MIT/LCS/TR-297

FUNDAMENTAL DESIGN PROBLEMS OF DISTRIBUTED SYSTEMS FOR THE HARD-REAL-TIME ENVIRONMENT

Aloysius Ka-Lau Mok

May 1983

This blank page was inserted to preserve pagination.

FUNDAMENTAL DESIGN PROBLEMS OF DISTRIBUTED SYSTEMS FOR THE HARD-REAL-TIME ENVIRONMENT

by

Aloysius Ka-Lau Mok

S.B., Massachusetts Institute of Technology
(1977)

S.M., Massachusetts Institute of Technology
(1977)

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May, 1983

© Massachusetts Institute of Technology 1983

Signature of Author ..... Aloysius Mok
Department of Electrical Engineering and Computer Science,
May 1983

Certified by .....
Thesis Supervisor

Accepted by .....
Chairman, Departmental Committee on Graduate Students

FUNDAMENTAL DESIGN PROBLEMS OF DISTRIBUTED SYSTEMS FOR THE HARD-REAL-TIME ENVIRONMENT

by **Aloysius K. Mok**

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 1983, in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

Abstract

Software designed to function in a hard-real-time environment where strict timing constraints must be met often entails implicit assumptions about a programming language and the underlying system which supports it. Programs which are logically correct, i.e., they implement the intended algorithms, may not function correctly if their assumed timing characteristics are not met. This can occur if the programming language is not expressive enough to permit an adequate specification of the desired timing characteristics of the software or if the expressible timing characteristics cannot be verified before run time. For distributed systems in particular, the software must be tailored to a myriad of implementation parameters, e.g., communication bandwidth, thus rendering subsequent modifications hazardous.

Our research investigates the basic problems in automating the design and maintenance of hard real-time software. After examining the limitations of the traditional approach to real-time software design via process-based models, we shall provide a graph-based computation model which is more suitable for expressing the computational requirements of the hard real-time environment. This model is an extension of CONSORT (Control Structure Optimized for Real-Time), an experimental software design system which has been implemented to generate process control application programs from block diagram schemata. While our graph-based model is abstract, it can serve as a useful intermediate representation between textual requirements specifications and target application programs. Using the graph-based model, the complexity of the relevant resource allocation problems for meeting stringent timing constraints is investigated.

Name and Title of Thesis Superviser:
Stephen A. Ward
Associate Professor of Electrical Engineering and Computer Science

Keywords and Phrases:
hard real-time systems, real-time scheduling, distributed systems, software engineering, design automation, embedded systems

Acknowledgements

I wish to thank my thesis supervisor Steve Ward for providing the anarchistic environment which is the RTS (Real Time Systems) Group of the Laboratory for Computer Science at MIT. His good nature and confidence in my abilities are perhaps the best support that a graduate student can hope for; and I shall always remember the advice: "Don't panic! Sweat" which Steve passed on to me from his own graduate advