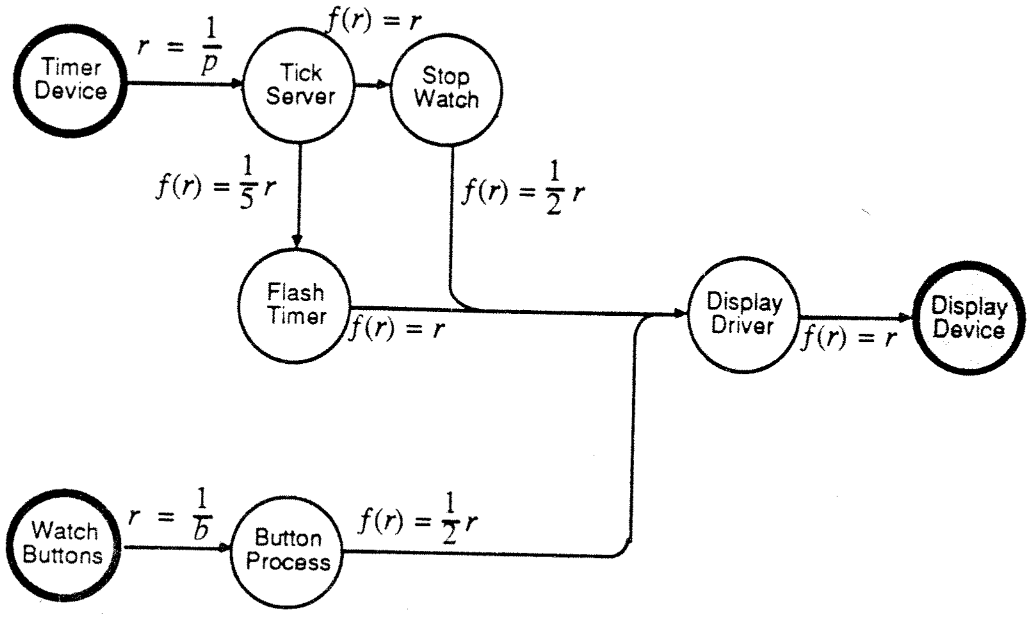


Figure 2.7.4: Transmission rate functions for the stopwatch design.

The derivation of these functions was performed by hand by modeling each process as a finite state machine as suggested in Section 2.6.1. A precise description of each process is not germane to the present discussion. Given the current level of description of the watch, some transmission rate functions have rather straightforward explanations. For example, since it takes two actions on a button (press and release) to have an effect on the watch, the Button Process process will always receive two messages before it emits a message. The transmission rate function on its asynchronous output channel is therefore one-half its input rate. In our design, the Stop Watch process receives messages at the same rate as the Tick Server; hence the channel between them has an identity transmission rate function. The watch is also designed so that the Flash Timer process processes messages five times slower than the Stop Watch process.

We will assume that in the worst case the hardware timer sends messages to the Tick Server every p time units, and that the buttons can be pressed at a maximum rate of once every b time units. We will assume that these two constants are provided to us. With these initial input rates we can solve the transmission rate functions and determine the worst case message transmission rate along each asynchronous channel. Figure 2.7.5 shows the transmission rates for each asynchronous channel.

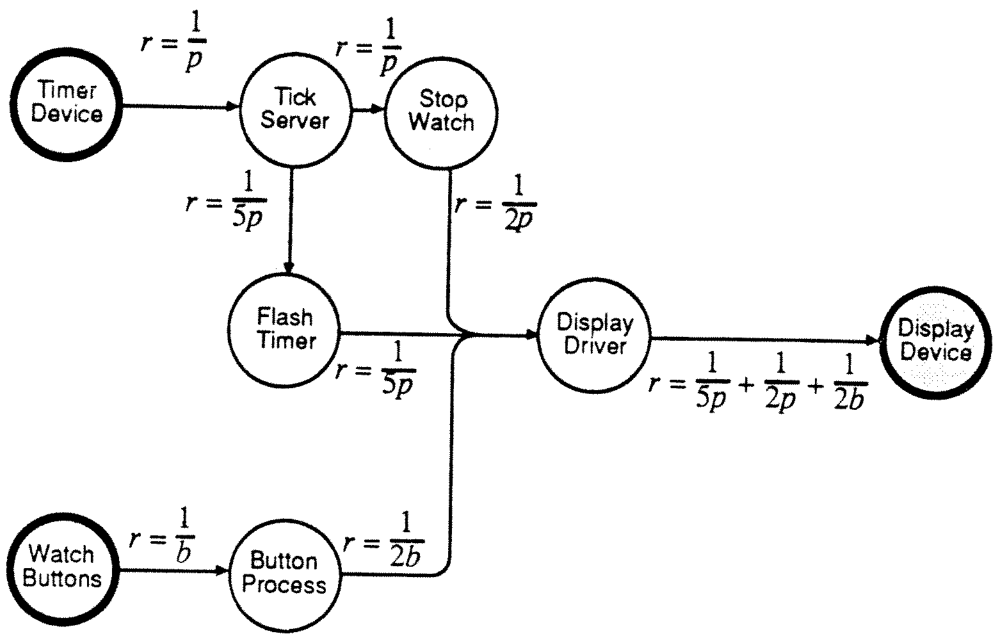


Figure 2.7.5: Transmission rates for stopwatch example.

This design graph contains a single MP/SC process: the Display Driver process. This process receives messages at a logical input rate of

$$r = \frac{1}{5p} + \frac{1}{2p} + \frac{1}{2b}.$$

Given the message transmission rates from Figure 2.7.4, we have an upper bound on the processing time of all messages. For example, all messages from the Tick Server to the Flash Timer will be consumed within $5p$ time units of their emission. Messages sent from the Button Process to the Display Driver will be consumed within $2b$ time units of their emission. Lastly, the physical display device must be able to handle display requests that arrive every

$$\frac{1}{\frac{1}{5p} + \frac{1}{2p} + \frac{1}{2b}} = \frac{10pb}{7b + 5p} \qquad (2.7.1)$$

time units. In general this quantity will be neither an integer nor an integer multiple of our basic time unit. However, as we remarked in Section 2.6.2, if our basic time unit is small enough, then (2.7.1) can be closely approximated by its truncated value.

2.8 Reasoning About Time

The semantics of message passing gives an upper bound on the delay between the emission and the processing of a single message. In this section we show how these semantics can be used to derive bounds on the propagation delay of a sequence of messages along a path in a design graph. Such information is useful since it allows a designer to bound the overall response time of the system to the occurrence of external or internal events. For example, for the stopwatch design in the previous section, we can determine bounds on the time between a press of the on/off button and the actual starting or stopping of the watch. Bounds on message propagation delays can also be useful for reasoning about logical properties of a design. For example, we can determine the maximum drift between the internal representation of time and absolute real-time in the watch design.

For an ordered sequence of processes, we would like to determine the maximum difference between the time a message arrives at the first process in the sequence, and the time a message is emitted from the last process. A closely related problem is to determine the maximum difference between the time a message arrives at the first process in the sequence, and the time when the last process in the sequence consumes a message. The former measure is useful for deriving response times to events while the latter measure is useful for determining temporal properties of data in data repositories. All the bounds we develop will be valid for any implementation of our real-time design discipline.

Informally, if we think of this sequence of processes as a single meta-process, we are trying to determine when this meta-process emits a message relative to the time it received a message. We call this time the *message propagation*, or *message delay time*. While no single message propagates through a path in a design graph, we choose this name as it is suggestive of the single process abstraction of a path.

2.8.1 Message Propagation Times Through Processes

The determination of the message propagation time through a sequence of processes depends on the rates at which messages are emitted and the slopes of the transmission rate functions for the channels the messages traverse. Let processes P_1 - P_k be the processes on an acyclic path through a design graph as shown in Figure 2.8.1. For this path, let r_i be the rate at which the i^{th} process P_i receives messages on the i^{th} channel on the path and let x_i be the denominator of the coefficient of process P_i's transmission rate function for its output channel. Assume processes P_1 - P_k emit messages at their maximum rate for some long time interval.

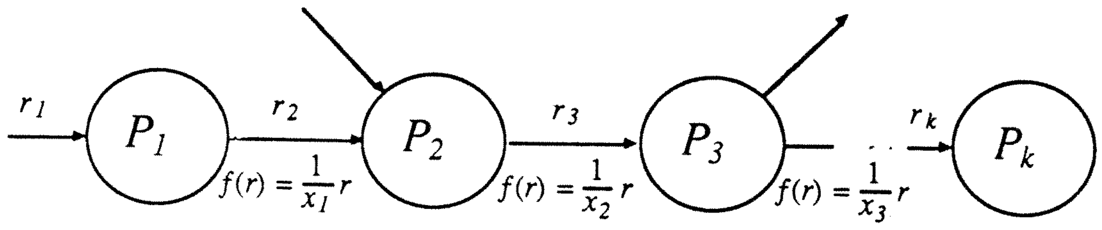


Figure 2.8.1: An acyclic sequence of k processes.

For a sequence consisting of only a single process P_1, when a message arrives at P_1, by Theorem 2.5.1, the message must be be consumed sometime within $1/r_1$ time units of its arrival. In addition, when process P_1 receives a message, it can wait for the arrival of at most $x_1 - 1$ additional messages before it must emit a message. Therefore, when process P_1 receives a message, P_1 may delay for at most

$$\frac{1}{r_1} + \frac{x_1 - 1}{r_1} = \frac{x_1}{r_1}, \qquad (2.8.1)$$

time units before emitting a message. If process P_1 is an SP/SC process, then (2.8.1) will be the minimum message inter-arrival time on the channel from P_1 to process P_2. If process P_1 is an MP/SC process, then (2.8.1) will be larger than the minimum message inter-arrival time on the channel from P_1 to process P_2.

For a sequence of k processes, the worst case delay between the arrival of a message at process P_1, and the emission of a message from process P_k can be at most

$$\sum_{i=1}^{k} \frac{x_i}{r_i}, \qquad (2.8.2)$$

time units. The worst case the delay between the arrival of a message at process P_1, and the consumption of a message at process P_k can be at most

$$\frac{1}{r_k} + \sum_{i=1}^{k-1} \frac{x_i}{r_i}, \qquad (2.8.3)$$

time units.

Both (2.8.2) and (2.8.3) correspond to the case where processes P_1 - P_k are least synchronized. That is, while each process is emitting messages at its maximum rate, each process waits for the largest possible number of message arrivals before emitting a message. A more optimistic scenario occurs when processes P_1 - P_k are all synchronized. That is, each process emits a message when it receives a message. In this case, the arrival of a message at process P_1 will directly cause a message to be emitted by process P_k. Hence the maximum propagation delay for *this message* will be at most

$$\sum_{i=1}^{k} \frac{1}{r_i}, \qquad (2.8.4)$$

time units. If all processes in the sequence have identity transmission rate functions then (2.8.2) and (2.8.4) are identical. Note that if all processes emit messages at their maximum rate, then in any sequence of $x_1 \cdot x_2 \cdot \ldots \cdot x_k$ message arrivals at process P_1, there will always be messages whose propagation delay is bounded from above by (2.8.2) and (2.8.4).

If $x_i \neq 1$, then process P_i can be viewed as always combining (at least) x_i messages from process P_{i-1} into a single output message. For example, in the stopwatch design, the Button Process process always combines two button messages ("press" and "release") into a single message to change the state of the watch. For applications such as the watch, (2.8.4) is a more interesting measure of message propagation delay than (2.8.2). Since the "release" message always causes the Button Process process to emit a message, the propagation delay for this message is more interesting since this is the message that is causing an action. The particular flavor of propagation delay that a designer will be interested in will of course be dependent on the application at hand.

On a related note, for paths through MP/SC processes, it may not always make sense to apply the above analysis of propagation delays. The foregoing analysis is only applicable to MP/SC processes that behave as *shared servers*. A shared server is a process that offers the same computational service to all its message producers. Shared servers have the property that their actual rate of output is not a function of the distribution of message arrivals across individual producers. An example of such a shared server is the Display Driver process in the stopwatch example. The Display Driver process provides the same service to all its message producers. As such, it always emits messages to the Display Device process independent of the message's origin. The Display Driver would not be a shared server if, for example, it waited to receive a message from each message producer before emitting a message. In this case, we could not determine the propagation delay of messages from the Watch Buttons device to the Display Device by examining only the processes on the path between these two nodes. In practice, we have yet to encounter an MP/SC process that was not a shared server.

If the worst case transmission rate is not realized on every channel simultaneously, then in order to determine a bound on the propagation delay for messages, we must have a lower bound on the transmission rate for each channel. Suppose a process P, with transmission rate function $f(r) = r/x$ on an output channel, pauses for at most $y > x$ input messages before emitting an output message on the channel. For a path through process P containing this channel, the worst case message propagation delay through P will simply be increased by y/r time units, where r is the input rate for process P. Note that the specification of an upper bound on transmission rates necessarily must come from the system designer. It cannot be inferred from a design. Presently we do not explicitly include a notion of worst case maximum transmission rate. The design discipline could be modified to include a specification of worst case maximum rates. Extensions to our discipline, such as this one, will be examined in Chapter 5.

If a graph contains a cycle then message propagation delays can be determined only for each pass through the cycle.

Finally, note that our ability to determine temporal properties of messages was not dependent on whether or not processes were in critical sections. Mutual exclusion, as we have included it in our design discipline, has no bearing on the real-time properties of processes or messages. Mutual exclusion is only a mechanism for ensuring logical properties.

2.8.2 Message Propagation for Output Devices

For purposes of deriving message propagation delays, output devices can be modeled as pseudo processes. If a process sends a message to an output device at rate r, then the device can delay at most $1/r$ time units. Since devices are not multiplexed among processes, we will actually expect that messages only delay for $c \leq 1/r$ time units where c is the maximum response time of the device.

2.8.3 An Example: Reasoning about the Stopwatch

The techniques for determining message propagation times can be used on the stopwatch example to address the questions posed at the start of this section. For example, when the watch is running we can determine the best and worst case drifts in the displayed and internal representation of elapsed time.

When the watch is running, the Tick Server and Stop Watch processes emit messages at their maximum rate. Recall that the Stop Watch process updates the value of Current Time on every message arrival from the Tick Server, and updates the display on every other message arrival from the Tick Server. In the worst case it can take up to $2p$ time units for a Timer Device message to propagate to, and be consumed by, the Stop Watch process (see Figure 2.7.4). Therefore, in the worst case:

- the internal value of time differs from external time by up to $2p$ time units. (This corresponds to 1 unit of stopwatch time.)

For a Timer Device message that causes the display to be updated, it can take up to $4p$ time units for a Timer Device message to propagate to, and be consumed by, the Display Driver process. Therefore, in the worst case:

- displayed time differs from the external value of time by less than $4p + c$ time units where c is the maximum latency of the display device ($c \leq r_{out}$ for the Display Driver process).

In the best case:

- the internal value of time differs from external time by less then p time units,

- displayed time differs from internal time by less than p time units.

Note that the worst case figures above can not be combined. The worst case behavior of the Display Driver is not independent of the worst case behavior of the Stop Watch. In the worst case, the displayed time differs from the external value of time by up to $4p + c$ time units. If the difference between the internal and external values of time is (within epsilon of) $2p$, then in the worst case the difference between the displayed time and external time will be in the interval $[2p, 4p + c]$ and the difference between the displayed time and internal time will be in the interval $[0, 2p + c]$.

In Chapter 5 we will present an implementation of our design discipline in which we can guarantee that in the worst case:

- the internal value of time differs from external time by less than p time units,

- displayed time differs from the internal value of time by less than $2p + c$ time units (no more than c plus one unit of stopwatch time).

We can determine the lag between the release of a button and a change in the display in a similar manner. Recall that the Button Process process emits messages only when a button is released. If the Watch Buttons and Button Process processes emit messages at their maximum rate, then

- in the best case, the time between the release of a button and an action of the display will be less then b time units,

- and up to $3b + c$ time units in the worst case.

With the implementation to be presented in Chapter 5, we can guarantee that in the worst case:

- the time between the release of a button and an action of the display will be less then 2*b* time units.

2.9 Discussion and Summary

This chapter has presented.the RT/PC paradigm of interaction as a means for modeling the behavior of real-time systems. The paradigm was motivated by Wirth's phrasing of interactions with the external world in terms of producer/consumer relationships [Wirth 77c]. The RT/PC paradigm extends these observations by focusing on the rates at which a producer produces data objects. The concept of a rate, defined in terms of the worst case inter-arrival time of a message, is central to our design discipline. The abstraction that a message consumer appears to a producer as being able to consume messages infinitely fast, is borrowed from the language ESTEREL. In ESTEREL the programmer is presented with the abstraction that all computational statements execute in zero time. We have chosen to generalize this abstraction. The RT/PC paradigm simply states that the computations required to consume a message are completed before the next message arrives.

Our design discipline is similar in appearance to several existing programming, design, and specification systems. This is primarily a reflection of the long standing popularity of message passing systems and their natural graphical representation. Of the systems mentioned in Chapter 1, DARTS, MASCOT, Statecharts, and SARTOR all use a graphical representation expressing designs. In addition, DARTS, MASCOT, SARTOR, and ESTEREL all employ some form of message passing. The SARTOR environment is most similar to our discipline as it also is based on an underlying directed graph model. Our discipline extends each of the aforementioned systems in three significant ways. These are:

- the inclusion of mutual exclusion as a first class entity in the discipline,
- the real-time semantics of message passing based on the RT/PC paradigm, and
- the ability to reason about the real-time behavior of messages along arbitrary paths in a design graph.

The costs and benefits of each of these components will become apparent in the following chapters.

Having formulated our design discipline, there are three fundamental issues to address next. First, can real-time systems be naturally and elegantly designed with our discipline? That is, can the real-time behavior of actual systems be expressed in our discipline? In Chapter 6 we will provide evidence that this question can be answered affirmatively. The second issue to address concerns the utility of the discipline. We must assess the benefit of designing systems with our discipline over more traditional methods. Chapter 6 will present some designs of actual systems to demonstrate the utility of the discipline. The third issue concerns the practicality of the discipline. We must demonstrate that is it possible to actually implement the RT/PC paradigm. This last issue is the subject of the next three chapters.

Chapter 3

Fundamental Scheduling Results for Cyclic Tasks

3.1 Introduction

The design discipline of Chapter 2 will be useful only if it is possible to realize the programming abstractions it presents. As a first step towards implementing this discipline, we study an abstract characterization of the processing requirements implied by the RT/PC paradigm.

A process in our discipline repeatedly accepts a message, consumes the message and then waits for the arrival of the next message. The repetitive behavior motivates a characterization of a process as a *cyclic task*. A cyclic task is simply a task that makes multiple requests for execution. We will distinguish between two types of cyclic tasks: *periodic* and *sporadic*. A periodic task makes execution requests at regular intervals in time while a sporadic task makes execution requests with only a lower bound on the duration between successive requests.

This chapter examines the problem of scheduling independent, cyclic tasks on a uniprocessor. The goal is to develop scheduling policies that can guarantee that each task completes execution within a specified time after it starts. We develop and analyze algorithms for preemptive and non-preemptive scheduling of periodic and sporadic tasks. Pure preemptive scheduling represents the most general environment for studying independent tasks while non-preemptive scheduling represents the most restrictive environment. The algorithms we present are analyzed for correctness and optimality. The results of this chapter will establish the tone and limits of the analysis of the more complex problems addressed in the following chapter. The results developed here will also serve to demonstrate the important differences between sporadic and periodic tasks.[1]

Section 3.2 introduces a characterization of a real-time system as a set of cyclic tasks. It describes our notation and states the basic scheduling problem we address. Section 3.4 describes the *earliest deadline first* (EDF) scheduling policy. This is the primary scheduling policy we use throughout this dissertation. The analysis of the EDF policy is primarily an analysis of the worst case processor demand that can exist in an interval of time. Section 3.4 develops some important lemmas concerning the processor demand of cyclic tasks.

Section 3.5 studies the problem of preemptive scheduling of cyclic tasks on a uniprocessor. We will show that the EDF scheduling policy is an optimal policy for both periodic and sporadic tasks. Section 3.6 studies the problem of non-preemptive scheduling. In this case the EDF policy is only optimal for sporadic tasks. Section 3.9 will later show that there does not exist an optimal policy for periodic tasks.

Section 3.7 examines the complexity of determining when the EDF scheduling policy can correctly schedule a set of tasks. Section 3.8 briefly discusses some alternative scheduling policies.

3.2 Basic Task Model, Notation, and Approach

A common model for the abstract study of real-time systems has been the cyclic task [Liu & Layland 73, Leung & Merrill 80, Mok 83]. These models are motivated by the observation that real-time processing is frequently repetitive. Unlike tasks in traditional multi-programming systems, real-time tasks are frequently executed from start to completion numerous times over the life of the system. In this section we present our simplest task model and describe our notation.

[1] The results presented in this chapter were first reported in [Jeffay 88] and [Jeffay & Anderson 88].

3.2.1 Cyclic Tasking Model

Formally, a *cyclic* task T is a 3-tuple (s, c, p) where

s = start or release time: the time of the first request for execution of task T,

c = computational cost: the time to execute task T to completion on a dedicated uniprocessor, and

p = period: the interval between requests for execution of task T.

The parameters s, c, and p are expressed as integer multiples of some indivisible time unit. In the majority of scheduling problems that follow, we will assume that the release times of tasks are unknown. The reasons for this assumption will become apparent in Chapter 5.

We distinguish between two types of cyclic tasks. The distinction is based on the characterization of a task's inter-request time. If a task makes request at regular intervals it is called *periodic*; otherwise it is termed a *sporadic* task. In both cases, a cyclic task must be completed before the next execution request is made. Periodic tasks arise frequently in applications, such as navigation and process control, in which accurate control requires sampling and processing at precise intervals. Sporadic tasks are more commonly associated with event driven processing. Since processes in our design discipline can in principle be used to design applications with either type of processing requirements, we are interested in studying both characterizations.

Formally, the behavior of a cyclic task is given by the following sets of execution rules. Let t_k be the time that task T makes its k^{th} request for execution. The behavior of a periodic task T is given by the following execution rules.

i) If T has period p and makes its first request for execution at time s, then T will make its k^{th} request for execution *exactly* at time $t_k = s + (k-1)p$, for all $k \geq 1$.

ii) The k^{th} execution request of T must be completed no later than the *deadline* $t_k + p = (s + (k-1)p) + p = s + kp = t_{k+1}$.

iii) Each execution request of T requires c units of execution time.

A sporadic task is a generalization of a periodic task. The behavior of a sporadic task T is slightly less constrained than a periodic task. Its behavior is given by the following rules.

i) Task T makes its first request for execution at time $t_1 = s$.

ii) If T has period p, then T makes its $(k+1)^{st}$ request for execution at time $t_{k+1} \geq t_k + p \geq s + kp$.

iii) The k^{th} execution request of T must be completed no later than the *deadline* $t_k + p$.

iv) Each execution request of T requires c units of execution time.

The "period" of a sporadic task is simply the minimum time between any two successive execution requests of the task. Beyond this, we make no assumptions on the nature of a sporadic task's execution requests. Sporadic tasks are independent in the sense that the time of their execution requests are dependent only upon the time of their last request and not upon those of any other task.

Once released, both periodic and sporadic tasks make requests for execution forever. If an execution request of a task has not completed execution by its deadline, then we say the task *misses a deadline*. More precisely, if a task has a deadline at time t_d, then if the task had not completed execution at time t_d, then the task has missed a deadline. For a set of cyclic tasks, the problem is to determine if it is possible to schedule the tasks on a uniprocessor such that no task misses a deadline. Given the real-time nature of these tasks, we would like to determine if it is possible to achieve this goal without actually scheduling the tasks.

The property of cyclic tasks we are interested in studying is termed *feasibility* [Leung & Merrill 80]. A set of periodic or sporadic tasks τ, is said to be *feasible* on a uniprocessor if it is possible to schedule τ on a uniprocessor, such that every execution request of every task T is guaranteed to have completed execution at or before its deadline. To demonstrate the feasibility of a set of tasks we must therefore provide a scheduling algorithm that is *correct with respect to the set of tasks*. A scheduling discipline is correct with respect to a set of tasks if the discipline can schedule the tasks such that no task ever misses a deadline. If a scheduling discipline is correct with respect to a set of tasks then we say that the discipline can *correctly schedule* the tasks. A scheduling discipline is said to be *optimal* if it can correctly schedule any task set that is feasible.

The use of the term *optimal* may be slightly confusing at first. In the context of scheduling, if a discipline is termed *optimal*, it does not imply that this discipline is *better* than all other disciplines. It simply means that the discipline is *as good as* any other discipline.

There are two central issues in the study of feasibility:

- for a particular class of scheduling disciplines, does an optimal scheduling discipline exist, and
- if an optimal discipline exists, what is the complexity of determining if the discipline will be correct with respect to a set of tasks.

The first issue is a determination of relative feasibility. Often we will limit our attention to a restricted class of scheduling disciplines motivated by practical concerns. Within this restricted domain we seek optimal scheduling disciplines. The second issue is to establish a lower bound on the complexity of the feasibility decision question. If there exist multiple optimal scheduling disciplines, it behooves us to use the most efficient decision procedure for feasibility.

3.2.2 On-line Versus Off-line Scheduling

The complexity of determining if a scheduling discipline will be correct with respect to a set of tasks, is important for distinguishing between optimal scheduling disciplines. There are two basic approaches to scheduling: *on-line* and *off-line* scheduling. In on-line scheduling tasks are selected for execution simply based on the state of the tasks at the current time. In off-line scheduling, a *schedule*, or list of tasks, is constructed prior to the execution of the tasks. When the tasks are executed, the execution sequence specified by the schedule is followed. Off-line scheduling is always optimal since in theory we could always generate all possible schedules. However, the time and space complexity of constructing these schedules is likely to be prohibitive. For a schedule to have finite length it must be repetitive. For a more refined model of periodic tasks, Leung and Merrill have shown that a repetitive schedule always exists for feasible tasks. However the length of this schedule has an upper bound proportional to the least common multiple of the periods of the tasks

[Leung & Merrill 80]. In general, it will take exponential time and space to generate this schedule. For our tasking model it is not known if this bound can be improved.

In this dissertation, we will limit our attention to on-line scheduling for two related reasons. First, given our interest in sporadic tasks, we do not view off-line scheduling as a viable option. A schedule can be constructed only if the times at which tasks make execution requests are known in advance. Secondly, we will consider scheduling problems for periodic and sporadic tasks in which the release times of tasks are not known.

3.2.3 Scheduling Diagrams

If task T makes its k^{th} execution request at time t_k, then the closed interval $[t_k, t_k+p]$ is called the k^{th} ***request interval*** (or simply a request interval) of task T. We will represent a request interval graphically with a three sided rectangle as shown in Figure 3.2.1. These rectangles are horizontally positioned on a time line.

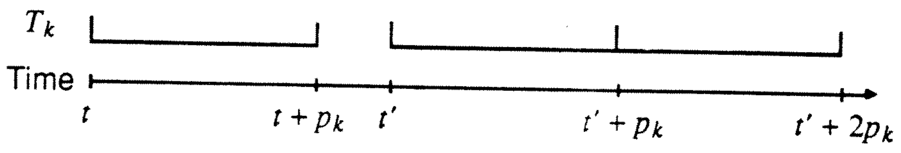


Figure 3.2.1: Graphical representation of a task's request intervals. This figure shows three request intervals.

The left side of the rectangle represents the start of a request interval and the right side represents the deadline of the request interval. The width of each request interval is the period of the task. In Figure 3.2.1, task T_k makes three requests for execution. These requests are made at times t, t', and $t' + p_k$. These execution requests have deadlines at times $t + p_k$, $t' + p_k$, and $t' + 2p_k$, respectively.

To illustrate the behavior of scheduling algorithms, we will use a modified Gantt chart that we call a "2-dimensional" Gantt chart. The intervals in time during which a request interval is executing on the processor are represented with a smaller, four sided, shaded rectangle superimposed on the request interval. The rectangle may be open ended to indicate either the preemption or resumption of the request interval. A rectangle open on the right side indicates that the task has been preempted. A rectangle open on the left side indicates that the task has been resumed. In each request interval, the total width of the shaded rectangles should not exceed the computational cost of the task.

Figure 3.2.2 illustrates these conventions. It shows the execution of two sporadic tasks $T_1 = (2, 3, 5)$ and $T_2 = (0, 3, 6)$. In this figure, an execution request of task T_2 commences execution at time 0. Task T_2 is preempted by task T_1 at time 2 and resumed at time 5. In the interval from time 6 to time 7, the processor is idle.

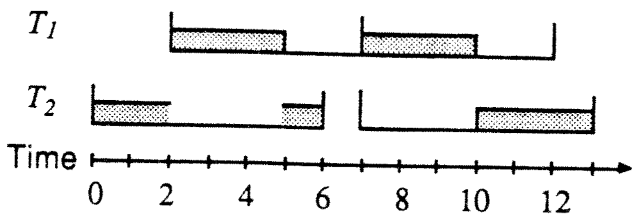


Figure 3.2.2: Graphical representation of task execution intervals. (A "2-dimensional" Gantt chart.)

The width of the shaded rectangles representonly the cost of executing a task and not the overhead of scheduling. For uniprocessor scheduling problems, any vertical line extending up from the time axis should pass through at most one shaded rectangle.

3.2.4 Approach

We will follow the same approach to solving all scheduling problems in this chapter. The first step is to develop *necessary* conditions for the feasibility of *periodic* tasks. These are conditions that must hold independent of the scheduling policy employed. Since a sporadic task can behave as a periodic task, any conditions necessary for the feasibility of a periodic task will also be necessary for the feasibility of a sporadic task. The second step is to develop conditions that are *sufficient* for the correctness of a specific scheduling policy when scheduling *sporadic* tasks. By the same argument as above, any conditions sufficient for ensuring the correctness of a sporadic task will also be sufficient for ensuring the correctness of a periodic task.

If conditions are developed that are sufficient for the correctness of a scheduling discipline, then these conditions are sufficient for the feasibility of a set of tasks. That is, if a set of tasks meet these conditions, then the scheduling policy under consideration can be used to correctly schedule the tasks. If the sufficient conditions are identical to the necessary conditions for feasibility, then the scheduling policy is an optimal policy. It can correctly schedule any set of tasks that is feasible.

3.3 The Earliest Deadline First Scheduling Policy

We will use the same scheduling policy for both preemptive and non-preemptive scheduling. The policy is the *earliest deadline first* (EDF) policy [Liu & Layland 73]. Our choice of the EDF policy is based on the observation that whenever uniprocessor scheduling problems with deadlines are considered, the EDF policy is typically shown to be an optimal policy.

The EDF scheduling policy dictates that when selecting a task for execution, the task with an uncompleted execution request whose deadline is closest to the current point in time is chosen for execution. Ties between tasks with identical deadlines are broken arbitrarily. The points in time at which the scheduler selects a task for execution are called *scheduling* or *dispatching points*. A preemptive scheduler will make a dispatching decision whenever a task makes, or terminates, an execution request. Unless the processor is idle, a non-preemptive scheduler will make dispatching decisions only when a task terminates an execution request. If the processor is idle then the first task to make an execution request is scheduled.

For the scheduling problems we consider, we will ignore the overhead associated with scheduling. We assume that tasks can be dispatched, or preempted, in zero time. Conceptually, an EDF scheduler will execute on an abstract machine with a single processor. This machine has the property that it can schedule tasks and allocate resources infinitely fast. The results we develop for this machine will establish limits for what is achievable on a real machine.

3.4 Bounding Processor Demand

For cyclic tasks, the analysis of the EDF scheduling discipline is primarily an analysis of the worst case computational demand on the processor that can occur over an interval in time. As a prelude to the study of preemptive and non-preemptive scheduling policies, we define the concept of processor demand for cyclic tasks and present some useful lemmas.

For a set of cyclic tasks τ, the *processor demand* in the interval $[a,b]$, written $d_{a,b}$, is defined as the minimal processing time required by τ in the interval $[a,b]$. That is, $d_{a,b}$ is the minimum amount of processor time required in the interval $[a,b]$ to ensure that no deadline is missed in the interval $[a,b]$. If the set of cyclic tasks τ is feasible, then for all a and b, $a < b$, we have $d_{a,b} \le b - a$.

For periodic tasks we give a useful lower bound on the processor demand in an interval.

Lemma 3.4.1: Let τ be a set of n periodic tasks with release times $s_1, s_2, \ldots, s_n$. Let s_{max} be the latest release time. For all points in time t_1 and t_2, $t_2 \ge t_1 \ge s_{max}$, in the interval $[t_1,t_2]$,

$$d_{t_1,t_2} \;\ge\; \sum_{i=1}^{n} \left(\left\lfloor \frac{t_2 - t_1}{p_i} \right\rfloor - 1 \right) c_i .$$

Proof: Since all tasks in τ have been released by time t_1, and since all tasks are periodic, in the interval $[t_1,t_2]$, each task will make one of the four patterns of execution requests shown in Figure 3.4.1. That is, each task will have execution request intervals that either contain one of the points t_1 or t_2, or it will have execution request intervals that have end points at points t_1 or t_2.

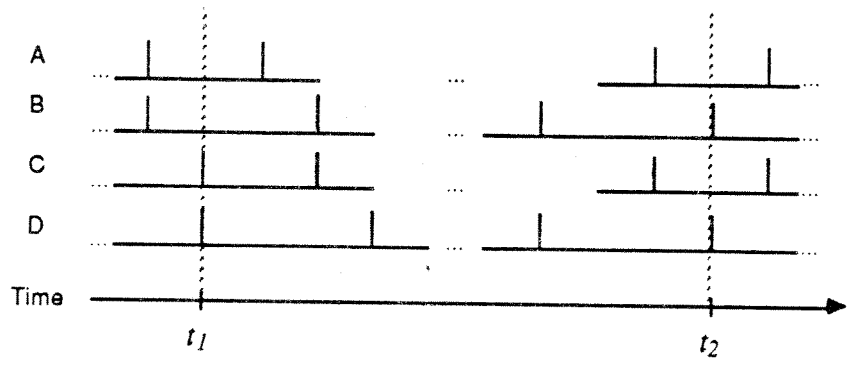


Figure 3.4.1: All possible patterns of execution requests for periodic tasks after time s_{max}.

If a task T_k makes execution requests corresponding to the pattern labelled D, then T_k has exactly

$$\frac{t_2 - t_1}{p_k}$$

execution requests that must be satisfied in the interval $[t_1,t_2]$. The processor demand for task T_k in the interval $[t_1,t_2]$, written $d^k_{t_1,t_2}$, is

$$d^k_{t_1,t_2} = \frac{t_2 - t_1}{p_k} c_k .$$

If task T_k makes execution requests corresponding to the pattern labelled C, then T_k has exactly

$$\left\lfloor \frac{t_2 - t_1}{p_k} \right\rfloor$$

execution requests that must be satisfied in the interval $[t_1,t_2]$. In this case task T_k will require a minimum of

$$d^k_{t_1,t_2} = \left\lfloor \frac{t_2 - t_1}{p_k} \right\rfloor c_k .$$

units of processor time to ensure that it does not miss a deadline in the interval $[t_1,t_2]$.

If task T_k makes execution requests corresponding to the pattern labelled B, then T_k again has at least

$$\left\lfloor \frac{t_2 - t_1}{p_k} \right\rfloor$$

execution requests that must be satisfied in the interval $[t_1,t_2]$. However, in this case task T_k can potentially have a higher processor demand. The processor demand for task T_k in the interval $[t_1,t_2]$, is

$$d^k_{t_1,t_2} = \left\lfloor \frac{t_2 - t_1}{p_k} \right\rfloor c_k + c'_k ,$$

where c'_k, $0 \leq c'_k \leq c_k$, is the fraction of the execution request containing the point t_1 that has not completed execution by time t_1.

Lastly, if task T_k makes execution requests corresponding to the pattern labelled A, then T_k has at least

$$\left\lfloor \frac{t_2 - t_1}{p_k} \right\rfloor - 1$$

execution requests that must be satisfied in the interval $[t_1,t_2]$. The processor demand for task T_k in the interval $[t_1,t_2]$, is

$$d^k_{t_1,t_2} = \left(\left\lfloor \frac{t_2 - t_1}{p_k} \right\rfloor - 1\right) c_k + c'_k \quad \geq \quad \left(\left\lfloor \frac{t_2 - t_1}{p_k} \right\rfloor - 1\right) c_k .$$

units of processor time to ensure that it does not miss a deadline in $[t_1,t_2]$.

From this analysis we conclude that each task's processor demand is a minimum when a task makes the pattern of execution requests labelled A. Therefore, for all points in time t_1 and t_2, $t_2 \geq t_1 \geq s_{max}$, in the interval $[t_1,t_2]$, the total processor demand is bounded by

$$d_{t_1,t_2} \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{t_2 - t_1}{p_i} \right\rfloor - 1\right) c_i . \qquad \Delta$$

Given the non-deterministic behavior of sporadic tasks, a meaningful lower bound on the processor demand in an arbitrary interval cannot be established for sporadic tasks. However, for sporadic tasks, we can give a useful upper bound on the processor demand in an interval. This bound will hold for periodic tasks as well.

Lemma 3.4.2: Let τ be a set of n cyclic tasks. Let t_0 be a point in time at which there are no outstanding requests for execution. More precisely, t_0 is a point in time such that if no task makes a request for execution at t_0, then the processor necessarily will be idle in the interval $[t_0, t_0 + 1]$. For all $t \geq t_0$,

$$d_{t_0,t} \leq \sum_{i=1}^{n} \left\lfloor \frac{t - t_0}{p_i} \right\rfloor c_i .$$

Proof: As in the previous lemma, in the interval $[t_0,t]$, each task will make one of the four patterns of execution requests shown in Figure 3.4.1. However, unlike the previous lemma, by the definition of the point t_0, all execution requests containing the point t_0 will have completed execution at or before time t_0. Therefore, for each pattern of execution requests, in the interval $[t_1,t_2]$, a task T_k makes at most

$$\left\lfloor \frac{t - t_0}{p_k} \right\rfloor$$

execution requests that must be satisfied in the interval $[t_0,t]$. Hence the total processor demand in the interval $[t_1,t_2]$ is bounded by

$$d_{t_0,t} \le \sum_{i=1}^{n} \left\lfloor \frac{t - t_0}{p_i} \right\rfloor c_i . \qquad \Delta$$

The utility of these bounds will become apparent in the following sections.

3.5 Preemptive Scheduling of Periodic and Sporadic Tasks

The work of this section borrows heavily from the seminal work of Liu and Layland [Liu & Layland 73]. In that work, they considered a more refined task model. Their task model consisted of periodic tasks with a release time of 0. They showed that when preemption is allowed at arbitrary points, the preemptive EDF algorithm is an optimal scheduling discipline for periodic tasks. We can extend their analysis in a straightforward manner to include arbitrary release times and sporadic tasks. The following theorem establishes a necessary condition for feasibility of periodic tasks with release times.

Theorem 3.5.1: A set of periodic tasks $\tau = \{T_1, T_2, \ldots, T_n\}$, with arbitrary release times can be feasible only if:

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \le 1.$$

Informally, the summation above can be thought of as a requirement that the system not be overloaded. If a periodic task T has a cost c and period p, then c/p is the fraction of processor time consumed by T over the lifetime of the system (i.e., the utilization of the processor by T). The above condition simply stipulates that the total processor utilization cannot exceed one.

Proof: Let s_{max} be the latest release time of the tasks in τ. By Lemma 3.4.1, for all t greater than s_{max}, in the interval $[s_{max},t]$ we have

$$\sum_{i=1}^{n} \left(\left\lfloor \frac{t - s_{max}}{p_i} \right\rfloor - 1 \right) c_i \le d_{s_{max},t} ,$$

so

$$\sum_{i=1}^{n} \left(\frac{t - s_{max}}{p_i} - 2 \right) c_i \le d_{s_{max},t} .$$

If τ is feasible then

$$\sum_{i=1}^{n} \left(\frac{t - s_{max}}{p_i} - 2 \right) c_i \le t - s_{max}$$

$$\sum_{i=1}^{n} \left(\frac{1}{p_i} - \frac{2}{t - s_{max}} \right) c_i \le 1.$$

Taking the limit as $t \to \infty$, we have

$$\lim_{t \to \infty} \left(\sum_{i=1}^{n} \left(\frac{1}{p_i} - \frac{2}{t - s_{max}} \right) c_i \le 1 \right) =$$

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \le 1.$$

Since tasks make execution requests forever, the above inequality is a necessary condition for feasibility.

Δ

Since sporadic tasks are a generalization of periodic tasks, Theorem 3.5.1 must also hold for sporadic tasks. The following theorem shows that the non-overload condition above is sufficient for ensuring the correctness of the preemptive EDF discipline for sporadic tasks. Since periodic tasks are a special case of sporadic tasks, this theorem also will hold for periodic tasks. Since the condition *necessary* for feasibility is *sufficient* for ensuring the correctness of the EDF discipline, the EDF discipline is therefore an optimal discipline for periodic and sporadic tasks. If any scheduling discipline can correctly sequence a set of periodic or sporadic tasks, then the EDF discipline must also be able to do so.

Theorem 3.5.2: The preemptive EDF scheduling algorithm can correctly schedule a set of sporadic tasks $\tau = \{T_1, T_2, \dots, T_n\}$, with arbitrary release times, if the cumulative processor utilization is less than or equal to one.

Proof: By contradiction.

Assume that there exists a set of sporadic tasks τ, with release times $s_1, s_2, ..., s_n$, whose cumulative processor utilization is less than or equal to one, such that a task misses a deadline at some point in time when τ is scheduled by the preemptive EDF algorithm. Let T_k be the first task to miss a deadline. Consider the first execution request of T_k to miss a deadline. Let t_d be the deadline of this request. For the remaining tasks whose release times are less than t_d, either they do, or do not, have a execution request interval that contains the point t_d. In the first case, a task either has a deadline at time t_d, or it is not active (has no execution request containing time t_d). For the tasks with request intervals that contain t_d, we consider the following two cases: either none of the request intervals that contain the point t_d are scheduled prior to t_d, or, some of the request intervals that contain the point t_d are scheduled prior to t_d.

Case 1: None of the request intervals that contain the point t_d are scheduled prior to t_d.

Let t_0 be the end of the last period in which the processor was idle. If the processor has never been idle let $t_0 = 0$. Since the EDF discipline never idles the processor if there is a task with an outstanding request for execution, all execution requests made prior to time t_0 will have completed execution by time t_0. Therefore, by Lemma 3.4.2, in the interval $[t_0,t_d]$, the processor demand is

$$d_{t_0,t_d} \leq \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_0}{p_j} \right\rfloor c_j .$$

Since there is no idle period in $[t_0,t_d]$ and since a task misses a deadline at t_d, it must be the case that $t_d - t_0 < d_{t_0,t_d}$. Therefore we have

$$t_d - t_0 < \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_0}{p_j} \right\rfloor c_j \leq \sum_{j=1}^{n} \frac{t_d - t_0}{p_j} c_j ,$$

or simply

$$1 < \sum_{j=1}^{n} \frac{c_j}{p_j} .$$

This is a contradiction of our assumption that the cumulative processor utilization was less than or equal to one. Therefore, if the non-preemptive EDF discipline fails, then some of the request intervals that contain the point t_d must have been scheduled prior to t_d.

Case 2: Some of the request intervals that contain the point t_d are scheduled prior to t_d.

Let T_i be a task that has a request interval that contains the point t_d. Assume that this request interval executes prior to the point t_d. Furthermore, assume task T_i is the last such task to execute prior to t_d. Let $t \leq t_d$ be the latest point in time that T_i executes in the interval $[0,t_d]$.

Case 2a: There exists a period in the interval $[t,t_d]$ in which the processor is idle.

If the processor was idle in this interval then we can apply the analysis of Case 1 on the interval $[t,t_d]$ and reach the same contradiction as in Case 1.

Case 2b: There is no idle time in the interval $[t,t_d]$.

Since tasks are selected for execution by earliest deadline, all execution requests of tasks with request intervals containing t, with deadlines less than or equal to t_d must have completed execution by t. Therefore, in the interval $[t,t_d]$ we must have

$$d_{t,t_d} \leq \sum_{j=1}^{n} \left\lfloor \frac{t_d - t}{p_j} \right\rfloor c_j .$$

As in the previous case, this leads to the contradiction of our assumption that the cumulative processor utilization was less than or equal to one.

We have shown that in all cases, if the preemptive EDF scheduling algorithm fails, then the processor utilization must have been greater than one. This proves the theorem. Δ

These results show that when preemption is allowed at arbitrary points, periodic and sporadic tasks are equivalent in terms of feasibility. They also show that their feasibility is independent of their release times. If a set of cyclic tasks is feasible, they will be feasible for all possible release times.

3.6 Non-Preemptive Scheduling of Sporadic Tasks

We next examine the opposite extreme: the case when preemption is disallowed. This section studies non-preemptive scheduling of sporadic tasks. Section 3.9 considers periodic tasks. For sporadic tasks, our approach is motivated by the early work of Sorenson [Sorenson 74, Sorenson & Hamacher 75].

An important issue concerning a non-preemptive scheduler concerns the use of *inserted idle time* [Conway et al. 67]. If tasks are scheduled by a discipline that allows itself to idle the processor when there exists a task with an outstanding request for execution, then that discipline is said to use inserted idle time. The non-preemptive EDF scheduling algorithm outlined in Section 3.3, does not use inserted idle time. All of the optimality results we develop in this section are relative to the class of scheduling disciplines that do *not* allow inserted idle time. In Section 3.8 we will discuss the effects of allowing inserted idle time.

The release time of a task plays a more significant role in the analysis of a non-preemptive scheduler. We start by deriving feasibility conditions for all possible release times and then extend these results to cover arbitrary release times. As in the previous section, we will first develop necessary conditions for feasibility and show these conditions are sufficient for ensuring the correctness of the non-preemptive EDF discipline.

There is a subtle but important difference between the two characterizations of release times. Arbitrary release times means that each task has a fixed, but arbitrarily chosen, release time. If a condition is necessary for feasibility with arbitrary release times, then the condition will be necessary for feasibility when all possible release times are considered. However, in general the converse does not automatically hold. When determining necessary conditions for feasibility, the consideration of all possible release times leads to a more general, and easier problem. For determining sufficient conditions for feasibility, the two characterizations are equivalent. If a condition is sufficient for feasibility for all possible release times, then the condition will also be sufficient for feasibility for arbitrary release times. The converse is also true.

3.6.1 Necessary Conditions for Feasibility

The following theorem establishes necessary conditions for ensuring the correctness of any non-preemptive discipline that does not use inserted idle time. For convenience, we assume throughout this section that our set of tasks is sorted in non-decreasing order by period ($p_i \geq p_j$ if $i > j$). The *index* of a task refers to its position in this sorted list.

Theorem 3.6.1: A set of periodic tasks $\tau = \{T_1, T_2, \ldots, T_n\}$, sorted in non-decreasing order by period, can be scheduled non-preemptively without inserted idle time for all possible release times only if:

1) $\displaystyle\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1,$

2) $\displaystyle\forall k,\ 1 \leq k < n:\ p_k \geq \underset{i:\, p_i > p_k}{\mathrm{MAX}} \left(c_i + \underset{0 < l < p_i - p_k}{\mathrm{MAX}} \left(-l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \right) \right).$

Condition (1) is simply the non-overload requirement. Condition (2), unfortunately, does not have such a straightforward explanation. Semantically, for each task T_k, $k < n$, the right hand side of the inequality in the second condition is the sum of the execution cost of T_k plus the worst case delay that can occur between the time T_k makes a request for execution and the time it is scheduled. Informally, if we think of a task's period as its maximum tolerable latency for each execution request it makes, then condition (2) simply requires that each task have a latency greater than or equal to the worst case delay that it can experience. The derivation of the worst case delay will be developed in the proofs below and is primarily a function of the relative sizes of tasks periods. For example, if all tasks have the same period, note that condition (2) is vacuous and condition (1) is the only necessary condition.

Proof: To show that these conditions are necessary we need only demonstrate that there exists a set of release times for which conditions (1) and (2) are necessary for τ to be feasible. We first show that condition (1) is necessary.

For all i, let $s_i = 0$ and let $t = p_1 \cdot p_2 \cdot \ldots \cdot p_n$. In the interval $[0,t]$, $\frac{t}{p_i} c_i$ is the total processor time spent executing task T_i. Therefore, if τ is feasible we must have

$$d_{0,t} = \sum_{i=1}^{n} \frac{t}{p_i} c_i \leq t,$$

or simply

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1.$$

For condition (2), choose a task T_k, where $k < n$, and values i' and l' such that $k < i' \leq n$, and $0 < l' < p_{i'} - p_k$. Let $s_{i'} = 0$, and $s_j = 1$ for $1 \leq j \leq n, j \neq i'$. This gives rise the pattern of task execution requests shown in Figure 3.6.1 below.

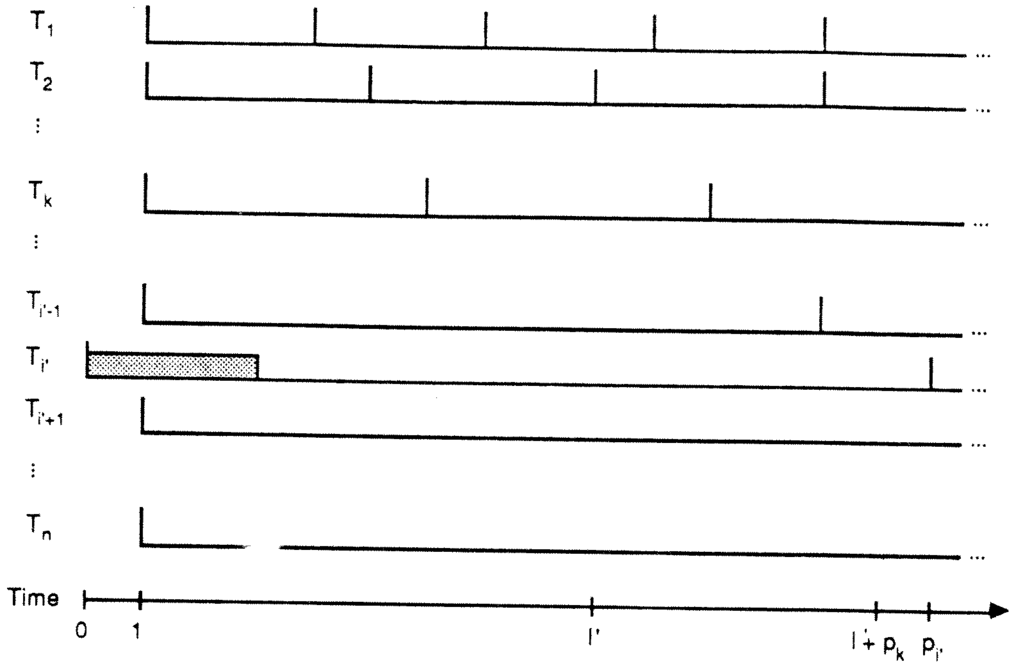


Figure 3.6.1: Construction for the necessity of condition (2).

In the interval $[0, l'+p_k]$, the total processor demand, $d_{0,l'+p_k}$, is given by

$$d_{0,l'+p_k} \geq c_{i'} + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j .$$

If τ is feasible then we must have

$$l'+p_k \geq u_{0,l'+p_k} \geq c_{i'} + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j ,$$

and hence

$$p_k \geq c_{i'} - l' + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j . \qquad \Delta$$

For a given task T_k, the parameter i in condition (2) refers to a task with a period larger than T_k's. The role of the parameter l is not as simple. An alternate construction for showing the necessity of condition (2) may make the role of the parameter l more intuitive. For a given task T_k, and a value of i, $i > k$, the parameter l can be thought of as the time between the scheduling of task T_i and the start of an execution request of task T_k. We will refer to this time as the *lag time*. Let l and i assume values that maximize condition (2) for task T_k. Let $s_i = 0$ as in the proof above. If condition (2) does not hold for task T_k, then there exist release times $s_1, \ldots, s_k, \ldots, s_{i-1}$ for tasks $T_1, \ldots, T_k, \ldots, T_{i-1}$, such that a task will miss a deadline at time $t = p_k + l$.

Assuming condition (1) holds, the maximum processor demand in the interval $[0, p_k + l]$ occurs when task T_i makes an execution request at time $t = 0$ and tasks T_1 - T_{i-1} make requests for execution time $t = 1$. This was the scenario illustrated in the proof of Theorem 3.6.1. A more intuitive construction is to shift each task's request intervals to the right so that tasks T_1 - T_{i-1} all have deadlines at the rightmost end of this interval; namely at time $p_k + l$ as shown in Figure 3.6.2 below.

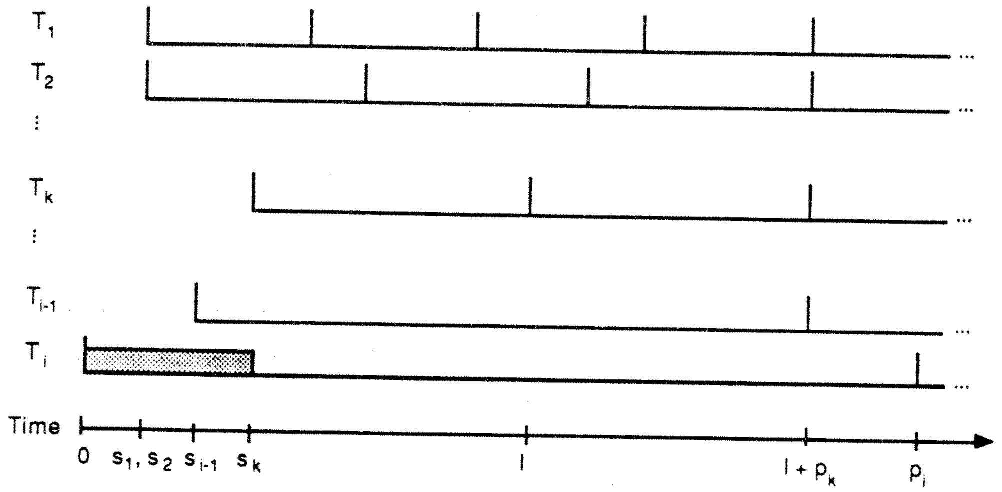


Figure 3.6.2: Alternative construction for the necessity of condition (2).

The total processor demand for the interval $[0, p_k + l]$ will be c_i plus the processor demand of tasks T_1 - T_{i-1} in the interval $[1, p_k + l]$. If τ is feasible then the total processor demand in the interval $[0, l{+}p_k]$, must satisfy

$$d_{0,l+p_k} \geq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k+l-1}{p_j} \right\rfloor c_j > l+p_k.$$

This implies condition (2). The scenario in Figure 3.6.2 occurs when tasks T_1 - T_{i-1} have release times

$$s_j = ((p_k + l - 1) \bmod p_j) + 1, \qquad (3.6.1)$$

for $1 \leq j < i$. If condition (2) does not hold and tasks T_1 - T_{i-1} have these release times, then a task will miss a deadline at time $p_k + l$. Therefore, if condition (2) does not hold for some task T_k, we can automatically generate a set of release times for the tasks in τ such that a task misses a deadline at a precise point in time.

For example, consider the following set of periodic tasks (recall T = (*release time*, *cost*, *period*)):

$$T_1 = (s_1, 2, 5),$$
$$T_2 = (s_2, 2, 9),$$
$$T_3 = (s_3, 1, 9),$$
$$T_4 = (s_4, 2, 10),$$
$$T_5 = (s_5, 3, 45).$$

The cumulative utilization of these tasks is 1.0, hence these tasks satisfy condition (1). For task $T_k = T_2$, the right hand side of condition (2) is maximized when $i = 5$, and $l = 2$. Condition (2) evaluates to

$$p_2 \geq c_5 - 2 + \sum_{j=1}^{5-1} \left\lfloor \frac{p_2+2-1}{p_j} \right\rfloor c_j, \text{ or}$$

$$9 \geq 3 - 2 + \sum_{j=1}^{5-1} \left\lfloor \frac{9+2-1}{p_j} \right\rfloor c_j = 10.$$

Since condition (2) does not hold for task T_2, by Theorem 3.6.1, these tasks cannot be feasible. Specifically, condition (2) tell us that if task $T_i = T_5$ makes an execution request 2 time units before an execution request of task T_2, then a task, say T_2, can miss a deadline at time $p_k + l = 9 + 2 = 11$. This can occur if task T_5 has release time $s_5 = 0$, and tasks T_1 - T_4 have release times $s_1 = 1$, $s_2 = 2$, $s_3 = 2$, and $s_4 = 1$. The latter release times are derived using (3.6.1). For these release times, the tasks cannot be correctly scheduled without inserted idle time. The actual task that will miss a deadline at time 11 will depend on the specific scheduling algorithm used. The behavior of the non-preemptive EDF algorithm on these tasks is shown in Figure 3.6.3 below. (Note that these tasks will also be infeasible when s_1, s_2, s_3, s_4 are all equal to 1.)

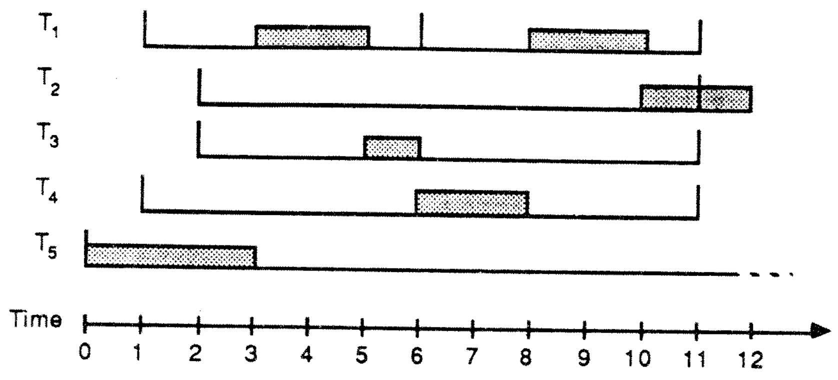


Figure 3.6.3: An example of the worst case blockage under condition (2).

We next show that conditions (1) and (2) from Theorem 3.6.1 are also necessary for the correct, non-preemptive scheduling of a set of sporadic tasks when inserted idle time is disallowed. This follows immediately from the previous theorem and our definition of sporadic tasks.

Corollary 3.6.2: A set of sporadic tasks $\tau = \{T_1, T_2, \ldots, T_n\}$, sorted in non-decreasing order by period, can be scheduled non-preemptively without inserted idle time for all possible release times only if conditions (1) and (2) from Theorem 3.6.1 hold.

Proof: Based on our definitions of periodic and sporadic tasks, a sporadic task is a generalization of a periodic task. Therefore, any conditions necessary for scheduling a set of periodic tasks must also be necessary for scheduling a set of sporadic tasks. By Theorem 3.6.1, a set of periodic tasks can be scheduled non-preemptively without inserted idle time for all possible release times only if conditions (1) and (2) hold, hence the same must hold for sporadic tasks. Δ

3.6.2 The Optimality of the EDF Discipline For All Possible Release Times

We now demonstrate the optimality of the non-preemptive EDF discipline over all non-preemptive disciplines that do not use inserted idle time for scheduling sporadic tasks for all possible release times. This means if any non-preemptive algorithm that does not use inserted idle time, can correctly schedule a set of sporadic tasks for all possible release times, then the non-preemptive EDF algorithm will also correctly schedule the tasks. To prove optimality, we show that conditions (1) and (2) are sufficient for ensuring the correctness of the non-preemptive EDF algorithm. This will show that if a set of tasks satisfy conditions (1) and (2), then no task will ever miss a deadline when scheduled by the non-preemptive EDF algorithm.

Theorem 3.6.3: Let τ be a set of sporadic processes $\{T_1, T_2, \dots, T_n\}$, sorted in non-decreasing order by period. The non-preemptive EDF discipline is correct with respect to τ for all possible release times if conditions (1) and (2) from Theorem 3.6.1 hold.

Proof: (By contradiction.)

Assume the contrary, i.e., that conditions (1) and (2) from Theorem 3.6.1 hold and yet there exists a set of values for the s_i such that a task T_k misses a deadline at some point in time when τ is scheduled by the non-preemptive EDF algorithm. Without loss of generality, assume that T_k is the first task to miss a deadline. (In the case of simultaneous missed deadlines, let T_k be the task with smallest index.) The proof has the same structure as the proof Theorem 3.5.1.

Consider the first execution request of T_k that misses a deadline. Let t_d be the deadline of this request. For the remaining tasks whose release times are less than t_d, as in the proof of Theorem 3.5.2, these tasks either do, or do not, have an execution request interval that contains the point t_d. For the tasks with request intervals that contain t_d, we consider the following two cases.

<u>Case 1</u>: None of the request intervals that contain the point t_d are scheduled prior to t_d.

The proof of this case is identical to Case 1 in the proof of Theorem 3.5.2. If the premise of this case is true then condition (1) must have been false, contradicting our original assumption. Therefore, if conditions (1) and (2) hold and the non-preemptive EDF discipline fails, then some of the request intervals that contain the point t_d must have been scheduled prior to t_d.

Case 2: Some of the request intervals that contain the point t_d are scheduled prior to time t_d.

Let T_i be a task who does not have a deadline at t_d, and whose request interval that contains t_d is scheduled prior to t_d. Without loss of generality, assume T_i is the last such task to be scheduled prior to t_d. Let $t_s = t_d - p_k$, i.e. t_s is the point in time at which task T_k makes its request for execution that has a deadline at t_d. Let $t_{s'}$ be the point in time at which T_i makes its request for execution that contains t_d. Let t_i be the point in time in which the request interval of T_i that contains t_d is scheduled. If task T_k misses a deadline at time t_d then we have the following four facts.

Claim 1: $t_i < t_s$. (The request interval of task T_i that contains t_d is scheduled before the request interval of task T_k that misses a deadline at t_d. (See Figure 3.6.4.))

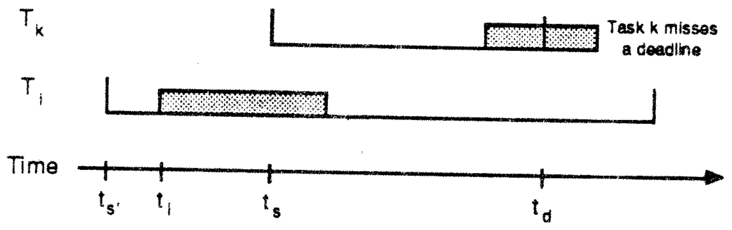


Figure 3.6.4: Task T_i is scheduled before time t_s.

Proof of claim: Let $t_k \geq t_s$ be the point in time at which the request interval of T_k with deadline at t_d is scheduled. By the assumption of Case 2, the request interval of T_i that contains t_d is scheduled before t_d ($t_i < t_d$). Since T_k has a nearer deadline than T_i throughout the interval $[t_s,t_d]$, if $t_i \geq t_s$, then the non-preemptive EDF discipline would schedule T_k before T_i. Therefore, if $t_i \geq t_s$, then it must be the case that $t_k + c_k \leq t_i < t_d$. However this implies that T_k does not miss a deadline at t_d which contradicts the assumption of the main proof. Therefore we must have $t_i < t_s$. Δ

Claim 2: No task with index greater than i is scheduled in the interval $[t_i,t_d]$.

Proof of claim: Since T_i is the last task with a request interval containing t_d scheduled prior to t_d, other than T_i, every task scheduled in $[t_i,t_d]$ has a deadline at or before t_d.

Assume there exists a task T_j, $j > i$, with a request interval that has a deadline somewhere in the interval $[t_i,t_d]$ as shown in Figure 3.6.5.

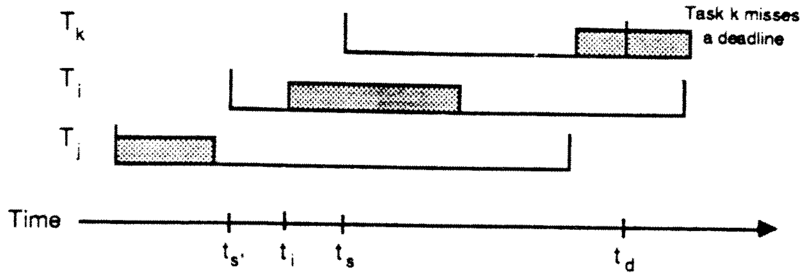


Figure 3.6.5: No task with index greater than i is scheduled in the interval $[t_i,t_d]$.

Since $t_d - t_{s'} < p_i \leq p_j$, the request interval of T_j that has a deadline in the interval $[t_i,t_d]$ must have started before $t_{s'}$. Therefore, since T_i has a deadline after t_d, the EDF algorithm will choose T_j before T_i in the interval $[t_{s'},t_d]$. Hence, it is not possible for a task with index greater than i to be scheduled in the interval $[t_i,t_d]$. Δ

Claim 3: Other than T_i, no task that is scheduled in $[t_i,t_d]$ could have made a request for execution at t_i.

Proof of claim: Again, since T_i is the last task with a request interval containing t_d scheduled prior to t_d, other than T_i, every task scheduled in $[t_i,t_d]$ has a deadline at or before t_d. Therefore, if a task $T_{i'}$, that is scheduled in $[t_i,t_d]$ had made a request for execution at t_i, the non-preemptive EDF discipline would have scheduled $T_{i'}$ instead of T_i at time t_i. Since T_i was scheduled at t_i, we can conclude that no task scheduled in $[t_i,t_d]$ made a request for execution at t_i. Δ

Claim 4: There cannot have been any idle time in the interval $[t_{s'},t_d]$.

Proof of claim: Assume the contrary, that is, there exists at least one idle period in the interval $[t_{s'},t_d]$. In order for this to be the case, the idle period clearly must have occurred after the execution request of T_i that contains t_d has completed execution. Let t_0 be the end of the last idle period that occurs in the interval $[t_{s'},t_d]$, $t_{s'} < t_i + c_i < t_0 \leq t_d$. The analysis of Case 1 of Theorem 3.5.2 can be applied on the interval $[t_0,t_d]$ to show that if task T_k misses a deadline at time t_d, then condition (1) could not have been true. As this contradicts our original assumption that conditions (1) and (2) both held, we conclude that there could not have existed any idle time in the interval $[t_{s'}, t_d]$.

Δ

Returning to the main proof of Case 2, we will show that if T_i is scheduled prior to t_d, then there must have existed enough processor time in $[t_s, t_d]$ to execute T_k. To do this we will derive a bound on the processor demand for the larger interval $[t_i, t_d]$. From Claim 2, we know that we need only consider tasks T_1 - T_{i-1} in computing d_{t_i, t_d}.

Since the request interval of T_i scheduled at t_i has a deadline after t_d, all outstanding requests for execution at t_i with deadlines before t_d must have been satisfied by t_i. Therefore, we need not consider request intervals of tasks T_1 - T_{i-1} that contain the point t_i when computing d_{t_i, t_d}. Similarly, since T_i was the last task with deadline after t_d scheduled prior to t_d, we need not consider request intervals of tasks T_1 - T_{i-1} that contain the point t_d. Lastly, from Claim 3 we know that none of tasks T_1 ... T_{i-1} made a request for execution at time t_i. These three facts can indicate that the total processor demand in $[t_i, t_d]$ is

$$d_{t_i, t_d} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t_d - (t_i + 1)}{p_j} \right\rfloor c_j .$$

For example, for the tasks below, the request intervals that need to be scheduled in $[t_i, t_d]$ are lightly shaded in Figure 3.6.6 below.

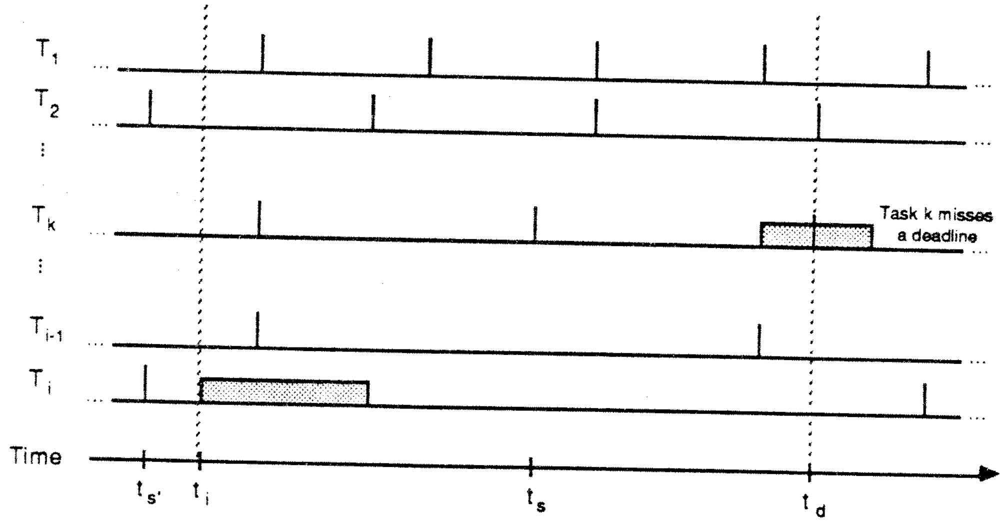


Figure 3.6.6: Request intervals that must be scheduled in the interval $[t_i, t_d]$.

Let $l = (t_d - t_i) - p_k$. Substituting l into the above inequality we get

$$d_{t_i,t_d} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j . \tag{3.6.2}$$

Now from Claim 4 we know that there cannot be any idle time in $[t_i,t_d]$. Therefore, because a task missed a deadline at t_d, we must have $t_d - t_i < d_{t_i,t_d}$, or from the definition of l,

$$l + p_k < d_{t_i,t_d} .$$

Combining this with (3.6.2) gives

$$l + p_k < c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j ,$$

or simply

$$p_k < c_i - l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j .$$

Now since $p_i > t_d - t_i$, we know that $0 < l < p_i - p_k$. Therefore the above inequality contradicts the fact that condition (2) was assumed to be true. This concludes Case 2.

We have shown that in all cases, if the non-preemptive EDF algorithm fails then either condition (1) or condition (2) must have been violated. This proves the theorem. Δ

Theorem 3.6.3 has shown that with respect to the class of scheduling policies that do not use inserted idle time, the non-preemptive EDF scheduling discipline is an optimal non-preemptive discipline for sporadic tasks when all possible release times are considered. We now show that the non-preemptive EDF discipline is also optimal for a set of periodic tasks for all possible release times. This again follows immediately from the previous theorem and our definition of a sporadic task.

Corollary 3.6.4: Let τ be a set of periodic processes $\{T_1, T_2, \ldots, T_n\}$, sorted in non-decreasing order by period. The non-preemptive EDF discipline is correct with respect to τ for all possible release times if conditions (1) and (2) from Theorem 3.6.1 hold.

Proof: Recall that a periodic task is a sporadic task. Therefore, if conditions (1) and (2) are sufficient for guaranteeing the correctness of the non-preemptive EDF discipline when scheduling a set of sporadi .sks for all possible release times, then conditions (1) and (2) are also sufficient for guara..eeing the correctness of the non-preemptive EDF discipline when scheduling a set of periodic tasks for all possible release times. Δ

3.6.3 The Optimality of the EDF Discipline For Arbitrary Release Times

We can in fact show a stronger optimality result for sporadic tasks. Based on our definition of a sporadic task, we can show that the non-preemptive EDF discipline is an optimal discipline for scheduling sporadic tasks with arbitrary release times.

Lemma 3.6.5: A set of sporadic tasks τ, can be feasible for an arbitrary set of release times only if τ is feasible for all possible release times.

Proof: By the definition of sporadic tasks, a sporadic task will wait for an arbitrary amount of time between the end of one request interval and the start of the next. Therefore, after all tasks have been released, there can exist a time t such that a task, or group of tasks in τ, make requests for execution at time t, and there are no outstanding requests for execution at time t. In other words, if these tasks had not made execution requests at t then the processor would have been idle for some non-zero interval starting at t. At time t, τ is effectively "starting over" with a set of "release times" that are independent from the initial release times. Therefore, a set of sporadic tasks with arbitrary release times can be feasible only if they are feasible for all possible release times. Δ

Theorem 3.6.6: With respect to the class of scheduling algorithms that do not use inserted idle time, the non-preemptive EDF discipline is an optimal discipline for scheduling sporadic tasks with arbitrary release times.

Proof: The proof follows immediately from Theorem 3.6.3 and Lemma 3.6.5. Δ

3.7 Complexity of Deciding Feasibility For Sporadic Tasks

Algorithms for answering a boolean question such as "Are a set of sporadic tasks feasible?" are termed *decision procedures*. In this section we develop a simple tabular decision procedure for deciding if a set of sporadic tasks are feasible on a uniprocessor when preemption is disallowed.

Since the non-preemptive EDF discipline is optimal for sporadic tasks, in order to decide if a set of sporadic tasks is feasible on a uniprocessor, we need only consider if conditions (1) and (2) from Theorem 3.6.1 hold. Determining if condition (1) holds is straightforward and can be computed in time $O(n)$. In this section we give an $O(n^2 p_n)$ decision procedure for determining if condition (2) holds.

For each task T_k, we can compute

$$\operatorname{MAX}_{i>k}\left(c_i + \operatorname{MAX}_{0<l<p_i-p_k}\left(-l + \sum_{j=1}^{i-1}\left\lfloor\frac{p_k+l-1}{p_j}\right\rfloor c_j\right)\right) \tag{3.7.1}$$

as follows. Let

$$f_{k,j}(l) = \left\lfloor\frac{p_k+l-1}{p_j}\right\rfloor c_j ,$$

and

$$S_{k,i}(l) = c_i - l + \sum_{j=1}^{i-1} f_{k,j}(l) .$$

To determine if condition (2) holds we compute $S_{k,i}(l)$ for $1 \le k \le n\text{-}1$ and $k < i \le n$. Table 3.7.1 lists in tabular form the $S_{k,i}$ that must be computed. The largest entry in each column gives the maximum value of condition (2) for a particular value of i. Comparing the maximum entry of the maximum's from each column gives the maximum value of condition (2) for task T_k. Therefore, in order to determine if task T_k alone is feasible, we need to compute all the entries in the table below and check whether p_k is greater than or equal to the largest entry in the table.

Table 3.7.1: A method for computing condition (2) from Theorem 3.6.1.

	$i = k + 1$	$k + 2$	$\dots$	$n - 1$	n
$l = 1$	$S_{k,k+1}(1)$	$S_{k,k+2}(1)$		$S_{k,n-1}(1)$	$S_{k,n}(1)$
2	$S_{k,k+1}(2)$	$S_{k,k+2}(2)$		$S_{k,n-1}(1)$	$S_{k,n}(2)$
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
$p_{k+1}-p_k$	$S_{k,k+1}(p_{k+1}-p_k)$	$S_{k,k+2}(p_{k+1}-p_k)$		$S_{k,n-1}(p_{k+1}-p_k)$	$S_{k,n}(p_{k+1}-p_k)$
$\vdots$		$\vdots$		$\vdots$	$\vdots$
$p_{k+2}-p_k$		$S_{k,k+2}(p_{k+2}-p_k)$		$S_{k,n-1}(p_{k+1}-p_k)$	$S_{k,n}(p_{k+2}-p_k)$
$\vdots$				$\vdots$	$\vdots$
$p_{n-1}-p_k$				$S_{k,n-1}(p_{n-1}-p_k)$	$S_{k,n}(p_{n-1}-p_k)$
$\vdots$					$\vdots$
p_n-p_k					$S_{k,n}(p_n-p_k)$

Note that for all l, $S_{k,i+1}(l) = S_{k,i}(l) + f_{k,i}(l) - c_i + c_{i+1}$. This suggests that we can reuse computations of $S_{k,i}$ as shown in Table 3.7.2.

Table 3.7.2: An improved method for computing equation (3.7.1).

	$i = k + 1$	$k + 2$	$\dots$	n
$l = 1$	$S_{k,k+1}(1)$	$S_{k,k+1}(1) + f_{k,k+1}(1) - c_{k+1} + c_{k+2}$		$S_{k,n-1}(1) + f_{k,n-1}(1) - c_{n-1} + c_n$
2	$S_{k,k+1}(2)$	$S_{k,k+1}(2) + f_{k,k+1}(2) - c_{k+1} + c_{k+2}$		$S_{k,n-1}(2) + f_{k,n-1}(2) - c_{n-1} + c_n$
$\vdots$	$\vdots$	$\vdots \quad \vdots$		$\vdots \quad \vdots$
$p_{k+1}-p_k$	$S_{k,k+1}(l)$	$S_{k,k+1}(l) + f_{k,k+1}(l) - c_{k+1} + c_{k+2}$		$S_{k,n-1}(l) + f_{k,n-1}(l) - c_{n-1} + c_n$
$p_{k+1}-p_k+1$		$S_{k,k+2}(l)$		$S_{k,n-1}(l) + f_{k,n-1}(l) - c_{n-1} + c_n$
$\vdots$		$\vdots$		$\vdots \quad \vdots$
$p_{k+2}-p_k$		$S_{k,k+2}(l)$		$S_{k,n-1}(l) + f_{k,n-1}(l) - c_{n-1} + c_n$
$p_{k+2}-p_k+1$				$S_{k,n}(l)$
$\vdots$				$\vdots$
p_n-p_k				$S_{k,n}(l)$

Since we can evaluate the function $f_{k,j}(l)$ in constant time, the function $S_{k,i}(l)$ can be computed in time $O(i)$. Therefore, in the worst case we can compute a row in this table in time $n + O(n)$ or simply $O(n)$. For task T_k we can compute the entire table in time $O(np_n)$.

Since we need to construct a table for $n - 1$ tasks, the total time required to evaluate condition (2) is no worse than $O(n^2 p_n)$.

Note that this bound depends on the value of one of the inputs. Since the size of an input cannot be expressed as a polynomial in the length of the input, our decision procedure is technically an exponential time algorithm. This does not necessarily imply intractability however. For any bound on the size of the inputs, our algorithm is polynomial in this bound. Therefore, if we impose an upper bound on the size of the inputs, say 2^{16}, then the decision procedure is polynomial for these restricted problems. Such an algorithm is said to run in *pseudo-polynomial time* [Garey & Johnson 79]. For the task descriptions that we are most likely to encounter in practice, we can efficiently determine the feasibility of the tasks. Lastly, we mention that it is not known if (3.7.1) can be computed in time polynomial in n alone. Therefore it is possible that the true complexity of evaluating (3.7.1), and hence the complexity of deciding the feasibility of a set of sporadic tasks, is NP-complete.

3.8 Scheduling With Inserted Idle Time

All of our optimality results for non-preemptive scheduling have been with respect to the class of non-preemptive disciplines that do not use inserted idle time. That adopting inserted idle time yields a more powerful scheduling algorithm can be seen by the following example. Consider the two periodic tasks

$$T_1 = (9, 8, 20), \text{ and}$$
$$T_2 = (0, 23, 40).$$

These tasks cannot be scheduled correctly using any non-preemptive discipline that does not use inserted idle time. Any such discipline would necessarily schedule T_2 at time 0 and T_1 at time 23 as shown in the simulation depicted in Figure 3.8.1.

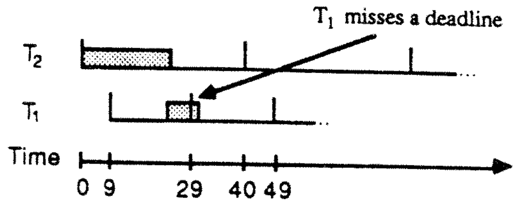


Figure 3.8.1: Scheduling without inserted idle time.

These choices of release times force T_1 to miss a deadline at time 29 when inserted idle time is disallowed. These two tasks can, however, be scheduled correctly by a non-preemptive algorithm that idles the processor as shown in Figure 3.8.2.

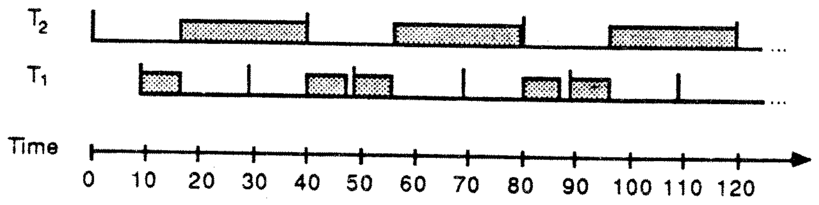


Figure 3.8.2: Scheduling with inserted idle time.

Note that inserted idle time is only of potential benefit in the non-preemptive scheduling domain. Unfortunately however, we are not aware of any literature concerning scheduling with inserted idle time for cyclic tasks. It remains an open problem to determine if an efficient decision procedure exists for deciding the feasibility of sporadic tasks when inserted idle time is allowed. Given the non-determinism in the behavior of sporadic tasks, we are skeptical that such a decision procedure can be found for sporadic tasks. In order for inserted idle time to function correctly, it would seem to require that the scheduler know when tasks will make their next requests for execution. In general, this will not be possible for sporadic tasks.

3.9 Non-Preemptive Scheduling of Periodic Tasks

For periodic tasks, we limited our attention in Section 3.6.2 to scheduling problems where all possible release times are considered. The existence of an optimal scheduling discipline for periodic tasks is quite sensitive to knowledge of the release times. In this section we provide evidence that there may not exist an optimal non-preemptive uniprocessor scheduling discipline for periodic tasks with arbitrary release times.

For this section, we will need to refine our definition of optimality to include some notion of efficiency. So far, we have assumed that a scheduling discipline can determine which task to schedule next in zero time. With this assumption, a scheduler that enumerated all possible schedules would be an optimal, albeit uninteresting scheduler. To be of practical use, we will require an optimal scheduling discipline to take time polynomial in the number of tasks to make each scheduling decision. This requirement is in addition to correctly scheduling all sets of tasks that are feasible on a uniprocessor. With this new notion of optimality, we will show that if there exists an optimal non-preemptive uniprocessor scheduling discipline for the problem of scheduling periodic tasks with arbitrary release times, then P = NP.

To begin, we first show that the complexity of deciding if a set of periodic tasks is feasible on a uniprocessor when one is allowed to consider any non-preemptive scheduling discipline (including those with inserted idle time) is NP-hard in the strong sense [Garey & Johnson 79]. This means that unless P = NP, a pseudo-polynomial time algorithm does not exist for deciding feasibility of periodic tasks with arbitrary release times. This demonstrates that the problem is intractable. This decision problem can be formally stated as follows.

NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS: Given a set of periodic tasks $\tau = \{T_i = (s_i, c_i, p_i) \mid 1 \le i \le n\}$ with $s_i, c_i, p_i \in \mathbf{Z}^+$, is it possible to correctly schedule τ non-preemptively, on a uniprocessor?

Theorem 3.9.1: NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS is NP-hard in the strong sense.

Proof: We shall give a pseudo-polynomial time transformation from the 3-PARTITION problem ([Garey & Johnson 79]) to NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS. An instance of the 3-PARTITION problem consists of a finite set A of $3m$ elements, a bound $B \in \mathbf{Z}^+$, and a "size" $s(a) \in \mathbf{Z}^+$ for each $a \in A$, such that each $s(a)$ satisfies $B/4 < s(a) < B/2$, and $\sum_{j=1}^{3m} s(a_j) = Bm$. The problem is to determine if A can be partitioned into m disjoint sets $S_1, S_2, \ldots, S_m$ such that, for $1 \le i \le m$, $\sum_{a \in S_i} s(a) = B$. (With the above constraints on the element sizes, note that every S_i will contain exactly three elements from A.)

The transformation is performed as follows. Let $A = \{a_1, a_2, a_3, \ldots, a_{3m}\}$, $B \in \mathbf{Z}^+$, and $s(a_1), s(a_2), s(a_3), \ldots, s(a_{3m}) \in \mathbf{Z}^+$, constitute an arbitrary instance of the 3-PARTITION problem. We create an instance of NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS by constructing a set τ of $n = 3m + 2$ periodic tasks as follows:

$$\begin{aligned}\tau = \{\ & T_1 = (0B, 8B, 20B),\\ & T_2 = (9B, 23B, 40B),\\ & \forall j,\ 3 \le j \le 3m+2:\ T_j = (8B, s(a_{j-2}), 40Bm)\ \}.\end{aligned}$$

This construction can clearly be done in polynomial time with the largest number created in the new problem instance being $40Bm$. In our instance of NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS, note that the system utilization is

$$\sum_{j=1}^{n} \frac{c_j}{p_j} = \frac{8}{20} + \frac{23}{40} + \frac{\sum_{j=1}^{3m} s(a_j)}{40Bm} = \frac{39}{40} + \frac{Bm}{40Bm} = 1.$$

By our choice of release times for T_1 and T_2, τ can be feasible only if T_1 and T_2 are scheduled according to the following Lemma.

Lemma 3.9.2: τ is feasible on a uniprocessor under any non-preemptive scheduling discipline only if T_2 is scheduled at points in time $9B + 40Bk$, and all the request intervals of T_1 that begin at time $20B + 40Bk$, are scheduled at time $40B(k+1) - 8$, for all $k \ge 0$ as shown in Figure 3.9.1 below.

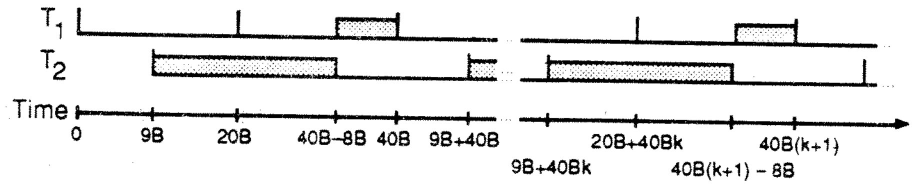


Figure 3.9.1: The only feasible execution schedule for τ.

Proof: Without loss of generality, assume $B = 1$ (i.e., assume all quantities are scaled by B). In order for τ to be feasible, the $(k+1)^{st}$ request interval of T_2 can be scheduled only in the interval

$$[s_2+kp_2,\ s_2+(k+1)p_2 - c_2] = [9+40k,\ (9+40(k+1)) - 23]$$
$$= [9+40k,\ 26+40k]$$

for all $k \geq 0$. If the $(k+1)^{st}$ request interval of T_2 is scheduled anywhere in this interval other than at $9 + 40k$, then the request interval of T_1 made at time $20 + 40k$ will miss its deadline. Similarly, In order for τ to be feasible, the $2(k+1)^{st}$ request interval of T_1 can be scheduled only in the interval

$$[s_1+(2(k+1)-1)p_1,\ s_1+2(k+1)p_1 - c_1] = [20(2(k+1) - 1),\ 20(2(k+1)) - 8]$$
$$= [20+40k,\ 32+40k]$$

for all $k \geq 0$. If the $2(k+1)^{st}$ request interval of T_1 is scheduled anywhere in this interval other than at $40(k+1) - 8$, then the request interval of T_2 made at time $9 + 40k$ must have been scheduled after the $2(k+1)^{st}$ request interval of T_1 and will therefore miss its deadline.

Δ

For example, when T_1 and T_2 are scheduled by the non-preemptive EDF algorithm, every $40B$ time units, there will exist an interval in which the processor will be idle for exactly B time units. It follows that in the interval $[0, 40Bm]$, there will be exactly m idle periods, each of duration B. Define $\tau_{3P} = \{T_3, T_4, T_5, \ldots, T_{3m+2}\}$. τ will be feasible under the non-preemptive EDF discipline if and only if the EDF algorithm can schedule the tasks in τ_{3P} in these m idle periods as shown in Figure 3.9.2. The following lemma generalizes this observation for all non-preemptive scheduling disciplines.

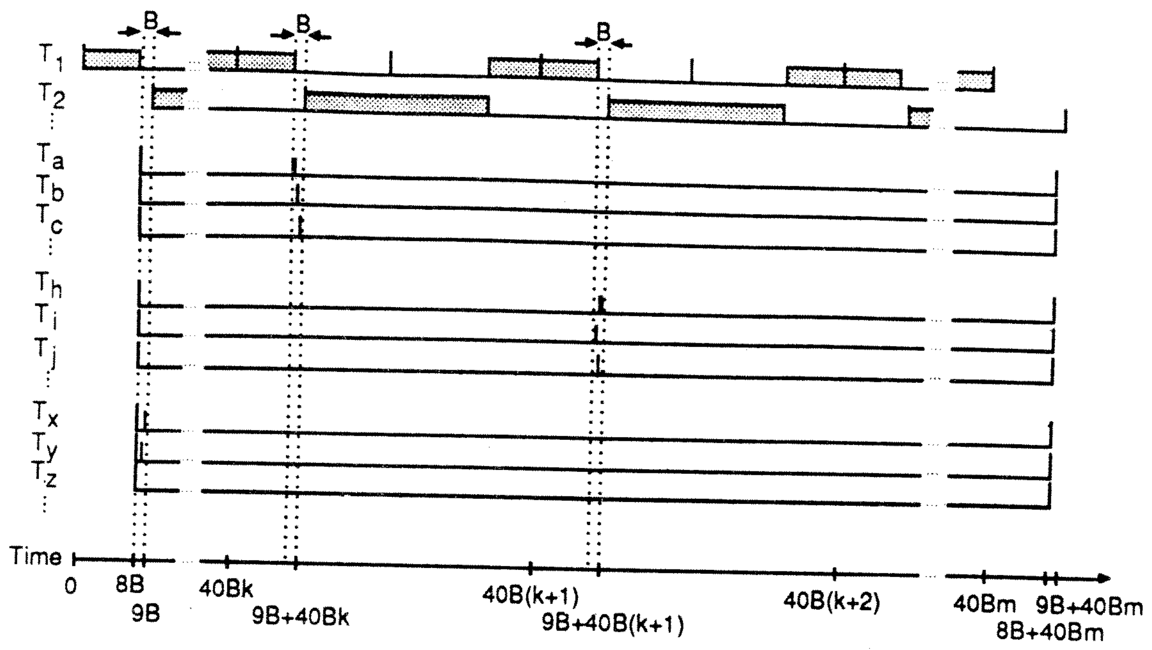


Figure 3.9.2: Execution schedule of τ for any scheduling discipline.

Lemma 3.9.3: τ is feasible on a uniprocessor under any non-preemptive scheduling discipline if and only if there exists a partition of the tasks in τ_{3P} into m disjoint sets S_1, $S_2, \ldots, S_m$, such that for each set S_i, $\sum_{T_j \in S_i} c_j = B$.

Proof: Sufficiency: Without loss of generality, again assume $B = 1$. If τ_{3P} can be partitioned into m disjoint sets $S_1, S_2, \ldots, S_m$ according to the restrictions above, then τ can be scheduled correctly by simply scheduling T_1 at all points in time $40k$ and $40(k+1) - 8$, scheduling T_2 at all points in time $9 + 40k$, for all $k \geq 0$, and during the interval $[40k+8, 9+40k]$ scheduling the tasks in S_{i+1} where $i = k \bmod m$.

Necessity: If all the request intervals of T_1 that begin at time $20B + 40Bk$, are scheduled at time $40B(k+1) - 8$, then if τ is to be feasible, all other request intervals of T_1 (those that begin at time $40Bk$ for all $k \geq 0$), must be scheduled in the interval $[40Bk, 40Bk+B]$ for all $k \geq 0$. Therefore, by Lemma 3.9.2, in every interval $I_k = [40Bk, 40B(k+1)]$, there is at least one, and at most two periods of time, whose cumulative length is B, in which neither T_1 nor T_2 executes. We call an interval of time in which neither T_1 nor T_2 executes a T_1T_2 *idle interval*. For each interval I_k, the T_1T_2 idle interval, or intervals, are contained within the larger interval $[40Bk, 9B+40Bk]$. Note that no T_1T_2 idle interval from I_k can overlap with any T_1T_2 idle interval from either I_{k+1} or I_{k-1}.

Since there are at least $m-1$ I_k intervals with T_1T_2 idle intervals wholly contained in $[8B, 8B+40Bm]$, τ is feasible on a uniprocessor only if there exists a partition of τ_{3P} into at least $m-1$ disjoint sets $S_1, S_2, \dots, S_{m-1}$, such that for each set S_i, $\sum_{T_j \in S_i} c_j = B$. However, given the constraints on the costs of tasks in τ_{3P} imposed by the 3-PARTITION problem, if there exists a partition of the tasks in τ_{3P} into these $m-1$ disjoint sets S_i, then there will exist three tasks in τ_{3P} that are not members of any of the S_i. We can combine these tasks into a new disjoint subset S_m. Note that it must be the case that $\sum_{T_j \in S_m} c_j = B$. Therefore we can conclude that τ is feasible on a uniprocessor under any non-preemptive scheduling discipline only if there exists a partition of tasks τ_{3P} into m disjoint sets $S_1, S_2, \dots, S_m$, such that for each set S_i, $\sum_{T_j \in S_i} c_j = B$. Δ

With Lemma 3.9.3 we can now use a solution to NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS to solve the 3-PARTITION problem. For an instance of 3-PARTITION, simply construct a set of periodic tasks as shown in the beginning of this proof, and feed this set of tasks into a decision procedure for NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS. By Lemma 3.9.3, the answer from the NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS decision procedure is the answer to the 3-PARTITION question for this problem instance. Since 3-PARTITION is known to be NP-complete in the strong sense [Garey & Johnson 79], NON-PREEMPTIVE SCHEDULING OF PERIODIC TASKS must be at least NP-hard in the strong sense. Δ

For periodic tasks, the intractability of deciding feasibility arises from our inability to efficiently determine if the worst case blockage that a task may experience while waiting to execute, can ever actually occur. Because of the small amount of non-determinism in the behavior of sporadic tasks, we were able to argue that a sporadic task can always experience its worst case blockage and hence did not have to worry if it actually occurred or not.

The situation is actually bleaker for periodic tasks. The construction of τ in Theorem 3.9.1 can also be used to show that if an optimal non-preemptive uniprocessor scheduling discipline existed for scheduling periodic tasks and this discipline took only a polynomial amount of time (in the length of the input) to make each scheduling decision, then P = NP. We will give a pseudo-polynomial time algorithm for deciding 3-PARTITION based on the existence of an optimal non-preemptive scheduling discipline for periodic tasks. The main idea is that we only need to check a pseudo-polynomial length portion of the schedule generated by an optimal non-preemptive discipline when scheduling τ, in order to decide the embedded 3-PARTITION problem instance.

Corollary 3.9.4: Let τ be a set of periodic tasks constructed from an instance of the 3-PARTITION problem as described in the proof of Theorem 3.9.1. If τ is not feasible then τ will miss a deadline in the deadline in the interval $[0, 8B+80Bm]$.

Proof: Assume τ is not feasible. By Lemma 3.9.3, there is no partition of τ_{3P} into m disjoint sets $S_1, S_2, \ldots, S_m$, such that for each set S_i, $\sum_{T_j \in S_i} c_j = B$. Therefore if τ does not miss a deadline during the interval $[0, 8B+40Bm]$, τ_{3P} must have utilized T_1T_2 idle intervals whose cumulative length is greater than Bm. That is, some of the T_1T_2 idle interval from interval I_m must have been utilized to schedule some of the first execution requests of tasks in τ_{3P}. This will leave T_1T_2 idle intervals in the interval $[8B+40Bm, 8B+80Bm]$ whose cumulative length is less than Bm, for the scheduling of tasks in τ_{3P}'s second execution requests. Therefore, since τ_{3P} will require an amount of computation time equal to Bm, in the interval $[8B+40Bm, 8B+80Bm]$, τ is certain to miss a deadline in this interval.
Δ

Corollary 3.9.5: If there exists an optimal, non-preemptive, uniprocessor scheduling discipline for scheduling periodic tasks then P = NP.

Proof: Assume there exists such an optimal scheduling discipline. From an instance of the 3-PARTITION problem, construct a set τ of periodic tasks as described in the proof of Theorem 3.9.1. Simulate the optimal scheduling discipline on τ for $8B + 80Bm$ time units and simply check to see if any tasks miss a deadline in the interval $[0, 8B+80Bm]$. The simulation and the checking of the schedule produced by the optimal discipline can clearly be performed in time proportional to $40Bm$. From Corollary 3.9.4 and Lemma 3.9.3, if some task missed a deadline then there is a negative answer to the 3-PARTITION problem instance and if no task missed a deadline then there is an affirmative answer. Therefore, since 3-PARTITION is NP-complete in the strong sense and since we have given a pseudo-polynomial time algorithm for deciding 3-PARTITION, P = NP.
Δ

Unless P = NP, Corollary 3.9.5 shows that we will not be able to develop an optimal non-[illegible]mptiv[illegible] [illegible]cheduling discipline for scheduling periodic tasks with arbitrary release times.

3.10 Discussion and Summary

Our task model is most similar to the model of Liu and Layland [Liu & Layland 73]. The task model outlined in Section 3.2 extends theirs with the addition of sporadic tasks and arbitrary release times. Other related models have generalized the concept of a deadline so that an execution request of a task can have a deadline that is nearer than the arrival of the next execution request [Mok 83]. These models are too general for our purposes.

In summary, we have the following key results. For preemptive scheduling, the EDF discipline is an optimal scheduling discipline if preemption is allowed at arbitrary points. Table 3.10.1 summarizes these results. The important aspects of these results are:

- feasibility is not dependent on knowledge of release times,
- sporadic and periodic tasks are equivalent in terms of feasibility, and
- feasibility can be determined efficiently.

Table 3.10.1: Optimal preemptive scheduling disciplines.

	Arbitrary Release Times	All Possible Release Times
Sporadic Tasks	Preemptive EDF	Preemptive EDF
Periodic Tasks	Preemptive EDF	Preemptive EDF

For non-preemptive scheduling, the EDF discipline is optimal for sporadic tasks and for periodic tasks only when all possible release times are considered. Unless P = NP, there does not exist an optimal scheduling discipline for periodic tasks with arbitrary release times. In all cases, the optimality is with respect to the class of scheduling policies that do not use inserted idle time. Table 3.10.2 summarizes these results. Table 3.10.3 gives the complexity measures for deciding feasibility. The important aspects of these results are:

- sporadic and periodic tasks are not equivalent in terms of feasibility,

- for sporadic tasks, feasibility is not dependent on knowledge of release times,
- for periodic tasks, feasibility is dependent on knowledge of release times,
- feasibility of sporadic tasks can be determined efficiently for tasks with bounded periods, and
- the problem of determining the feasibility of a set of periodic tasks with arbitrary release times is intractable.

Table 3.10.2: Optimal non-preemptive scheduling disciplines.

	Arbitrary Release Times	All Possible Release Times
Sporadic Tasks	Non-preemptive EDF	Non-preemptive EDF
Periodic Tasks	If a polynomial time algorithm algorithm exists, then P = NP	Non-preemptive EDF

Table 3.10.3: Complexity of deciding feasibility when preemptive is disallowed.

	Arbitrary Release Times	All Possible Release Times
Sporadic Tasks	Pseudo-polynomial time $O(n^2 p_n)$	Pseudo-polynomial time $O(n^2 p_n)$
Periodic Tasks	NP-hard in the strong sense	Pseudo-polynomial time $O(n^2 p_n)$

Although there may not exist an optimal algorithm for scheduling periodic tasks with arbitrary release times, Corollary 3.6.4 has established sufficient conditions for ensuring the correctness of the non-preemptive EDF discipline. This is certainly useful for problems such as scheduling periodic tasks when the release times are unknown.

Chapter 4

An Implementation Strategy For Sporadic Tasks with Shared Resources

4.1 Introduction

In the previous chapter, tasks were presumed to be able to commence execution when they made a request for execution. This is not a realistic assumption as tasks often share resources that can only be used by a single task at a time. Therefore, when a task makes a request for execution, it may require a resource that is already in use by some other task. In this chapter we extend the task model of the previous chapter to include a set of serially reusable resources that are shared among the tasks. The resources will be software objects that a task will require during a portion of its computation. We will assume that each resource can be used by at most one task at a time. The problem is to schedule the tasks on a uniprocessor such that all tasks meet their deadlines and all resources are never used simultaneously by more than one task.

A solution to this problem will require an integration of the basic results for preemptive and non-preemptive scheduling. Given the intractability results for non-preemptive scheduling of periodic tasks, for the current problem we will limit our attention to the study of sporadic tasks. Any feasibility condition developed in this section for sporadic tasks will be a sufficient condition for the feasibility of periodic tasks.

To be precise, the problem of sequencing tasks with shared resources extends beyond the realm of a simple processor scheduling problem. Besides developing a policy for determining which task should execute next on the processor, we will also need a policy for synchronizing access to the shared resources. Therefore, we will phrase this problem as one of determining an *implementation strategy* for sporadic tasks with shared resources. An implementation strategy will be the integration of a processor scheduling policy and a synchronization discipline. We propose an implementation strategy for sporadic tasks where access to each shared resource is controlled by a simple lock, implemented by a monitor with `WAIT` and `BROADCAST` synchronization primitives [Hoare 74, Lampson & Redell 80]. For task scheduling, we further propose that tasks be selected for execution according to an *earliest deadline first* (EDF) selection rule as in Chapter 3. We will show that for certain characterizations of a task's resource requirements, these two disciplines can, in principle, result in an implementation strategy for sporadic tasks that will be optimal. That is, under restricted conditions, this combination of synchronization and scheduling disciplines will be able to correctly execute a set of sporadic tasks whenever it is possible to do so.

We will consider three characterizations of a task's resource requirements (in order of increasing complexity):

- a set of sporadic tasks that share a single resource,
- a set of sporadic tasks that share a set of resources, but where each task requires only a single resource, and
- a set of sporadic tasks that share a set of resources and where each task may require multiple resources but will only use one resource at a time.

We begin in Sections 4.2 and 4.3 by extending the task model of Chapter 3, and formally defining our implementation strategy. Sections 4.4 - 4.6 address the each of the characterizations of resource usage listed above. Section 4.7 compares our approach to others that have appeared in the literature and briefly discusses some extensions of our scheduling results to multiple processors.

The results of this study of cyclic tasks will be used in the following chapter to show that there exists an implementation strategy for ensuring the temporal correctness of designs constructed using our discipline.[1] A design is temporally correct if all interconnected pairs of processes are guaranteed to adhere to the RT/PC paradigm.

4.2 A Tasking Model With Shared Resources

4.2.1 Serially Reusable resources

As in Chapter 3, a sporadic task T is a 3-tuple (s, c, p) where s is the release time, c is the computational cost, and p is the period of the task. We will assume that a set of sporadic tasks τ, shares m software resources $R_1, R_2, ..., R_m$. A software resource could be, for example, a pool of buffers, a portion of a database, or a global data structure. Whenever a task uses a shared resource, the task must be guaranteed exclusive access to the resource for the duration of its use. Each execution request of task T_i consists of a sequence of n_i *phases* labeled $r_{i1}, r_{i2}, r_{i3}, ..., r_{in_i}$. The j^{th} phase of task T_i is represented by an integer r_{ij}, $0 \le r_{ij} \le m$, indicating the resource required by T_i during the j^{th} phase of its computation. We will assume that each phase of each task will require access to at most one resource. If $r_{ij} = 0$, then the j^{th} phase of task T_i's computation requires no shared resources. Conceptually, if $r_{ij} = 0$, then the j^{th} phase of task T_i requires the special resource R_0. Resource R_0 is the only resource that can be allocated to multiple tasks simultaneously.

If a task never requires a resource (all phases use only resource R_0) then that task is called a *non-resource consuming task.* If at least one phase of a task ever requires a resource, then the task is called a *resource consuming task.* Without loss of generality, we will assume that for each resource R_j, $1 \le j \le m$, there are at least two distinct tasks T_a and T_b such that $r_{ax} = r_{by} = j$. This is simply a requirement that each resource is in fact shared. If a resource is only used by a single task then we can treat that resource as an R_0 resource.

The computational cost, c_i, of task T_i, is a given by:

$$c_i = \sum_{j=1}^{n_i} c_{ij} ,$$

[1] The results reported in this chapter first appeared in [Jeffay 89].

where c_{ij} represents the computational cost of phase j. That is, c_{ij} is the time to execute phase j of task i to completion on a dedicated processor. If phase j requires a shared resource ($r_{ij} \neq 0$) then c_{ij} represents only the cost of using the resource and not the cost of obtaining and releasing the resource. In later sections we will often wish to refer to the period of the "smallest" task that uses resource R_i. For resource R_i, let P_i represent this period. That is,

$$P_i = \underset{1 \leq j \leq n}{\mathrm{MIN}} \left(p_j \mid r_{jk} = i \text{ for some } k,\ 1 \leq k \leq n_j\right).$$

(Task T_k is the task with the smallest period that uses resource r_i.) As in the previous chapter, the analysis of the scheduling disciplines will assume that tasks can be dispatched or preempted in zero time.

4.2.2 Absolute and Relative Feasibility

A set of sporadic tasks with resource requirements is said to be *feasible* on a uniprocessor if it is possible to execute the tasks, on a uniprocessor, such that every execution request of every task is guaranteed to have completed execution at or before its deadline, and such that exclusive access to shared resources is maintained. For this task model, it will be useful to broaden the scope of temporal correctness.

The concept of feasibility we have used so far is an *absolute* boolean measure of temporal correctness. It either is possible for a set of tasks to meet their deadlines or it is not. Feasibility is a property of a set of tasks that is independent of the method used to sequence the tasks on a processor. For a given implementation strategy, one can often derive conditions under which a set of sporadic tasks, implemented using this strategy, can be guaranteed to be *viable*. An implementation of a task system is viable if every execution request of every task can be guaranteed to complete execution at or before its deadline. Viability is a *relative* measure of temporal correctness. Viability is a property of tasks that is relative to a particular implementation strategy. Viability is a stronger measure of temporal correctness than feasibility. For a given implementation strategy, a *viable* set of tasks will always be *feasible*. This converse is not true. Viability is a more restrictive concept than feasibility.

To compare implementation strategies we need a notion of optimality. An implementation strategy is said to be *optimal* for a uniprocessor if every *feasible* sporadic task system implemented using this strategy can be guaranteed to be *viable*.

4.3 Implementation Strategy Definition

4.3.1 Programming Model and Synchronization Discipline

Task's synchronize access to shared resources, according to the schema shown in Figure 4.3.1 below. We will assume that the body of each task is implemented following this template. At the start of each phase, a task requests a lock for the resource it requires for that phase. At the end of each phase the task releases the lock it held during that phase.

```
BEGIN
   ResourceRi.Request();   -- Phase 1
   < use resource $R_i$, $i = r_{x1}$ >
   ResourceRi.Release();

   ResourceRj.Request();   -- Phase 2
   < use resource $R_j$, $j = r_{x2}$ >
   ResourceRj.Release();
           :
           :
           :
   ResourceRk.Request();   -- Phase $n_x$
   < use resource $R_k$, $k = r_{xn}$ >
   ResourceRk.Release();
END
```

Figure 4.3.1: Task schema for task T_x.

During the lifetime of a system of sporadic tasks, each task is always in one of four states. Before a task is released it is *idle*. Each execution request of a task begins with an (implicit) request for the processor. Conceptually, this is the execution of the `BEGIN` statement above. While a task is waiting for the processor it is in the *ready* state. Once the processor is allocated to the task, then the task is in the *executing* state. If a task makes a lock request for a resource that is not available (the resource is already locked) then the task will be *blocked*. At the end of the last phase the task conceptually executes the `END` statement. This returns the task to the idle state. The task will remain idle until the start of its next request interval. During the course of execution the task may be preempted. When a task is preempted it maintains possession of any resources it may have held. When preempted, a task returns to the *ready* state. A state transition diagram for the states is shown in Figure 4.3.2 below. We will assume that the `BEGIN` and `END` statements take no time to execute and hence are not part of the task's computational cost.[2]

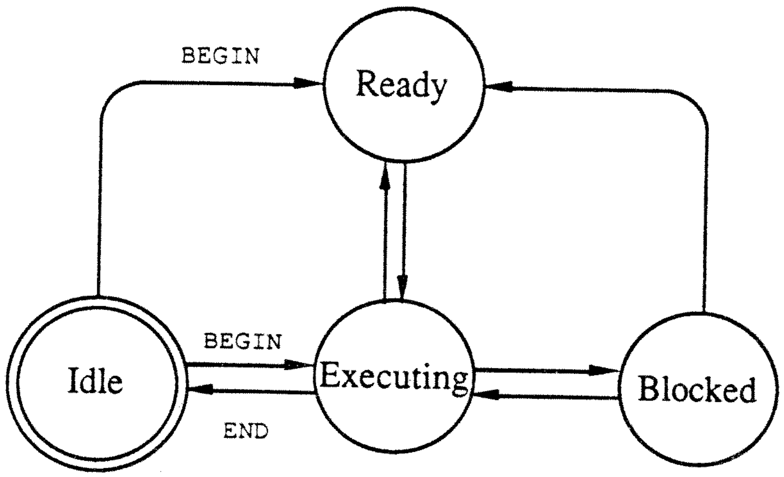


Figure 4.3.2: Partial state transition diagram for sporadic tasks.

The locks controlling access to each resource R_i, $1 \leq i \leq m$, are implemented by a monitor that exports the `request` and `release` operations. A pseudo-code schema for such a monitor is shown in Figure 4.3.3.

The semantics of the monitor ensures that only one task is ever executing code inside the monitor at any time. When a task wishes to gain access to resource R_i, it calls the `Request` entry of the monitor corresponding to R_i's lock. Whenever a task returns from this call it is guaranteed to have exclusive access to the resource. If a task attempts to request a resource lock that is not available, then the task will execute the `WAIT` statement and be blocked on the monitor's condition variable. When a task releases a resource lock, the task executes the `BROADCAST` statement that will wake-up all tasks that were blocked on that monitor's condition variable. When a task is woken-up, it is placed in either the *ready* or *executing* state. For resource R_0, assume that the `request` and `release`

[2] The cost of the END statement could be included in the task's cost with little effort.

operations are null statements. We will assume that for all resources, the `request` and `release` operations take no time to execute.[3]

```
MONITOR ResourceRi =
   BEGIN
      var available : BOOLEAN := TRUE; -- Initialization
      var resource  : CONDITION;

   ENTRY PROCEDURE Request() =
      BEGIN
         WHILE( NOT available) DO
            WAIT( resource);
         END
                                    -- Caller has acquired exclusive
         available := FALSE;        -- access to Resource Ri.
      END

   ENTRY PROCEDURE Release() =
      BEGIN
         available := TRUE;

         BROADCAST( resource); -- Wake-up all waiting tasks.
      END

   END -- Of Monitor
```

Figure 4.3.3: Monitor for resource lock acquisition and release.

An important feature of this synchronization scheme is that the broadcast operation does not explicitly allocate the newly freed resource lock to any task. Each task that is woken-up by a broadcast must re-attempt to acquire the resource lock on its own. (This style of synchronization is borrowed from the Mesa language [Lampson & Redell 80].) The decision as to which task will actually acquire the resource lock will unwittingly be made by the scheduler since it is the scheduler who decides which released task will execute first.

For this synchronization discipline, we can complete the state transition diagram in Figure 4.3.2. For a given task, the following state transition diagram describes the task's behavior. The underlined operations are operations that are performed by other tasks. For

[3] In principle, the cost of these operations could be included in the cost of a task's phase. However, if these operations took time to execute then we would have to broaden the discussion to include details concerning the implementation of the `WAIT` and `BROADCAST` primitives. As such details are not central to the presentation of our results, we postulate a zero execution time cost.

example, the only way a task can leave the blocked state is for some other task to perform a broadcast. A transition from the executing state to the ready state corresponds to a preemption of the executing task. Preemptions in the system occur only as the result of one of two actions. A task may be preempted by another task executing its BEGIN statement, or a task may be preempted when it performs a BROADCAST. The complete state transition diagram is shown in Figure 4.3.4.

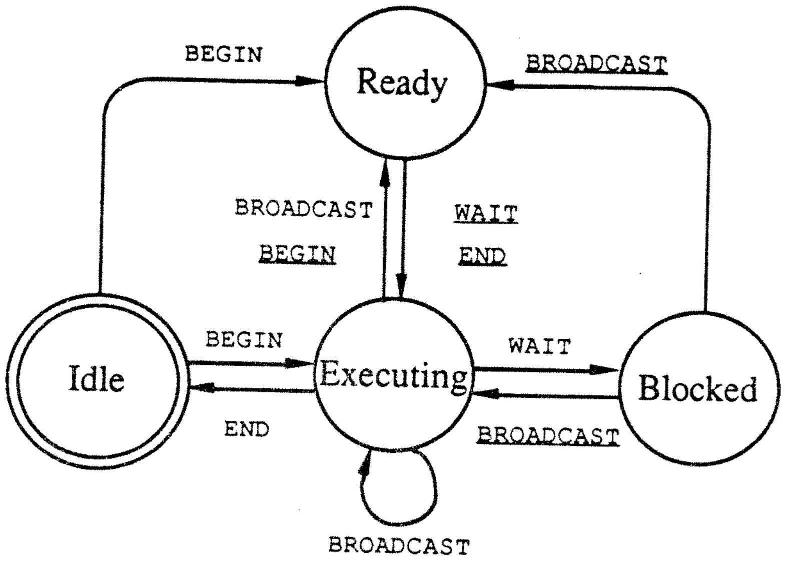


Figure 4.3.4: Complete transition diagram for task states.

4.3.2 Processor Scheduling Discipline

For processor scheduling we propose a policy based on the EDF selection rule (see Chapter 3, Section 3.3). When making scheduling decisions, an EDF scheduler will dispatch the ready task whose deadline is nearest to the current value of real-time. Recall that ties among tasks with the same deadline may be broken arbitrarily. In our implementation strategy, the scheduler is preemptive. It makes scheduling decisions each time a task makes a request for execution (executes the BEGIN statement), requests an unavailable resource lock (performs a WAIT), completes a phase of its execution (performs a BROADCAST), or completes an execution request (executes the END statement).

The overall goal of our implementation strategy is to ensure that the task with the nearest deadline gets whatever resources it requires as soon as possible. To achieve this goal we will need a slightly more sophisticated synchronization mechanism than the simple WAIT and BROADCAST operations described above. Since our implementation strategy allows tasks to preempt one another, when a task T_i performs a request operation for a lock on resource R_k, it may be the case that R_k's lock is allocated to a (preempted) task T_j whose current deadline is greater than that of task T_i. Since resource R_k is not available, T_i will become blocked on the condition variable associated with R_k's lock. To ensure that T_i's deadline can be respected, task T_j will have its deadline shortened to that of task T_i for the duration of T_j's current phase. This advancing of T_j's deadline occurs as a side effect of T_i's execution of the WAIT statement. In this manner, when T_i becomes blocked, T_j will have a deadline that is less than or equal to the deadlines of all the ready tasks. Therefore, under an EDF scheduler, T_j is quite likely to resume execution after T_i becomes blocked. When T_j performs the release operation at the end of its current phase, T_j will have its deadline restored (increased) to its original value as a side effect of the execution of the BROADCAST statement. At this point, all the tasks blocked on R_k's condition variable will become ready. The scheduler will preempt T_j and dispatch T_i (or some other task with a deadline less than or equal to T_i's deadline). This deadline advancement technique is a limited version of the scheme proposed in [Lampson & Redell 80] and [Sha et al. 87]. The analysis in the following sections will validate this implementation strategy.

4.4 Single Resource, Single Phase Systems

We begin with an analysis of the simplest form of resource requirements. This is a task system with a single shared resource R_1, and one where every task consists of only a single phase. This latter simplification means that if a task requires access to resource R_1, then each execution request of the task will require exclusive access to R_1 for the entire duration of its computation. For single phase systems, we will let $r_i = r_{i1}$ (= 0 or 1) denote the resource requirement of the (single) phase of task T_i.

For single resource, single phase tasks, we can derive necessary and sufficient conditions for a task system to be viable on a single processor under our implementation strategy. We start by establishing necessary conditions for ensuring the viability of any implementation strategy whose scheduler does not use inserted idle time.

Theorem 4.4.1: Let $\tau = \{T_1, T_2, \ldots, T_n\}$ be a set of single phase sporadic tasks that share a single resource R_1. Assume that the tasks in τ are sorted in non-decreasing order by period. In the absence of inserted idle time, τ will be feasible for arbitrary release times only if:

1) $\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1,$

2) $\forall k,\ 1 \leq k < n,\ r_k \neq 0:\ p_k \geq \underset{i:\, I_1}{\text{MAX}}\left(c_i + \underset{l:\, I_2}{\text{MAX}}\left(-l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \right)\right),$

where $I_1 = (k < i \leq n) \wedge (r_i = r_k) \wedge (r_i \neq 0)$,[4]
$I_2 = 0 < l < p_i - p_k$.

3) $\forall k,\ 1 \leq k < n,\ r_k = 0:\ p_k \geq \underset{i:\, II_1}{\text{MAX}}\left(c_i + \underset{l:\, II_2}{\text{MAX}}\left(-l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \right)\right),$

where $II_1 = (k < i \leq n) \wedge (r_i \neq 0)$,
$II_2 = \text{MAX}(0, P_1 - p_k) < l < p_i - p_k$.

As in Theorems 3.5.1 and 3.6.1, condition (1) is the requirement that the system not be overloaded (cumulative processor utilization less than 1). Conditions (2) and (3) are a generalization of condition (2) from Theorem 3.6.1. The current condition (2) applies to resource consuming tasks and condition (3) applies to non-resource consuming tasks. They both require that a task's period (its maximum tolerable latency) be greater than the worst delay that can be encountered while the task is waiting to be scheduled and while waiting to access the resource it needs.

Note that a set of single phase sporadic tasks τ, where $r_i = 0$, for all i, $1 \leq i \leq n$, corresponds to a set of tasks with no preemption constraints. In this case the above conditions reduce to condition (1) alone (I_1 and II_1 are both empty). This agrees with Theorem 3.5.1. A set of single phase sporadic tasks where $r_i = 1$, for all i, $1 \leq i \leq n$, corresponds to a set of tasks that must be scheduled non-preemptively. In this case, the above conditions reduce to those in Theorem 3.6.1.

[4] Since r_j can only be either 0 or 1, the notation used for predicates I_1 and II_1, may appear overly general. We choose this notation because it will allow for easier comparison with the feasibility conditions for the more general problems that follow.

Proof: To show that conditions (1), (2), and (3) are necessary for arbitrary release times, by Lemma 3.6.5, we need only show that there exist one set of release times for which these conditions are necessary. The construction used in the proof of Theorem 3.5.1 is sufficient for showing the necessity of condition (1) and hence we will not repeat it here.

To see that condition (2) is necessary, choose tasks T_k and $T_{i'}$, such that $k < i' \le n$, $r_k \ne 0$, $r_{i'} = r_k$, and a value l' such that $0 < l' < p_{i'} - p_k$. Let $s_{i'} = 0$, and $s_j = 1$, for $1 \le j \le n, j \ne i'$. This gives rise the pattern of task execution requests shown in Figure 4.4.1 below.

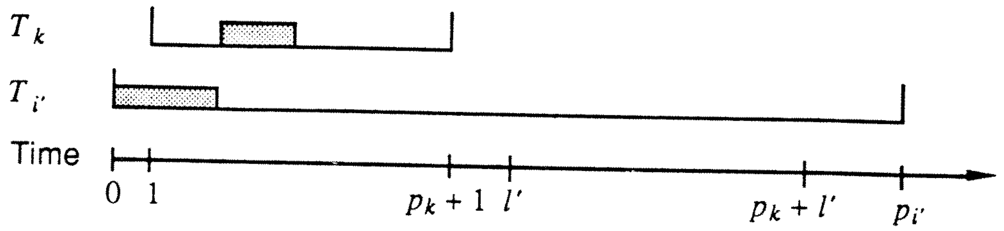


Figure 4.4.1: Construction for the necessity of condition (2).

Because inserted idle time is not allowed, $T_{i'}$ will be dispatched at time zero. Since T_k and $T_{i'}$ both require the same resource (R_l), the execution request of $T_{i'}$ begun at time 0 must be completed by time $t = (p_k - c_k) + 1$ if τ is to be feasible. Therefore, in the interval $[0,l'+p_k]$, the total processor demand $d_{0,l'+p_k}$, is bounded by

$$l'+p_k \ \ge\ d_{0,l'+p_k} \ \ge\ c_{i'} + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j ,$$

hence

$$p_k \ \ge\ c_{i'} - l' + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j .$$

To see that condition (3) is necessary, we use a construction similar to the one above. Choose tasks T_k and $T_{i'}$ such that $k < i' \le n$, $r_k = 0$, $r_{i'} \ne 0$, and a value l' such that $\mathrm{MAX}(0, P_l - p_k) < l' < p_{i'} - p_k$. Let $s_{i'} = 0$, and $s_j = 1$, for $1 \le j \le n, j \ne i'$. This gives rise the pattern of task execution requests shown in Figure 4.4.2 below. Let T_m be the task with $r_m = 1$, whose period is P_l. (T_m is the task with the smallest period that accesses R_l.)

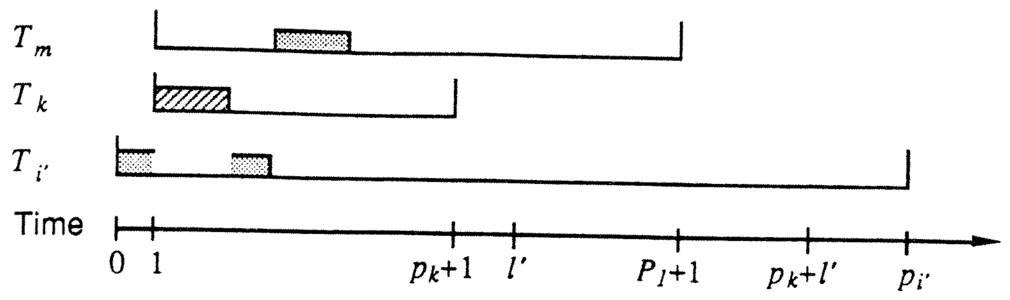


Figure 4.4.2: Construction for the necessity of condition (3).

At time $t = 0$, task $T_{i'}$ may be preempted by task T_k (or by some other non-resource consuming task in with a nearer deadline). However, as in the previous case, $T_{i'}$ must still complete execution before time $t = (p_m - c_m) + 1$, if task T_m is to meet its deadline. Therefore, in the interval $[0, l'+p_k]$, the total processor demand is again bounded by

$$l'+p_k \ \geq \ d_{0,l'+p_k} \ \geq \ c_{i'} + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j \,,$$

and hence

$$p_k \ \geq \ c_{i'} - l' + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j \,. \qquad \Delta$$

The next theorem [illegible]ws that the conditions of Theorem 4.4.1, are sufficient for ensuring the viability of a s[illegible] of single phase sporadic tasks that share a single resource under our implementation strategy. Since these conditions are necessary conditions for the feasibility of such a set of sporadic tasks, Theorem 4.4.2 also shows that the implementation strategy of Section 4.3 is an optimal strategy for resource consuming, single phase tasks that share a single resource. Again, the optimality is with respect to the class of implementation strategies whose schedulers do not use inserted idle time. This optimality result means that if a set of single phase sporadic tasks that share a single resource can be viable under any implementation strategy that does not use inserted idle time in its scheduler, then the tasks must be viable under our strategy.

Theorem 4.4.2: Let τ be a set of single phase sporadic tasks as in Theorem 4.4.1. Under the implementation strategy of Section 4.3, τ will be viable for arbitrary release times, if conditions (1), (2), and (3) from Theorem 4.4.1 hold.

Proof: (By contradiction.) The proof proceeds by enumerating all possible types of blockage that a task can encounter. For each case we show that if a task misses a deadline then one the conditions from Theorem 4.4.1 must have been false. The proof is similar to the proof of Theorem 3.6.1. The full text of the proof appears in Appendix A. Δ

4.5 Multiple Resource, Single Phase Systems

We next examine the problem of executing single phase sporadic tasks that share a set of resources. This problem differs from the previous problem in that it is now possible for resource consuming tasks to preempt one another provided they do not use the same resource. The main result of this section is to demonstrate that often it is inappropriate for an implementation strategy to allow these preemptions. Unfortunately, determining when it is appropriate to allow this additional preemption remains an open problem.

The implementation strategy in Section 4.3 uses a separate lock to synchronize access to each resource. For a system with multiple resources, this does not lead to an optimal implementation strategy. Often a better strategy is to use a single lock to synchronize access to a group of resources. (Assume this shared lock is implemented with a dedicated monitor.) That is, in terms of optimality, there will sometimes be an advantage to ignoring the distinction between certain resources and to treat them as a single logical resource. Note that treating a group of resources as a single logical resource and using a single lock to control access to these resources does not effect the logical correctness of the system. The following example illustrates the effect of controlling access to a group of resources with a single lock.

Example 4.5.1: Consider the following set of single phase tasks that share two resources R_1 and R_2.

$T_1 = (4,1,4),\ \ r_1 = R_1 \qquad T_3 = (2,1,6),\ \ r_3 = R_2 \qquad T_2 = (3,1,5),\ \ r_2 = R_0$
$T_5 = (0,3,17),\ \ r_5 = R_1 \qquad T_4 = (1,3,15),\ \ r_4 = R_2$

Our implementation strategy will use a separate lock for controlling access to each resource. This means that our strategy will allow preemptions between tasks that use different resources. Under our implementation strategy, the above tasks will be executed as shown in Figure 4.5.1 below. Note that task T_3 misses a deadline at time $t = 8$.

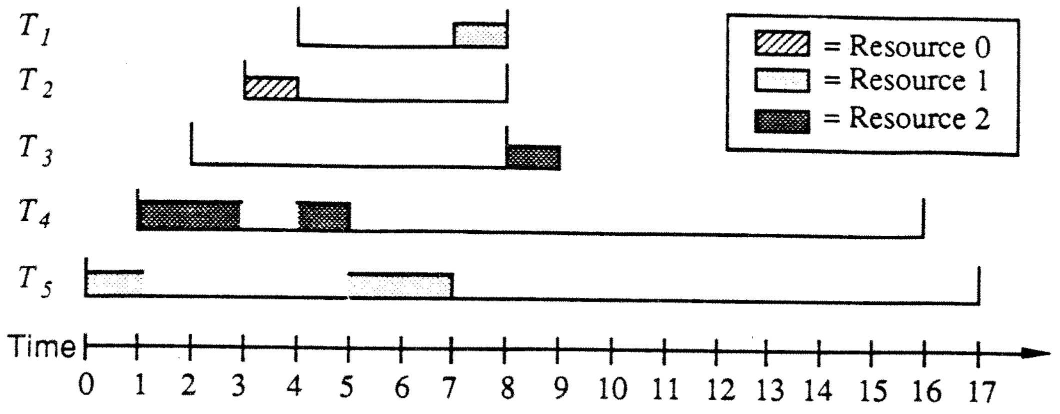


Figure 4.5.1: Execution of tasks with one lock per resource.

Had we modified our characterization of these tasks so that R_1 and R_2 were treated as a single logical resource, then these same tasks could have been executed correctly by our implementation strategy. If we treat this multiple resource system as a single resource system by requiring that all tasks requesting R_1 or R_2 actually request the meta-resource $R_{1\&2}$, then conditions (1), (2), and (3) from Theorem 4.4.1 hold for the resulting single resource system. By using a single lock for controlling access to R_1 and R_2, we are disallowing preemption between the resource consuming tasks in this example. By disallowing this preemption, task T_4 would not have been scheduled at time 1 and T_3 will complete execution before its deadline as shown in Figure 4.5.2 below.

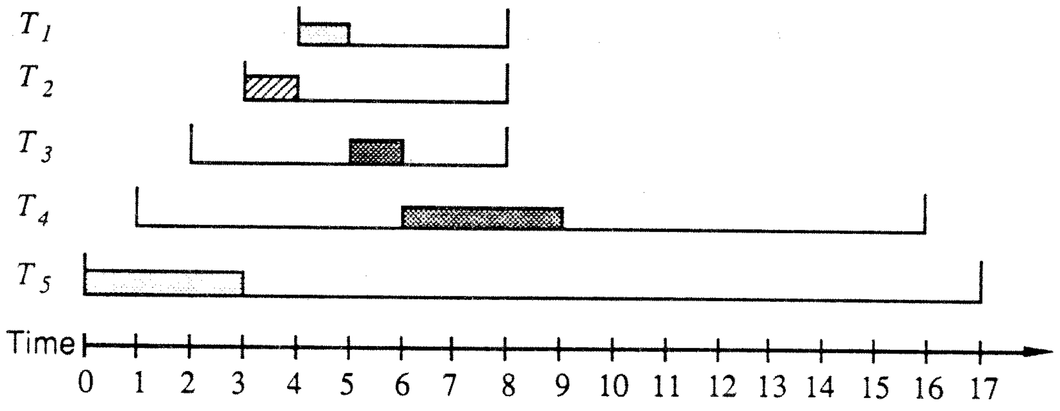


Figure 4.5.2: Execution of tasks with one lock for all resources.

This seemingly anomalous be[illegible]vior can be explained by closely examining the effect of preemption among resource c[illegible]uming tasks. In the previous section, the worst case delay that a task T_k could experience while waiting to be scheduled, occurred when a *single* resource consuming task T_i with a larger period made an execution request l time units before T_k's request. Under the right circumstances, T_k had to wait for the execution request of this larger task to complete before task T_k was allowed to complete. By allowing resource consuming tasks to preempt one another, it is now possible for task T_k to have to wait for *multiple* resource consuming tasks with further deadlines to complete before being scheduled. For the specific tasks above, we do better by prohibiting this behavior from occurring.

However, always disallowing preemption among resource consuming tasks does not lead to an optimal implementation strategy. Under certain circumstances, allowing resource consuming tasks to preempt one another makes previously unviable tasks sets viable.

Example 4.5.2: Consider the following set of single phase tasks that share two resources R_1 and R_2. For the implementation strategy of Section 4.3 with separate locks used for accessing resources R_1 and R_2, the following tasks can be shown to be viable.

$T_1 = (1,1,4),\ r_1 = R_0$ $\quad T_2 = (1,1,5),\ r_2 = R_1$ $\quad T_4 = (0,5,25),\ r_5 = R_2$

$\quad T_3 = (1,1,6),\ r_3 = R_1$ $\quad T_5 = (0,5,28),\ r_4 = R_2$

Under our implementation strategy, these tasks will be executed as shown in Figure 4.5.3 below.

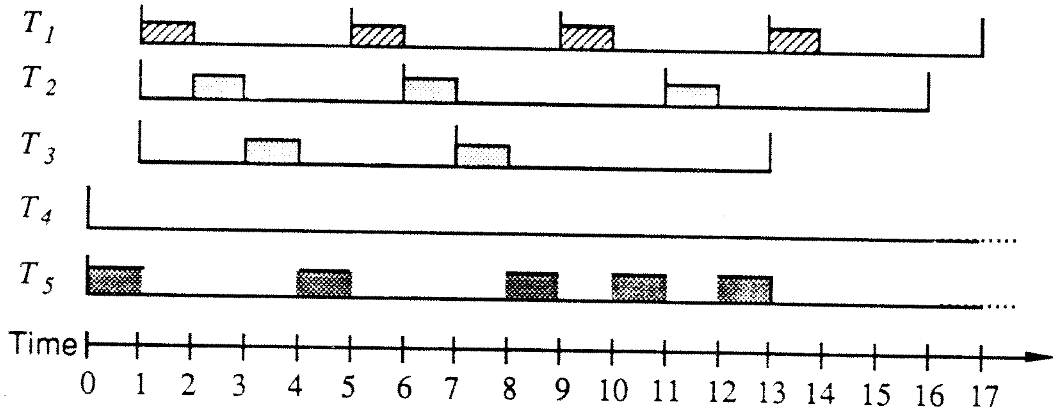


Figure 4.5.3: Execution of tasks with one lock per resource.

However, had we ignored the distinction between resources R_1 and R_2 and used a single lock for accessing these resources, then these tasks would have been quite unviable under our implementation strategy as shown in Figure 4.5.4.

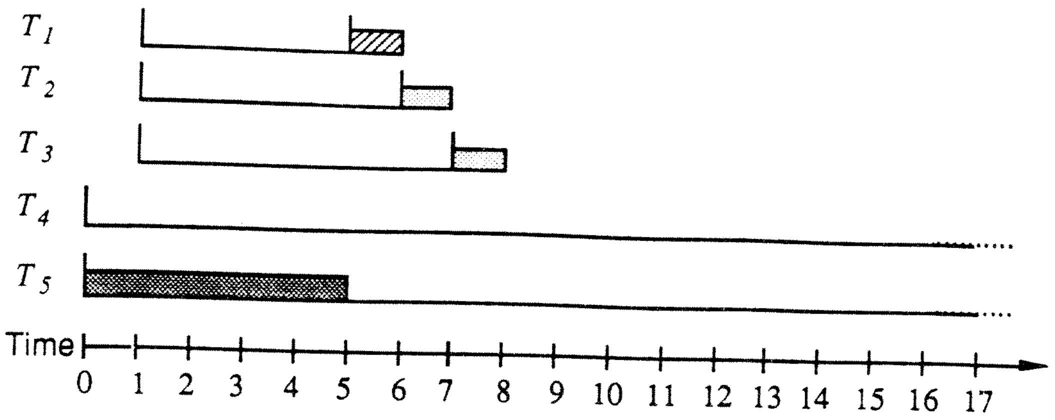


Figure 4.5.4: Execution of tasks with one lock for all resources.

An important difference between the tasks in these two examples is that in the second example, the periods of the tasks that used R_1 were all less than the periods of the tasks that used resource R_2. Because of this, for any request of task T_k, with $r_k \neq 0$, it is not possible for more than one resource consuming task with a further deadline to execute while T_k has an outstanding request for execution. In order for multiple resource consuming tasks with further deadlines to execute while T_k has an outstanding request for execution, there must exist tasks T_i and T_j with $r_i = r_k$, $r_j \neq r_k$, $r_j \neq 0$, and $p_k < p_j < p_i$. When this occurs we say that there exists an *overlap* in the periods of the tasks that consume resources. For example, consider the three tasks shown in Figure 4.5.5 below. tasks T_i and T_k share resource R_1. Since there exists a resource R_2 consuming task T_j such that $k < j < i$ (i.e., $p_k < p_j < p_i$), there exists an overlap in the periods of the resource consuming tasks.

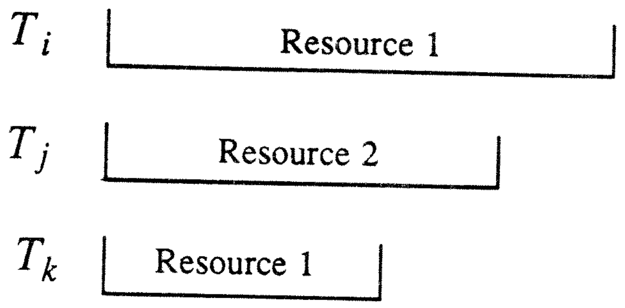


Figure 4.5.5: An overlap in the periods of resource consuming tasks.

A precise characterization of feasibility and viability conditions for single phase tasks that share a set of resources, is the object of an on-going study. One special case that has been solved is the case where there is no overlap in the periods of the tasks that consume resources.

Theorem 4.5.1: Let $\tau = \{T_1, T_2, \ldots, T_n\}$ be a set of single phase sporadic tasks that share m resources R_1 - R_m. Assume that the tasks in τ can be labeled such that if $k < i$, then $p_k \le p_i$, and if $k < j < i$, and $r_k = r_i$ ($\neq 0$), then either $r_j = 0$, or $r_k = r_j = r_i$. In the absence of inserted idle time, τ will be feasible for arbitrary release times only if:

1) $\displaystyle\sum_{i=1}^{n} \frac{c_i}{p_i} \le 1$,

2) $\forall k,\ 1 \le k < n,\ r_k \neq 0$: $\displaystyle p_k \ge \underset{i:\,I_1}{\mathrm{MAX}}\left(c_i + \underset{l:\,I_2}{\mathrm{MAX}}\left(-l + \sum_{j=1}^{i-1}\left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j\right)\right)$,

where $I_1 = (k < i \le n) \wedge (r_i = r_k) \wedge (r_i \neq 0)$,
$I_2 = 0 < l < p_i - p_k$.

3) $\forall k,\ 1 \le k < n$: $\displaystyle p_k \ge \underset{i:\,II_1}{\mathrm{MAX}}\left(c_i + \underset{l:\,II_2}{\mathrm{MAX}}\left(-l + \sum_{j=1}^{i-1}\left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j\right)\right)$,

where $II_1 = (k < i \le n) \wedge (r_i \neq r_k) \wedge (r_i \neq 0)$,
$II_2 = \mathrm{MAX}(0, P_{r_i} - p_k) < l < p_i - p_k$.

Conditions (1) and (2) are identical to those in Theorem 4.4.1. Condition (3) is identical in spirit to the corresponding condition in that theorem but now applies to both resource consuming and non-resource consuming tasks.

Proof: The necessity of these conditions follow immediately from the related constructions in the proof of Theorem 4.4.1. Δ

For multiple resource systems, it can be shown that the implementation strategy of Section 4.3 is an optimal strategy for single phase sporadic tasks with no overlap in the periods of the tasks that consume resources. As in the previous section, the optimality is shown by demonstrating that the conditions necessary for feasibility are sufficient conditions for ensuring the viability of our implementation strategy.

Theorem 4.5.2: Let τ be a set of single phase sporadic tasks as in Theorem 4.5.1. Under the implementation strategy of Section 4.3, τ will be viable for arbitrary release times, if conditions (1), (2), and (3) from Theorem 4.5.1 hold.

Proof: The proof is largely identical to the proof of Theorem 4.4.2 and will not be repeated here. Δ

For sets of single phase tasks with arbitrary overlap in the periods of resource consuming tasks, the above results suggest a heuristic for ensuring viability. A set of meta-resources is created by treating groups of resources as a single resource in such a manner that there no longer is any overlap in the periods of meta-resource consuming tasks. The conditions from Theorem 4.5.1 can then be used to determine viability of the transformed system.

4.6 Multiple Phase Systems

We finally consider the problem of determining feasibility conditions for a set of sporadic tasks when each task consists of a set of phases. This characterization addresses tasks that require multiple resources to execute. A task can request to use a resource multiple times; however, we are still assuming that each phase of a task requires at most one resource. The main result of this section is to show that the analysis of the previous sections is sufficient to analyze these systems. We will show how a multiple phase task can be thought of as a set of single phase tasks and how our implementation strategy for single phase tasks can be modified to execute multiple phase tasks.

4.6.1 From Multiple Phase Tasks to Single Phase Tasks

Let τ be a set of multiple phase sporadic tasks. We can create an equivalent set τ', of single phase sporadic tasks such that τ' will be feasible if and only if τ is feasible. The set τ' is constructed as follows. For a task T_k in τ with n_k phases, we will create n_k single phase tasks $T_{ki} = (s_k, c_{ki}, p_k)$, $1 \le i \le n_k$. All tasks derived from T_k will make requests for execution at the same points in time at which T_k would have made execution requests. These tasks can be scheduled as ordinary single phase tasks except that for each T_k in τ, a precedence order must be maintained between tasks T_{ki}, $1 \le i \le n_k$, in τ'. Each execution request of task T_{ki}, $i > 1$, made at time t, cannot be scheduled, or allocated resources, until after the execution request of task T_{ki-1} made at time t has completed execution.

To execute the set of single phase sporadic tasks τ', we supplant the scheduler in our implementation strategy with similar scheduler based on a refined EDF scheduling policy that we will call the EDF* policy. An EDF* scheduler behaves exactly as an EDF scheduler except when choosing among tasks with the same deadline. Recall that the EDF rule allows for an arbitrary choice among tasks with the same deadline. We will exploit this feature to enforce the precedence constraints on the tasks in τ'. When there are multiple ready tasks with the nearest deadline, the EDF* policy will choose a ready task T_{ki} only if there does not exist another task $T_{kj}, j < i$, in the ready or blocked state. For our modified implementation strategy, the transformation from a set of multiple phase sporadic tasks, to a larger set of single phase sporadic tasks, outlined above is correct in the sense that for each k, the aggregate behavior of the tasks T_{ki}, $1 \leq i \leq n_k$, will be indistinguishable from the behavior of T_k. Such an implementation strategy can correctly execute τ' if and only if it can correctly execute τ.

Unfortunately, the problem of deciding feasibility for τ' is more complicated than the problem of deciding feasibility for single phase tasks. Recall that we have assumed that the times at which tasks made execution requests were independent. The times at which the single phase tasks derived from a multiple phase task, make execution requests are not independent. For all k, tasks T_{k1} - T_{kn_k} will always make execution requests at the same time. This fact must be reflected in the feasibility analysis since these tasks can *never* interfere with each other. They will never compete for resources or block one another. Therefore, the feasibility conditions from the previous sections are only sufficient conditions for the feasibility of a single phase system derived from a multiple phase system. They must be generalized slightly in order to become necessary conditions.

4.6.2 Single Resource, Multiple Phase Tasks

The following theorem extends Theorem 4.4.1 for multiple phase tasks. It establishes necessary conditions for the feasibility of multiple phase tasks that share a single resource. This corresponds to a set of tasks where in each request for execution, a task alternates between using the resource and not using the resource.

Theorem 4.6.1: Let $\tau = \{T_{11}, T_{12}, ..., T_{1n_1}, T_{21}, T_{22}, ..., T_{2n_2}, ..., T_{n1}, T_{n2}, ..., T_{nn_n}\}$ be a set of single phase sporadic tasks derived from a set of multiple phase sporadic tasks $\{T_1, T_2, ..., T_n\}$ that share a single resource R_1. Assume τ is sorted such that if

$i < j$, then $p_{ix} \le p_{jy}$ for all x and y, $1 \le x \le n_i$ and $1 \le y \le n_j$ (recall that for all k, $1 \le k < n_i$, $p_{ik} = p_i$). For tasks T_{i1} - T_{in_i}, define the cost function $C_i(h)$ to be

$$C_i(h) = \begin{cases} 0 & \text{if } h = 1, \\ \sum_{j=1}^{h-1} c_{ij} & \text{if } 1 < h \le n_i \;. \end{cases}$$

In the absence of inserted idle time, τ will be feasible for arbitrary release times only if:

1) $\displaystyle\sum_{i=1}^{n} \frac{c_i}{p_i} \le 1,$

2) $\forall k,\ 1 \le k < n,\ \forall g,\ 1 \le g < n_k$: $r_{kg} \ne 0$:

$$p_k \ \ge \ \underset{i,h:\, I_1}{\text{MAX}} \left(c_{ih} + \underset{l:\, I_2}{\text{MAX}} \left(-l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \right) \right),$$

where $I_1 = (k < i \le n) \wedge (1 \le h \le n_i) \wedge (r_{ih} \ne 0) \wedge (r_{ih} = r_{kg})$,
$I_2 = 0 < l < (p_i - p_k) - C_i(h)$,

3) $\forall k,\ 1 \le k < n,\ \forall g,\ 1 \le g < n_k$: $r_{kg} = 0$:

$$p_k \ \ge \ \underset{i,h:\, II_1}{\text{MAX}} \left(c_{ih} + \underset{l:\, II_2}{\text{MAX}} \left(-l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \right) \right),$$

where $II_1 = (k < i \le n) \wedge (1 \le h \le n_i) \wedge (r_{ih} \ne 0)$,
$II_2 = \text{MAX}(0, P_1 - p_k) < l < (p_i - p_k) - C_i(h)$.

Conditions (2) and (3) are semantically equivalent to the corresponding conditions in Theorem 4.4.1. These conditions differ from those of Theorem 4.4.1 in that for a task T_{kg}, conditions (2) and (3) do not include any delay due to *waiting* for other tasks T_{kj} to complete execution. As in Theorem 4.4.1, condition (3) only applies to those tasks that never use any resources.

Conditions (2) and (3) also differ in that the range of the lag time parameter l is more restricted than in Theorem 4.4.1. This is a reflection of the existence of a precedence relation among subsets of the tasks. For the previous problems, the worst case blockage of a task T_k occurred when a task T_i with a larger period commenced execution l time units before a request interval of T_k. In those cases, the maximum that the lag could be was

$p_i - p_k - 1$ time units. In the current problem, since there exists a precedence relation on tasks, task T_{ih} can never be scheduled until $C_i(h)$ time units after it makes an execution request. The cost function $C_i(h)$ represents the cost of the execution of the $h - 1$ tasks that must precede each execution request of task T_{ih}. Therefore, when assessing the blockage due to a task T_{ih} executing just prior to an execution request of a task T_{kg} with a smaller period, we need only consider a maximum lag between these two tasks of

$$l < (p_i - p_k) - C_i(h)$$

time units.

Proof: We will only demonstrate the necessity of conditions (2) and (3).

For condition (2), choose tasks T_{kg} and $T_{i'h}$, such that $k < i' \leq n$, and $1 \leq h \leq n_i$, $1 \leq g \leq n_k$, $r_{ih} \neq 0$, and $r_{ih} = r_{kg}$. $T_{i'h}$ is a task that requires the same resource that T_{kg} needs. Choose a value l' such that $0 < l' < (p_{i'} - p_k) - C_i(h)$. Let $s_{i'} = 0$, and $s_j = C_i(h) + 1$, for $1 \leq j \leq n, j \neq i'$. This gives rise the pattern of task execution requests shown in Figure 4.6.1 below. (Recall that shaded cost rectangles denote resource phases.)

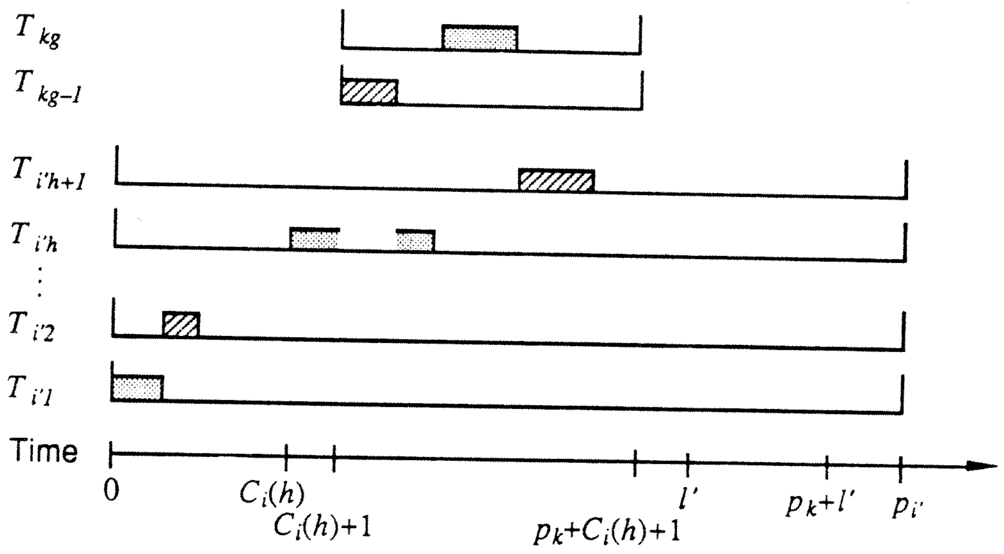


Figure 4.6.1: Necessity construction for condition (2).

Following the analysis of Theorem 4.4.2, we have in the interval $[0,l'+p_k]$, the total processor demand $d_{0,l'+p_k}$, bounded by

$$l'+p_k \ \geq \ d_{0,l'+p_k} \geq \ c_{i'h} + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j \, ,$$

hence

$$p_k \ \geq \ c_{i'h} - l' + \sum_{j=1}^{i'-1} \left\lfloor \frac{p_k+l'-1}{p_j} \right\rfloor c_j \, .$$

The construction showing the necessity of condition (3) is similar. Δ

The next theorem shows that the above conditions are sufficient for the ensuring the viability of a set of tasks under our implementation strategy. Therefore, the implementation strategy in Section 4.3, modified with an EDF* scheduler, is an optimal strategy for multiple phase tasks that share a single resource.

Theorem 4.6.2: Let τ be a set of multiple phase sporadic tasks as in Theorem 4.6.1. Under the implementation strategy of Section 4.3 with an EDF* scheduler, τ will be viable for arbitrary release times, if conditions (1), (2), and (3) from Theorem 4.6.1 hold.

Proof: Excluding some minor changes in notation, the proof is identical to the proof of Theorem 4.4.2 and will not be repeated here. Δ

Although we have only considered multiple phase tasks that share a single resource, we can draw several conclusions that will be true for all multiple phase systems. First, note that the feasibility of a set of multiple phase tasks is not a function of the number of phases of the tasks. What is important is the relative sizes of only the resource consuming phases. The relative sizes of the non-resource consuming phases is immaterial. For example, adjacent non-resource consuming phases may be coalesced without affecting the feasibility of the system. Of course resource consuming phases should be as short as possible. Therefore, there is a potential benefit, in terms of feasibility, to separating a resource consuming phase into multiple phases. This may be possible if, for example, a phase consists of a sequence of disjoint operations. This technique amounts to defining allowable preemption points within a resource consuming phase.

4.6.3 Multiple Resource, Multiple Phase Tasks

Since we can always map a set of multiple phase tasks into a set of single phase tasks with precedence constraints, our ability to determine viability conditions for multiple resource, multiple phase tasks, will depend on our ability to determine viability conditions for multiple resource, single phase tasks. As we remarked in Section 4.5, this remains an open problem. However, it is apparent that the complexity of this problem lies not in the existence of multiple phases but in the use of multiple resources.

4.7 Discussion

4.7.1 Desirable Interactions Between Scheduling and Synchronization Policies

There are two important features of our implementation of locks that were critical to the optimality of our implementation strategy. The first feature is that resource locks are never assigned to a task by another task. When a task that is waiting on a lock's condition variable is awakened, the task must reattempt to acquire the lock. Since each task must explicitly acquire a resource's lock on its own, a task must be executing when it acquires a lock. Therefore, in our idealized environment, a task is guaranteed to execute for at least one time unit immediately after acquiring the resource. Put in an other way, resources are allocated to tasks as late as possible. They are allocated to tasks only when the acquiring task is guaranteed to commence execution.

The second notable feature of our implementation is that available resources are always acquired by the ready task with the nearest deadline (of all the ready tasks in the system). The feature is possible because of our use of the `BROADCAST` primitive. Note that it would be quite awkward to guarantee that an implementation based on the more common `WAIT` and `SIGNAL` synchronization primitives would have the features above. This is because the signal operation only releases a single waiting task that then typically has priority over the signaling task [Hoare 74]. This is not acceptable if , for example, the signaling task has a nearer deadline than the signaled task. In addition, if the resource is allocated to the signaled task, then problems arise if a task with a nearer deadline, that desires the same resource, becomes ready (makes a request for execution) before the signaled task resumes execution.

Our implementation strategy has the additional interesting property that a task will execute the wait statement in the request operation at most once for any resource request. This is due to the fact that we use deadline scheduling. Once a task T_k has been blocked and is made ready by a `BROADCAST`, T_k will execute when it has the nearest deadline. In the meantime the resource it requires will be used only by tasks with a nearer deadline. Since we use an EDF scheduler, these tasks will be dispatched and complete execution before task T_k is considered. When T_k is dispatched the resource it requested must be available. Therefore, the loop in the `request` monitor entry can be replaced by an *if* statement. This observation is important for assessing the cost (overhead) of our synchronization scheme. Although we are currently ignoring the cost of the `request` and `release` operations, this observation simplifies the determination of their cost since the loop is executed at most once.

4.7.2 Bounding the Preemption Required in an Implementation Strategy

Another issue related to the overhead of our implementation strategy concerns the cost of preemption. In our strategy, tasks that require different resources may preempt one another. While we have been ignoring the cost of this preemption, in practice there is a high cost associated with a context switch. Therefore, for a given task system, it would be useful to determine if the preemption in our implementation strategy is necessary for viability. The analysis of the previous sections can be used to determine the minimum amount of preemption necessary for ensuring the feasibility of a set of sporadic tasks.

A set of sporadic tasks with m shared resources will require a maximum of m locks to ensure preemption constraints are respected. As the number of locks used in a task system decreases, the amount of preemption allowed in the system also decreases. Therefore, in theory, an implementation strategy could be more efficient, in terms of the fraction of processor utilization consumed by overhead, if it used only $k < m$ locks. Let $m' \leq m$ be the maximum number of logical resources into which the m resources can be grouped such that there is no overlap in the periods of the tasks that consume the logical resources. We can determine a lower bound on the number of locks required to ensure feasibility by computing the viability conditions for each possible implementation strategy with $1 \leq k \leq m' + 1$ locks. While the number of possible implementation strategies is proportional to $m!$, this procedure may well be practical for task sets with a small number of resources.

4.7.3 Feasibility Versus Processor Utilization: Comparison with other approaches

Previous work in the area of cyclic tasks with resource requirements has focused on heuristic solutions for periodic tasks. One approach has been to reduce the analysis of a set of periodic tasks with preemption or mutual exclusion constraints to the analysis of a set of independent periodic tasks and then apply results developed for independent tasks [Mok et al. 87, Sha et al. 87]. For periodic tasks with no preemption constraints, the conditions that are necessary and sufficient for feasibility are stated in terms of the processor utilization U of the system:

$$U = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq \alpha,$$

where the value of α, $\alpha \leq 1$, varies according to constraints imposed on the scheduler. In our tasking model we had $\alpha = 1$. For the purposes of discussion, we can consider α to be a constant. The reductions from the constrained task system to the non-constrained task system employed by other researchers have typically imposed further restrictions on the utilization of the system. A common form for the viability conditions for task sets with preemption constraints is $U \leq \alpha - B$, where B is a function of the durations that tasks in the system can be blocked [Mok et al. 87, Sha et al. 87]. The reduction process results in conditions that are sufficient for ensuring the temporal correctness of a set of tasks but that are not necessary. In effect, these methods are sacrificing processor utilization to gain viability.

Our approach to study of feasibility has been based solely on the examination of the relative sizes of the task's periods and costs and not on utilization. The majority of our analysis has been directed at determining if a task's period is large enough to accommodate the blockage due to other tasks' executions. The constraints we impose on a task's period (e.g., conditions (2) and (3) in Theorem 4.4.1) are not a function of processor utilization. It is also the case that processor utilization is not directly effected by these constraints. For example, for the feasibility conditions of Theorem 4.4.1, it is possible to have task sets that satisfy condition (1) (a utilization constraint) but that do not satisfy condition (2) or (3). Similarly, it is possible to have task sets that satisfy conditions (2) and (3), but that do not satisfy condition (1). We conclude from these observations that, assuming that the processor is not overloaded (condition (1) holds), the viability conditions of task sets with preemption constraints need not be considered a function of processor utilization. One does not have to trade-off utilization to ensure viability.

4.7.4 Employing Multiple Processors

In this chapter we have considered two problems in a rich spectrum of feasibility problems. To illustrate this spectrum we state, but do not solve, a third problem. The problem involves determining feasibility conditions for cyclic tasks, with resource requirements, on a set of completely connected, identical, processors.

For the task model of Section 4.3, consider the problem of determining the feasibility of tasks when each task executes on a dedicated processor. This problem represents one end-point of a spectrum of solution techniques to feasibility problems. When we considered the feasibility of tasks on a uniprocessor when preemption was not allowed, a solution consisted solely of the definition and analysis of a scheduling algorithm. When we considered tasks on a uniprocessor when preemption was allowed, a solution consisted of the integration of a scheduling and synchronization discipline. For tasks on dedicated processors, a solution will consist solely of a synchronization discipline. This spectrum of solution requirements is illustrated in the figure below. While the implementation environments for the end-points of this spectrum are unique, the environments in the middle are quite far ranging.

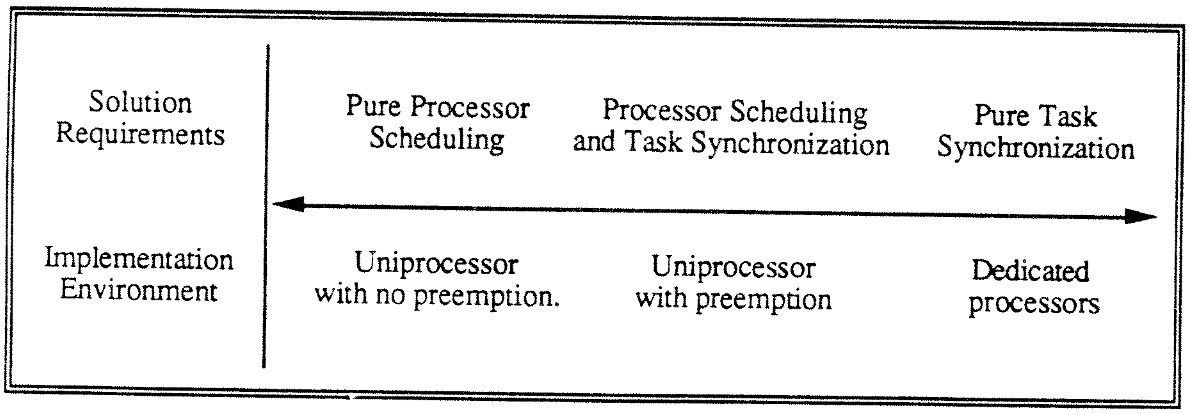

Solution Requirements
Pure Processor Scheduling
Processor Scheduling and Task Synchronization
Pure Task Synchronization
Implementation Environment
Uniprocessor with no preemption.
Uniprocessor with preemption
Dedicated processors

Figure 4.7.1: A continuum of solutions to feasibility problems.

While we offer no solution for the feasibility problem on dedicated processors, we can make a few observations about the nature of such a solution. For a task T, a necessary condition for feasibility will certainly be that $c \leq p$. To be a sufficient condition some be included. If a task has n phases then a necessary and sufficient feasibility condition would be

$$\sum_{i=1}^{n} c_i + B_i \quad \leq \quad p \ ,$$

where B_i represents the worst case blockage that the task can encounter while attempting to gain access to the resource required for phase i. This is an interesting and important departure from the feasibility conditions of the previous implementation environments. Since tasks are dedicated to processors, each B_i represents an interval of time for which the processor executing task T must (in the worst case) be idle. Therefore feasibility now *is* a function of processor utilization. If there is any sharing of resources then processors necessarily cannot be fully utilized if the release times of tasks are independent.

Note that this observation will also hold for implementation environments in which tasks that share resources can execute on different processors. An implication of this is that simply adding additional processors to an implementation of an infeasible task system will not necessarily guarantee feasibility. Adding additional processors will not reduce a task's cost and, after a point, it will not reduce the blockage a task can experience. On the other hand, replacing processors with faster processors always has the potential for making infeasible task sets feasible.

4.8 Summary

This chapter has studied a characterization of real-time systems based on sporadic tasks with resource requirements. We have developed a model of an implementation termed an *implementation strategy*. An implementation strategy consists of a scheme for synchronizing access to shared resources and for scheduling tasks. We have presented and analyzed an implementation strategy based on monitors, with wait and broadcast synchronization primitives, and deadline scheduling.

Several problems of executing sporadic tasks with resource requirements, on a machine that can dispatch and preempt tasks in zero time, have been studied. We have shown that our implementation strategy is an optimal strategy for single and multiple phase sporadic tasks that share a single resource. The optimality is again with respect to the class of implementation strategies whose schedulers do not use inserted idle time.

For tasks that share more than one resource, it was shown that allowing preemption between resource consuming tasks often can lead to sub-optimal strategies. If there is no overlap in the periods of the resource consuming tasks, then our implementation strategy is again an optimal strategy. If there is an overlap in the periods, then the determination of necessary conditions for the feasibility of multiple resource single phase systems remains an open problem. The viability conditions are sufficient for these systems. Lastly, we have shown how multiple phase systems can be modeled by a single phase system with precedence constraints. The implementation strategy can be easily extended to incorporate these precedence constraints. While our focus has been on a model of sporadic tasks, all of the results are applicable (as sufficient conditions) to periodic tasks.

Chapter 5

Realizing the Real-Time Producer/Consumer Paradigm

5.1 Introduction

In this chapter we investigate the problem of implementing designs created using the discipline presented in Chapter 2. The key problem in implementing the discipline is to ensure that all interconnected processes obey the RT/PC paradigm.

A design graph is said to be *realizable* on a uniprocessor, if it is possible to implement the design on a uniprocessor so that the design will be temporally correct. An implementation of a design graph is *temporally correct* if every pair of vertices connected with an asynchronous channel is guaranteed to adhere to the RT/PC paradigm. Realizability is to design graphs what feasibility is to cyclic tasks; it is an absolute measure of temporal correctness. Given a fast enough processor, in principle, we will always be able to implement a design graph that is realizable. Design graphs that are not realizable are impossible to implement.

The first problem is to identify the class of design graphs that are realizable. In order for a design to be realizable we must be able to determine the message transmission rates on each asynchronous channel in the graph. Section 5.2 develops a characterization of design graphs that are realizable. Whether or not a graph is realizable will be a function of the existence and structure of cycles in the graph. Since this analysis is not dependent on the actual methodology used to implement a design graph, we term this analysis *processor independent analysis.*

For those designs that are realizable, we must develop a methodology for mapping a graph onto a processor in such a manner that the temporal correctness of this mapping is apparent. Section 5.3 demonstrates how one decomposes a design into a set of sporadic tasks such that the feasibility decision procedures of Chapters 3 and 4 can be used to determine the temporal correctness of the implementation.

In Sections 5.4 and 5.5 we revisit some of the design issues in Chapter 2. If we fix an implementation strategy then we can exploit properties of this strategy and customize the design discipline for this strategy. Section 5.4 shows that the bounds on message propagation delays from Chapter 2 can be significantly tightened for the implementation strategies of Chapter 4. In addition, Section 5.4 also demonstrate how the range of these bounds will be a function of the number of processors used in an implementation. Section 5.5 examines some extensions to the discipline from the perspective of viability.

A prototype kernel that implements processes and channels according to the strategy outlined in this chapter and the previous one, has been constructed and used to implement the digital stopwatch. The scheme described below for implementing design graphs has been incorporated into the kernel. The design and performance of the kernel are described in Appendix C.

5.2 Processor Independent Analysis

The semantics of message passing require that the RT/PC paradigm hold on every asynchronous channel in a design graph. In Chapter 2 we showed this meant that each *process* was required to consume messages within $1/r$ time units of their arrival, where r was the worst case maximum transmission rate on the channel delivering the message. To implement, or reason about a design, we must be able to compute these rates. Section 2.6 has discussed the determination of maximum message transmission rates for asynchronous channels in acyclic design graphs. Given a fast enough processor, we will always be able to implement these designs. (Recall from Chapter 2 that the semantics of message passing on synchronous channels is defined in terms of the message transmission rates on asynchronous channels. Therefore, to implement or reason about a design, we need only compute the transmission rates on asynchronous channels.)

For cyclic design graphs, it may not be possible to determine the maximum rates for channels in a cycle. That is, there may not exist a solution to the transmission rate functions for the channels in a cycle. If it is not possible to determine maximum transmission rate on a channel then the behavior of the design is undefined.

Our ability to determine these rates will be a function of:

- the slopes of the transmission rate functions of the channels in a cycle and
- whether or not cycles are nested.

The purpose of this section is to formulate a characterization of the class of realizable design graphs. From this characterization we can derive a set of rules for the construction of designs with cycles. Adherence to these rules will be a necessary and sufficient condition for realizability.

We will study the case of disjoint cycles in a design graph and a special form of nested cycles. Two cycles in a graph are *disjoint* if no process appears in both cycles.

5.2.1 Disjoint Cycles

Consider the cycle of n processes and asynchronous channels shown in Figure 5.2.1 below. Each channel in the cycle is labelled with both the transmission rate function and the transmission rate. Let r'_i be the sum of the rates at which process P_i receives messages from processes not in the cycle.

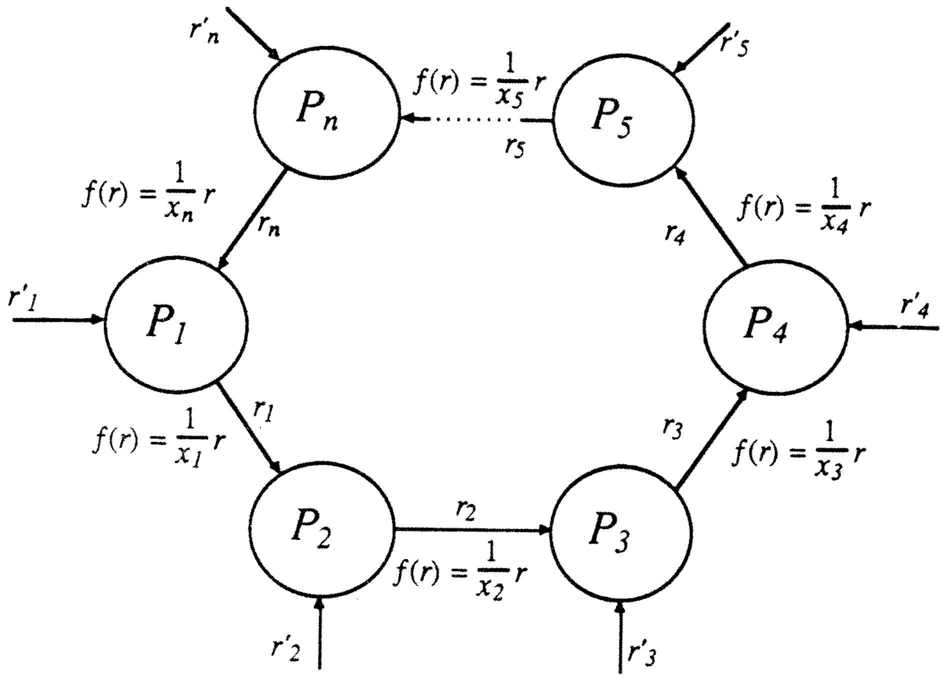


Figure 5.2.1: A cycle of asynchronous channels.

The transmission rates along channels in a cycle can be described with a set of simultaneous linear equations. In Figure 5.2.1 above, MP/SC process P_1 receives messages at a logical input rate of $r'_1 + r_n$. Its output on the channel from P_1 to P_2 is simply

$$r_1 = \frac{r'_1 + r_n}{x_1} .$$

Working our way around the processes in the cycle, we have the following set of equations for the transmission rates on the other channels in the cycle:

$$\left.\begin{aligned} r_1 &= \frac{r_n + r'_1}{x_1} \\ r_2 &= \frac{r_1 + r'_2}{x_2} \\ r_3 &= \frac{r_2 + r'_3}{x_3} \\ &\vdots \\ r_n &= \frac{r_{n-1} + r'_n}{x_n} . \end{aligned}\right\} \qquad (5.2.1)$$

These equations can easily be rewritten in terms of the rate on a single channel. Rewriting (5.2.1) in terms of r_n yields:

$$\left.\begin{aligned}
r_1 &= \frac{r_n}{x_1} + \frac{r'_1}{x_1} \\
r_2 &= \frac{r_n}{x_1x_2} + \frac{r'_1}{x_1x_2} + \frac{r'_2}{x_2} \\
r_3 &= \frac{r_n}{x_1x_2x_3} + \frac{r'_1}{x_1x_2x_3} + \frac{r'_2}{x_2x_3} + \frac{r'_3}{x_3} \\
&\vdots \\
r_k &= \frac{r_n}{\prod_{i=1}^{k} x_i} + \sum_{i=1}^{k} \frac{r'_i}{\prod_{j=i}^{k} x_j} \, .
\end{aligned}\right\} \tag{5.2.2}$$

The following theorem establishes necessary and sufficient conditions for solving these equations for the case when all cycles in a design graph are disjoint.

Theorem 5.2.1: Let G be a design graph with a cycle C of n distinct processes and asynchronous channels. Let $1/x_1$, $1/x_2$, ..., $1/x_n$ be the slopes of the transmission rate functions for the channels in C. If C is disjoint from other cycles in the graph, then the transmission rate functions for the channels in C can be solved if and only if

$$\prod_{i=1}^{n} x_i > 1 \, . \tag{5.2.3}$$

This condition is simply a requirement that at least one channel in the cycle have a non-identity transmission rate function ($x_i \neq 1$ for some transmission rate function). The denominator of the coefficient in a transmission rate function corresponds to the number of messages a process receives before it emits a message on the channel. Therefore, (5.2.3) requires that some process in the cycle must be guaranteed to always delay for at least two input messages before emitting a message on a channel in the cycle.

Proof: Each transmission rate in (5.2.2) is given in terms of the rate r_n. Therefore, if we can determine r_n we can determine the rates of all channels in the cycle.

According to (5.2.2), for process P_n, we have its output rate given by

$$r_n = \frac{r_n}{\prod_{i=1}^{n} x_i} + \sum_{i=1}^{n} \frac{r'_i}{\prod_{j=i}^{n} x_j} \, .$$

Let $R_k = \sum_{i=1}^{k} \frac{r'_i}{\prod_{j=i}^{k} x_j}$. Solving the above for r_n gives

$$r_n = \frac{r_n}{\prod_{i=1}^{n} x_i} + R_n ,$$

$$= \frac{R_n \prod_{i=1}^{n} x_i}{\prod_{i=1}^{n} x_i - 1} . \qquad (5.2.4)$$

Therefore, the transmission rates for channels in the cycle can be determined only if

$$\prod_{i=1}^{n} x_i - 1 \neq 0 .$$

Since the x_i are positive integers, this is a requirement that product of the x_i simply be greater than 1. This establishes the necessity of condition (5.2.3). Note this condition is not dependent on where in the cycle we chose to start deriving transmission rates (it is independent of the labelling of the processes).

If (5.2.3) holds, then replacing r_n in (5.2.2) yields for all $k < n$:

$$r_k = \frac{R_n \prod_{i=k+1}^{n} x_i}{\prod_{i=1}^{n} x_i - 1} + R_k . \qquad (5.2.5)$$

If cycles in a graph are disjoint, then for all processes P_i in a cycle, the r'_i (the sum of the rates at which process P_i receives messages from processes not in the cycle) can be treated as constants. Since, R_k is a function of the r'_i and x_j, which are constants, for all k, R_k can be computed. Therefore, if the product of the x_i is greater than 1, then the transmission rates of channels in the cycle can be efficiently computed according to (5.2.4) and (5.2.5).

Δ

5.2.2 Non-disjoint Cycles

If a graph contains non-disjoint cycles, then the analysis is considerably more complex. The following theorem gives necessary conditions for determining transmission rates in these cycles. The main idea is to concentrate on *simple* cycles in the graph. A simple cycle is a cycle in which all vertices are disjoint.

Theorem 5.2.2: Let G be a design graph with non-disjoint cycles. Let C_1 and C_2 be two non-disjoint, simple, cycles. Assume C_1 and C_2 contain n and m processes respectively. If processes in C_1 are interconnected with channels whose transmission rate functions have slopes $1/x_1$, $1/x_2$, ..., $1/x_n$, and processes in C_2 are interconnected with channels whose transmission rate functions have slopes $1/y_1$, $1/y_2$, ..., $1/y_m$, then the transmission rate functions for the channels in C_1 and C_2 can be solved only if

$$\left(\prod_{i=1}^{n} x_i - 1\right)\left(\prod_{i=1}^{m} y_i - 1\right) > 1 . \qquad (5.2.6)$$

This condition requires that each cycle have at least one channel with a non-identity transmission rate function. That is, some process in each cycle must always delay for at least two messages before emitting a message. In addition, in one of the cycles there must a process, or processes, who cumulatively delay for at least three messages. That is, there must exist a sequence of processes in one of the cycles, in which at least three message are always required to be sent to the first process in the sequence, before a message can ever be emitted from the last process in the sequence.

Proof: If C_1 and C_2 are not disjoint then there exists a process P_B in C_1 and C_2 that receives messages from a process P_A in cycle C_1 but not in cycle C_2, and from a process $P_{A'}$ in cycle C_2 but not in cycle C_1. Let r'_B be the sum of the rates at which process P_B receives messages from processes not in cycle C_1 or C_2. Assume that each channel (and the process that emits messages on it) in each cycle is numbered consecutively around the cycle. Let r_i and s_j represent the transmission rates on channels in cycles C_1 and C_2 respectively. If a process or channel appears in both cycles then it will have one label in cycle C_1 and a potentially different label in cycle C_2. Assume that the channels in both cycles are labelled in such a manner that process P_B receives messages from process P_A on channel number k in cycle C_1, and messages from process $P_{A'}$ on channel number k in cycle C_2 as shown in Figure 5.2.2 below. Note that channel k+1 in cycle C_1 may be the same physical channel as channel k+1 in cycle C_2.

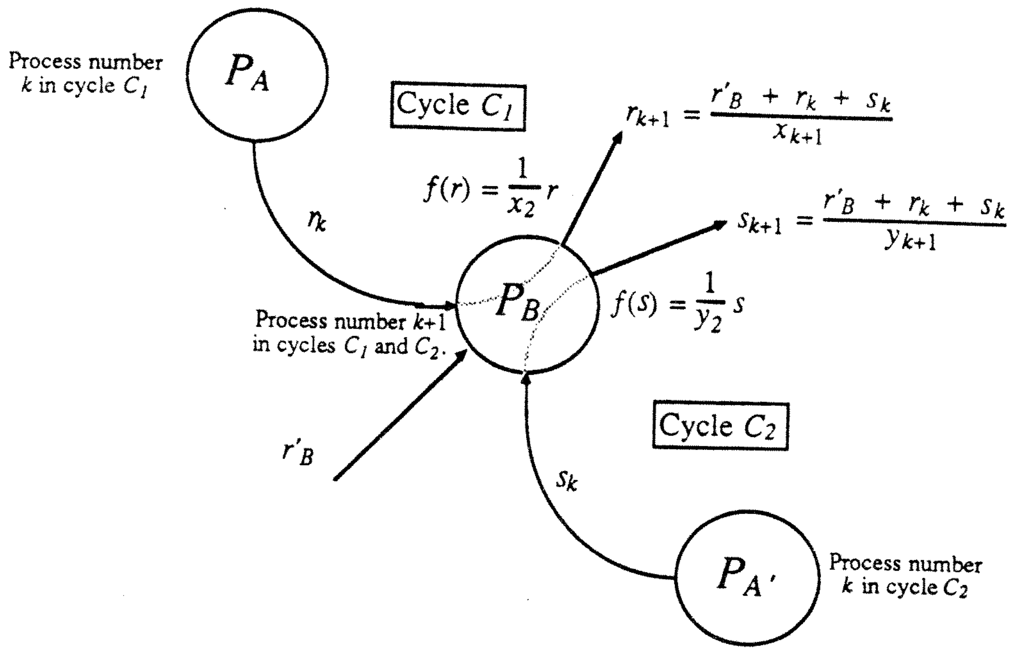

Process number k in cycle C_1
P_A
Cycle C_1
$r_{k+1} = \frac{r'_B + r_k + s_k}{x_{k+1}}$
$f(r) = \frac{1}{x_2} r$
r_k
$s_{k+1} = \frac{r'_B + r_k + s_k}{y_{k+1}}$
P_B
$f(s) = \frac{1}{y_2} s$
Process number k+1 in cycles C_1 and C_2.
Cycle C_2
r'_B
s_k
$P_{A'}$
Process number k in cycle C_2

Figure 5.2.2: A pair of intersecting cycles.

For ease of notation, let

$$X = \prod_{i=1}^{n} x_i, \quad \text{and} \quad Y = \prod_{i=1}^{m} y_i .$$

From Theorem 5.2.1 we know that in order to determine the transmissions rate r_i and s_j, we must have $X > 1$, and $Y > 1$. If these conditions hold, then as in the previous theorem, the transmission rates of each channel in C_1 and C_2 are given by

$$r_z = \frac{R_n \prod_{i=z+1}^{n} x_i}{X - 1} + R_z , \tag{5.2.7}$$

$$s_z = \frac{S_m \prod_{i=z+1}^{m} y_i}{Y - 1} + S_z , \tag{5.2.8}$$

where S_z is simply R_z rewritten in terms of s and y for processes in C_2. Unlike the previous theorem, (5.2.7) and (5.2.8) cannot be solved immediately. Recall that $R_k\,(S_k)$ is defined in terms of $r'_i\,(s'_j)$ which are the rates at which processes emitting on channels in $C_1\,(C_2)$ receive messages from processes not in $C_1\,(C_2)$. Since process P_B receives messages from processes in C_1 and C_2, we have

$$r'_{k+1} = r'_B + s_k, \quad \text{and} \quad s'_{k+1} = r'_B + r_k.$$

That is, unlike the previous theorem where the r'_i and s'_j were constants, r'_{k+1} and s'_{k+1} are functions of the rates at which messages travel in the other cycle. Since (5.2.7) and (5.2.8) depend on these values, (5.2.7) and (5.2.8) must be solved simultaneously.

The equations for r_k and s_k can combined to eliminate the dependence of r'_{k+1} and s'_{k+1} on r_k and s_k. Rewriting (5.2.7) for r_k in terms of s_k we have

$$\begin{aligned}
r_k &= R_n \left(\frac{\prod_{i=k+1}^{n} x_i}{X - 1}\right) + R_k \\
&= \sum_{i=1}^{n} \frac{r'_i}{\prod_{j=i}^{n} x_j} \left(\frac{\prod_{i=k+1}^{n} x_i}{X - 1}\right) + R_k \\
&= \left(\frac{r'_{k+1}}{\prod_{j=k+1}^{n} x_j} + \sum_{i=1, i\neq k+1}^{n} \frac{r'_i}{\prod_{j=i}^{n} x_j}\right) \left(\frac{\prod_{i=k+1}^{n} x_i}{X - 1}\right) + R_k \\
&= \frac{r'_{k+1}}{\prod_{j=k+1}^{n} x_j} \left(\frac{\prod_{i=k+1}^{n} x_i}{X - 1}\right) + \sum_{i=1, i\neq k+1}^{n} \frac{r'_i}{\prod_{j=i}^{n} x_j} \left(\frac{\prod_{i=k+1}^{n} x_i}{X - 1}\right) + R_k \\
&= \frac{r'_{k+1}}{\prod_{j=k+1}^{n} x_j} \left(\frac{\prod_{i=k+1}^{n} x_i}{X - 1}\right) + R'_k \\
&= \frac{r'_{k+1}}{X - 1} + R'_k \\
&= \frac{s_k + r'_B}{X - 1} + R'_k \\
&= \frac{s_k}{X - 1} + \frac{r'_B}{X - 1} + R'_k \\
&= \frac{s_k}{X - 1} + R''_k \qquad (5.2.9)
\end{aligned}$$

where R'_k and R''_k are the compilation of terms containing only constants. In a similar manner we can derive an expression for s_k in terms of r_k. Substituting one into the other yields

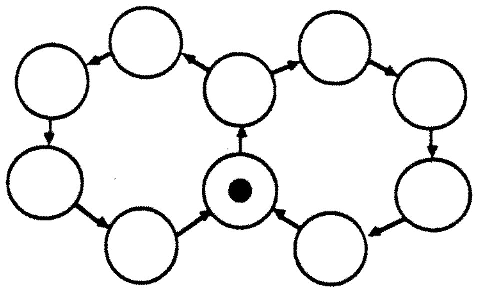
$$\begin{aligned} s_k &= \frac{r_k}{Y-1} + S''_k \\ &= \left(\frac{s_k}{X-1} + R''_k\right)\frac{1}{Y-1} + S''_k \\ &= \frac{s_k}{(X-1)(Y-1)} + \frac{R''_k}{Y-1} + S''_k \\ s_k(X-1)(Y-1) &= s_k + R''_k(X-1) + S''_k\,(X-1)(Y-1) \\ s_k((X-1)(Y-1)-1) &= R''_k(X-1) + S''_k\,(X-1)(Y-1) \\ s_k &= \frac{R''_k(X-1) + S''_k(X-1)(Y-1)}{(X-1)(Y-1)-1} \end{aligned} \tag{5.2.10}$$

Therefore, the transmission rates for channels in C_1 and C_2 can be determined only if

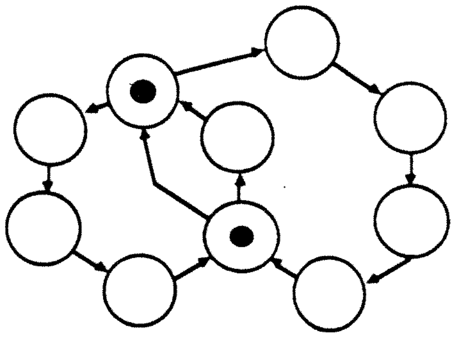
$$(X-1)(Y-1)-1 = \left(\prod_{i=1}^{n} x_i - 1\right)\left(\prod_{i=1}^{m} y_i - 1\right) - 1 \neq 0 .$$

Since the x_i and y_j are positive integers, then the above must be greater than 1. Δ

Unfortunately, we do not have sufficient conditions for determining when transmission rates can be computed for arbitrary patterns of nested cycles. However, if we limit ourselves to graphs in which each pair of simple cycles has at most a single process that receives messages from a distinct process in each cycle, then (5.2.6) is also a sufficient condition for determining transmission rates. For example, consider the cycles in Figure 5.2.3.



Cycles for which (5.2.6) is a sufficient condition for determining transmission rates.



Cycles for which (5.2.6) is not a sufficient condition for determining transmission rates.

Figure 5.2.3: Examples of nested cycles.

Corollary 5.2.3: Let G be a design graph with non-disjoint cycles. If every pair of cycles in G contain at most one process that receives messages from a distinct process in each cycle, then (5.2.6) is also a sufficient condition for determining transmission rates.

Proof: Assume every pair of cycles in G contains at most one process that receives messages from a distinct process in each cycle. For two non-disjoint cycles in G, the rates for the channels in both cycles can be expressed in terms of (1) the slopes of the transmission rate functions around the two cycles, and (2) the rates at which messages arrive from outside the two cycles. Some of these latter rates may be a function of the rates of channels in other cycles in the same strongly connected component of the design graph. However, performing this analysis for all pairs of non-disjoint cycles will yield an expression for the rate of each channel in a strongly connected component of the graph, in terms of slopes of transmission functions and the rates at which message arrive from outside of the strongly connected component. These latter rates will be constants. At this point all the rates can be solved directly. Δ

If a design graph contains non-disjoint cycles with more pathological interconnections, then to determine if worst case maximum rates exist, one will have to write, and then solve, the full set of simultaneous linear equations for these rates. In this manner one can determine if transmission rates can be determined for an arbitrary design graph. For the designs that have been studied to date, equations (5.2.4) and (5.2.5) have been sufficient for computing rates in all of the cycles encountered.

5.3 Processor Dependent Analysis

With the results of the previous section, we can identify those design graphs that in principle are realizable. In this section we show how these design graphs can be implemented using the sporadic tasking model of Chapter 4. This will serve to demonstrate two points. First, it will show that our design discipline is tractable in the sense that it can be implemented and that the temporal correctness of an implementation can be guaranteed. Secondly, we will argue that the task model of Chapter 4 is a good vehicle for implementing our designs in the sense that it is likely to be difficult to do any better model in terms of realizing actual designs.

To employ the sporadic tasking model of Chapter 4 to implement our design discipline we must (1) determine how to map the components of our discipline into a set of sporadic tasks and (2) determine how properties of the model can be used to infer the temporal correctness of this implementation of a design. Throughout this section we will assume designs are implemented on a uniprocessor.

5.3.1 Design Decomposition

The natural decomposition to consider is to create a task for each process. Conceptually, tasks will make requests for execution when their corresponding process receives a message. Tasks will terminate when they have consumed the message. It will turn out that there is a slight advantage to taking a different approach and creating a task for each *asynchronous channel*. In this scheme a task will execute the code of the process that consumes the messages sent on the channel. For SP/SC processes, this mapping is equivalent to creating a task for each process. For MP/SC processes, if the process consumes messages from *n* sources, then *n* tasks are created with each of these *n* tasks executing the same process code. Since an MP/SC process constitutes an implicit mutual exclusion region, these *n* tasks will not be allowed to preempt one another. The reason for this choice of mappings will become apparent when we discuss the application of the viability analysis in Section 5.3.4.

Note that since processes do not contain persistent data, if we create multiple copies of an MP/SC process, there will be no consistency problems with any data used by the original process.

Logically, each task is a multiple resource/multiple phase task. There will be two types of resources in an implementation of our discipline: *data resources* and *lock resources*. Data resources are MP/SC data repositories and lock resources are mutual exclusion locks. Tasks will be partitioned into a number of phases based on the number of messages sent to MP/SC data repositories. Specifically, phases are delineated by synchronous message emissions to MP/SC data repositories.

For example, consider the process code shown in Figure 5.3.1. The code is for a process that emits messages on three synchronous channels. This process is configured in a design graph in which the data repositories *A* and *B* are MP/SC data repositories. That is, these data repositories are shared among a set of processes. Therefore, the S_EMIT statement for the channels connected to data repositories *A* and *B* will comprise a single phase. The synchronous emit statements for channel out_mesg2 and channel out_mesg3 will each comprise a separate phase. The remaining code between these S_EMIT statements, and between these S_EMIT statements and the ACCEPT and END statements will also form a phase as shown in Figure 5.3.1.

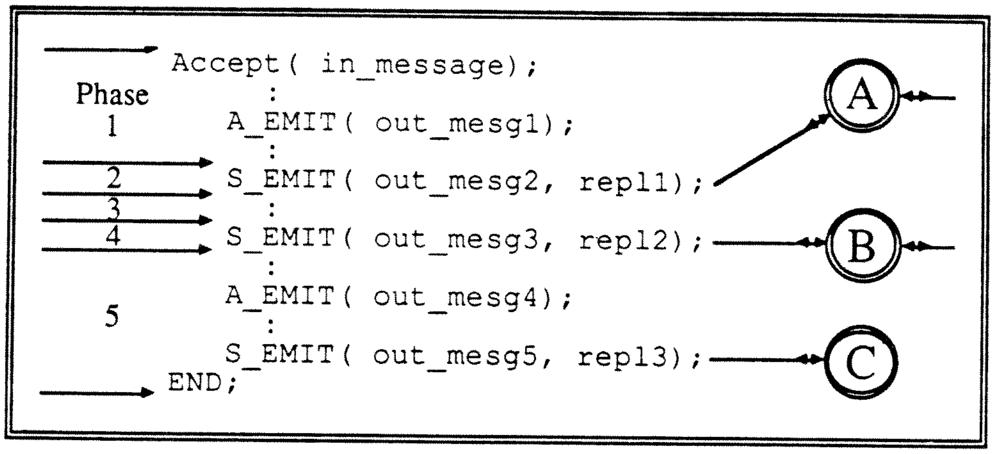

```
Accept( in_message);
        :
    A_EMIT( out_mesg1);
        :
    S_EMIT( out_mesg2, repl1);
        :
    S_EMIT( out_mesg3, repl2);
        :
    A_EMIT( out_mesg4);
        :
    S_EMIT( out_mesg5, repl3);
END;
```

Figure 5.3.1: Task phase definitions for a process.

Using the notation of the previous chapter, the ACCEPT statement is conceptually expanded into:

```
< Wait for a message to arrive. >
BEGIN
    ResourceR0.Request(); ,
```

the S_EMIT statement for a channel connected to an MP/SC data repository expands into:

```
        ResourceR0.Release();
        ResourceRj.Request();

            < send a message to MP/SC data repository j
              and wait for a reply >

        ResourceRj.Release();
        ResourceR0.Request(); ,
```

and the END statement becomes:

```
        ResourceR0.Release();
    END; .
```

If a message is sent to an MP/SC data repository from within a loop, then conceptually the loop is unrolled and then the phases are determined. (Recall that for each loop, the maximum number of iterations of a loops is assumed to be known.) Since the number of phases of a task depends on whether or not the data repositories its process uses are shared, the number of phases in a task can vary for uses of the same process in different designs.

The only form of nested resource usage that can occur is when a process in a mutual exclusion region sends a message to an MP/SC data repository (see Appendix B). The task corresponding to the process will need to posses a lock resource and a data resource simultaneously. In this case the entire body of process code above is bracketed with a request and release operation for the appropriate mutual exclusion lock. While the implementation strategies of Chapter 3 do not support general nested resources, this form of resource usage can be incorporated with little trouble. The problem with arbitrary nested resource requirements is that a task can deadlock while trying to access a shared resource. If a task can deadlock then the worst case blockage a task can experience while trying to gain access to a shared resource cannot be computed. In our discipline deadlock is not possible (see Appendix B for a proof of this). Therefore, for this form of nested resource requirement, we are always able to assess the blockage. Section 5.3.4 discusses how the viability decision procedures can be modified to account for this form of nested resource requirements.

Finally, note that certain very simple MP/SC data repositories need not be treated as data resources. This is because the hardware of a system may be able to guarantee mutually exclusive message processing. For example, for the stopwatch system in Chapter 2, the Watch State MP/SC data repository could be implemented with a simple integer variable (see Figure 2.7.3). In this case non-preemptive read and write operations would be implemented in hardware (i.e., by the bus).

5.3.2 Determination of Task Parameters

We have chosen to implement a design as a set of sporadic tasks because processes in our discipline may pause for an arbitrary number of input messages before emitting an output message. This means that the task corresponding to the channel on which these messages are emitted, will potentially make sporadic requests for execution.

Recall that a sporadic task is characterized by three parameters: *release time*, *computational cost*, and *period*. We will assume that the initial release times of tasks are unknown. Since a process may choose to wait for an arbitrary number of input messages before emitting a message, it will not be easy to determine the exact starting time of each task. We are not motivated to pursue this issue since the results of Chapter 3 indicate that there is no advantage to knowing release times if tasks actually make aperiodic requests.

Each task will have a computational cost equal to the worst case maximum processing time for the messages that travel on its corresponding channel. The total cost of a task will be composed of

- the cost of executing the code of the corresponding process, plus
- the overhead cost of sending and receiving messages.

We will need the cost of a task broken down into the cost of individual phases. The execution time cost of a phase consists of either

- the cost of executing the sequential code in the phase, plus the cost of sending synchronous messages to SP/SC data repositories multiplied by the number of times that a message is sent to each repository,

or

- the cost of sending synchronous messages to an MP/SC data repository.

The former cost is the cost of a non-resource consuming phase and the latter cost is of a resource consuming phase. On a uniprocessor, the cost of sending a message to a data repository is

- the cost of processing a message at the data repository, plus
- the cost of the emit and reply operations.

Since data repositories do not emit any messages, their execution time cost can be determined independently of their actual use. Note that the execution time cost for an MP/SC data repository is strictly the time required for it to consume a message. It does not (and need not) include any measure of the delay incurred while waiting to gain access to the shared repository.

We do not formally address the problem of determining the execution time costs of actual code blocks. There are two basic approaches to this problem. The first is primarily an analytic approach based on an examination of the source code and detailed knowledge of the languages implementation [Shaw 89]. The second is a more empirical approach and involves the careful measurement of execution times of program fragments. This problem area is an emerging area within the real-time community. Given the current state of analytic techniques, we believe that it is appropriate to measure these costs empirically by executing processes and data repositories in isolation and recording the worst case execution time for the messages they are expected to receive. This worst case time must exist since because of the constraints imposed in Chapter 2 on the implementation of the language used to construct these components. The primary drawback to this approach is that for non-trivial programs, it is difficult to determine if the worst case execution time has indeed been measured. This problem is exacerbated when processes are executed together and interfere with one another in subtle ways [Park & Shaw 89].

The semantics of message passing are embedded in a task's period. A task's period will be the worst case minimum message interarrival time on the asynchronous channel corresponding to the task. The minimum interarrival time will be the reciprocal of worst case maximum message transmission rate for a channel. For design graphs without pathological cycles, the periods of tasks can be efficiently determined by combining the results of Section 2.6 and Section 5.2.

5.3.3 Task Activation Schema

When describing the semantics of message passing we assumed that processes could consume a message in zero time. With this abstraction we could assume that there existed a minimum message interarrival time on all asynchronous channels in a design graph. When a design is decomposed into a set of tasks, this abstraction is no longer appropriate. A task will take time to consume a message. As discussed below, an implication of this is that tasks cannot be activated (make requests for execution) by the arrival of messages.

Recall that a task, with period p, must make execution requests in such a manner that it never makes two requests for execution within p time units. This can be troublesome if a task makes execution requests when it receives a message. In principle, a task may emit a message at any time during one of its request intervals. For example, consider the two SP/SC processes shown below. Two tasks, T_A and T_B, with periods $1/r_A$ and $1/r_B$ respectively, will be created for these processes. Suppose task T_A make execution requests at times t_1 and $t_1 + 1/r_A$. During the first request interval task T_A emits a message to task T_B at time t_2 as shown in Figure 5.3.2 below.

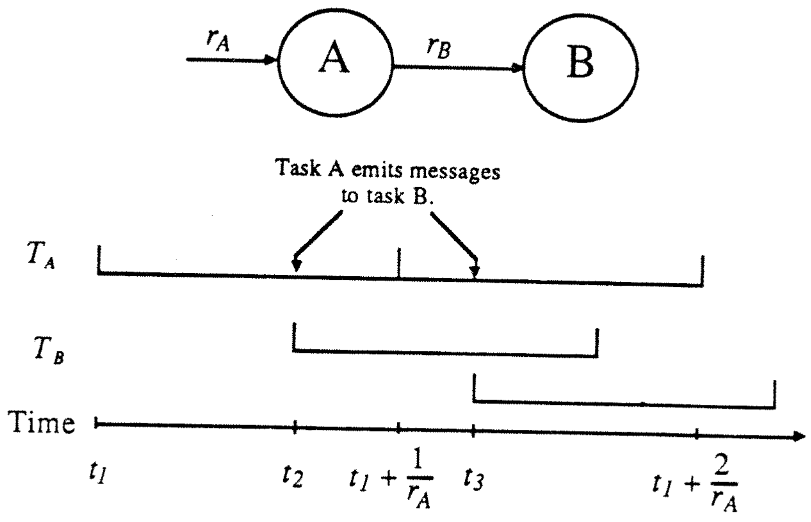


Figure 5.3.2: Message emission times versus task execution request times.

During its second request interval, task T_A may emit another message to task T_B at time $t_3 < t_2 + 1/r_B$. In this case task T_B cannot make execution requests at both times t_2 and t_3.

We will phrase the problem of determining when a message receiving task makes execution requests in terms of when a message sending task actually transmits asynchronous messages. That is, tasks will make requests for execution when they receive a message but the implementation's run-time system will not actually transmit a message when the sending task performs the emit operation. The task activation problem then reduces to the problem of determining when messages can physically be transmitted.

For tasks derived from SP/SC processes, a simple solution is to buffer the emission of messages until a fixed point in each request interval. For example, all asynchronous message could be buffered until the end of the current request interval of the message emitting task as shown in Figure 5.3.3 below. Since a task will never emit more than one message to any other task per execution request, this buffering scheme will ensure that all messages sent on asynchronous channels have the appropriate minimum interarrival time. Another, more optimistic approach will be presented in Section 5.4.

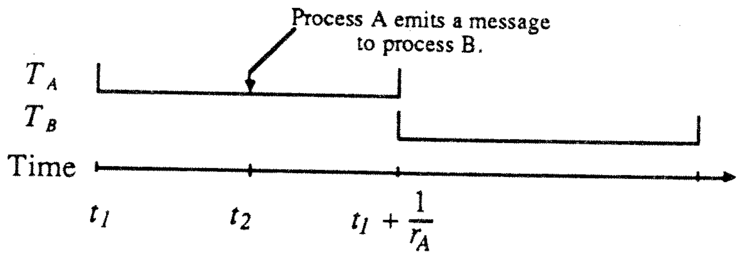


Figure 5.3.3: Buffering message emission times for the tasks of figure 5.3.2.

The situation is more complex for tasks derived from an MP/SC process. The asynchronous messages emitted by these tasks must be buffered in such a manner that they arrive at the receiving task with a minimum interarrival time equal to the inverse of the sum of the periods of the sending tasks. For each MP/SC process, an additional emitting task will be created. The emitting task will periodically check to see if a message emitted by a task derived from the MP/SC process can be transmitted.

5.3.4 Assessing Temporal Correctness

Given the proposed decomposition of a design into a set of sporadic tasks, we have three choices of an implementation strategy for these tasks. As outlined in Chapter 3, for a set of sporadic tasks with R physical resources, there exist implementation strategies with one, two, or (under restricted conditions) $k \leq R+1$ (including R_0), logical resources. If the sporadic tasks generated from a design graph are viable for one of these implementation strategies, then the resulting use of this strategy will be temporally correct. Viability ensures that no task will miss a deadline, that is, each execution request of each task will finish within $1/r$ time units. In terms of channels, every message sent on an asynchronous channel will be guaranteed to be consumed before the next message is sent on that channel.

Design graphs may generate tasks with nested resource requirements. Nested resource requirements result from processes in a mutual exclusion relation emitting messages to MP/SC data repositories. The viability decision procedures of Chapter 4 are sufficient, but not necessary, for ensuring the viability of these designs. They are sufficient since they assess the worst case blockage that the task can endure while waiting for both resources. They are not necessary in the event that there exists an MP/SC data repository that receives messages from multiple processes in the same mutual exclusion relation. Such a scenario is illustrated in Figure 5.3.4 below. In this case, the worst case blockage determined by the viability decision procedure for processes in the mutual exclusion relation can never occur. It can never occur since processes in the mutual exclusion relation can never compete with each other for access to the MP/SC data repository. For these tasks, the viability decision procedure can be modified by hand to exclude the effects of the tasks that never execute simultaneously.

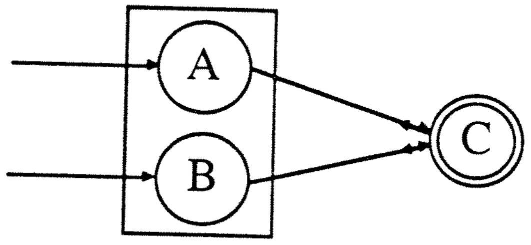


Figure 5.3.4: An MP/SC data repository that is effectively an SP/SC data repository.

In addition to determining temporal correctness, the decision procedure for viability can also be used as an analysis tool to derive tolerable input rates for devices whose input rates may be unknown. Recall from Chapter 2 that if the worst case minimum message interarrival time from an input device is not be known, then in the decomposition of a design graph, certain tasks will have their periods specified symbolically. The algorithms for deciding viability could be used to determine the smallest value for this variable that could be tolerated by the current implementation of the design. If multiple devices have unknown worst case behavior, then an optimization procedure could be employed to determine suitable values for these variables. In any case, since these devices are not likely to be guaranteed to respect these artificially derived interarrival time bounds, the design must be modified to ensure that the derived interarrival time is respected. An example of such a modification is presented in Chapter 5 (Section 5.2).

An alternate and more obvious task decomposition scheme would have been to simply create a task for each process. For MP/SC processes this task would have a period of $1/r_{in}$ where r_{in} is the process's logical input rate. By creating a task for each channel, a task derived from an MP/SC processes will have a period larger than $1/r_{in}$. When a task requires shared resources, there is an advantage, in terms of feasibility, for the task to have as large a period as possible. The larger the task's period, the more blockage it can endure and the less blockage it imposes on others. Therefore, in the event that an MP/SC process sends messages to MP/SC data repositories, there is an advantage to having tasks derived from MP/SC processes have as large a period as possible.

5.3.5 Temporal Correctness Versus Viability

We have commented that an affirmative answer from a viability decision procedure implies temporal correctness. However, a negative answer from a viability decision procedure does not necessarily imply an implementation will be temporally incorrect. Viability is a sufficient but not necessary condition for temporal correctness. It is important to understand the relationship between viability and temporal correctness so that when an implementation of a design is not viable, a designer can determine how best to expend her efforts to make the design viable. For a non-viable graph, a designer can either rework the viability analysis taking into account more specific information concerning the particular design or she can alter the design.

One reason why viability is not equivalent to temporal correctness is that the execution requests of tasks derived from a design graph are not independent. The task model of Chapter 3 assumes that the points at which a task makes a request for execution are independent of the execution requests of other tasks. Because of this, the viability analysis assumes that all possible interleaving of task execution requests are possible. This is clearly not the case for tasks derived from a design graph since tasks make execution requests based on the receipt of messages from other tasks. For example, consider a subgraph in which a process emits messages to one of two tasks according to the result of a branching decision, as shown in Figure 5.3.5 below.

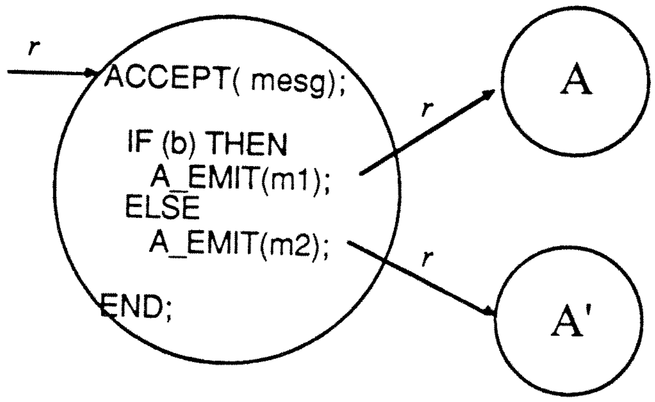


Figure 5.3.5: Processes that will never consume messages simultaneously.

If the two output channels shown above have the identity transmission rate function, then the tasks derived from processes A and A' will never execute at the same time. Therefore, these tasks can never interfere with one another. If processes A and A' send messages to an MP/SC data repository, or are in a mutual exclusion relation, then the viability analysis will be considering execution sequences that can never occur. In addition, the utilization computation will be overly pessimistic.

The problem of assessing the viability of dependent tasks is similar to the problem of nested resources mentioned in the previous section. However, unlike nested resources, the effect of dependent execution requests is not isolated to a small, well defined group of processes. In general, processes A and A' above can be the sources of subgraphs with arbitrary behavior. It is possible for certain portions of these subgraphs to never interfere with one another, while other portions interfere only in limited ways. A precise characterization of feasibility conditions for cyclic tasks with dependencies on execution requests remains an open problem.

Another reason why viability is not equivalent to temporal correctness is that the viability analysis assumes that worst case costs and execution rates are realizable simultaneously. Accounting for this behavior in a rigorous manner will also require a more sophisticated tasking model.

For design graphs that cannot be guaranteed to be temporally correct, a designer has the following options (if he wishes to use the unadulterated decision procedures of Chapter 3):

- rewrite the design graph,
- reduce the computational costs of processes and data repositories by either employing a faster processor or rewriting the algorithms in these components,
- increase the worst case message interarrival time of messages at processes that share data repositories or are in a mutual exclusion relation by rewriting the algorithms.
- introduce buffering data repositories between existing processes.

Note that the style of feasibility analysis we have pursued is constructive. In assessing feasibility and viability, we explicitly construct scenarios in which task will fail if conditions are not met. Therefore, if a task set is non-viable, we can demonstrate a set of conditions under which a task can fail. This information is likely to be useful to a designer charged with making a non-viable design viable.

5.4 Reasoning About Time Revisited

In Chapter 2 we demonstrated how one could determine message propagation delays through paths in a design graph. It was shown that for a sequence of k processes P_1 - P_k, with message arrival rates r_1 - r_k, the worst case delay between the arrival of a message at the first process in the sequence, P_1, and the emission of a message from the last process in the sequence, P_k, can be at most

$$\sum_{i=1}^{k} \frac{x_i}{r_i}, \tag{5.4.1}$$

time units. The maximum propagation delay for the message whose arrival at process P_I causes a message to be emitted from process P_k will be at most

$$\sum_{i=1}^{k} \frac{1}{r_i}, \tag{5.4.2}$$

time units. In this section we show that for the implementation of our design discipline with the cyclic tasking model of Chapter 3, in the worst case messages will be delayed for only a fraction of these amounts. We will concentrate on the latter form of message propagation delay.

The actual amount of time that a sequence of messages will be delayed along a path in a design graph will be a function of the number of processors used in the implementation. For a design with n processes, we will consider implementations on 1, n, and $k < n$ processors. In the case of multiple processors we will assume that the processors are identical and tightly coupled so that a task can communicate with tasks on different processors as efficiently as they can communicate with tasks on the same processor. In the case of $k \leq n$ processors, we further assume that there exists a partitioning of tasks onto processors and that this partitioning is fixed for the life of the system.

5.4.1 Single Processor Implementations

Our ability to determine the propagation delay for a sequence of messages hinges on knowing when tasks make execution requests relative to the time a message is logically sent to them. In the previous section we argued that tasks derived from SP/SC processes had to have their asynchronous output messages buffered until the end of their request interval. While this scheme ensures that there will always exist a minimum messages interarrival time, it leads to an implementation in which the best case message propagation delay for a sequence of SP/SC processes is exactly (5.4.2). We can do substantially better than this bound by using a technique we call *pre-scheduling.*

Let A be an SP/SC process that receives message at rate r_A, and let T_A be the task corresponding to process A. Let B be a process that receives messages from process A at rate $r_B = r_A/x_A$, and let T_B be the task corresponding to process B. Let t_1 correspond to the arrival time of a message at task T_A that will cause T_A to send a message to task T_B. If task T_A sends a message to task T_B at time t_2, then T_B will make a request for execution sometime at or after time t_2 as shown in Figure 5.4.1 below.

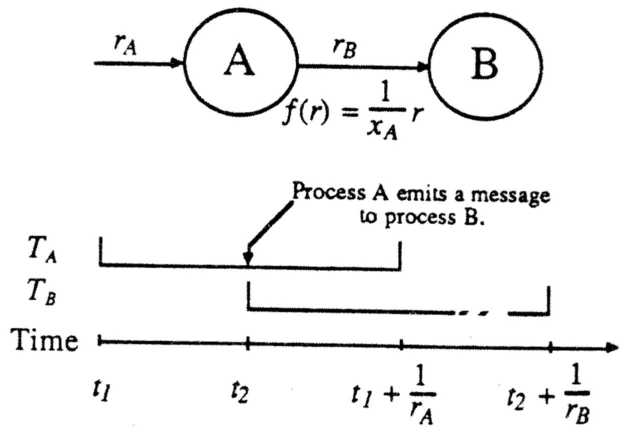


Figure 5.4.1: A process receiving a message from an SP/SC process and their associated tasks.

Instead of having task T_B make its execution request after the message is emitted, we can actually have T_B make its request for execution *before* the message is emitted as shown in Figure 5.4.2 below. Conceptually, the request interval of task T_B that will consume the message sent at time t_2, will begin at the same time as the request interval of task T_A that emitted the message.

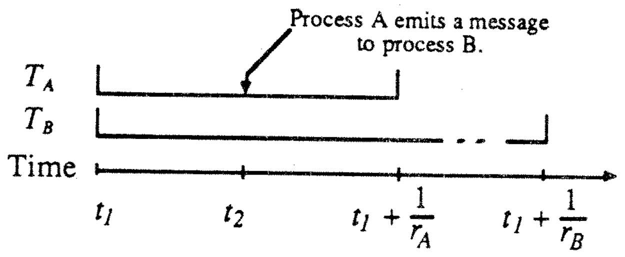


Figure 5.4.2: Simultaneous scheduling of tasks corresponding to an SP/SC process and one of its message receivers.

Since an SP/SC process's output rates are never greater than its input rate, the period of the task corresponding to the SP/SC process will always be less than or equal to the period of a task corresponding to a channel on which the SP/SC process emits messages. Therefore, when a task is created for an SP/SC process, the period of this task will always be less than or equal to the periods of the tasks created for the SP/SC process's output channels. If the receiving tasks never have a smaller period, then if the tasks make simultaneous execution requests, the receiving tasks can never have a nearer deadline than the sending task. For example, the execution request of task T_B made at time t_I can not have a nearer deadline than the execution request of task T_A made at the same time. Since our implementation strategies all use deadline scheduling, task T_B will not be scheduled until after the execution request of task T_A made at time t_I, has completed execution. (If tasks T_A and T_B have the same period (i.e., $x_A = 1$), then, as in Section 3.4, the scheduler can biased to always choose T_A over T_B if they ever have the same deadline.) When task T_B is finally scheduled, there will be a message from task T_A waiting for it.

We will refer to the simultaneous scheduling of execution requests of a sending and receiving task for a given message as the *pre-scheduling of the receiving task*. In general, when task T_A receives a message, T_A will not know if it will emit a message on a particular output channel. Therefore, the primary drawback to pre-scheduling tasks is the fact that we cannot determine those message arrivals at task T_A for which task T_B should be pre-scheduled. In practice this can be remedied with the following trick. Whenever a task corresponding to an SP/SC process emits a message, the task destined to receive the message will be inserted into the scheduler's ready queue with a deadline equal to the deadline it *would have had* if the receiving task *had* been pre-scheduled. This trick will not effect the functioning of the system since even if the receiving task *had* made its request for execution at the same time as the sending task, on a uniprocessor the receiving task still would not have been scheduled prior to the emission of the message since it will not have a nearer deadline than the sending task.

The pre-scheduling of task T_B does not effect the feasibility or viability of the task system. Chapter 3 has shown that the feasibility of a set of sporadic tasks is not dependent on their relative execution request times. Also, note that task T_B will never make execution requests less than $1/r_B$ time units apart if task T_A does not make execution requests more often than every $1/r_A$ time units and respects its output rates. This demonstrates that pre-scheduling does not pervert the periodicity of tasks.

The advantage of pre-scheduling a task is that we can guarantee a better worst case propagation delay for the message that arrives at task T_A and causes a message to be emitted to task T_B. With pre-scheduling this delay will be for at most $1/r_B$ time units.

From (5.4.2), without pre-scheduling the propagation delay could be as great as $1/r_B + 1/r_A$ time units. The technique of pre-scheduling a task can be applied transitively to a sequence of tasks derived from processes that receive messages from SP/SC processes. Consider the sequence of processes shown in Figure 5.4.3 below.

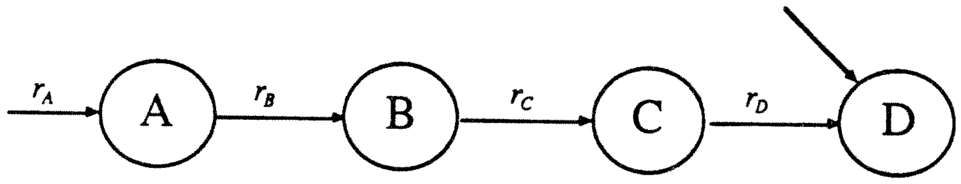


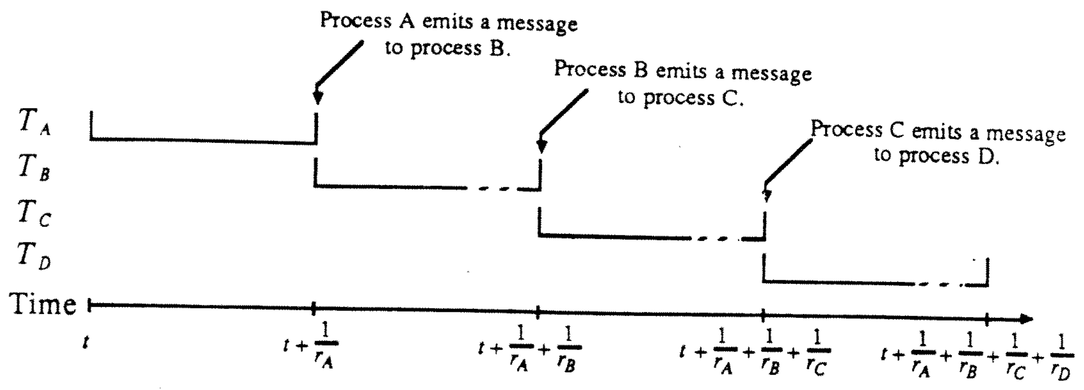
Figure 5.4.3: A chain of SP/SC processes.

As described above, the tasks derived from processes *A*, *B*, and *C*, can each be pre-scheduled with their predecessor. Therefore, when a message arrives at process *A* that causes a message to be emitted at process *C*, the four tasks corresponding to the processes can *all* be scheduled simultaneously. Therefore, with pre-scheduling, the worst case propagation delay for this message will be only $1/r_D$ time units. Without pre-scheduling the worst case delay could be up to

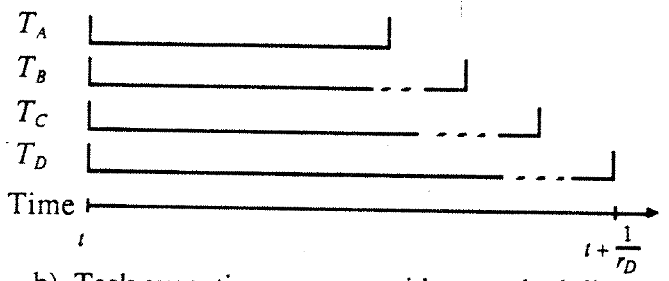
$$\frac{1}{r_A} + \frac{1}{r_B} + \frac{1}{r_C} + \frac{1}{r_D}$$

time units. Figure 5.4.4 below illustrates the difference between the pre-scheduling and normal scheduling of these tasks.

With the proposed implementation of our design discipline, pre-scheduling can not be applied to tasks that receive messages from tasks derived from MP/SC processes. Recall that an MP/SC process can emit messages at a faster rate than the rate at which messages arrive on a given channel. Therefore, when a message is sent by a task corresponding to an MP/SC process, it is possible for the receiving task to have a nearer deadline than the sending task. In this case, if we attempt to schedule the receiving task early then the receiving task will be scheduled and will execute before the sending task. For this reason, the technique of pre-scheduling tasks only works for tasks whose processes receive messages from SP/SC processes.


Process A emits a message to process B.
Process B emits a message to process C.
Process C emits a message to process D.
T_A
T_B
T_C
T_D
Time
t
$t+\frac{1}{r_A}$
$t+\frac{1}{r_A}+\frac{1}{r_B}$
$t+\frac{1}{r_A}+\frac{1}{r_B}+\frac{1}{r_C}$
$t+\frac{1}{r_A}+\frac{1}{r_B}+\frac{1}{r_C}+\frac{1}{r_D}$

a) Task execution requests without pre-scheduling



b) Task execution requests with pre-scheduling

Figure 5.4.4: Execution sequences for tasks derived from Figure 5.4.3.

Pre-scheduling processes effectively reduces the worst case behavior of a series of interconnected SP/SC processes to the behavior of the last process in the sequence. Therefore, the effect of pre-scheduling on worst case message propagation delays along a path in a design graph, is to reduce the delay to only the delay incurred when passing through MP/SC processes and through the last process in the path. For a sequence of k processes, the maximum propagation delay for the message whose arrival at process P_1 causes a message to be emitted from process P_k will be at most

$$\sum_{\substack{i=1,\\ P_i \in MP}}^{k-1} \frac{1}{r_i} + \frac{1}{r_k}, \qquad (5.4.3)$$

time units, where MP is the set of MP/SC processes in a design graph.

For example, for the stopwatch system in Chapter 2 (see Figure 2.7.3), with pre-scheduling we can guarantee that in the worst case:

- the internal value of time differs from external time by less than p time units,

- displayed time differs from the internal value of time by less than $2p + c$ time units (no more than c plus one unit of stopwatch time).

- the time between the release of a button and an action of the display will be less then $2b$ time units.

where p is the period of the hardware timer, c is the maximum response time of the display, and b is the minimum duration between button presses. These bounds are significant improvements over the previous worst case figures of $2p$, $4p + c$, $3b + c$, respectively.

A final advantage of pre-scheduling is that it allows a designer to make liberal use of SP/SC processes. SP/SC processes can be either expanded into a sequence of processes or collapsed into a single process without incurring any penalty in terms of worst case message propagation delay. Applying this observation to the tasks that implement SP/SC processes we note that a sequence of SP/SC processes can be logically combined and then decomposed into only a single task. With fewer tasks, the actual implementation of a design graph is likely to be more efficient.

A prototype kernel that implements our message passing discipline with a sporadic tasking model is described in Appendix C. The kernel is non-preemptive and implements pre-scheduling.

5.4.2 Dedicated Processor Implementations

We next examine the utility of dedicating a processor to each process in the design graph. More precisely, we will dedicate a processor only to each process not in a mutual exclusion relation (either implicit or explicit). These processor will only have a single task execute on them. We will also dedicate a processor to each mutual exclusion relation. Since processes in a mutual exclusion relation can never execute simultaneously, there is no advantage to placing these processes on different processors. These processors will have more than one task executing on them and the tasks must be scheduled non-preemptively.

Consider the two SP/SC processes of Figure 5.4.1. Assume task T_A makes an execution request at time t_1 and then emits a message to task T_B at time t_2. If task T_B makes its execution request at time t_2, T_B runs the risk of executing too often. However, since task T_A executes on a dedicated processor, T_B is guaranteed that T_A will emit its message no later than time $t_1 + c_A$. (Recall from Chapter 3 that c_A is the computational cost for task T_A.) Therefore, (in the worst case) the earliest that T_B can make an execution request is c_A time units after an execution request of task T_A. In this example, task T_B will delay its request until at least time $t_1 + c_A$ as shown in Figure 5.4.5 below.

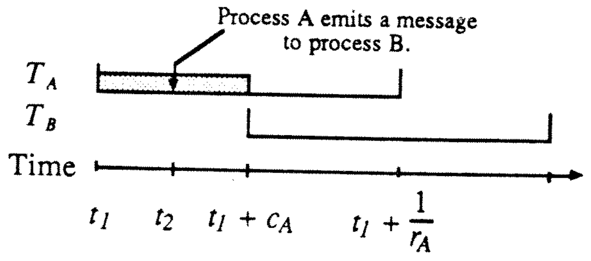


Figure 5.4.5: Task execution request time versus message emission time on dedicated processors.

As in the previous section, this observation can be applied transitively to a sequence of tasks derived from processes that receive messages from SP/SC processes. For the processes in Figure 5.4.3, the maximum propagation delay for a message whose arrival at process A causes a message to be emitted at process D is simply the sum of the costs of the processes in the sequence. This is illustrated in Figure 5.4.6.

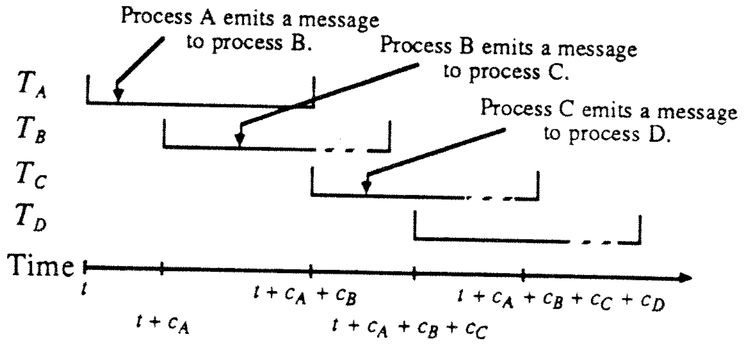

Process A emits a message to process B.
Process B emits a message to process C.
Process C emits a message to process D.
T_A
T_B
T_C
T_D
Time
t
$t + c_A$
$t + c_A + c_B$
$t + c_A + c_B + c_C$
$t + c_A + c_B + c_C + c_D$

Figure 5.4.6: Task execution requests for tasks of Figure 5.4.3.

We have the same problem with message emission from MP/SC process on dedicated processors as on a uniprocessor. That is, when a message is emitted from a task derived from an MP/SC process, the message may have to be buffered for up to $1/r_{in}$ time units where r_{in} is the logical input rate for the MP/SC process.

Dedicating processors to processes reduces the worst case behavior of a series of interconnected SP/SC processes to the sum of the costs of the tasks implementing the processes in the sequence. Therefore, for a sequence of k processes, the maximum propagation delay for the message whose arrival at process P_1 causes a message to be emitted from process P_k will be at most

$$\sum_{i=1}^{k} f(i)\,, \tag{5.4.4}$$

time units, where

$$f(x) = \begin{cases} c_x & \text{if } P_x \in SP \\ \frac{1}{r_x} & \text{if } P_x \in MP\ , \end{cases}$$

where MP and SP are the sets of MP/SC processes and SP/SC processes, respectively, in a design graph.

For a design graph that is viable on a uniprocessor (5.4.4) will never provide a worse (greater) bound than (5.4.3). In all cases (5.4.4) is strictly less than the original bound of Chapter 2 (5.4.2).

5.4.3 General Multiprocessor Implementations

Lastly we briefly examine the case where the n processes are mapped onto a set of $k < n$ processors. There are at least two interesting ways in which processes could be mapped onto processors. The first scheme would be geared towards ensuring viability. Processes would be mapped onto processors according solely to their resource requirements. In the worst case, the processes on a given path would be on separate processors. To ensure the tasks derived from these processes respected their periods, a message receiving task would have to wait until the end of the request interval of the sending task before the receiving task could make its execution request. Therefore, for a sequence of k processes, the maximum propagation delay for the message whose arrival at process P_l causes a message to be emitted from process P_k will be at most

$$\sum_{i=1}^{k} \frac{1}{r_i} ,$$

time units. This is exactly (5.4.2).

The second scheme would be oriented towards minimizing message propagation delays for critical paths in a design graph. In this case the message propagation delays will be a hybrid of (5.4.3) and (5.4.4).

5.5 Model Extensions

The previous sections have demonstrated one method for realizing a design graph: the decomposition into a set of sporadic tasks. Given this implementation, we can extend our design discipline in several dimensions without adding additional complexity to the assessment of temporal correctness. In this section we discuss some extensions to the design discipline. We are interested in exploiting properties of the cyclic tasking model to create a more expressive design discipline. We will examine the effect of relaxing some programming restrictions and adding a specification of minimum message transmission rates.

5.5.1 Some Programming Restrictions Revisited

In Section 2.3 we listed a number of restrictions on the composition of programs. The initial motivation for many of these restrictions was the desire to create as simple a programming model possible and yet still be able to construct useful programs. We are now in a position to evaluate some of these restrictions in terms of their role in the feasibility and viability analysis. By understanding the impact of a particular restriction on the feasibility analysis we would be able to trade off whatever expressive power the absence of the restriction was deemed to bring to the model versus its hindrance of the viability process.

When describing our programming discipline, we required the following three restrictions:

- processes and data repositories may not be combined in a mutual exclusion relation,
- data repositories may not emit any output messages, and
- each process and data repository could only appear in a single mutual exclusion relation.

Relaxing these restrictions could possibly make the discipline easier to use. For example, allowing processes and data repositories to be in a mutual exclusion relation together would allow for the construction of a more general form of abstract data type. Allowing data repositories to perform output would be useful for the specification of a hierarchical data type. However, these restrictions are in place to ensure that processes do not have arbitrary nested resource requirements. It was for this reason that we were able to claim that all design graphs will be deadlock free. Without these restrictions, designs can easily reach a deadlock state. For example, consider the design fragment in Figure 5.5.1 below in which a process and data repository are placed in a mutual exclusion relation.

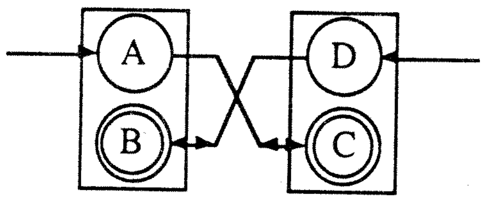


Figure 5.5.1: Processes and data repositories in a mutual exclusion relation.

Assume the following sequence of events occur. Process *A* receives a message and commences processing. Process *D* receives a message that gives process *D* a nearer deadline than process *A*. Process *D* preempts process *A* and commences processing its message. While processing, process *D* sends a message to data repository *B*. Since process *A* and data repository *B* are in a mutual exclusion relation and there exists an uncompleted message at process *A*, the processing of *D*'s message at data repository *B* cannot commence. Because of this, process *D* becomes blocked. Process *A* resumes execution and sends a message to data repository *C*. Since the processing of *A*'s message at data repository *D* cannot commence, process *A* becomes blocked as well. Since there is no action of the program that will ever cause either process *A* or process *D* to become unblocked, processes *A* and *D* are deadlocked.

A similar situation can arise if data repositories are allowed to emit messages as shown in Figure 5.5.2 below.

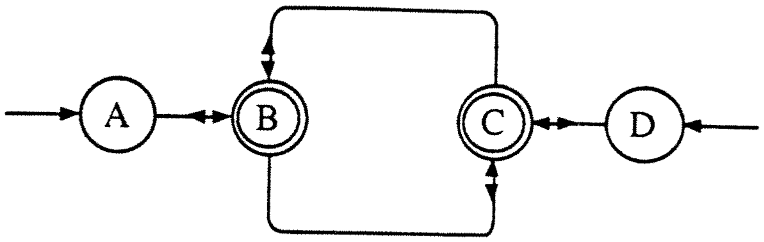


Figure 5.5.2: Data repositories with output.

In this case, the mutual exclusion requirement on processing of messages at shared data repositories can cause processes *A* and *D* to deadlock.

If deadlock is possible in the design discipline, then we would be unable to employ the design decomposition of Section 5.3.1.

However, these restrictions can be lifted if we adopt the implementation strategy from Chapter 3 that treats all physical resources as a single logical resource. In this strategy tasks that hold resources never block and hence can never become deadlocked. An implementation strategy with only a single logical resource is a non-preemptive strategy with respect to all the resource consuming tasks. Such a non-preemptive implementation strategy provides a form of syntactic sugar to the designer. All of the graphs above conceptually are being implemented as if all the processes were in a mutual exclusion relation.

An alternate view is that in effect, the non-preemptive implementation strategy is allowing us to realize the abstraction that messages are consumed in no time for reasoning about the logical behavior of designs. If our initial zero-time hypothesis were true, then none of the above examples would cause any problems. The viability decision procedure for the single logical resource implementation strategy gives us necessary and sufficient conditions under which we can realize the zero-time abstraction.

5.5.2 Minimum Transmission Rates

In Section 2.3 we remarked that in order to determine an upper bound on message propagation delays we needed to be assured of a minimum message transmission rate on all asynchronous channels. It should now be clear that such specifications could easily be added to our discipline. Requiring a minimum rate on channels is equivalent to requiring a maximum inter-execution request time for the tasks derived from a design. If a set of tasks derived from a design graph are viable, then they will remain viable no matter what maximum inter-execution request time is specified. Therefore, in principle, specifying minimum transmission rates has no bearing on our ability to realize a design.

If tasks have a specified maximum inter-execution request time, then each task will conceptually require a timer to be associated with the task. The timer will prod the task whenever it has been idle for too long. For example, the message from the timer could be an exception message since its emission likely implies that an error has occurred with the task's usual message producer. Although we do not provide for a specification of minimum transmission rates, a designer can achieve this effect by making the aforementioned scheme explicit and simply making each process for which a minimum rate is desired into an MP/SC process with a timer.

5.6 Summary

This chapter has shown how a design graph constructed using our design discipline can be implemented. There are two factors that determine whether a not a design graph can be realized. The first concerns the existence of cycles. We have demonstrated conditions on cycles that are necessary and sufficient for solving transmission rate functions on channels around a cycle. The second condition concerns the viability of a decomposition of a design graph into a set of sporadic tasks. If the tasks created in the decomposition are viable then the implementation will be temporally correct. All interconnected pairs of processes will adhere to the RT/PC paradigm. If the tasks are not viable then the implementation may or may not be temporally correct. In all cases the realizability and viability of a design graph can be determined efficiently. This demonstrates that it is possible to realize the RT/PC paradigm. In addition, our development of non-preemptive implementations argues that design graphs can be implemented efficiently.

We have also shown that there are significant advantages to using the implementation strategies of Chapter 4 to implement a design graph. We have shown that the pre-scheduling of message emitted from tasks derived from SP/SC processes can eliminate the worst case delay incurred when a message passes through an SP/SC process. This means that in terms of the temporal behavior of a design, SP/SC processes can be liberally used in a design with no added delay in message propagation times.

Chapter 6

Designing Real-Time Systems: Some Experiments

6.1 Introduction

In this chapter we discuss an implementation of our design discipline and present paper and real designs for some actual real-time systems. These exercises will serve three purposes. First, they will demonstrate that our design discipline is expressive enough to describe the types of synchronization and timing constraints found in actual real-time systems. Secondly, they will demonstrate some tangible benefits of using the discipline to design real-time systems. By adopting our discipline, a designer will be able to derive interesting and important real-time properties of the system being constructed. Lastly, they will provide evidence that our discipline is practical. They will show that the scheduling policies we have investigated can be implemented efficiently and effectively.

In Section 6.2 we discuss some of our experiences with an implementation of a prototype programming system to support the abstractions of the design discipline. Sections 6.3 - 6.6 present and discuss some designs for actual real-time systems. We will examine four real-time systems:

- a digital stopwatch,
- a computer music system,
- a control program for a six legged walking robot, and

- a naval communications subsystem.

Each system has been chosen to either illustrate the use of our discipline to express a particular class of timing constraints or to illustrate a particular programming style. Each system corresponds to an actual operational real-time system. The stopwatch will be used to illustrate how the concept of *modes* of a system's operation, and *mode changes*, can be incorporated into the design discipline. Modes play an important role in both the structuring and viability analysis of designs. The watch will also serve as a vehicle for presenting a solution to the problem of managing input devices that emit messages at either an erratic or unbounded rate.

The computer music system was selected for its emphasis on output timing constraints. Throughout this dissertation we have concentrated on predictable responses to inputs. There is no explicit mechanism for expressing timing constraints on output in our discipline. We will present a design for a small computer music system to illustrate how timing constraints on outputs can be reduced to timing constraints on inputs. This will demonstrate that the discipline is sufficient for expressing and reasoning about real-time input and output interactions. The design we present is based on a system constructed by Maloney [Maloney 89]. Descriptions of similar systems have been reported separately by Bloch and Dannenberg and by Vercoe [Bloch & Dannenberg 85, Vercoe 84].

The walking robot control program is used to illustrate that our design discipline can be applied to a real system with a tangible benefit. Based on an examination of the actual control program, we propose a design for a significant subsystem of the program. By using our discipline we are able to provide important guarantees of performance that were desired, but not achievable, in the original implementation. The original design and implementation of the walking program was reported by Donner [Donner 84].

The final system we consider is a naval communications subsystem. The naval system will be used to compare our discipline to the design methodology in the SARTOR development environment [Mok 85]. The SARTOR project is concerned with the development of efficient and reliable real-time software. The design we present for the communication subsystem is based on the design reported in [Mok et al. 87, Mok et al. 88].

For each of these systems we will present a hypothetical design using the design discipline of Chapter 2. The design will then be analyzed to assess (1) the elegance of the discipline,

(2) the appropriateness of the semantics of message passing, and (3) the implications of a viability assessment for a design. Where the design discipline exhibits shortcomings, we will propose and discuss extensions to the discipline. Each extension will be considered from the standpoint of added expressiveness as well as its impact on our ability to realize a design.

6.2 Implementation Experiences

To demonstrate the practicality of our design discipline, we implemented a prototype programming system to support the abstractions of the discipline. The implementation consisted of a small kernel which implemented processes, message passing communications, and earliest deadline first scheduling with pre-scheduling. Appendix C provides a more detailed description of the implementation. In this section we describe our experiences with the implementation.

The stopwatch application presented in Chapter 2 was implemented according to the design graph given in Figure 2.7.3. In actual fact the design graph in Figure 2.7.3 was the result of several experiments with the stopwatch application and the prototype kernel. The experiences and observations reported here are based on several experimental designs for the stopwatch.

6.2.1 Benefits of Non-Preemptive Scheduling

Chapter 4 has described several implementation strategies for our design discipline. We experimented with actual implementations of two of these strategies: an implementation with a single logical resource and an implementation with a two logical resources. The first scheme corresponds to a completely non-preemption system. This is the implementation described in Appendix C. The second scheme corresponds to a system in which preemption is allowed between non-resource consuming phases of tasks and resource consuming phases of tasks.

Our experiences with these two implementations has convinced us that the completely non-preemptive implementation is superior to any others that allow preemption. Although such an implementation is more restrictive in terms of the number of task sets that can be realized under the implementation, we feel that the benefits of non-preemption in terms of ease of system construction and low overhead costs greatly outweigh the loss of generality. Section 5.5.1 has already commented on some extensions to the discipline that are possible through the adoption of non-preemptive scheduling. Some additional examples of the benefits of non-preemption are enumerated below.

Implementation of Tasks

If tasks are not allowed to preempt one another then they need not be implemented as traditional tasks. That is, tasks need not be treated as separate programs with individual program counters and stacks. Since tasks will always execute to completion without preemption, they may be implemented as procedures that are simply invoked via a procedure call whenever the corresponding task is dispatched.

There is a significant savings in overhead costs for adopting this strategy. In the implementation described in Appendix C, a null procedure call (call with no parameters and immediate return with no returned result) took approximately 14 microseconds to complete. (See Table C.2.1 in Appendix C.) Tasks were implemented as separate threads of control in an address space. The operation of dispatching a task took approximately 188 microseconds. This was simply the time to transfer control to the task. By implementing tasks as procedures we can "dispatch" tasks an order of magnitude faster. In addition, the design of the run-time system would be greatly simplified.

Implementation of Data Repositories and Synchronous Channels

With a non-preemptive scheduler, once a task commences execution it is guaranteed that it will have exclusive access to any data repositories that it requires. Therefore no access mechanism for MP/SC data repositories need be implemented. Our prototype implementation implements data repositories as ordinary procedures. With this scheme there is no need to implement synchronous channels. The emission of a synchronous message is simply a procedure call. Non-preemptive scheduling again greatly simplifies the design of the run-time system.

Buffering Outputs From MP/SC Tasks

In Section 5.3.3 we mentioned that the outputs of tasks created from an MP/SC process would sometimes have to be buffered to ensure a minimum message inter-arrival time on the MP/SC process's output channels. In that section, it was suggested that an additional task be created to handle this buffering chore. If preemption is not allowed then the creation of this additional task is not necessary. A scheduler can simply check after the completion of each execution request to see if any buffered messages can be released.

Polling for Interrupts

The implementation of the design discipline described in Appendix C creates an interrupt handler for each input device. Each interrupt handler is treated as an ordinary task. If the set of tasks derived from a design graph are viable under a non-preemptive implementation strategy, then interrupts need not be serviced when they arrive. In particular, an implementation of a design may execute tasks with interrupts disabled and only poll for interrupts when making scheduling decisions (when a task completes execution). In this manner, the cost of servicing interrupts is reduced significantly. The reduction is due to the elimination of a preemption due to the arrival of the interrupt. If a task set is viable then we are guaranteed that each interrupt will be serviced before the next interrupt from the same source arrives. Therefore we can guarantee that interrupts can not be lost.

Assessing Viability

The cumulative effect of the benefits described above is that the assessment of viability is both simplified and more credible. To assess viability one must know the computational cost of the tasks. In practice, this cost must include the overhead of such operations as scheduling and interrupt processing. Since in each cases above, non-preemptive scheduling either eliminates or greatly simplifies the operation of the implementation of the discipline, it correspondingly simplifies the determination of a task's worst case cost. The overhead cost of operations such as interrupt processing are often the most difficult parameters to calculate. By eliminating or simplifying these operations we make the viability analysis more credible since we can be more confident about the quality of the data going into the procedure.

6.2.2 Software Engineering Issues

When processes were defined in Chapter 2, we presented the abstraction that each process received messages from a single source. This means that conceptually a process does not know if it is an MP/SC or SP/SC process. The prototype implementation has adopted this abstraction of a single input source per process. This has aided in the use of the discipline.

The implementation of processes described in Appendix C allowed for the separate compilation of tasks. This was achieved by adopting the single input abstraction and by developing a scheme for the dynamic binding of tasks to channels. The result was that it was easy to manipulate a design graph and experiment with different configurations of processes. Changing a graph only required the recompilation of a program with a number of statements proportional to the number of vertices and edges in the graph.

6.2.3 Conclusions

The primary conclusion we draw from these experiences is that the benefits of a completely non-preemptive implementation out weight its drawbacks as they are currently perceived. The primary drawback is the more restrictive nature of the correctness conditions for a non-preemptive implementation. It is not known how serious a drawback this will be in practice. It should be noted however, that since viability is a function of the computational cost of tasks, and since this cost must account for the overhead of the scheduling and synchronization primitives, a completely non-preemptive implementation is likely to yield a more favorable viability analysis for task sets with small computational demands than a more sophisticated strategy.

6.3 A Digital Stopwatch

The stopwatch system described in Chapter 2 will be used to illustrate how one can design and reason about a system with *mode changes* and unbounded input rates.

Real-time systems often have physical components such as radars, displays, or signal processing hardware, that can be used in a variety of modes. For example, the display in the stopwatch had two basic modes of operation. When the watch was running it could either display the continuously updating value of elapsed time or it could display the so-called split time. It is important to be able to express the operating modes of a system, or subsystem, for three reasons. First, making the modes of a system explicit would aid in the presentation and logical analysis of a design. Second, it would also aid in the construction of designs as it provides a means for structuring and modularizing designs.

Lastly, it is important to isolate the components of the system's modes to improve the quality of the assessment of temporal correctness of a design. Modes in a system typically imply the existence of mutually exclusive functions [Hood & Grover 86]. For example, in the stopwatch system, the Flasher and Stop Watch processes will never send messages to the Display Driver process at the same time. If we could explicitly represent this fact then when assessing the temporal correctness of a design we would know that these processes could never interfere with one another.

We will examine two issues regarding modes and mode changes:

- how can the modes of a system be made explicit in a design, and
- for the implementation strategies of Chapter 4, how can the viability decision procedures be modified to incorporate modes and mode changes.

The stopwatch will also serve as an example of a system with a potentially unbounded input rate. A fundamental premise of our design discipline is that the worst case maximum rate at which inputs enter the system is known. In the event that these rates are unknown, we suggested in Chapter 5 that a viability decision procedure could be used to derive maximum allowable inputs. This technique by itself is insufficient for assessing the temporal correctness of the design since we cannot be guaranteed that inputs will not physically arrive at a higher rate than the one derived for them. We will use the button driver process of the stopwatch system to illustrate a simple buffering and polling technique that will ensure that artificially derived rates are not violated.

6.3.1 Expressing System Modes

Currently, the existence of modes cannot be inferred from a design graph. For example, given the current design graph for the stopwatch (see Figure 2.7.3), it is not possible to determine that the Flasher and Stop Watch processes will never emit messages simultaneously. While we can talk about mutual exclusion at the level of processing a single message, what is needed is a mechanism for describing mutually exclusive sub-graphs in a design graph.

Such a mechanism could be provided with the introduction of a new program component called a *switch*. A switch controls the flow of messages on a set of channels that pass through the switch. Graphically, a switch could be represented with a transparent box placed over a set of channels. For example, for the stopwatch system, a switch could be employed as shown in Figure 6.3.1 below.

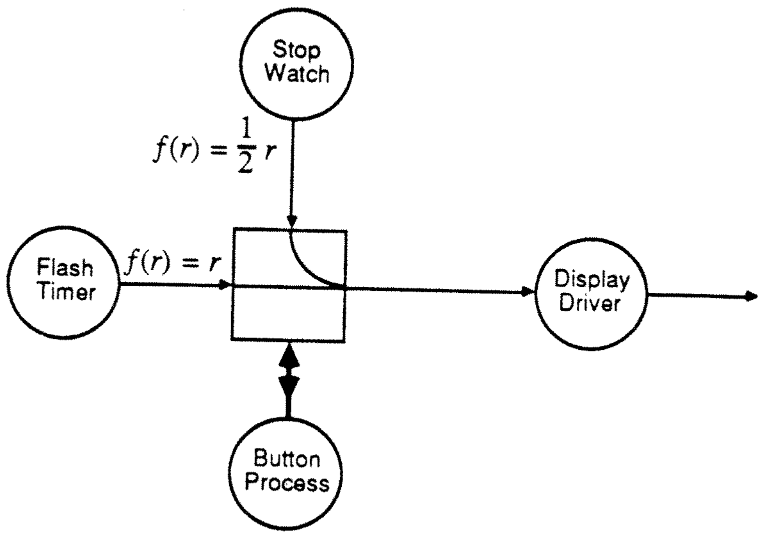


Figure 6.3.1: A design graph for the stopwatch with a switch.

A switch allows only a subset of the channel connections shown in the switch be active simultaneously. The subset of active channel connections is referred to as the *switch setting*. The switch itself behaves similar to a data repository since it maintains state information (namely the current switch setting) between mode changes. In the design above, the switch replaces the Watch State data repository, thereby making the control functions of the data repository more explicit. The use of a switch in this design would denote that messages to the Display Driver process come either from the Flash Timer process or from the Stop Watch process. The choice of which path is active is made by sending a synchronous message to the switch. In this case the process that initiates mode changes is the Button process.

From the standpoint of message receivers, the semantics of message passing are uneffected by the imposition of a switch. The Display Driver process will still consume messages from both the Flash Timer and Stop Watch processes according to the RT/PC paradigm. For processes with output channels that pass through a switch, the semantics of sending a message will now be a function of the mode of the system. In the design above, the Display Driver process has been reduced from an MP/SC process to an SP/SC process.

The actual rate at which it consumes messages, and hence the rates at which it emits messages, will depend on the mode of the system. For a given mode, the rate at which messages are transmitted on channels leaving a switch will simply be the sum of the rates at which messages arrive at that channel for the mode's switch setting. Since a switch is conceptually a data repository, it will consume a message (perform a mode change) according to the rate at which the sending process consumes messages (see Section 2.4.5).

Note that switches could be employed on synchronous channels as well as asynchronous channels. In the case of synchronous channels, the process that initiates mode changes would have its synchronous channel graphically distinguished to avoid confusion with the synchronous channels that the switch controls.

6.3.2 Modes, Mode Changes, and Temporal Correctness

We define a *mode* as a sub-graph whose sources are channels passing through a single switch. A *mode change* is the emission of a message from a process to a switch that causes a change in the switch settings. Since a switch is logically a data repository (from the point of view of its message producers), it will respond to mode change messages according to the semantics of message passing on a synchronous channel (see Chapter 2, Section 2.5.3). We will distinguish between two types of mode changes: *preempt/resume* and *termination* [Locke 88].

Conceptually, a mode change acts like a hardware interrupt on the processes in the currently active mode. When a preempt/resume mode change is made, the processing of all messages in the currently enabled mode is suspended. When the suspended mode is reentered, the processing of the suspended messages is resumed. Under a termination mode change, the processing of all messages in the sub-graph corresponding to the current mode are aborted. The type of mode change desired could either be hard-wired into a switch or could be specified as a parameter in the synchronous message sent to the switch.

To assess the temporal correctness of design graphs with modes we can employ the design decomposition of Chapter 5 and the viability decision procedures of Chapter 4. As we demonstrated in Chapter 5, the results of these decision procedures are sufficient but not necessary conditions for the temporal correctness of the decomposition of a design graph. The specification of modes allows us to improve upon this situation significantly.

The problem of assessing the temporal correctness of design graphs with modes, represents a tractable special case of assessing the viability of sporadic tasks with dependent execution requests. Recall that one reason why viability was not a necessary condition for temporal correctness was the fact that certain processes in a graph could be shown to never receive or consume messages simultaneously. Incorporating this information into the viability analysis would be complex. It would require an identification of processes in the subgraphs rooted at these non-interfering processes, that could and could not interfere with one another. Unlike these sub-graphs, messages can never be processed in two modes simultaneously. Therefore, when assessing the temporal correctness of a design with modes, it is sufficient to only consider the viability of the design with each mode in isolation. If a design is viable for all modes then its implementation with the strategies of Chapter 5 will be temporally correct. For our proposed decomposition of design graphs into sporadic tasks, viability will be a necessary condition for the temporal correctness of graphs with disjoint modes. For these graphs viability is equivalent to temporal correctness. This means that if the decomposition of a design is not viable, then either an alternate decomposition must be used or the graph must be redesigned.

A more tedious aspect of the assessment of temporal correctness is the determination of the cost of performing a mode change. Since switches are modeled after data repositories, the cost of a mode change is the cost of sending a message to a switch plus the cost of performing the mode change. Depending on the type of mode change, this will either be the cost of preempting, resuming, or aborting, whatever outstanding messages exist in the currently enabled mode. In either case, the cost will be proportional to the number of processes in the mode (the number of processes in the appropriate sub-graph emanating from the switch).

6.3.3 Buffering and Polling Erratic Inputs

If a worst case minimum separation of inputs cannot be enforced by the external environment then inputs must be buffered and polled at the desired rate. For example, since there does not exist a minimum separation between button presses (or its value is too small to be of practical use), the presses may be buffered and sampled as shown in Figure 6.3.2 below. The replacement of the button process's ill-behaved input source with a well defined input source ensures that the RT/PC paradigm is logically obeyed between the button device and the button process. Note that we are only ameliorating the problem of unbounded input rates. If the Watch Buttons input device sustains an input rate greater than $1/b$, then data will be lost. For a given number of buffers, one can determine conditions for the input process under which data will not be lost.

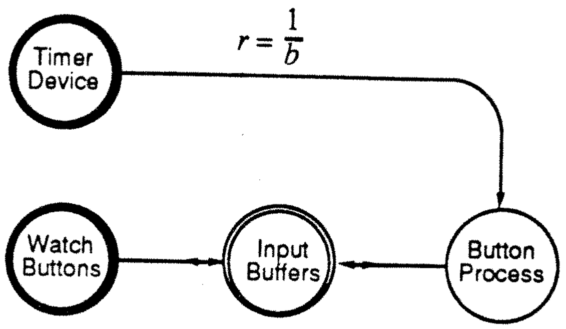


Figure 6.3.2: Buffering button presses.

This buffering technique is similar to the technique mentioned in Section 4.3.5 to re-engineer non-viable designs. However, in this case the buffering must be performed by the input hardware. If the buffering were performed in software then the problem of unbounded input rates has just been pushed into a different portion of the design.

6.4 A Computer Music System

The entire emphasis in this thesis has been on the study of timing constraints related to the processing of inputs. In this section we will use a real-time computer music system to demonstrate that it is sufficient to concentrate solely on input constraints. We will show that timing constraints on outputs necessarily map into constraints on inputs.

6.4.1 The Synthesis of Music as a Real-Time Process

Loosely speaking, a computer music system is a computer system that generates music based on a set of inputs. The inputs can range from a textual description of a piece of music, to a sequence of codes representing the actions of a performer on a musical instrument such as a keyboard instrument. The output of the system is typically a sequence of commands to a synthesizer or group of synthesizers that actually produce the audible notes. Alternately, the outputs of the system could be a textual representation of music such as sheet music. We will focus on the former type of music system.

On the surface, a computer-driven music synthesis system does not embody many of the characteristics of a hard-real-time system such as a high monetary cost of failure, or an abundance of physical and logical concurrency. However, in its essence, a computer music system is a pure hard-real-time system. Music is defined completely in terms of time. Notes are expressed in terms of frequency, amplitude, and duration. The correctness of a computer music system is fundamentally a function of time. If, in the generation of audible music, a single note is begun too early or too late then the resulting product is considered erroneous music. The music produced in this case does not correspond to the music specified by the score. Therefore, for a music synthesis system, there is a high esthetic cost of failure associated with not operating in real-time.

We are interested in studying a very simple computer music system. We consider a system that reads a piece of music, processes the score, and then controls a set of synthesizers in real-time to perform the music [Maloney 89]. Such a system could be used in the development of musical arrangements, especially those with multiple parts. For concreteness, we will assume that the computer music system controls its synthesizers through a MIDI[1] interface. (The actual interface is immaterial.) The MIDI interface provides primitive operations of *turn on note*, *turn off note*. The computer music system has timing constraints on output since notes must be turned on and off at precise times. This computer music system is unique in that all of the real-time aspects of the system are related solely to the output process. Although there are no facilities for specifying output constraints in our discipline, we can achieve the desired effect by binding output constrained events to constraints on a stream of inputs.

6.4.2 Mapping Output Timing Constraints to Input Timing Constraints

Abstractly, an output constraint specifies that an output operation be performed during a particular interval of real-time. The endpoints of the interval may be specified relative to the occurrence of other events in the system or in the external environment. In order to ensure that an output constraint is adhered to, the system must be able to measure the passage of time in the units of time in which the constraint is specified. For example, if a computer music system must turn on a note on the sixth beat of a measure, the system must be able to

[1] Musical Instrument Digital Interface.

measure the passage of beats. The system need not measure beats directly but it must have available some reference stream of inputs from which it can accurately infer the passage of beats in real-time. A constraint on output can therefore be mapped into a constraint on the processing of the input reference stream.

The implementation of the computer music system consists of two separate phases. In the first phase, the input score is compiled into an ordered list of time-stamped commands for the output synthesizers. The second phase consists of the sequencing of this list in real-time. A design for a prototype of the second phase of the system is given in Figure 6.4.1 below.

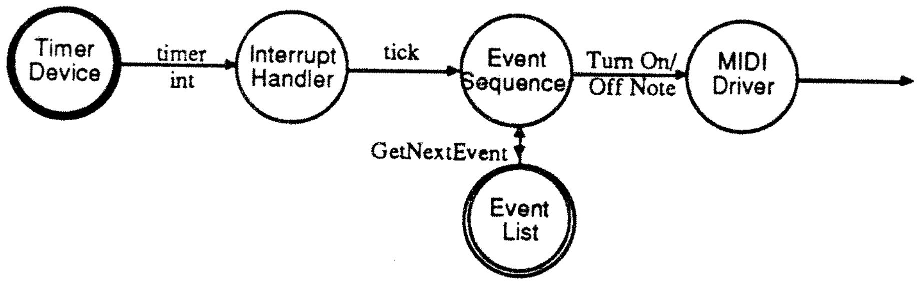


Figure 6.4.1: A prototype computer music system.

An external timer device serves as a metronome for the system. It delivers interrupts at a rate that is sufficient for accurately keeping time in the piece of music. The Event Sequencer process receives tick messages at this rate. The sequencer maintains time and schedules events on the synthesizers based on an event list. All transmission rate functions are the identity function. Using the timing analysis techniques of the previous chapters, we can provide upper bounds on the lag between the tick of the metronome and the emission of a message from the MIDI driver. We can also specify performance constraints on the MIDI interface. With these bounds the adherence of the system to the output constraints can readily be assessed.

6.5 A Six Leg Walking Robot

In his dissertation, Donner described the design and implementation of a control system for a six legged, 1600 pound, walking machine [Donner 84]. The purpose of the control program was to synchronize the actions of the legs in such a manner that the machine was stable (did not tip over) and would make forward progress. The structure of the walking program consisted of a high-level process for controlling each leg plus a suite of processes for sensor monitoring, data logging, and exception handling. The majority of the code was concentrated in the leg process that was instantiated for each leg. We will give an alternate design for the leg process.

6.5.1 Existing Program Design

The behavior of each leg can be described with a four state finite state machine. When a leg was is contact with the ground, and bearing a portion of the load of the machine, the leg is in the *load* state. From this state the robot can be moved forward by driving the leg backward. During this motion the leg is in the *drive* state. When the leg has reached the end of its range of travel, it must be picked up and returned to its forward-most position. When the leg is being lifted it is in the *unload* state and when it is being returned to its starting position it is in the *recover* state. The simple finite state machine for a leg is shown in Figure 6.5.1 below.

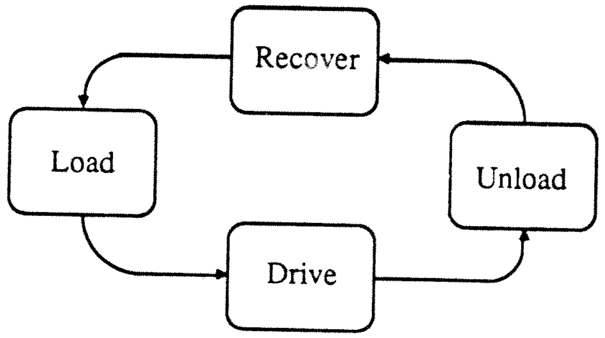


Figure 6.5.1: Finite state machine for the high-level leg process.
(From [Donner 84].)

Each leg is able to communicate with the legs adjacent to it on the same side of the machine, and the one leg on the opposite side of the machine as shown below in Figure 6.5.2. There are two types of communications between legs: *inhibition* and *excitation*. The purpose of inhibition is to maintain the stability of the machine. Each leg maintains a value representing its level of inhibition. If this value is below a threshold the leg will not make a transition from either the drive to the unload state, or from the unload to the recover state. If a leg is in the recover state and not bearing any load, then it will inhibit a transition from the drive to the unload state of its neighbors. Inhibition consists of subtracting an amount from the inhibition levels of a leg's neighbors. Excitation is concerned with even forward progress of the machine. The goal of excitation is to establish rear-to-front waves of leg recoveries. When a leg completes a recovery it will add a value to the inhibition level of its forward neighbor. For each leg, the condition for determining when a state change can occur is a function of the states of the leg's neighbors.

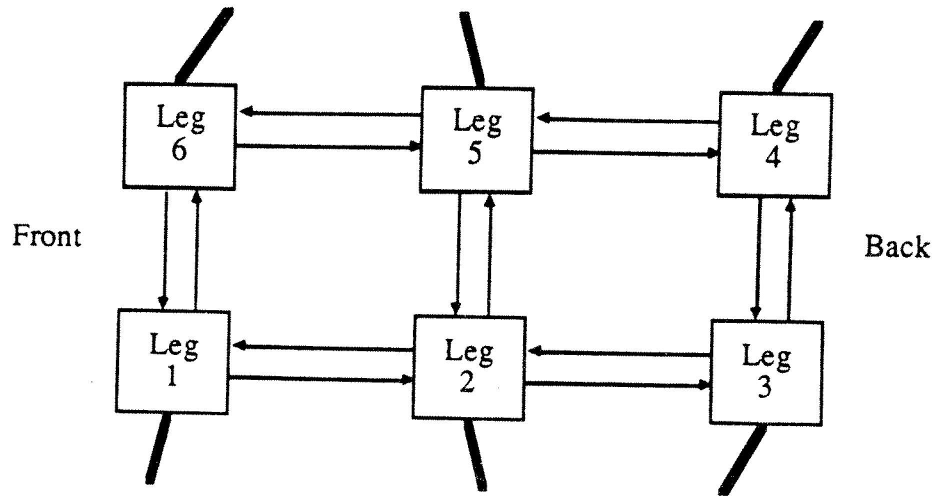


Figure 6.5.2: Communication paths between legs. Desired direction of travel is to the left. (Adapted from [Donner 84].)

The existing leg process consists of a main loop that sequences the leg through each of the four states. Specifically, the loop contains the following major steps; each executed in succession:

- Load: attempts to place the leg in contact with the ground until a threshold force is reached.
- Excite and Inhibit: after the leg is loaded the fore (front), aft (back), and across neighbors are inhibited, and the aft neighbor is excited.
- Drive: the leg is driven backward. If the leg's inhibition is too small then it will wait until its threshold is reached.
- Inhibit: fore, aft, and across neighbors are inhibited.
- Unload: the leg is raised from the ground.
- Recover: the leg is placed back in its forward-most position.

The leg process receives no direct inputs or interrupts from the robot's sensors. All communication with the external world is via a segment of memory that is mapped into the address space of an auxiliary processor that controls the actuators and monitors the sensors.

There are no explicit timing constraints for the leg process, however, the real-time behavior of the process is critically important. Each operation that physically effects the leg consists of a loop that checks the value of a force sensor for the leg and then commands an actuator appropriately. The implementation of the walking program would typically preempt the leg process after each iteration of this inner loop. There was no mechanism for ensuring that a preempted loop will be resumed with a specified bound. This would be desirable since this bound would limit the duration a problem with an actuator could remain undetected. Using ad hoc techniques, the system designer was able to convince himself that the worst case time a loop could be delayed between iterations was approximately 6 ms. Since sensors were sampled by the auxiliary processor every 10 ms., this latency was deemed acceptable [Donner 89].

6.5.2 Proposed Design

Below is a proposed new design for the leg process. Each of the six major steps above is a process that is driven by a timer. The processing associated with each step has been constructed as a mode. Activation of each step constitutes a mode. At the completion of each step the switch is set for the process corresponding to the next step. The sensor process executes on a separate processor.

Upon receipt of a message from the timer, each of the four processes that physically manipulate the leg (*load*, *drive*, *unload*, *recover*) will check the appropriate force(s) on the leg and emit a message to an actuator for a joint on the leg. The purpose of the timer is to bound the duration between iterations of each process. By adopting our discipline, we can guarantee worst case maximum latencies to each control loop in the original program. In principle, the maximum rates at which each process can be activated can be computed using the viability decision procedures in Chapter 4. Our design is presented in Figure 6.5.3. The leg process communicates with other leg processes (other instances of Figure 6.5.3) through inhibition and excitation data repositories.

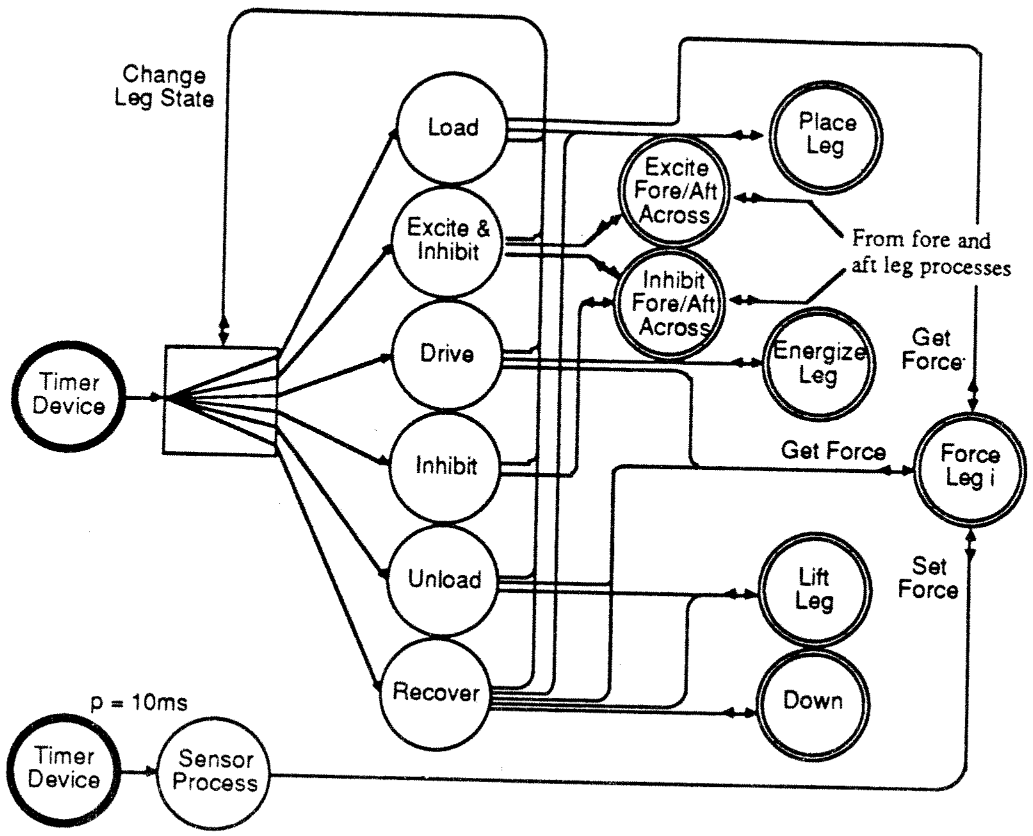


Figure 6.5.3: Partial design graph for the leg process.

6.6 A Naval Communications Subsystem

Mok has described a redesign of a sanitized version of an naval communication sub-system originally developed for the US Navy by the Lockheed Missiles and Space Company [Mok et al. 87, Mok et al. 88]. The subsystem, termed the Link-11 preprocessor, is one component of a tactical information communication system. We will give an alternate design for the Link-11 preprocessor.

The Link-11 design that we have studied was developed with the SARTOR system [Mok 85]. The computational model underlying SARTOR is a directed, acyclic graph of input and computation nodes. Nodes communicate with one another by sending and receiving messages according to a set of rules. For an arbitrary graph, sufficient conditions exist for ensuring that all messages sent are received.

6.6.1 Existing Link-11 Design

The Link-11 preprocessor monitors transmissions on the Navy Tactical Data System (NTDS) network via a passive tap. The preprocessor is responsible for the filtering, quality assurance, correlation, and formatting of data related to the tracking of vessels. The preprocessor maintains a database of tracks. It sends portions of its database periodically to a Data Management Processor (DMP) that maintains the master database of track files. A high-level view of the communication system architecture is shown in Figure 6.6.1 below.

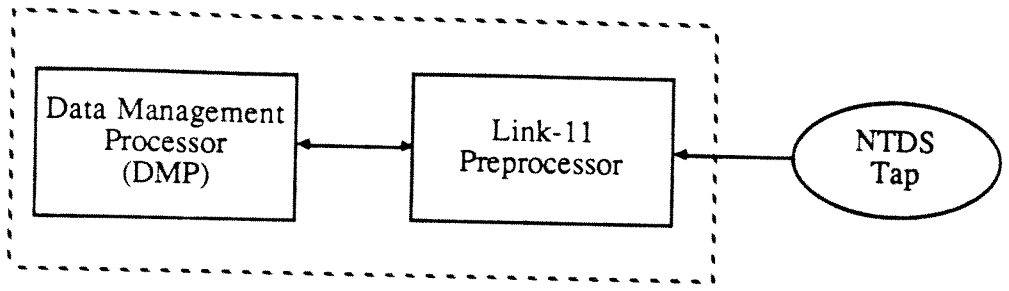


Figure 6.6.1: Naval data display system (from [Mok et al. 87]).

The design presented by Mok consists of the six "processing threads" (connected components) shown in the figure below. In their notation circles represent input/output nodes, squares represent computation nodes, and cylinders represent database operations. Input nodes are executed periodically according to a specified period. Computation and database operation nodes are also executed periodically with a period equal to the smallest period of the nodes from which it receives a message. The design below represents 3000 lines of C code.

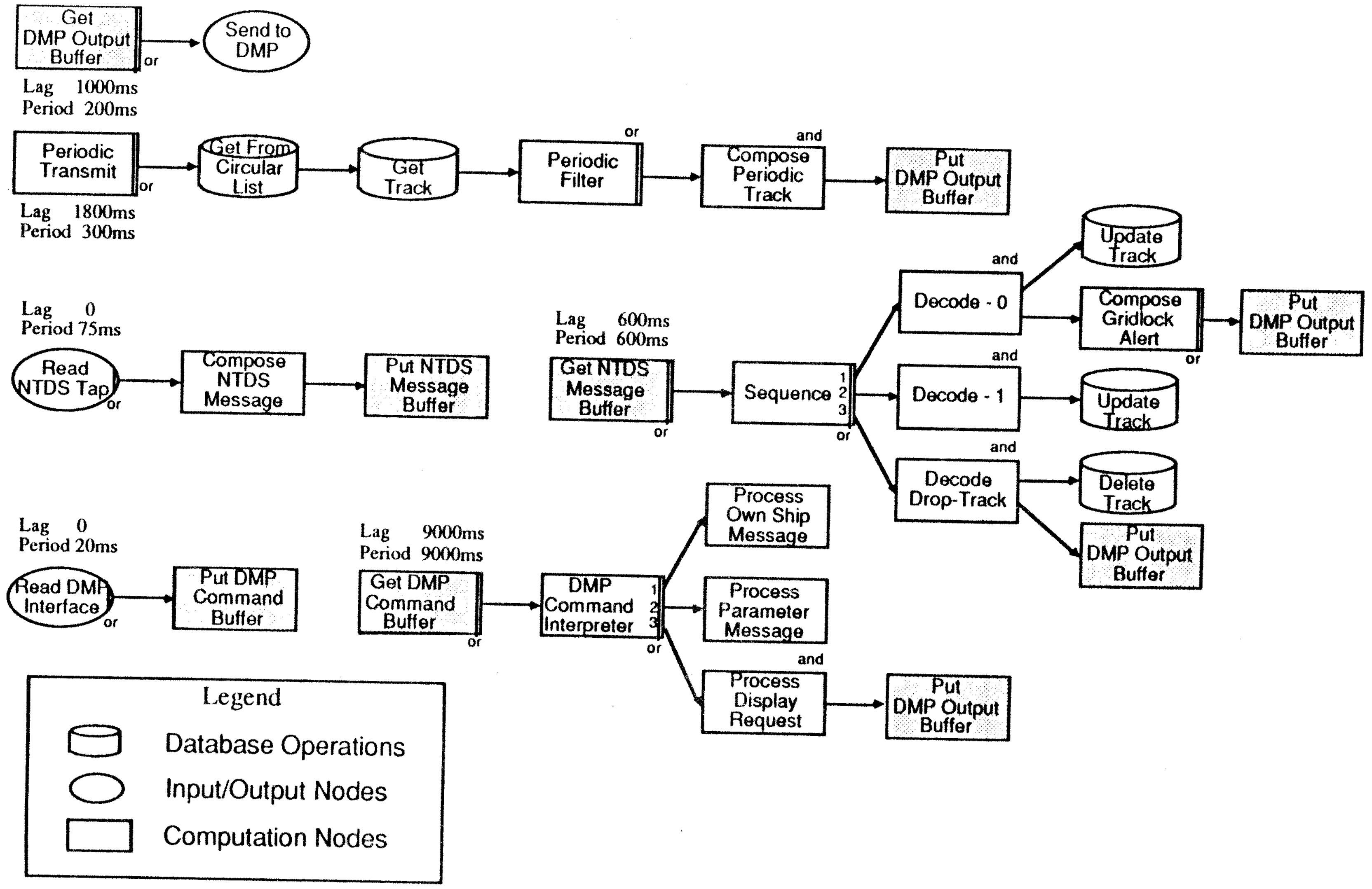


Figure 6.6.2: SARTOR Link-11 design (from [Mok et al. 88]).
Shaded computation nodes are our emphasis.

The source node for each processing thread is labelled with the period at which the thread is activated. The parameter "lag" refers to the initial release time.

For purposes of comparison with our discipline, we are primarily interested in the structure and not the content of this design. For a more complete explanation of the functions of each node above, the reader is referred to the original reference ([Mok et al. 87]).

6.6.2 Comparison

There are three important differences between our discipline and the SARTOR method. The first is our use of the transmission rate formalism. In our discipline, a process's behavior is completely, and automatically, specified by the rates at which messages arrive on its input channel. By describing interactions in terms of transmission rates, we are able to develop and analyze cyclic designs (those that meet the conditions specified in Chapter 5). In SARTOR, if tasks compute at different rates then buffers must be inserted in between them. For example, in the design above, three sets of buffers are needed. With our discipline there is no need for these buffers. The forced imposition of buffering in SARTOR clouds the behavior of the resulting design. While SARTOR can ensure that all messages are received, when buffers are imposed between design components, the utility of this feature is diminished. For example, in the SARTOR design above, since the PERIODIC TRANSMIT, GET NTDS MESSAGE BUFFER, and GET DMP COMMAND BUFFER processing threads all buffer data items in the GET DMP BUFFER buffer, after a computable amount of time, the DMP output buffers will be overwritten and data will be lost.[2] This is because DMP buffer items are not consumed at the rate at which they are produced. From Figure 5.5.2, we see that DMP output buffers are produced at a rate of

$$\frac{1}{9000} + \frac{1}{600} + \frac{1}{300} \text{ ms.}$$

which is greater than the rate at which DMP output buffers are consumed (1/200 ms).

The second difference is the sophistication of the underlying formal model. A SARTOR graph represents a set of periodic tasks. A design in our discipline represents a set of

[2] However it is quite possible that this was the intent of the designers.

sporadic tasks with shared resources. This means we can directly develop designs for systems with event driven processing requirements. Without sporadic tasks, polling must be used to respond to sporadic events. Both disciplines use variants of deadline scheduling. However, we have demonstrated implementation strategies that are optimal with respect to strategies that schedule without inserted idle time. We have also demonstrated that our strategies need not sacrifice processor utilization to ensure viability. The scheme for realizing SARTOR designs imposes an upper bound (strictly less than 1) on the achievable processor utilization. This bound is a function of the number of tasks in the system and the size of their periods. This is not true of our discipline. Therefore, any SARTOR design that is viable under their proposed implementation strategy will be viable under ours. The converse is not true.

A final point is that SARTOR has no mechanism for explicitly representing critical sections or shared data.

6.6.3 Proposed Design

Using our discipline, we have constructed the design shown in Figure 6.6.3 below. This construction is based solely upon the description of the Link-11 system reported in [Mok et al. 87]. We did not have the privilege of examining the actual code of the system.

Our design deviates from the original in that we have eliminated all use of buffering. (The shaded nodes in Figure 6.6.2.) In place of the buffers, we have imposed an asynchronous channel with an appropriate transmission rate function (derived from the textual description of the processes in the SARTOR design). By eliminating buffering, we are able to provide more meaningful guarantees concerning the loss of data in the system. Our processes are guaranteed to consume messages at the rate they are sent. Therefore, if the input devices respect their input rates, then no data will be lost.

At first glance, our design may appear to be more complex than the design in Figure 6.6.2. However, note that the SARTOR design is presented with several nodes duplicated (e.g., the "Put DMP Output Buffer" node). Had we chosen to adopt a similar convention our designs would likely be more digestible.

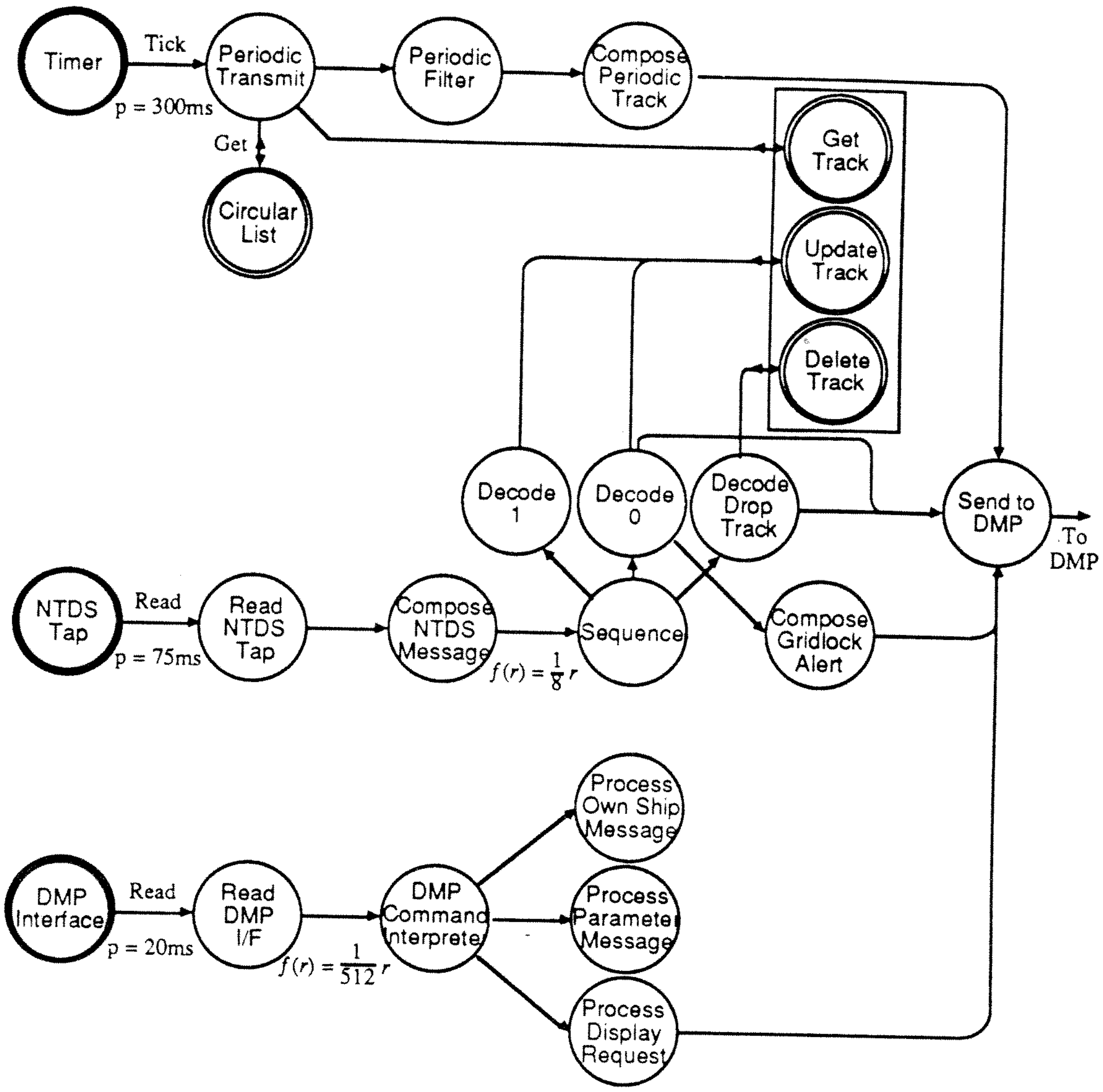


Figure 6.6.3: Proposed Link-11 design.

6.7 Summary

This chapter has examined the applicability of our design discipline by examining an implementation of the discipline and by constructing designs for four systems. This has demonstrated both that the discipline is practical, that it is expressive enough to design actual systems, and that the semantics of message passing lead to a meaningful analysis of the design.

Based on our experiences with a prototype programming system based on the design discipline, we conclude that there are significant benefits to the adoption of non-preemptive scheduling.

We have demonstrated that the discipline can be easily extended to explicitly include the concept of operating modes of a system. The *switch* component was introduced for this purpose. This extension significantly extends the discipline in two dimensions. First it provides a mechanism for modularizing designs, and secondly, it circumvents a portion of the problem of determining the feasibility of graphs with dependent output streams. Note that with the initial abstract abstraction of infinitely fast message consumers, the addition of a switch to our discipline does not add any expressive power to the discipline. If all consumers are are infinitely fast then a switch in a design can always by replaced by a data repository. The real benefit to the switch occurs in practice where consumers actually take time to consume messages.

Although the RT/PC paradigm only deals with an input process, we have shown that output timing constraints naturally can be expressed in terms of input constraints and that the RT/PC paradigm of interaction is sufficient for expressing these constraints. Lastly, our experiments have shown that the design discipline can be effectively used to determine important real-time characteristics of systems.

Chapter 7

Conclusions and Contributions

7.1 Thesis Summary

The domain of real-time computing covers a spectrum of computing environments in which the processing requirements of the external world dictate the pace of activity within the computer system. Hard-real-time systems represent one extreme on this spectrum. Hard-real-time systems require strict guarantees of a system's adherence to all performance constraints. These strict guarantees are required because often there is a high cost of failure associated with not adhering to the temporal processing constraints of the external world. In the thesis we have studied the problem of designing and analyzing the real-time behaviors of hard-real-time systems. Hard-real-time systems are an important area of study because all real-time systems contain some hard-real-time component or subsystem.

Our approach to the study of hard-real-time systems has been to develop a discipline for designing such systems. The discipline is composed of a

- a graphical and graph notation for specifying the logical design of real-time systems,
- an operational semantics for the notation for specifying and reasoning about performance constraints, and
- a formal model of an implementation of a real-time system.

In this discipline, a real-time system is expressed as a directed graph where vertices are either *processes*, *data repositories*, *input devices*, or *output devices*. Edges are either synchronous or asynchronous unidirectional communication channels. Vertices communicate with one another by sending messages on channels. The choice of message passing communication is motivated by a natural paradigm of process interaction for hard-real-time processes called the *real-time producer/consumer* (RT/PC) *paradigm*. This paradigm states that to ensure information is not lost when communicating with a real-time process, data must be consumed at the precise rate at which it is produced. In this thesis we have defined a rate in terms of the worst case minimum message inter-arrival time.

The fundamental abstraction that is presented in the design discipline is that all interconnected pairs of vertices adhere to the RT/PC paradigm. Each producer of messages is guaranteed that each message it produces will be consumed by its recipient. We have shown that if the behavior of the producer is unconstrained, then an implication of this guarantee is that the message consumer must consume each message before the next message is sent (i.e., without buffering). The semantics of message passing therefore specify the real-time behavior of each message consumer in a design graph. The semantics formally define the behavior of each vertex in a graph in terms of the rates at which messages are received at the vertex. With these semantics, a designer can derive best and worst case bounds on the propagation time of messages along arbitrary paths in a design graph. These bounds are expressed in terms of the rates at which messages are transmitted on channels. In this manner, the temporal behavior of a design is characterized completely by the rates at which vertices send messages to one another.

In order for the design discipline to be useful it must be possible to realize the abstractions it presents. The final component of the design discipline is a formal implementation model of a design. Abstractly, a design graph is a set of processes that make multiple requests for execution and have strict processing requirements. We develop a model of these processes consisting of a set of independent cyclic tasks. Cyclic tasks make multiple requests for execution; the time of each request being constrained by a set of execution rules. Tasks have arbitrary release times and each request of each task has a deadline for completion. The goal is to schedule the tasks on a uniprocessor in such a manner that each execution request of each task completes before its deadline.

Two characterizations of a task's behavior were considered. We examined and compared a model of *periodic* and *sporadic* tasks. Periodic tasks make requests for execution at precise intervals and sporadic tasks make requests at random but with a minimum separation time between each request. If preemption is allowed at arbitrary points, then the *earliest deadline first* (EDF) scheduling discipline is shown to an optimal scheduling policy for both characterizations. When preemption is disallowed, the EDF discipline is optimal for sporadic tasks and for periodic tasks if all possible release times are considered. The optimality in this case is with respect to the class of scheduling disciplines that do not use inserted idle time. For arbitrary release times, we show that if an optimal non-preemptive scheduling discipline exists for periodic tasks, then P = NP. This demonstrates an important difference between periodic and sporadic tasks.

The complexity of determining when the EDF discipline can correctly schedule a set of tasks was also studied. When preemption is allowed this decision question can be answered in linear time. When preemption is not allowed, then the decision question has pseudo-polynomial time complexity for sporadic tasks. For periodic tasks with arbitrary release times the problem is NP-hard in the strong sense.

These basic scheduling results are next extended to encompass more realistic task models. A model of sporadic tasks with shared software resources was studied. For this problem we develop the concept of an *implementation strategy*. An implementation strategy is the integration of a processor scheduling discipline and a task synchronization discipline. A strategy integrating EDF scheduling with WAIT and BROADCAST synchronization primitives on condition variables was developed. For various characterizations of a task's resource requirements, this strategy was shown to be an optimal strategy.

The abstract scheduling studies have identified fundamental properties of tasks which must be satisfied if the tasks' deadlines are to be respected. The analysis has been constructive and hence suggests a method for implementing designs created with our notation and semantics. The problem of realizing designs was examined next. The determination of whether or not a design graph can be realized has two components. The first component is concerned with the derivation of message transmission rates for each channel in the graph. For a cyclic design graph, it may not be possible to realize the design - independent of a choice of implementation strategy. We have identified properties of cycles which will make an implementation of the design impossible. For acyclic and some cyclic graphs, one can efficiently determine whether or not they will be realizable. The second component deals with the capacity of a processor and the processes' resource requirements. Based on the study of the abstract tasking model, we have proposed a concrete implementation for design graphs constructed using our discipline. With the concrete implementation, the bounds for message propagation delays developed earlier can be significantly tightened.

Lastly we have argued that our design discipline is practical and expressive by demonstrating that actual real-time systems can be designed with the notation. Designs for four actual real-time systems developed by other researchers were presented and compared with the original design. The systems included a digital stopwatch, a computer music synthesis system, a control program for a walking robot, and a Naval communications subsystem. In each case there was a tangible benefit to adopting our discipline over the method used in the original design. The discipline could concisely and effectively determine interesting and important real-time properties of the system. To demonstrate that the discipline is practical we have developed and experimented with a prototype programming system to implement the design abstractions of the discipline. Our experiences with the prototype have been reported.

7.2 Conclusions

From this research we conclude that our design discipline is

- simple enough to be implemented and analyzed efficiently, and
- expressive enough to both construct actual systems, and to determine interesting and important temporal properties of the system.

We conclude that the RT/PC paradigm can be a useful and powerful tool for specifying and reasoning about the behavior of hard-real-time systems.

For realizing our discipline on a single processor, we conclude that an underlying implementation model based on sporadic tasks is superior to a model based on periodic tasks. We also conclude that the feasibility of tasks with preemption constraints is not a function of processor utilization.

7.3 Contributions

The thesis makes contributions to the study of real-time systems in the areas of

- programming and design,
- computational models, and
- theory of deterministic scheduling and resource allocation.

7.3.1 Design of Real-Time Systems

Our development and use of the RT/PC paradigm contributes to the systematic specification and analysis of the real-time behavior of a system. By demonstrating that the design discipline is realizable and useful, we have contributed a novel discipline for constructing hard-real-time systems. The discipline allows for the specification of mutual exclusion and shared data. A method for reasoning about this behavior given only the rates at which processes exchange messages has been demonstrated.

The study of cyclic tasks has lead to the development of an efficient, constructive procedure for assessing the viability of a design. The procedure is sufficient for ensuring the temporal correctness of an implementation strategy. With the addition of the concept of modes, the viability assessment becomes necessary. In addition to assessing viability, the procedure can be used to derive maximum allowable input rates and to tune the system in the event that it is non-viable.

7.3.2 Formal Models of Real-Time Systems

In the area of computational models of real-time systems we have developed feasibility results for the preemptive, and non-preemptive execution of periodic, and sporadic tasks with arbitrary release times on a uniprocessor. A pseudo-polynomial time procedure for deciding the feasibility of sporadic tasks when preemption is not allowed has been developed. For periodic tasks, the intractability of this decision problem has been established. This points out a fundamental difference between periodic and sporadic tasks.

We have extended traditional task models to include shared resource requirements. For simple characterizations of these requirements necessary and sufficient conditions for the feasibility of sporadic tasks have been derived. In all cases we have shown that feasibility for tasks with preemption constraints is not a function of processor utilization. Therefore, task sets with these constraints can, in principle, fully utilize the processor.

7.3.3 Scheduling Theory

Necessary and sufficient conditions on sporadic tasks for the correctness of preemptive, and non-preemptive earliest deadline first scheduling have been developed. There exist efficient algorithms for determining if a set of tasks adhere to these conditions. Problem domains for which these schedulers are optimal (with respect to inserted idle time) have been described.

When tasks have resource requirements, we have shown that the combination of deadline scheduling with wait and broadcast synchronization can lead to an optimal implementation strategy for sporadic tasks on a uniprocessor. A technique for exploiting properties of a deadline scheduler to incorporate precedence constraints into the basic task models has been developed.

7.4 Future Work

Based on the results of our research we recommend the following problems for future consideration.

7.4.1 Further Validation of the Design Discipline

With the exception of the stopwatch system, we have validated the use of our discipline with pen and paper experiments. The crucial test of our design discipline will be its actual use on the construction of a real system.

There are two potential pitfalls for the actual use of our discipline for implementing systems. First, the cost of implementing and supporting our programming abstractions may be too great. Our initial experience with the prototype kernel indicates that this will not be the case. However, further investigations are warranted. The second area of concern is the ability of actual task sets to satisfy the viability conditions stated in Chapter 4. It remains an open question to determine how restrictive these conditions are in practice. We are confident that the viability conditions will not unduly hinder the implementation of real-time systems with our discipline. Given the generally primitive state of reasoning about the real-time behavior of production systems, systems tend to be constructed with rather conservative processor utilizations. For example, United States military systems are required to not exceed 50% processor utilization. While we have demonstrated that viability is not a function of processor utilization, there is quite likely to be a high correlation between low processor utilization and viability.

7.4.2 Software Engineering of Hard-Real-Time Systems

The simplicity of our design discipline lends itself to much automation. The development of tools for constructing design graphs, deriving transmission rate functions, analyzing cycles, and assessing viability, would contribute greatly to the use of our discipline.

A structured graph editor would aid in the development, management, and presentation of designs. For restricted models of programs, such as finite state machines, the transmission rate functions can be derived automatically. Alternately, one could envision the development of programming constructs or macros to aid in their derivation. Ideally, this tool would be integrated with the sequential language compiler.

The most complex tool would easily be the viability assessment tool. An ideal version of such a tool would actually consist of a suite of tools for performing the following functions:

- determination of transmission rates on asynchronous channels,
- determination of execution time costs for processes and data repositories, and
- viability computations and tuning.

Given transmission rate functions, the analysis in Section 5.2 can be applied to solve these functions. A vital open problem concerns the computation of execution time costs for processes and data repositories. Given a piece of sequential code, the problem is to determine what its worst case execution time will be. This can either be done empirically or analytically. Promising analytical techniques for assessing this cost have been developed by Shaw [Shaw 89]. Finally, the data synthesized from a design needs to be evaluated for viability. The basic calculation is straightforward. A more powerful viability analysis tool would include provisions for the interactive assessment of viability. In the event a design is non-viable, the tool would identify patterns of worst case blockage and report them to the designer for guiding the modification process.

7.4.3 Adaptive Scheduling Policies

In Chapters 5 and 6, we commented on how buffers could be selectively used to both make non-viable designs viable and to guarantee a minimum separation between input messages. The primary drawback to this buffering approach is that it is not adaptive. It is quite possible that when actually executing a system designed with our discipline, there can exist an interval in time in which the processor utilization will be well below the statically computed utilization. Since the maximum rate at which buffers are sampled is hardwired in the sampling process, it is possible that inputs could be lost (buffered and then overwritten) even though there existed sufficient capacity to temporarily consume at a higher rate. This problem is symptomatic of the inability of the scheduling policy we have studied to assess, and adapt to a dynamic workload.

What is needed is a mechanism for dynamically assessing the load of the system and adjusting the rates at which buffer sampling tasks execute so that these tasks always execute at the maximum possible rate. In the process of performing these adjustments other tasks with fixed execution rates must be guaranteed not to miss a deadline.

7.4.4 Alternate Paradigms of Real-Time Interaction

Our use of the RT/PC paradigm has assumed that producers could produce data at their maximum rate for an arbitrary interval. If this is not the case then much of our analysis is overly pessimistic. We will construct and validate a system against a set of conditions which can possibly never occur. This motivates the investigation into alternate paradigms of real-time interaction.

If there existed a bound on the number of messages, a process could emit at its worst case rate, then we could utilize a buffering strategy to allow us to consume at an *average* rate. This is simply introducing the RT/PC paradigm at higher level of abstraction. Rather than applying the RT/PC paradigm at the level of an individual message, we are applying it to a group of messages. The group of messages will be consumed before the next group arrives. At this higher level, the analysis will still have the flavor of our deterministic, worst case analysis. Within a group of messages the analysis would take on more of a stochastic flavor.

Note that this alternate style of interaction is only appropriate for a design constructed under the current RT/PC paradigm which is non-viable.

7.4.5 Extending Formal Models

A number of outstanding problems remain in the abstract study of cyclic tasks. For our purposes the most interesting is the incorporation of multiple processors. There are several characterizations of the problem that we are interested in pursuing. The first assumes tasks are bound to processors. The challenge in this problem is to formulate synchronization policies for resources shared between tasks on different processors. These policies should guarantee a maximum delay for accessing a resource Scheduling and resource allocation for tasks on the same processor needs to be reworked since now tasks can be idle while waiting for access to remote resources. During this idle period the processor can execute other tasks.

Other open problems include the development of partitioning algorithms for tasks onto processors.

7.5 Summary

Hard-real-time systems require strict guarantees of adherence to timing constraints. The RT/PC paradigm of interaction can effectively be used to describe these constraints and to reason about the real-time behavior of systems. The design discipline we have constructed based on the RT/PC paradigm is simple enough to suggest efficient implementations and yet is expressive enough to derive interesting real-time properties.

This is just the beginning. Experimentation with additional real-time systems is planned to gain further insights in both the use of our discipline as well as the applicability of the tasking models we have considered.

Glossary

The following is a glossary for the key terms and notations used in this dissertation. Following each entry is a page number for the original definition, and use, of the term.

c_i	Computational cost. The time required to execute task T_i to completion on a dedicated processor. (p. 56)
c_{ij}	Computational cost. The time required to execute the j^{th} phase of task T_i to completion on a dedicated processor. (p. 96)
correctness	
producer/consumer systems	A producer/consumer system in which the *producer* and *consumer* are synchronized in such a manner that all data objects produced by the *producer* are ultimately consumed by the *consumer*. (p. 17)
scheduling disciplines	A scheduling policy that can guarantee that each execution request of each task will complete execution at, or before, its deadline. (p. 57)
cyclic task	A task that makes multiple requests for execution. (p. 56)
$d_{a,b}$	Processor demand. The minimal processing time required by a set of cyclic tasks in the closed interval $[a,b]$ to ensure that no tasks misses a deadline in this interval. (p. 61)
decision procedure	An algorithm for determining whether a specific decision question is true or false. (p. 80)
E	The set of edges in a design graph. (p. 23)
EDF	Earliest Deadline First scheduling policy. (pp. 55,61)

feasibility	A boolean measure of temporal correctness for cyclic tasks. A set of tasks is *feasible* if and only if it is possible to schedule the tasks such that no task ever misses a deadline. (p. 57)
G	A design graph. A directed graph. (p. 23)
implementation strategy	A strategy for the execution of cyclic tasks on a uniprocessor. A strategy consists of a policy for allocating resources to tasks and a policy for scheduling the tasks. (p. 94)
inserted idle time	The period of time between the time that the processor is idled when there exists a unsatisfied request for execution, and the time a task is dispatched. (p. 69)
MP/SC	Multiple Producer/Single Consumer. A *process* or *data repository* that receives messages from more than one *process*. (p. 35)
message transmission rate	The worst case rate at which message arrive on a (physical) asynchronous channel in a design graph. The rate is defined as the inverse of the worst case minimum inter-arrival time of messages at the channel's receiver. (p. 31)
optimality	A metric of comparison for scheduling disciplines (and implementation strategies). A discipline (strategy) is optimal if it can correctly schedule (execute) any feasible set of cyclic tasks. (pp. 57, 97)
overlap	A property of a set of single phase cyclic tasks that share resources. An overlap exists if there exist tasks T_i, T_j and T_k with $r_i = r_k$, $r_j \neq r_k$, $r_j \neq 0$, and $p_k < p_j < p_i$. In other words, tasks T_k and T_i share one resource, task T_j uses a different resource, and task T_j's period lies between the periods of tasks T_i, T_j. (p. 108)
P_i	The period of the task with smallest period that uses resource R_i. (p. 96)
period	The duration between execution requests of cyclic tasks. For periodic tasks this duration is a constant. For sporadic tasks the duration is a lower bound. (p. 56)
periodic task	A cyclic task that makes requests for execution at regular intervals. (p. 56)
processor demand	See $d_{a,b}$.
producer/consumer system	A producer/consumer system consists of two agents a *producer* and a *consumer*. The *producer* produces data objects that are consumed by the *consumer*. (p. 17)

pseudo polynomial time	A measure of the computational complexity of an algorithm. An algorithm has pseudo polynomial time complexity if its running time is polynomial in the length and size of the inputs. (p. 83)
R_i	The i^{th} software resource. Resource R_0 refers to the special null resource. This is the only resource that can be allocated to multiple tasks simultaneously. (p. 95)
r_{ij}, r_i	The resource required by the j^{th} phase of task T_i. An integer in the range $0 \le r_{ij} \le m$. With a single subscript the second subscript is understood to be equal to 1. (p. 95)
RT/PC	Real-Time Producer/Consumer. (pp. 16,20)
real-time producer/consumer paradigm	A producer/consumer paradigm of process interaction wherein the i^{th} output of the *producer* is consumed by the *consumer before* the $(i+1)^{st}$ output is produced. (p. 22)
real-time producer/consumer system	A producer/consumer system whose correctness has a temporal component. For a real-time producer/consumer system to be correct, the *consumer* must consume data objects at the rate at which they are produced. (p. 20)
realizability	A boolean measure of temporal correctness for design graphs. A design graph is *realizable* if and only if it is possible to implement the design such that all communications on asynchronous channels are guaranteed to adhere to the RT/PC paradigm. (p. 121)
release time	The point in real-time when a cyclic task makes its first request for execution. (p. 56)
request interval	The (closed) interval between the time a task makes a request for execution and the time the request must be completed. (p. 59)
SP/SC	Single Producer/Single Consumer. A *process* or *data repository* that receives messages from only one *process* (or *input device*). (p. 35)
sporadic task	A cyclic task that makes request for execution such that there exists a minimum separation time between execution requests. (p. 56)
τ	A set of cyclic tasks. (pp. 57, 95)
T_i	The i^{th} cyclic task. (p. 56)
temporal correctness Of cyclic tasks	The ability of a set of tasks to meet their deadlines. (p. 57)

Of design graphs	The ability of a design graph to adhere to its semantics. A design graph is temporally correct if all communications on asynchronous channels adhere to the RT/PC paradigm. (p. 121)
viability	The ability of a set of tasks to meet their deadlines when executed under a particular implementation strategy. For a given implementation strategy, a set of tasks will be *viable* if and only if the strategy can execute - schedule and allocate resources to - the tasks such that no task ever misses a deadline. (p. 96)
V	The set of vertices in a design graph. (p. 23)
well-formed design graph	A design graph in which there exists a path from an *input device* to each *process* in the graph. (p. 25)

List of References

[Barth et al. 85]
Barth, P., Guthery, S., Barstow, B., *The Stream Machine: A Data Flow Architecture for Real-Time Applications*, Proc. of the Eighth IEEE Int. Conf. on Softw. Eng., London, 1985, pp. 103-110.

[Bernstein & Harter 81]
Bernstein, A., Harter, P.K., *Proving Real-Time Properties of Programs With Temporal Logic*, Proc. of the Eighth Symp. on Operating Systems Principles, **ACM Operating Systems Review**, Vol. 15, No. 5, (December 1981), pp. 1-11.

[Berry et al. 83]
Berry, G., Moisan, S., Rigault, J.-P., *ESTEREL: Towards a Synchronous and Semantically Sound High-Level Language for Real Time Applications*, Proc. IEEE Real-Time Systems Symp., Arlington, VA, December 1983, pp. 30-37.

[Berry & Cosserat 85]
Berry, G., Cosserat, L., *The ESTEREL Synchronous Programming Language and its Mathematical Semantics*, Lecture Notes in Computer Science, Vol 197 (1985) pp. 389-448.

[Bertossi & Bonuccelli 83]
Bertossi, A.A., Bonuccelli, M.A., *Preemptive Scheduling of Periodic Jobs in Uniform Multiprocessor Systems*, **Information Processing Letters**, Vol. 16, No. 1, (January 1983), pp. 3-6.

[Bloch & Dannenberg 85]
Bloch, J.J., Dannenberg, R.D., *Real-Time Accompaniment of Keyboard Performances*, Proc. of the International Computer Music Conference, B. Traux ed., Computer Music Assoc., San Francisco, August 1985, pp. 279-289.

[Conway et al. 67]
Conway, R.W., Maxwell, W.L., Miller, L.W., **Theory of Scheduling**, Addison-Wesley, Reading, MA, 1967.

[Coulas et al. 87]
Coulas, M.F., MacEwen, G.H., Marquis, G., *RNet: A Hard Real-Time Distributed Programming System*, **IEEE Trans. on Computers**, Vol. C-36, No. 8, (August 1987), pp. 917-932.

[Dannenberg 84]
Dannenberg, R.B., *Arctic: A Functional Language for Real-Time Control*, ACM SIGPLAN , pp. 96 - 103.

[Dasarathy 85] Dasarathy, B., *Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them*, **IEEE Trans. on Soft. Eng.**, Vol. SE-11, No. 1, (January 1985), pp. 80 - 86.

[Dhall & Liu 78]
Dhall, S.K., Liu, C.L., *On a Real-Time Scheduling Problem*, **Operations Research**, Vol. 26, No. 1, (January 1978), pp. 127-140.

[Donner 83] Donner, M.D., *The Design of OWL: a language for walking*, ACM SIGPLAN Symp. on Programming Language Issues in Software Systems, SIGPLAN Notices, Vol. 18, No. 6, (June 1983), pp. 158 - 165.

[Donner 84] Donner, M.D., *Control of Walking: Local control and real-time systems*, Ph.D. Thesis, Carnegie-Mellon University, Department of Computer Science, Report #CMU-CS-84-121, May, 1984.

[Donner 89] Donner, M.D., Personal communication, April 1989.

[Frederickson 83]
Frederickson, G.N., *Scheduling Unit-Time Tasks with Integer Release Times and Deadlines*, **Information Processing Letters**, Vol. 16, No. 4, (May 1983), pp. 171-173.

[Garey & Johnson 77]
Garey, M.R., Johnson, D.S., *Two-Processor Scheduling with Start-Times and Deadlines*, **SIAM J. Computing**, Vol. 6, No. 3, (September 1977), pp. 416-426.

[Garey & Johnson 79]
Garey, M.R., Johnson, D.S., **Computing and Intractability, A Guide to the Theory of NP-Completeness**, W.H. Freeman and Company, New York, 1979.

[Garey et al. 81]
Garey, M.R., Johnson, D.S., Simons, B.B., and Tarjan, R.E., *Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadlines*, **SIAM J. Computing**, Vol. 10, No. 2, (May 1981), pp. 256-269.

[Gautier et al. 87]
Gautier, T., Le Guernic, P., Besnard, L., *SIGNAL: A Declarative Language for Synchronous Programming of Real-Time Systems*, Functional Programming Languages and Computer Architectures, G. Kahn, ed., **Springer-Verlag Lecture Notes in Computer Science**, Vol. 274, (September 1987), pp. 257-277.

[Gomaa 84] Gomaa, H., *A Software Design Method for Real-Time Systems*, **CACM**, Vol. 27, No. 9, (September 1984), pp. 938-949.

[Harel 79] Harel, D., *Statecharts: A Visual Formalism for Complex Systems*, **Science of Computer Programming**, North-Holland, Vol. 8, (1987), pp. 231-274.

[Harter 87] Harter, P.K., *Response Times in Level-Structured Systems*, ACM Trans. on Computer Systems, Vol. 5, No. 3, (August 1987), pp. 232-248.

[Hennessy et al. 75]
Hennessy, J.L., Kieburtz, R.B., Smith, D.R., *TOMAL: A Task-Oriented Microprocessor Applications Language*, **IEEE Trans. on Industrial Electronics and Control Instrumentation**, Vol. IECI-22, No. 3, (August 1975), pp. 283-289.

[Hoare 74] Hoare, C.A.R., *Monitors: An Operating System Structuring Concept*, **Comm. of the ACM**, Vol. 17, No. 10, (October 1974), pp. 549-557.

[Holt 72] Holt, R.C., *Some Deadlock Properties of Computer Systems*, **ACM Comp. Surveys**, Vol. 4, No. 3, (September 1972) pp. 179-196.

[Holt 83] Holt, R.C., **Concurrent Euclid, The UNIX System, and TUNIS**, Addison-Wesley, Reading, MA, 1983.

[Hood & Grover 86]
Hood, P., Grover, V., *Designing Real-Time Systems in Ada*, SofTech Inc., Technical Report #1123-1, January 1986.

[Jahanian & Mok 86]
Jahanian, F., Mok, A. K.L., *Safety Analysis of Timing Properties in Real-Time Systems*, **IEEE Trans. on Soft. Eng.**, Vol SE-12, No. 9, (September 1986), pp. 890-904.

[Jeffay 88] Jeffay, K., *On Optimal, Non-Preemptive Scheduling of Periodic Tasks*, University of Washington, Department of Computer Science, Technical Report #88-10-03, October 1988.

[Jeffay & Anderson 88]
Jeffay, K., Anderson, R., *On Optimal, Non-Preemptive Scheduling of Periodic and Sporadic Tasks,* University of Washington, Department of Computer Science, Technical Report #88-11-06, November 1988. (Submitted for publication.)

[Jeffay 89] Jeffay, K., *Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints*, University of Washington, Department of Computer Science, Technical Report #89-07-02, June 1989. (To appear in Proc. Tenth IEEE Real-Time Systems Symp., December 1989.)

[Jensen et al. 85]
Jensen, E.D., Locke, C.D., Tokuda, H., *A Time-Driven Scheduling Model for Real-Time Operating Systems*, Proc. IEEE Real-Time Systems Symp., San Diego, CA, December 1985, pp. 112-122.

[Kligerman & Stoyenko 86]
Kligerman, E., Stoyenko, A.D., *Real-Time Euclid: A Language for Reliable Real-Time Systems*, **IEEE Trans on Soft. Eng.**, Vol. SE-12, No. 9, (September 1986), pp. 941 - 949.

[Knuth 73] Knuth, D.E., **The Art of Computer Programming**, Vol. 1, *Fundamental Algorithms*, Addison-Wesley, Reading, MA, Second Ed., 1973.

[Lampson & Redell 80]
Lampson, B.W., Redell, D.D., *Experience with Processes and Monitors in Mesa*, **Comm. of the ACM**, Vol. 23, No. 2, (February 1980), pp. 105 - 117.

[Lawler & Martel 81]
Lawler, E.L., Martel, C.U., *Scheduling Periodically Occurring Tasks on Multiple Processors*, **Information Processing Letters**, Vol. 12, No. 1, (February 1981), pp.9-12.

[Leinbaugh 80]
Leinbaugh, D.W., *Guaranteed Response Times in a Hard-Real-Time Environment*, **IEEE Trans. on Soft. Eng.**, Vol. SE-6, No. 1, (January 1980), pp. 85-91.

[Leinbaugh & Yamini 86]
Leinbaugh, D.W., Yamini, H.-R., *Guaranteed Response Times in a Distributed Hard-Real-Time Environment*, **IEEE Trans. on Soft. Eng.**, Vol. SE-12, No. 12, (December 1986), pp. 1139-1144.

[Leung & Merrill 80]
Leung, J.Y.-T., Merrill, M.L., *A Note on Preemptive Scheduling of Periodic, Real-Time Tasks*, **Information Processing Letters**, Vol. 11, No. 3, (November 1980), pp.115-118.

[Leung & Whitehead 82]
Leung, J.Y.-T., Whitehead, J., *On the Complexity of Fixed Priority Scheduling of Periodic, Real-Time Tasks*, **Performance Evaluation**, Vol. 2, No. 4, (1982), pp.237-250.

[Liu & Layland 73]
Liu, C.L., Layland, J.W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, **Journal of the ACM**, Vol. 20, No. 1, (January 1973), pp. 46-61.

[Locke 88] Locke, C.D., Personal communication, December 1988.

[Maloney 89] Maloney, J., Personal communication, May 1989.

[Martin 78] Martin, T., *Realtime Programming Language PEARL - Concepts and Characteristics*, Proc. of the Second IEEE Computer Society COMPSAC, Chicago, IL, November 1978, pp. 301-306.

[Mok 83] Mok, A.K.-L., *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Thesis, MIT, Department of EE and CS, MIT/LCS/TR-297, May 1983.

[Mok 85] Mok, A.K.-L., *SARTOR - A Design Environment for Real-Time Systems*, Proc. of the Ninth IEEE COMPSAC, October 1985, pp. 174 - 181.

[Mok & Dertouzos 78]
Mok, A.K.-L., Dertouzos, M.L., *Multiprocessor Scheduling in a Hard Real-Time Environment*, Proc. of the Seventh Texas Conf. on Computing Systems, IEEE, Houston, TX, (1978), pp. 5.1-5.12.

[Mok & Sutanthavibul 85]
Mok, A.K.-L., Sutanthavibul, S., *Modeling and Scheduling of Dataflow Real-Time Systems*, Proc. IEEE Real-Time Systems Symp., San Diego, CA, December 1985, pp. 178-187.

[Mok et al. 87]
Mok, A.K.-L., Amerasinghe, P., Chen, M., Sutanthavibul, S., Tantisirivat, K., *Synthesis of a Real-Time Message Processing System with Data-driven Timing Constraints*, Proc. IEEE Real-Time Systems Symp., San Jose, CA, December 1987, pp. 133 - 143.

[Mok et al. 88]
Mok, A.K.-L., Amerasinghe, P., Chen, M., Sutanthavibul, S., Tantisirivat, K., *Automated Synthesis of a Real-Time System with Data-Driven Timing Constraints*, University of Texas at Austin, Department of Computer Science, SARTOR Project Technical Report, April 1988.

[Park & Shaw 89]
Park, C.Y., Shaw, A.C., *Design and Experiences With a Source Level Tool for Predicting Deterministic Execution Times of Programs*, University of Washington, Department of Computer Science, Technical Report (to appear).

[Schwan et al. 87]
Schwan, K., Bihari, T., Weide, B.W., Taulbee, G., *High-Performance Operating System Primitives for Robotics and Real-Time Control Systems*, **ACM Trans. on Computing Systems**, Vol. 5, No. 3, (August 1987), pp. 189-231.

[Sha et al. 87] Sha, L., Rajkumar, R., Lehoczky, J.P., *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, Carnegie Mellon University, Department of Computer Science, Technical Report CMU-CS-87-181, November 1987.

[Shaw 74] Shaw,A.C., **The Logical Design of Operating Systems**, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[Shaw et al.. 85]
Shaw, A.C., Binding, C., Hu, W.L., Jeffay, K., *Research in Real-Time Systems*, IEEE Third Workshop in Real-Time Operating Systems, Boston, MA, February 1986 (also available as University of Washington Department of Computer Science Technical Report TR-85-12-05 (December 1985)).

[Shaw 89] Shaw, A.C., *Reasoning About Time in Higher-Level Language Software*, **IEEE Trans. on Soft. Eng.**, Vol. SE-15, No. 7, (July 1989), pp. 875-889.

[Simpson & Jackson 79]
Simpson, H.R., Jackson K., *Process Synchronization in MASCOT*, **The Computer Journal**, Vol. 22, No. 4, (1979) pp. 332-345.

[Sorenson 74]
Sorenson, P.G., *A Methodology for Real-Time System Development*, Ph.D. Thesis, University of Toronto, June 1974.

[Sorenson & Hamacher 75]
Sorenson, P.G., Hamacher, V.C., *A Real-Time Design Methodology*,. **INFOR**, Vol. 13, No. 1, (February 1975), pp. 1-18.

[Stoyenko 87a]
Stoyenko, A.D., *A Schedulability Analyzer for Real-Time Euclid*, Proc. IEEE Real-Time Systems Symp., San Jose, CA, December 1987, pp. 218 - 227.

[Stoyenko 87b]
Stoyenko, A.D., *A Real-Time Language With a Schedulability Analyzer*, University of Toronto, Computer Systems Research Institute, Technical Report #CSRI-206, December 1987.

[US DoD 83] **Reference Manual for the Ada Programming Language**, ANSI /Military standard MIL-STD-1815A, US DoD, January 1983.

[Vercoe 84] Vercoe, B., *The Synthetic Performer in the Context of Live Performances*, Proc. of the International Computer Music Conference, W. Buxton ed., Computer Music Assoc., San Francisco, October 1984, pp. 199-200.

[Wirth 77a] Wirth, N., *Modula: A Language for Modular Multiprogramming*, **Software Practice and Experience**, Vol. 7, No. 1, (January 1977), pp. 3-35.

[Wirth 77b] Wirth, N., *The Use of Modula*, **Software Practice and Experience**, Vol. 7, No. 1, (January 1977), pp. 37-65.

[Wirth 77c] Wirth, N., *Toward a discipline of real-time programming*, CACM Vol. 20, No. 8 (Aug. 1977), 577-583.

[Zhao et al. 87a]
Zhao, W., Ramamritham, K., Stankovic, J.A., *Preemptive Scheduling Under Time and Resource Constraints*, **IEEE Trans. on Computers**, Vol. C-36, No. 8, (August 1987), pp. 949 - 960.

[Zhao et al. 87b]
Zhao, W., Ramamritham, K., Stankovic, J.A., *Scheduling Tasks with Resource Requirements in Hard Real-Time Systems*, **IEEE Trans. on Soft. Eng.**, Vol. SE-13, No. 5, (May 1987), pp. 564 - 577.

[Zwarico & Lee 85]
Zwarico, A., Lee, I., *Proving a Network of Real-Time Processes Correct*, Proc. IEEE Real-Time Systems Symp., San Jose, CA, December 1987, pp. 169 - 177.

Appendix A

The Correctness of an Implementation Strategy For Single Phase Sporadic Tasks That Share a Single Resource

The appendix provides a proof of correctness for the implementation strategy introduced in Section 4.4 of Chapter 4. The correctness is shown for the case of single phase sporadic tasks that share a single resource. The proof is similar to the proof of Theorem 3.3.3 and is presented in full detail.

Theorem 4.4.2: Let $\tau = \{T_1, T_2, \ldots, T_n\}$ be a set of single phase sporadic tasks, sorted in non-decreasing order by period, that share a single resource R_1. Under the implementation strategy of Chapter 3, τ will be viable for arbitrary release times, if:

1) $\displaystyle\sum_{i=1}^{n} \frac{c_i}{p_i} \le 1,$

2) $\forall k,\ 1 \le k < n,\ r_k \ne 0$: $\displaystyle p_k \ge \underset{I_1}{\mathrm{MAX}}\left(c_i + \underset{I_2}{\mathrm{MAX}}\left(-l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \right)\right),$

where $I_1 = (k < i \le n) \wedge (r_i = r_k) \wedge (r_i \ne 0)$,
$I_2 = 0 < l < p_i - p_k$.

3) $\forall k,\ 1 \le k < n,\ r_k = 0$: $\displaystyle p_k \ge \underset{II_1}{\mathrm{MAX}}\left(c_i + \underset{II_2}{\mathrm{MAX}}\left(-l + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \right)\right),$

where $II_1 = (k < i \le n) \wedge (r_i \ne 0)$,
$II_2 = \mathrm{MAX}(0, P_1 - p_k) < l < p_i - p_k$.

Proof: (By contradiction.)

Assume the contrary, i.e., assume that conditions (1), (2), and (3) above hold and yet there exists a set of values for the release times s_i such that a task T_k misses a deadline at some point in time when τ is implemented using the strategy of Chapter 3. Without loss of generality, assume that T_k is the first task to miss a deadline. (In the case of simultaneous missed deadlines, let T_k be the task with smallest index.)

Consider the first execution request of T_k that misses a deadline. Let t_d be the deadline of this request. For the remaining tasks whose release times are less than t_d, either they have a request interval with deadline at t_d or they have a request interval that contains t_d. For the tasks with request intervals that contain t_d, we consider the following two major cases.

<u>Case 1</u>: None of the request intervals that contain the point t_d are scheduled prior to t_d.

Let t_0 be the end of the last period in which the processor was idle. If the processor has never been idle then let $t_0 = 0$. Define $d_{a,b}$ as the processor demand required by τ in the interval $[a,b]$. That is, $d_{a,b}$ is the minimum amount of processor time required in the interval $[a,b]$ to ensure that no deadline is missed in the interval $[a,b]$. In the interval $[t_0,t_d]$, the total processor time demand is

$$d_{t_0,t_d} \leq \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_0}{p_j} \right\rfloor c_j .$$

Since there is no idle period in $[t_0,t_d]$ and since a task misses a deadline at t_d, it must be the case that $t_d - t_0 < d_{t_0,t_d}$. Therefore

$$t_d - t_0 < \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_0}{p_j} \right\rfloor c_j \leq \sum_{j=1}^{n} \frac{t_d - t_0}{p_j} c_j ,$$

or simply

$$1 < \sum_{j=1}^{n} \frac{c_j}{p_j} .$$

This is a contradiction of condition (1). Therefore, if conditions (1), (2), and (3) hold and τ is not viable under our implementation strategy, then some of the request intervals that contain the point t_d must have been scheduled prior to t_d.

Case 2: Some of the request intervals that contain the point t_d are scheduled prior to t_d.

Let T_i be a task who does not have a deadline at t_d, and whose request interval that contains t_d is the last such request interval to execute prior to t_d. Let $t_s = t_d - p_k$ (the point in time at which T_k makes its request for execution that has a deadline at t_d). Let $t_{s'}$ be the point in time at which the execution request of T_i that contains t_d is scheduled. That is, $t_{s'}$ is the point in time at which the execution request of T_i that contains t_d commences execution (is scheduled for the first time). At time $t_{s'}$, all request intervals containing $t_{s'}$ with deadline less than or equal to t_d, have completed execution. Let t_i be the point in time when the request interval of T_i containing t_d last ceases to execute prior to t_d. These points are illustrated in Figure A.1.1 below.

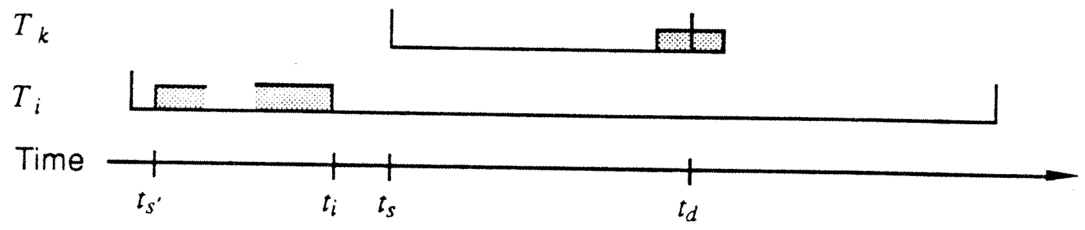


Figure A.1.1: Definition of points $t_{s'}$, t_i, and t_s.

Note that if T_k misses a deadline at t_d, then there can not have been any period in the interval $[t_{s'},t_d]$ in which the processor was idle. Such an idle period could have occurred only after the request interval of T_i containing t_d had completed. If such an idle period existed we could apply the analysis of Case 1 and arrive at a contradiction of condition (1).

There are two main sub-cases to consider depending on the value of r_i (whether or not T_i is a resource consuming task).

Case 2.1: $r_i = 0$ (T_i is a non-resource consuming task.)

Since T_i is preemptable, all request intervals containing t_i with deadlines less than or equal to t_d must have been satisfied by t_i. Therefore, for the interval $[t_i,t_d]$, we have

$$d_{t_i,t_d} \le \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_i}{p_j} \right\rfloor c_j .$$

Since there is no idle period in $[t_i,t_d]$ and since a task misses a deadline at t_d, it must be the case that $t_d - t_i < d_{t_i,t_d}$. This leads to a contradiction of condition (1) as in Case 1.

Case 2.2: $r_i \neq 0$ (T_i is a resource consuming task.)

We need to further sub-divide this case into two cases depending on the value of r_k. The analysis for both cases, however, will be largely identical.

Case 2.2.1: $r_i \neq 0$, $r_k \neq 0$ (Tasks T_k and T_i are resource consuming tasks.)

Since $r_k \neq 0$, we ultimately seek to establish a contradiction involving condition (2). As in the previous cases, we will derive a bound on the processor demand for some interval in time ending at t_d. This is a more complex problem for the present case since it is possible for non-resource consuming tasks with request intervals containing t_d to execute in the interval $(t_{s'}, t_i]$ as shown in Figure A.1.2 below.

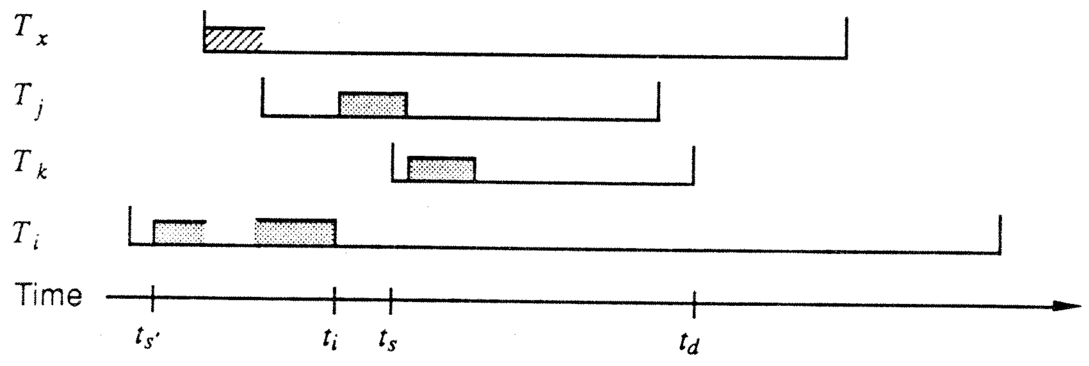


Figure A.1.2: Execution sequence with a non-resource consuming task executing in $(t_{s'}, t_i]$.

Since our implementation strategy uses deadline scheduling, all request intervals containing $t_{s'}$, with deadlines less than or equal to t_d, must have been satisfied by $t_{s'}$. Therefore, for the interval $[t_{s'}, t_d]$, we have

$$d_{t_{s'},t_d} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t_d - (t_{s'}+1)}{p_j} \right\rfloor c_j .$$

If we let $l = (t_d - t_{s'}) - p_k$, then the above inequality reduces to

$$d_{t_{s'},t_d} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j .$$

If there are no non-resource consuming tasks with request intervals containing t_d that execute in the interval $[t_{s'}, t_i]$, then because task T_k missed a deadline at t_d we must have

$t_d - t_{s'} < d_{t_{s'},t_d}$. From the definition of l, this is just $l + p_k < d_{t_{s'},t_d}$. Combining this with the previous inequality gives

$$l + p_k \;<\; c_i \;+\; \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \,,$$

or simply

$$p_k \;<\; c_i \;-\; l \;+\; \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \,.$$

Now since $p_i > t_d - t_{s'}$, we have $0 < l < p_i - p_k$. Therefore the above inequality contradicts the fact that condition (2) was assumed to be true.

If there are non-resource consuming tasks with request intervals containing t_d that execute in the interval $[t_{s'},t_i]$, then we cannot claim that $t_d - t_{s'} < d_{t_{s'},t_d}$. However, we can show that if T_k misses a deadline at t_d and these non-resource consuming tasks execute in the interval $[t_{s'},t_i]$, then there exist a set of release times such that T_k must miss a deadline at t_d without these non-resource consuming tasks executing in the interval $[t_{s'},t_i]$. We can then apply the analysis above to show that condition (2) could not have been true. The following procedure will generate these new release times.

The first step is to normalize τ with respect to the point in time $t_{s'}$. We create a set of sporadic tasks τ' identical to τ except that the tasks in τ' possibly have different release times. τ' releases its first task at time $t_{s'}$ and has the same behavior in the interval $[t_{s'},t_d]$ as τ. By normalizing τ, we are ignoring the history of τ before $t_{s'}$. More formally, let τ' be the set of sporadic tasks obtained from τ by modifying the release times of the tasks in τ such that:

i) If a task j in τ has no execution request that executes in the interval $[t_{s'},t_i]$, then task j in τ' will have a release time of infinity. (These tasks played no role in T_k missing a deadline at t_d and hence we will ignore them.)

ii) Task T_i is assigned a release time of $t_{s'}$ in τ'. (In τ', $s_i = t_{s'}$.)

iii) For the remaining tasks in τ', we assign to each a release time equal to the time of the first execution request made by the corresponding task in τ after time $t_{s'}$.

We will assume that in the interval $[t_{s'},t_d]$, all tasks in τ' released before t_d, will make execution requests at the same points in time as their corresponding task in τ did. Therefore, in the interval $[t_{s'},t_d]$, the behavior of τ', will be the same as the behavior of τ under our implementation strategy.[1] That is, if the scheduler dispatches task j at time t, $t_{s'} \le t \le t_d$, when executing τ, then the scheduler will dispatch task j at time t when executing τ'. If task k in τ misses a deadline at t_d, then task k in τ' must also miss a deadline at t_d.

The second step is to iteratively advance the release times of the tasks in τ' in such a manner that task T_k always misses a deadline at t_d, and with each iteration, in the interval $[s_i, t_i]$, there is less processor time consumed by non-resource consuming tasks with request intervals containing t_d. Consider the execution of τ' in the interval $[s_i,t_d]$ ($= [t_{s'},t_d]$). Let I represent the interval $[s_i,t_i]$. The interval I can be partitioned into $2h + 1$ contiguous sub-intervals, $I_1 I'_1 I_2 I'_2 I_3 I'_3 \dots I_h I'_h I_{h+1}$, where

$I_j =$ $[tt_j,tt'_j]$, where $tt_1 = s_i$ and $tt'_{h+1} = t_i$. I_j is an interval of time in which either T_i executes, or, request intervals of tasks with deadlines less than or equal to t_d execute. In the latter case, these tasks will be only non-resource consuming tasks. (Recall that any resource consuming task, other than T_i, cannot execute until after t_i.)

$I'_j =$ $[tt'_j,tt_{j+1}]$. I'_j is an interval of time in which request intervals of tasks, other than T_i, with deadlines after t_d execute. I'_j will contain execution intervals from only non-resource consuming tasks.

In the interval I, no resource consuming tasks other than T_i execute. The following procedure creates a sequence of task sets $\tau_0, \tau_1, \tau_2, \dots, \tau_{h-1}$, that differ from τ' (and hence τ) only in their assignment of release times. The release times for the tasks in these sets are

[1] Assume that when choosing among a set of tasks when identical deadlines, our EDF scheduler utilizes a deterministic procedure to choose the task to execute next.

chosen such that in all task sets τ_j, T_k fails at t_d and $[s_i,t_i]$ contains j intervals in which request intervals of non-resource consuming tasks with deadlines after t_d execute. The sequence of task sets is generated (in reverse order) as follows.

Step 0: Initialization step.

For the set of tasks τ', partition the interval $I = [s_i,t_i]$ into $2h + 1$ contiguous sub-intervals as described above. (Assume $h \geq 1$.)

$j \leftarrow h$. (j will be an iteration counter.)

$\tau_h \leftarrow \tau'$.

Step 1: Release time computation step.

This step derives a set of tasks τ_{j-1} from τ_j by creating new release times for the tasks in τ_j. The new release times are chosen such that τ_{j-1} will contain only $j - 1$ intervals in which non-resource consuming tasks with request intervals containing t_d execute prior to t_d.

If $j > 0$, then I contains at least three sub-intervals $I_1I'_1I_2$. In the sub-interval I_1, T_i does not complete its execution request. Task T_i was either the only task to execute in I_1, or T_i was preempted (and possibly resumed later in I_1) by a non-resource consuming task with a request interval completely contained within $[s_i,t_d]$. In the latter case the execution request of the non-resource consuming task will execute to completion in I_1. Let c be the total cost of the execution requests of those non-resource consuming tasks that execute in I_1. Let c' be the length of the sub-interval I'_1, $c' = tt_2 - t'_1$. Let $s = s_i + c + c'$.

Let τ_{j-1} be the set of sporadic tasks created from τ_j by modifying the release times of the tasks in τ_j such that:

i) If task g in τ_j has no execution request that executes in the interval $[s,t_i]$, then task g in τ_{j-1} will have a release time of infinity.

ii) Each remaining task g in τ_j with a request interval containing t_d that is scheduled prior to t_d, is assigned a release time of $s_g + c + c'$ in τ_{j-1}.

iii) The remaining tasks in τ_{j-1}, are assigned a release time equal to the time of the first execution request made by the corresponding task in τ_j after time s.

These new release times effectively defer the computations that occurred in I'_1. We will again assume that in the interval $[s,t_d]$, all tasks in τ_{j-1} released before t_d, will make execution requests at the same points in time as their corresponding task in τ_j did. Therefore, in the interval $[s,t_d]$, the behavior of τ_{j-1} will be the same as τ_j (or τ for that matter) under our implementation strategy. If task k in τ_j misses a deadline at t_d, then task k in τ_{j-1} must also miss a deadline at t_d.

$j \leftarrow j - 1$

Step 2: Termination step.

If $j = 0$ then for the set of tasks τ_0, no non-resource consuming task with request interval containing t_d executes in the interval $[s_i,t_i]$. From the definition of T_i, no non-resource consuming task with request interval containing t_d executes in the interval $[s_i,t_d]$ as well. Therefore the analysis at the start of this case shows that condition (2) could not have been true for τ_0 if T_k missed a deadline at t_d. This terminates the process.

Step 3: Partition step. ($j > 0$)

For the set of tasks τ_j, partition the interval $I = [s_i,t_i]$ into $2j + 1$ contiguous sub-intervals as described above.

Go to Step 1.

The following example illustrates this process.

Example A.1: Let τ consist of the following single phase tasks:

$T_1 = (6,1,4), \quad r_1 = 0$
$T_3 = (0,1,6), \quad r_3 = 0$
$T_4 = (4,3,19), \quad r_4 = 0$
$T_5 = (5,3,25), \quad r_5 = 0$

$T_2 = (0,2,5), \quad r_2 = 1$
$T_6 = (2,4,50), \quad r_6 = 1$

For this example, only tasks T_2 and T_6 are resource consuming tasks. Assume that the tasks in τ have the pattern of execution requests shown in Figure A.1.3 below. Note that T_2 misses a deadline at time $t_d = 18$ under our implementation strategy.

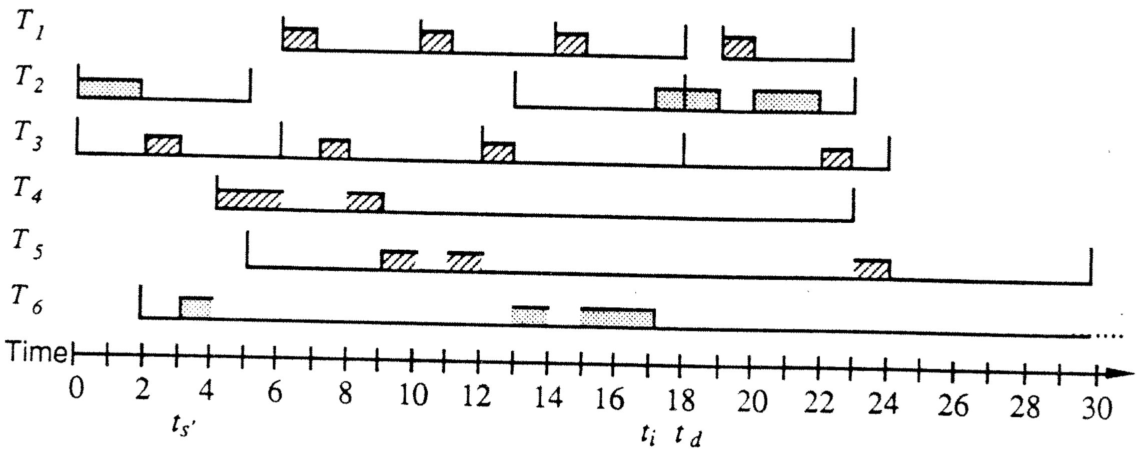


Figure A.1.3: Execution of the tasks in Example A.1.

For the purposes of this example let $T_k = T_2$, $T_i = T_6$, $t_{s'} = 3$, $t_i = 17$, and $t_d = 18$. The first step in the process is to normalize τ about $t_{s'} = 3$. Normalizing τ, gives the following for τ':

$T_1 = (6,1,4), \quad r_1 = 0$
$T_3 = (6,1,6), \quad r_3 = 0$
$T_4 = (4,3,19), \quad r_4 = 0$
$T_5 = (5,3,25), \quad r_5 = 0$

$T_2 = (13,2,5), \quad r_2 = 1$
$T_6 = (3,4,50), \quad r_6 = 1$

Tasks in τ' will make the pattern of execution requests shown in Figure A.1.4. Note that T_2 still misses a deadline at time $t_d = 18$.

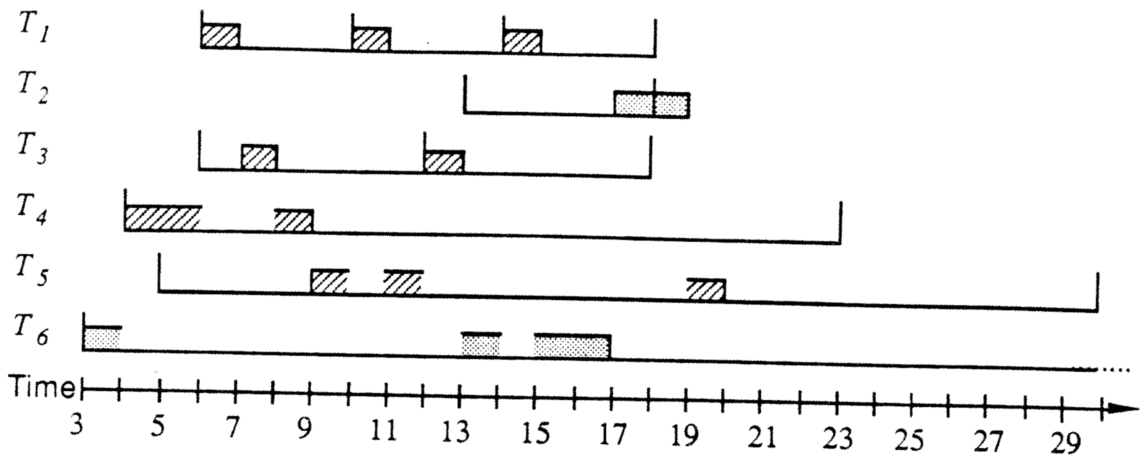


Figure A.1.4: Execution of the normalized tasks τ'.

For τ', in the interval $I = [s_i, t_i] = [3,17]$, there are three sub-intervals in which non-resource consuming tasks with request intervals containing t_d execute (i.e., $h = 3$). Therefore, I can be partitioned into the $2h + 1 = 7$ contiguous sub-intervals shown in Figure A.1.5 below.

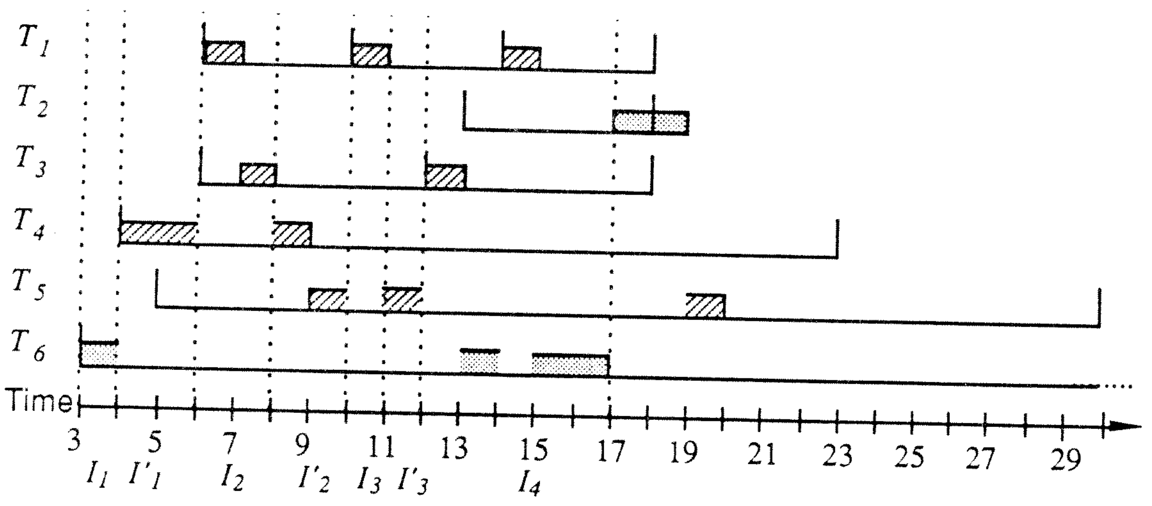


Figure A.1.5: Partitioning of the execution sequence of τ'.

We let $\tau_h = \tau_3 = \tau'$, and create the following sequence of task sets. τ_2 is created from τ_3 by eliminating I'_1 (deferring the computations of the non-resource consuming tasks that occur in the interval [4,6]) and advancing the release times of tasks with request intervals that contain t_d (T_4, T_5, and T_6) by 2 time units. Modifying the release times of the tasks in τ_3 yields the following for τ_2:

$T_1 = (6,1,4),\ \ r_1 = 0$ $\qquad$ $T_2 = (13,2,5),\ \ r_2 = 1$
$T_3 = (6,1,6),\ \ r_3 = 0$ $\qquad$ $T_6 = (5,4,50),\ \ r_6 = 1$
$T_4 = (6,3,19),\ \ r_4 = 0$
$T_5 = (7,3,25),\ \ r_5 = 0$

Tasks in τ_2 will make the pattern of execution requests shown in Figure A.1.6 with T_2 still missing a deadline at time $t_d = 18$.

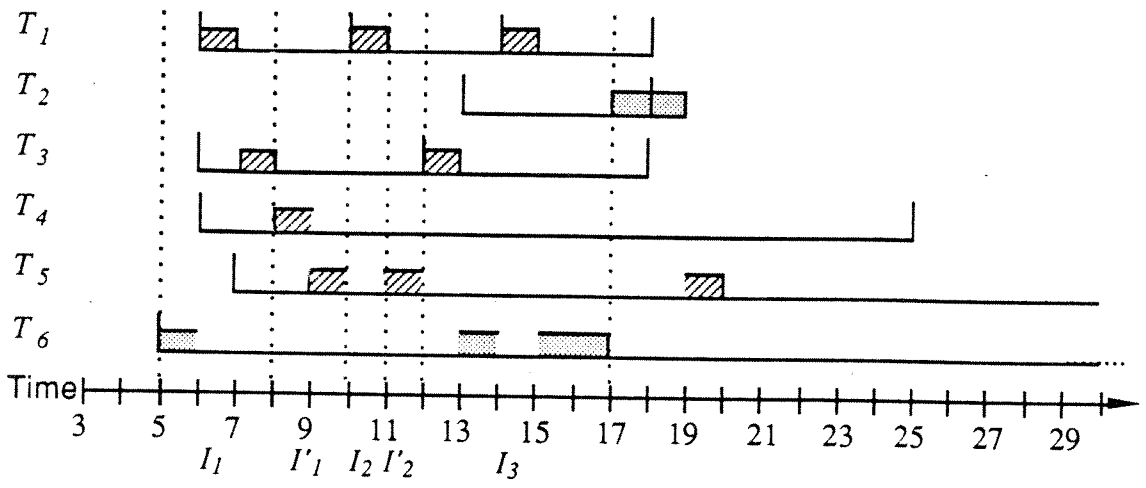


Figure A.1.6: Partitioning of the execution sequence of τ_2.

For τ_2, $I = [s_i,t_i] = [5,17]$, contains two sub-intervals in which non-resource consuming tasks with request intervals containing t_d execute. We create τ_1 from τ_2 by advancing the release times of tasks with request intervals that contain t_d four time units. Two time units come from deferring the computations in $I'_1 = [8,10]$, and 2 time units come from factoring out the execution requests that complete in $I_1 = [5,8]$. Tasks T_1 and T_3 have execution requests that complete execution in I_1 hence their release times in τ_1 will the time of their next execution request. Task T_4 does not execute in the interval $[s_i + 4,t_i] = [9,17]$, hence it will have a release time of infinity in τ_1. This yields the following for τ_1:

$T_1 = (10,1,4),\ \ r_1 = 0$ $\qquad$ $T_2 = (13,2,5),\ \ r_2 = 1$
$T_3 = (12,1,6),\ \ r_3 = 0$ $\qquad$ $T_6 = (9,4,50),\ \ r_6 = 1$
$T_4 = (\infty,3,19),\ \ r_4 = 0$
$T_5 = (11,3,25),\ \ r_5 = 0$

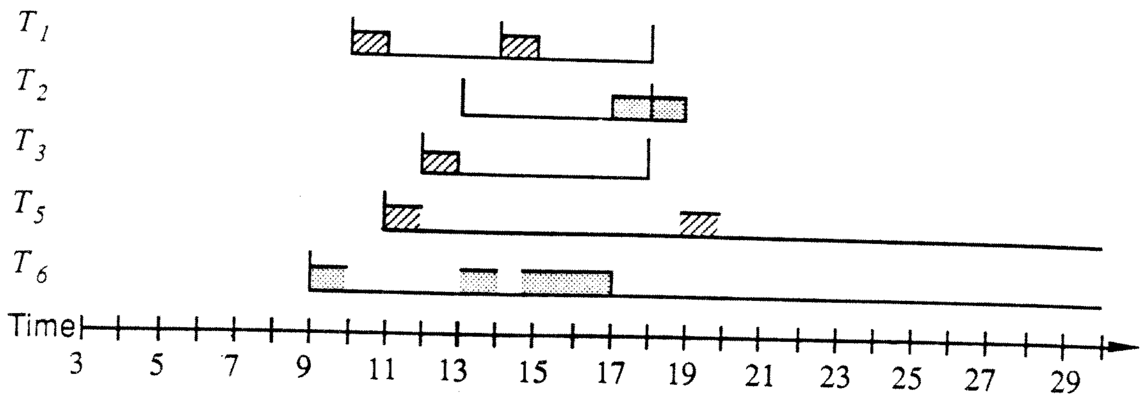


Figure A.1.7: Execution sequence for τ_1.

Repeating this process a final time, we modify the release times of the tasks in τ_1 yielding the following for τ_0:

T_1 = (14,1,4), $r_1 = 0$
T_3 = (12,1,6), $r_3 = 0$
T_4 = (∞,3,19), $r_4 = 0$
T_5 = (∞,3,25), $r_5 = 0$

T_2 = (13,2,5), $r_2 = 1$
T_6 = (11,4,50), $r_6 = 1$

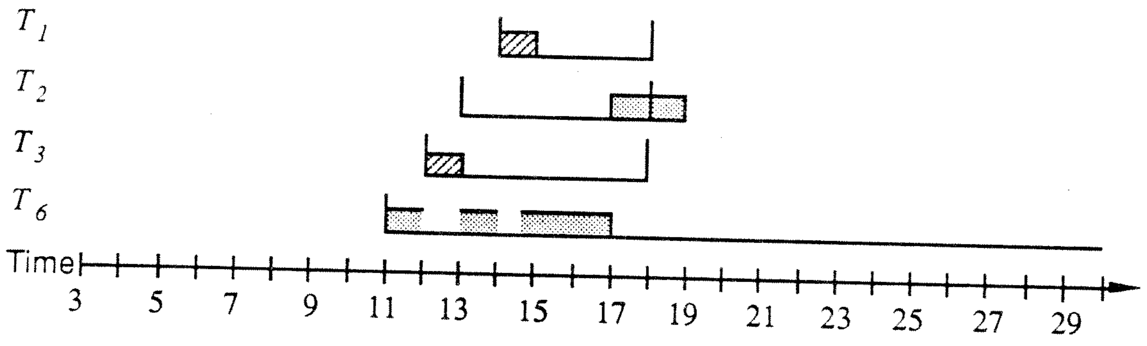


Figure A.1.8: Execution sequence for τ_0.

We have derived a set of release times for the tasks in τ such that task T_2 misses a deadline at time t_d =18 without any non-resource consuming task with request interval containing t_d executing prior to t_d. This completes the example.

We have shown that if T_k misses a deadline at t_d, and there are non-resource consuming tasks with request intervals containing t_d that execute in the interval $[t_{s'},t_i]$, then there exist a set of release times such that T_k must miss a deadline at t_d without these non-resource consuming tasks executing in the interval $[t_{s'},t_i]$. The analysis at the beginning of this case allows us to conclude that condition (2) could not have been satisfied by τ. This is the contradiction we seek.

Note that condition (2) does not depend on the release time of any task. Therefore, when creating the sequence of h task sets, that differed only in choice of release times, if condition (2) was not met by a task set τ_i, then condition (2) could not have been met by any task set in the sequence (including $\tau_h = \tau$).

Case 2.2.2: $r_i \neq 0$, $r_k = 0$ (T_i is a resource consuming task, T_k is a non-resource consuming task.)

The analysis of this final case is essentially identical to the previous case. The primary difference is that we will establish a contradiction involving condition (3). As in the previous case, the analysis is complicated by the existence of non-resource consuming tasks with request intervals containing t_d that execute in the interval $(t_{s'},t_d]$ as shown in Figure A.1.9 below.

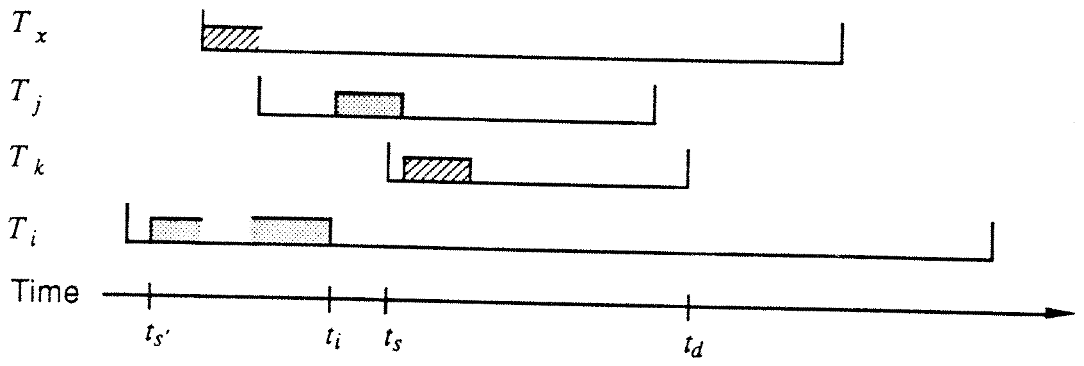


Figure A.1.9: Execution sequence with a non-resource consuming task executing in the interval $(t_{s'},t_i]$.

The first step is to again derive a bound on the processor demand for some interval ending at t_d. If there exists a resource consuming task T_j with period $p_j < t_d - t_{s'}$, as shown above, then in the interval $[t_{s'},t_d]$,

$$d_{t_{s'},t_d} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t_d - (t_{s'}+1)}{p_j} \right\rfloor c_j .$$

If we let $l = (t_d - t_{s'}) - p_k$, then the above inequality reduces to

$$d_{t_{s'},t_d} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j .$$

If there are no non-resource consuming tasks with request intervals containing t_d that execute in the interval $[t_{s'}, t_i]$, then because task T_k missed a deadline at t_d, $t_d - t_{s'} < d_{t_{s'}, t_d}$, or from the definition of l, $l + p_k < d_{t_{s'}, t_d}$. Combining this with the previous inequality gives

$$l + p_k \; < \; c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \,,$$

or simply

$$p_k \; < \; c_i \; - \; l \; + \sum_{j=1}^{i-1} \left\lfloor \frac{p_k + l - 1}{p_j} \right\rfloor c_j \,.$$

Now since $p_i > t_d - t_{s'}$, we know that $l < p_i - p_k$, and hence $l > p_j - p_k \geq P_1 - p_k$. Therefore the above inequality contradicts the fact that condition (3) was assumed to be true. If there are non-resource consuming tasks with request intervals containing t_d that execute in the interval $[t_{s'}, t_i]$, then the derivations employed in Case 2.2.1 can be employed here as well to show that condition (3) must still have been false.

If there does not exist a resource consuming task with period less than $t_d - t_{s'}$, then T_j need not complete its request interval that contains t_d prior to t_d. In this case, the analysis of Case 2.1 can be applied to show that condition (1) could not have been true if T_k missed a deadline at t_d. This concludes the analysis of this case.

This exhausts all possible cases. We have shown that in all cases, if a task misses a deadline then at least one of the conditions in the statement of the theorem must have been false. This contradicts our original assumption that a task missed a deadline and yet these conditions held. We therefore conclude that a task could not have missed a deadline. Hence, under the implementation strategy of Chapter 3, conditions (1), (2), and (3) are sufficient for guaranteeing the viability of a set of sporadic tasks that share a single resource.

Δ

Appendix B

On the Absence of Deadlock

B.1 Introduction

In this appendix we show that the design discipline presented in Chapter 2 is deadlock free. Specifically, we show that it is not possible for a process in a well-formed design graph to ever become deadlocked while waiting for access to either a MP/SC data repository or to a mutual exclusion lock. These are the only times a process can become blocked due to contention. The absence of deadlock is important for the scheduling analysis performed on design graphs. Without this property we would be unable to implement the design discipline by creating a task for each process. If processes could become deadlocked then we would be unable to derive a finite bound on the worst case blockage a process can experience while waiting for access to shared resources.

B.2 The Absence of Deadlock

Deadlock problems are usually expressed in terms of the state of a system's resource requests and allocations. A design graph may contain the following types of resources[1]:

[1]The terminology, and resource graph notation, used in this section is taken from [Holt 72]. Our style of presentation of resource graphs (circles for processes) is borrowed from [Shaw 74]. We use the notation of the latter reference for resource graphs as it is closer to our own graph notations.

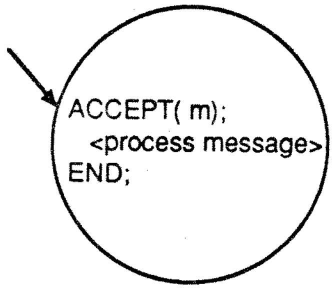
- Messages. The messages sent between processes are resources. A separate message resource exists for every asynchronous channel in a design graph.

- Process locks. Mutual exclusion locks are resources. Process locks are required by processes in a mutual exclusion relation. A separate process lock resource exists for every mutual exclusion relation (implicit and explicit) containing processes in a design graph.

- Data locks. Data locks are required by data repositories in a mutual exclusion relation. A separate process data resource exists for every mutual exclusion relation (implicit and explicit) containing data repositories in a design graph.

Message resources are a consumable resource while both types of lock resources are serially reusable resources. For each mutual exclusion relation, there is only a single unit of the corresponding lock resource.

We will use a simplified version of the task schemata developed in Chapter 4. Processes make requests for these resources as shown in Figure B.2.1 below. At a minimum, every process will require a message resource. In the simplest case, the process repeatedly requests a unit of the message resource `message_i`. Once acquired, the process simply consumes the message.

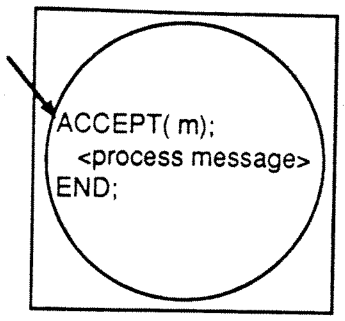


```
LOOP
   REQUEST( message_i)
      <consume message>
   END REQUEST
END LOOP
```

Figure B.2.1: Message resource request schema.

If the process is in a mutual exclusion relation, it must additionally acquire the process lock resource for its mutual exclusion relation before it may proceed. Assume that process P_i is in a mutual exclusion relation (appears inside of a box in a design graph) whose process lock resource is called `process_lock_j`. A schema for process P_i appears in Figure

B.2.2. Note that the process only attempts to obtain the process lock resource after it has acquired a message resource.



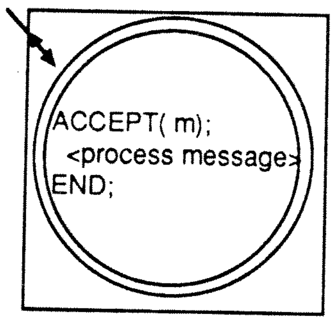
```
LOOP
   REQUEST( message_i)
   REQUEST( process_lock_j)

      <consume message>

   RELEASE( process_lock_j);
   END REQUEST /* for
message */
END LOOP
```

Figure B.2.2: Message and process lock resource request schema.

In both cases above, the processing of a message by a process may involve processing at one or more data repositories. For each data repository in a mutual exclusion relation, the calling process will have to make the additional lock resource requests shown in Figure B.2.3 below. Assume that the data repository in Figure B.2.3 is in the mutual exclusion relation whose data lock resource is called `data_lock_j`. Note that since data repositories do not emit messages, a process will never hold or request more than one data lock resource at a time.



```
REQUEST( data_lock_j)
   <process code>
RELEASE( data_lock_j)
```

Figure B.2.3: Data lock resource request schema.

Furthermore, note that in all cases, processes make only single-unit requests for any resource.

The state of a program's resource allocations and requests can be represented by a *general resource graph* (GRG). A GRG is a bipartite, directed graph. In a GRG circles represent processes and squares represent resources. An edge from a consumable resource (message resources) to a process indicates that the process is a producer of units of that resource. An edge from a process to a consumable resource indicates a request for a unit of that resource. An edge from a serially reusable resource (a lock resource) to a process indicates an assignment of a unit of that resource to the process. An edge from a process to a serially reusable resource to a process indicates a request by the process for a unit of that resource.

For example, consider the design graph in Figure B.2.4 below.

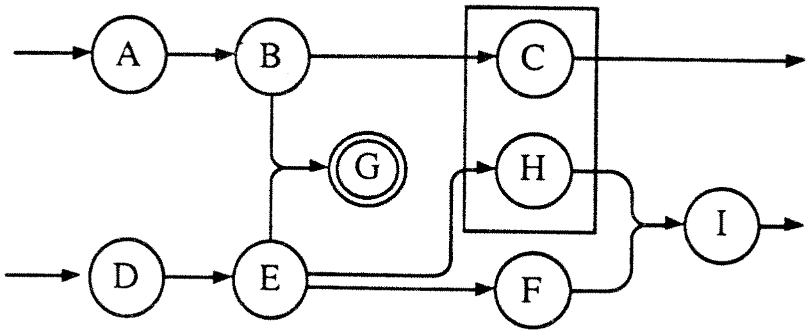


Figure B.2.4: An example design graph.

A GRG for this design is shown in Figure B.2.5. Message resources are labelled with an M and lock resources are labelled DL and PL for data lock resources and process lock resources respectively. This particular GRG shows all the possible resource request edges. (Note that the GRG in Figure B.2.5 does not correspond to any actual state of the system since it shows processes *C* and *H* requesting a message and process lock simultaneously.)

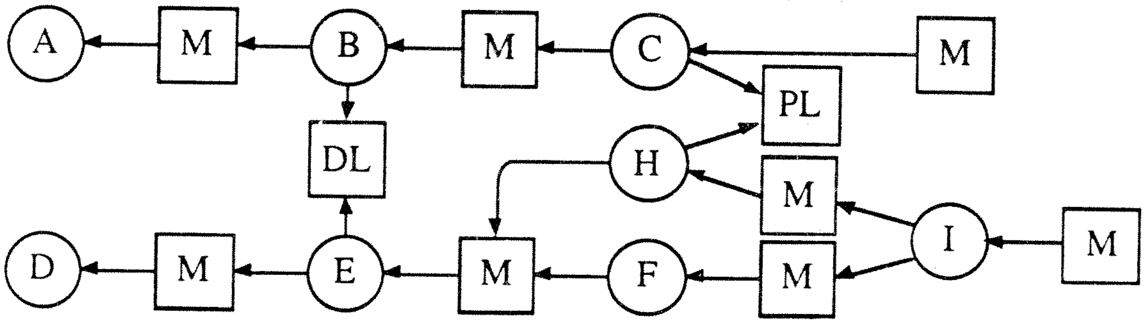


Figure B.2.5: General resource graph with all possible request edges.

The GRG in Figure B.2.6 below represents the state of the system wherein all processes except processes *B*, *C*, and *E* are waiting for a message (they have made a request for a unit of their respective message resources). Processes *B*, *C*, and *E* have received messages (acquired a unit of their respective message resource). Processes *B* and *C* are processing a message. That is, they have acquired a unit of its message resource and their lock resource for their mutual exclusion relation. Process *E* is blocked on a request for the data lock resource held by process *B*.

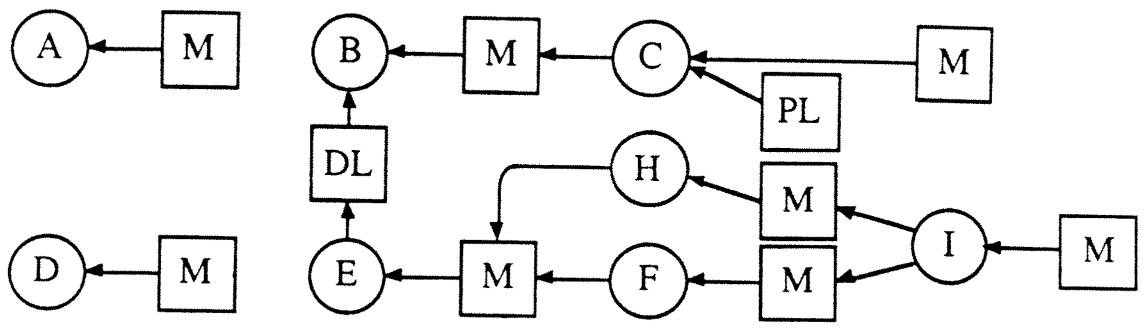


Figure B.2.6: General resource graph.

An *expedient general resource graph* (EGRG) is a general resource graph in which all requests for available resources have been granted. EGRGs have the following property.

Theorem: [Holt 72] An EGRG with single unit resource requests represents a deadlocked system state if and only if the EGRG contains a knot.

We can use this theorem to show the absence of deadlock in our design discipline by showing that no GRG representing a valid system state can contain a knot.

Theorem B.2.1: Every design graph constructed in our discipline is deadlock free.

Proof: It suffices to show that for an arbitrary design graph, no GRG corresponding to a valid state of the design can contain a knot. Since a necessary condition for the existence of a knot in a graph is a cycle, we first examine the nature of the possible cycles in a GRG.

Lemma B.2.1: If a cycle exists in a GRG then the only resource nodes that can appear in the cycle must correspond to message resources.

Proof of Lemma: By the construction of our discipline, a process will not request any lock resource until it has acquired a unit of its message resource (received a message). Because of this, the subgraph shown in Figure B.2.7 can never appear in a valid GRG (where L is one of DL or PL).

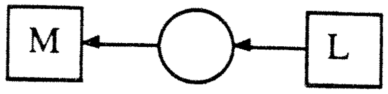


Figure B.2.7: An acquired lock resource and a request for a message resource.

Therefore, any resource nodes in a cycle in a valid GRG must be either only message resources or only lock resources. We will show that the latter case cannot occur.

Since data repositories perform no output, the subgraph shown in Figure B.2.8 can also never appear in a valid GRG.

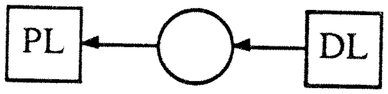


Figure B.2.8: An acquired data lock and a request for a process lock resource.

Therefore, if a cycle in a valid GRG contains a lock resource, then all resources nodes in the cycle must be of the same type (either all process locks resources or all data lock resources). This implies that if a cycle contains a lock resource then one of the two subgraphs in Figure B.2.9 must appear in the GRG.

Figure B.2.9: Simultaneous lock acquisition and request.

However neither of these subgraphs may appear in a valid GRG. The first subgraph corresponds to a system state where a process holds one data lock and requests another. This can only occur if a data repository sends a message to another data repository. This cannot happen as we have disallowed output from data repositories. The second subgraph corresponds to a system state where a process holds one process lock and requests another. This could occur only if a process were in multiple mutual exclusion relations. However we have specifically disallowed this construction.

Therefore, if a cycle exists in a valid GRG, it can contain only message resource nodes. Δ

Lemma B.2.2: A process node in a GRG corresponding to an external input process, cannot appear in a knot in a GRG.

Proof of Lemma: Since an external input process does not receive any messages, it never makes any resource requests. In terms of a GRG, this means that a process node corresponding to an external input process, has no edges directed outward from itself. Therefore the set of reachable nodes from such a process node in a GRG is the empty set. Since the set of nodes reachable from any node in a knot must be equal to the set of nodes in the knot, a process node in a GRG corresponding to an external input process cannot appear in a knot.

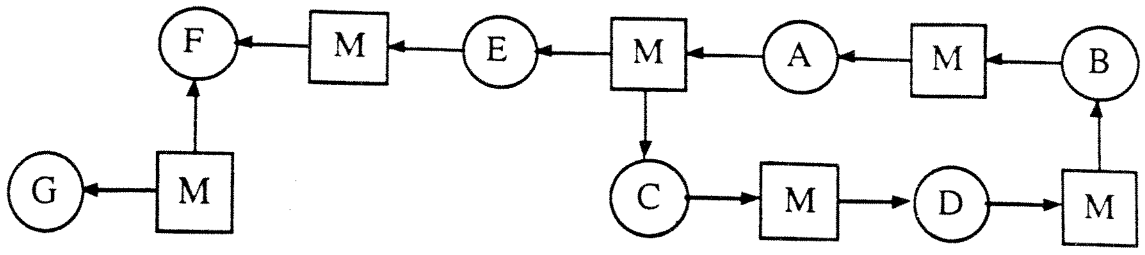
Δ

We now show that a knot cannot exist in a valid GRG. Assume there exists a cycle in some GRG corresponding to a valid state of an arbitrary design graph. By Lemma B.2.1 we know that all the resource nodes in this cycle must correspond to message resources. Recall from Chapter 2 (Section 2.3.1), that there must exist a path in the graph from at least one external input source to each process in the cycle. Let T_i be some process whose corresponding process node is in the cycle in the GRG. Consider the set of processes along a path from an external input process to T_i. In the GRG, consider the chain of process nodes and message resource nodes that correspond to the processes along this path.

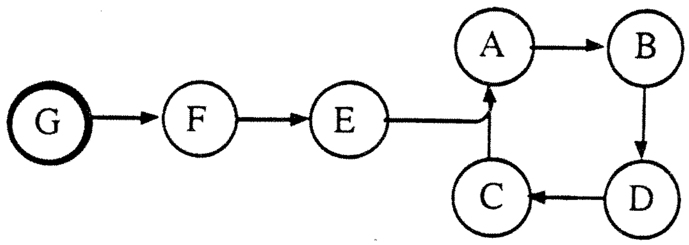
For example, consider the design graph with a cycle shown in Figure B.2.10(b). The corresponding GRG is shown in Figure B.2.10(a).

If the GRG represents a system state where any of the process nodes in this chain does not have a message resource request edge directed outward from itself, then the cycle in question cannot form a knot. If the all of the process nodes in this chain do have message resource request edges directed outward from themselves, then all of the nodes in this chain are members of the set of nodes reachable from nodes in the cycle. Therefore, if the cycle is in a knot then all the nodes along this chain must also be in the knot. In particular, a process node corresponding to an external input process would appear in the knot. However, from Lemma B.2.2 we know that this cannot be the case. Therefore the cycle cannot be in a knot. Since we have shown that no cycle may appear in a knot, and since a cycle is a necessary condition for a knot to exist, we conclude that no knot can exist in any GRG corresponding to a valid system state.

Δ



a) Sample GRG with a cycle.



b) corresponding design graph.

Figure B.2.10: Sample GRG with a cycle.

Appendix C

A Prototype Kernel For a Message Passing Hard-Real-Time System

C.1 Introduction

Section 6.2 has described our experiences with an implementation of our design discipline. In this appendix we provide a more detailed description of the implementation. Specifically, we present the design of a prototype kernel to support the execution of real-time programs designed using our discipline. The term *kernel* refers to a small set of library routines that implement the abstractions of our design discipline. The kernel implements *processes*, *asynchronous channels*, and *messages*. As described below, our use of non-preemptive scheduling in the kernel eliminated the need to implement *data repositories* and *synchronous channels*.[1]

The kernel is currently implemented in the C programming language [Kernighan & Ritchie 78] with the occasional use of in-line assembly language. The code was developed on a Sun 2 workstation using the development tools of the UNIX operating system. The next section describes the implementation from the perspective of a programmer. The kernel entries are defined and an example of their use is given. The example is taken from an implementation of the stopwatch design described in Chapter 2. Section C.3 describes the

[1] Recall from Section 2.3 that the final two components of our design discipline: *input devices* and *output devices* are specifications and not implementable components.

design of the kernel and discusses our implementation of *earliest deadline first* (EDF) scheduling and of *pre-scheduling*. Some preliminary performance figures are given for the basic kernel operations. We conclude with some brief remarks on planned extensions to our kernel.

C.2 Programmer Interface

The kernel provides a set of entry procedures and data types for implementing design graphs. A design graph is implemented by creating a task for each process and then interconnecting the tasks according to the structure of the design graph. The programming discipline we use for each of these steps is described below.

C.2.1 Process Definition and Task Construction

A *process* is a block of C code structured according to the schema given in Section 2.3.1. Messages are sent and received by calling kernel entries `Emit()` and `Accept()` respectively. A *process* is implemented as an infinite loop inside of a C procedure. The enclosing procedure is called a *task*. A sample task is shown in Figure C.2.1 below. The task corresponds to a *process* that accepts a message, performs some computation and then emits a message.

```
GenericTask()
 { channel   outputCh;
   long      inMesg, outMesg;

/* Waiting for channel bindings...
*/
   GetChannelBindings( 1, &outputCh);

   <Task initialization code goes here>

   while( 1) {
      Accept( &inMesg);

      <Body of process goes here>

      Emit( outputCh, outMesg);
    }
 }
```

Figure C.2.1: A task created from a sample process.

There are three parts to a task:

- channel declarations and bindings,
- task or process initialization, and
- process body.

The first portion of a task defines and initializes the channels that the process uses for output.[2] As described in Section 2.3, a process is an object with a single input channel (port) and some number of output channels (ports). In our implementation, each task "owns" the channels it sends messages on. A channel is a data type. The data structures that comprise a channel are bound to the task that uses the channel for output. Variables that are declared as type `channel` appear in a task only as a parameter to the Emit routine.

In principle, a task need not explicitly know where it is sending a message when it performs an emit. Among other reasons, this separation allows tasks to be defined independent of their use. However, there is a significant performance enhancement if tasks know the identity of their message recipients. Because of this, a facility is provided for tasks to dynamically determine how their channels are bound. This is accomplished by calling

```
GetChannelBindings( numChannels, ch1, ch2, ch3, ch4, ch5)
   int        numChannels;
   channel   *ch1, *ch2, *ch3, *ch4, *ch5;
```

The calling task passes in the number of channels it owns (the number of output channels it uses) and pointers to each of these channels. The call will suspend the task until all of its channels have been created. Channels are created when the tasks are interconnected. This will be discussed in further detail below.

Once the communication connections have been established, the initialization of data repositories is performed. (Recall from Section 2.3 that processes can not contain global variables or variables whose values persist across message arrivals. Therefore, processes

[2] Since synchronous channels are largely ignored in the current prototype, assume that the term *channel* always refers to an *asynchronous channel*.

themselves have no data to initialize.) Data repositories are implemented as C procedures. A synchronous message emission is simply a procedure call. Access control for shared data repositories is eliminated by our use of non-preemptive scheduling.

The final part of a task is the code that constitutes the body of the process. This block of code is encapsulated in an infinite loop. When tasks are executing process code, a task will accept a message, consume the message and then loop back in order to accept its next message. (It is assumed that the Accept kernel entry appears only once in each task.) Since processes conceptually never terminate, there is no mechanism for terminating a task.

A message is simply a C pointer. In the C language this means that a message can effectively be any data type. We assume that programmer discipline dictates how messages are interpreted by the receiving task. Messages are received by calling the kernel entry:

```
Accept( mesg)
   long  **mesg;
```

Aside from initialization and set-up procedures, this is the only kernel entry that can potentially suspend the execution of the caller. If there is no message for the caller or there exists a message but the caller is ineligible to continue execution (e.g., the caller does not have the nearest deadline), then the caller will be suspended.

Message are sent using the kernel entry:

```
Emit( channelHandle, message)
   channel   channelHandle;
   long       *message;
```

This routine will send message `message` on channel `channelHandle`. The parameter `channelHandle` is a channel that is defined (owned) by the caller.

C.2.2 Graph Definition

Once tasks have been created they need to be instantiated and interconnected. These functions are combined in a procedure that is executed by the C procedure `main()`. We refer to this routine as the *null task*.

A task is instantiated by a call to the kernel entry

```
taskIdPtr CreateTask( taskName, procedure, stackSize, parameter)
   char*        taskName;
   void        (*procedure)();
   unsignedint  stackSize;
   long*        parameter;
```

This routine makes the procedure whose name is given by the `procedure` parameter into an independent thread of control. It is expected that this parameter is a task as described in the previous section. A stack of size `stackSize`, is created and the routine `procedure` commences execution on this new stack. The routine `CreateTask` transfers control to the instantiated task. The task executes until it blocks. Typically the task will block while it waits for its channel bindings. When the task blocks the null task is resumed. This created task may be invoked with the optional parameter `parameter`. For debugging purposes tasks may have a name (a simple character string). `CreateTask` returns a symbolic handle for the task just created. This handle is used to identify tasks when interconnecting them.

Once all the tasks have been created, we interconnect output channels to input ports according to the structure of the design graph. Tasks are interconnected by calling the kernel entry

```
channel CreateChannel( sender, senderChannelID, receiver, period)
   taskIdPtr   sender;
   channelId   senderChannelID;
   taskIdPtr   receiver;
   long        period;
```

This creates a connection between the `sender` and `receiver`. The sending tasks channel `senderChannelId` is bound to the receiving task. The minimum interarrival time of messages on this connection is specified by the parameter `period`. Its value is determined using the procedures outlined in Chapters 2 and 5. CreateChannel returns a symbolic handle for the channel just created. This handle is primarily used for debugging as described below.

Channels whose message emiters are input devices (processes in the external world) require special handling. These channels are connected to tasks by calling

```
channel CreateInputChannel( interruptReceiver, period)
   taskIdPtr   interruptReceiver;
   long        period;
```

The sender on this channel will be an interrupt handler. The interrupt handler will use the handle for the channel that is returned to send messages to the task `interruptReceiver`. The fact that input channels require special attention has no effect on the tasks that receive messages on these channels. A fundamental principle of our design discipline is that tasks have no knowledge of who they are receiving messages from. This feature, combined with the dynamic binding of channels, allows for the separate compilation of tasks.

With these procedures the structure of a design graph can be specified. When the graph has been defined the system is ready to commence execution. The end of the graph description phase is signaled with the procedure

```
EndGraphDefinition().
```

This routine readies the system for operation. This routine informs each task that it may now initialize itself and wait for its first message. Upon return, all tasks are waiting to receive a message on one of their input channels.

We say the system "starts" when inputs from the external world begin arriving at tasks corresponding to source vertices in the design graph. Prior to this event, it is often useful for the programmer to artificially insert messages onto channels to "prime" the system. Since tasks have no knowledge of who is actually sending them a message, this priming facility is very useful for task debugging and performance evaluation. To insert a message on a channel, the null task simply calls the `Emit` kernel entry.

Having primed the system (if desired), it only remains to enable the system's interrupt handlers and wait for inputs from the external world to arrive. The latter function is achieved by calling

```
StartSys().
```

This routine places the null task into an idle loop and will never return. From this point forward the null task executes only when there are no messages in the system to be consumed. When the null task does execute it performs performance monitoring functions such as recording the amount of time that the system is idle. From this figure the overall processor utilization can be determined.

C.2.3 Example

The procedure shown in Figure C.2.2 is taken from our implementation of the stopwatch design discussed in Chapter 2. The procedure connects the tasks according to the design graph shown in Figure 2.7.3. (This "is" the null task for the stopwatch implementation.)

```
main ()

 { unsigned int   stack_sz     = 4096;
   LONG           period       = 9999;
   taskId         stopw, tickServer, display, flasher, buttons;
   channel        ch;

/* Fork off tasks...
*/

   stopw      = CreateTask( "StopW",   StopWTask,   stack_sz, NULL);
   flasher    = CreateTask( "Flasher", FlasherTask, stack_sz, NULL);
   display    = CreateTask( "Display", DisplayTask, stack_sz, NULL);
   buttons    = CreateTask( "Buttons", ButtonTask,  stack_sz, NULL);
   tickServer = CreateTask( "TServer", TServerTask, stack_sz, ticksPerTimeUnit);

/* Bind message communication channels...
*/

   ch = CreateChannel( tickServer, 1, stopw,   GetPeriod( tickserver, stopw));
   ch = CreateChannel( tickServer, 2, flasher, GetPeriod( tickserver, flasher));
   ch = CreateChannel( flasher,    1, display, GetPeriod( flasher, display));
   ch = CreateChannel( stopw,      1, display, GetPeriod( stopw, display));
   ch = CreateChannel( buttons,    1, display, GetPeriod( buttons, display));

   tickCh   = CreateInputChannel( tickServer, 1);
   buttonCh = CreateInputChannel( buttons, 1);

   EndChannelBindings();

   TurnOnSigIO();
   TurnOnSigAlarm( secs, u_secs);

   StartSys();
 }
```

Figure C.2.2: Null task for the stopwatch implementation.

Five tasks are created and then five channels are created. In this implementation we used a custom made function `GetPeriod` to set the minimum message inter-arrival times on channels. (The values are essentially computed by hand however.) Input channels for the hardware timer and button driver are created next and then the tasks are allowed to initialize themselves. Lastly, interrupts are enabled and the system is started.

A sample task is shown in Figure C.2.3 below. The task illustrated is a condensed version of the task that corresponds to the button process from the stopwatch design in Figure 2.7.3. In this implementation, the buttons of a mouse on a workstation were used to simulate the watch buttons. The button task receives messages from the mouse device interrupt handler and sends messages to the display task. In our watch implementation, the watch state data repository of Figure 2.7.3 is implemented by a procedure. The task constructed for the button process initializes this data repository.

```
ButtonTask()

 { channel         toDisplay;
   buttonStruct   *buttonPositions
   register BYTE   buttons, curState;

   GetChannelBindings( 1, &toDisplay);

   InitWatchState();

   while (1) {
      Accept( (long *) &buttonPositions);

      curState   = GetMouseData( buttonPositions);
      buttons    = GetWhatChanged( curState, buttonPositions);

      if ((buttons & LEFT_CHANGED) != 0) {
         if ((curState & LEFT_CHANGED) == 0) {     /* ON/OFF button released */
            switch( SetWatchState( onOff)) {
               case SP_ON_SW_ON:    break;
               case SP_OFF_SW_ON:   Emit( toDisplay, UPDATE); break;
               case SP_ON_SW_OFF:   break;
               case SP_OFF_SW_OFF:  break;
            }
         } /* else ON/OFF button depressed */
      }
      if ((buttons & MIDDLE_CHANGED) != 0) {
         if ((curState & MIDDLE_CHANGED) == 0) {  /* SPLIT button released */
            switch( SetWatchState( split)) {
               case INIT:           Emit( toDisplay, UPDATE); break;
               case SP_ON_SW_ON:    break;
               case SP_OFF_SW_ON:   Emit( toDisplay, UPDATE); break;
               case SP_ON_SW_OFF:   break;
               case SP_OFF_SW_OFF:  break;
            }
         } /* else SPLIT button depressed */
      }
   }
}
```

Figure C.2.3: Task created for button process.

The button process accepts a message giving the current state of the buttons. It then determines which buttons have been pressed or released. If a button was only pressed (and not released) then the process does nothing. If a button was released then the process sends a message to the watch state data repository (calls the procedure `SetWatchState`). It receives back the new state of the watch. In Figure C.2.3, the constants of the form `SP_xx_SW_yy` are symbolic codes for the state of the watch. (Recall the finite state machine of Figure 2.7.2.) The letters SP denote the state of the split function (on or off) and the letters SW denote the state of the timer (also on or off). Depending on the state of the watch, the button process either does nothing or sends a message to the display process telling it to update the display.

C.3 Kernel Internal Design

The prototype system and the stopwatch application are currently implemented in the C programming language. The kernel has two routines that use in-line assembly language. These are routines that perform context switches between tasks. All of the code presently executes on a Sun 2 workstation as a user process on top of the UNIX operating system. Input and output is managed with the UNIX signal handling facilities. This allows us to simulate input interrupts. The entire kernel consists of approximately 1000 lines of commented, sparse C code. The stopwatch application is of comparable size.

C.3.1 Implementation of Design Discipline Components

The kernel implements the basic components in our programming model: *processes*, *asynchronous channels*, and *messages*. Processes are implemented as sporadic tasks that are scheduled non-preemptively using the EDF discipline described in Chapter 3. A sporadic task is implemented as a body of code and a stack to execute the code on. A channel is implemented as a single slot buffer. Tasks are implemented as active entities (executable code), while channels and messages are implemented as passive entities (data structures).

We create UNIX signal handlers to perform logical I/O. A signal handler is.associated with each logical input source. In the terminology of the design discipline, we are constructing a signal handler for each *input device*. When an input arrives, the UNIX operating system generates a software interrupt called a *signal*. The signal is delivered to the kernel. Since the kernel uses non-preemptive scheduling, signals are fielded only at well-defined points in the program's operation. When a task is consuming a message, software interrupts are disabled. When the kernel is executing (after a task completes execution), interrupts are enabled. If the tasks created for a design are viable, then no signals will be lost (be missed or overwritten). Since the UNIX user process that executes our kernel is being periodically interrupted by the operating system, our kernel is in essence a virtual time simulation of an actual real-time system.

C.3.2 Task Scheduling

Tasks begin and terminate execution at the Accept statement. In the terminology of Chapters 3 and 4, tasks make execution requests when a message is *sent* to them. Execution requests terminate (complete execution) when they have consumed the message and become blocked on the Accept statement again.

Recall that the minimum message inter-arrival time on a channel is specified when the channel is created. This value is referred to internally as the *period* of the channel. When a message is sent on a channel it is given a *deadline* equal to the deadline of the sending task plus the period for the channel. Messages sent from input devices are assigned a deadline equal to the current value of real-time plus the channel's period. The task receiving the message will adopt the message's deadline as the deadline of its current execution request. Under this scheme, if the tasks created from a design graph are viable, then a single-slot buffer suffices for each channel.

The principal structure within the kernel is a system wide run queue that is ordered by deadline. At the start of the execution of a design, each task is blocked on an Accept statement waiting to receive a message. The run queue contains only the null task which is executing in an idle loop. When a task sends a message to another task, a copy of the message the message is buffered for the receiving task and the receiving task is placed in the run queue. A task is dispatched when it has the nearest deadline of all tasks in the system. Once dispatched, tasks execute to completion without preemption. When a task completes an execution request, all messages that arrived from the external world during the execution of the task (i.e., all pending interrupts), are sent to the appropriate tasks.

The kernel uses the technique of pre-scheduling tasks described in Section 5.4. With pre-scheduling, messages sent to SP/SC tasks are delivered with an earlier deadline than the one described above. Messages contain a header that is used by the kernel to store the starting time of the message sender's current execution request. If a task can be pre-scheduled then it is assigned a deadline equal to the start time encoded in the message plus the channel's period. As discussed in Chapter 5, the technique of pre-scheduling allows us to guarantee a better response to tasks.

C.3.3 Viability Analysis

Currently, the determination of the viability of an implementation of a design is not integrated with the kernel. There exist a separate set of programs for determining if the conditions from Chapter 3 for the correct functioning of the non-preemptive EDF discipline, are satisfied by a set of tasks. We will briefly comment of the operation of these programs.

To determine if a set of tasks is viable we need the period of each channel and the worst case computational cost of consuming a message on each channel. A task's period is a function of the algorithms used by the process the task implements. As such a task's period is independent of the mechanisms used to implement the task. The determination of periods is therefore not germane to the present discussion. (Computing minimum message inter-arrival times is discussed in detail in Chapters 2 and 5.)

The cost of a task is the time required to receive a waiting message, consume the message and then re-enter the kernel to wait for the arrival of the next message. In this manner the cost we measure includes the overhead of the scheduling process itself. The costs of receiving and consuming a message are assumed to be worst case costs. For this prototype all costs were determined empirically based on a manual examination of the code to determine worst case execution paths. Both source, and compiled assembly language code, were examined.

C.3.4 Multiple Producer/Single Consumer Tasks

The kernel associates a type with each task created based on the type of the process from which the task was created (MP/SC or SP/SC). Although the syntax of the message passing operations are identical for MP/SC and SP/SC tasks, the message passing operations for MP/SC tasks are implemented differently from those for SP/SC tasks. In addition, the types of each task is required to determine if tasks can be pre-scheduled. As discussed in Section 5.4, tasks receiving messages from MP/SC tasks cannot be pre-scheduled.

The kernel treats the emission of a message to an MP/SC task differently from an emission to an SP/SC task. Given our creation of a single task for each process, a task can receive messages from multiple sources and can receive messages simultaneously. However, if the tasks are viable then the task is guaranteed to never have to buffer more than a single message from each sending task. After all tasks and channels are created by the null task, the kernel determines if each task is used in the design as an MP/SC task or an SP/SC task. If an MP/SC task receives messages from *n* other tasks, then we create an *n* slot message queue for the task. Messages in this queue are ordered by deadline.

Recall from Section 5.3 that MP/SC tasks MP/SC tasks must have their message emissions buffered to ensure a minimum interarrival time at a receiver. This function is handled by the kernel. Prior to dispatching a task the kernel checks to see if any undelivered messages sent by MP/SC tasks are now ready to be delivered.

C.3.5 Process to Task Mappings Revisited

In Section 5.3 we advocated creating a separate task for each *asynchronous channel* in a design graph. Although the kernel described here only creates a task for each *process*, its underlying task model is the same. That is, conceptually we are creating a task for each *asynchronous channel*.

Our choice of design graph-to-task mapping is a minor optimization of the scheme presented in Chapter 5. (In particular, our scheme is not to be confused with the approach mentioned briefly at the end of Section 5.3.4.) The difference between our prototype and the implementation scheme outlined in Chapter 5, is that the scheme outlined in Section 5.3 will create *n* tasks for an MP/SC process that consumes messages from *n* processes. Our kernel implementation creates only a single task for MP/SC processes. However, *conceptually* there are multiple tasks for each MP/SC process in the kernel. Since the tasks associated with an MP/SC process must be scheduled non-preemptively with respect to one another (recall from Section 2.3.6 that an MP/SC process defines in an implicit mutual exclusion region), these tasks can never execute simultaneously in any implementation. The kernel takes advantage of this fact and uses a single task to simulate several tasks.

C.3.6 Performance

Table C.2.1 below lists the relative performance for the kernel operations used while an application system is running (i.e. we are not interested in the time it takes to create tasks or channels).

Most operations are invoked by procedure calls. Routines listed in all capital letters are macros whose code is inserted in-line in the caller. The first eight routines are internal kernel routines. They performed the functions of: inserting a task into the scheduler's run-queue, removing a task from the scheduler's run-queue, disabling and enabling interrupts, getting the time of day, performing a context switch between two tasks, dispatching a task, and preempting a task. Preemption was implemented as an experiment, however, the version of the kernel described here did not use it. It is provided to indicate the potential cost of more complex implementation strategies.

These timings were obtained by executing the code in question several thousand times (typically 100,000) in a loop and timing the execution of the loop. All of these measurements was performed in UNIX's single user mode on the workstation. The reported times represent worst case times (i.e., the largest number we ever measured). For purposes of comparison, a null C procedure call takes approximately 14 microseconds.

Table C.2.1: Kernel operation execution times.

Operation	**Cost** (in microseconds)
Null procedure call	14
`InsertIntoRQ()`	54 + (number of tasks in queue) * 5.6
`RemoveFromRQ()`	37
`DISABLE_INTS`	262
`ENABLE_INTS`	262
`GET_TIME`	4
`Xfer()`	125
`Dispatch()`	188
`PreemptFor()`	396
`Accept()`	SP/SC case = 211 MP/SC case = 268
`Emit()`	SP/SC case = 108 MP/SC case = 131

C.4 Future Plans

The kernel we constructed to support the development of real-time systems based on our design discipline has demonstrated the practicality of the discipline. Given our initial success with its use, we are motivated to proceed with the kernel's further development and refinement. In particular the following extensions and enhancements are planned.

There is no type checking performed on the senders and receiver of a channel. Ultimately we would like to guarantee that interconnected tasks send and receive the same type of messages. Currently, ensuring the correct interpretation of messages is left to programmer discipline. A solution to this problem will likely require the development of a preprocessor which will read a graph description and then parse the text of tasks to determine channel types. Such a preprocessor would also allow for a more concise specification of tasks. For example, processes could be specified in using the schemas of Chapter 2. Given a set of schemata, a preprocessor could generate the tasks automatically from a template.

Our experiences with the use of non-preemptive scheduling have been very positive. We feel that the benefits of non-preemption, in terms of efficiency, out weigh the loss of generality provided by the more complex implementation strategies of Chapter 4. We therefore anticipate remaining with the current non-preemptive scheduler. Given this inclination, tasks need not be implemented as separate threads of control as mentioned in Chapter 6. A re-implementation of tasks is planned.

A final enhancement would be the creation of a set of tools to support the determination of execution costs of tasks. Even for the simple stopwatch implementation, the determination of execution time costs was considerably more complex than the actual development of the process code. It is expected that this situation will only worsen with more complex processes.

Vita

Kevin Jeffay was born in Chicago, Illinois on April 8, 1960, the son of Henry and Ana Idalia Jeffay. Following completion of high school in Glen Ellyn, Illinois, he attended the University of Illinois at Urbana/Champaign where he received the B.S. degree with Highest Distinction in Mathematics in May 1982. From 1982 to 1984, he attended the University of Toronto in Toronto, Ontario, Canada, and received the M.Sc. degree in Computer Science in November 1984.

Kevin Jeffay is currently Assistant Professor of Computer Science at the University of North Carolina at Chapel Hill.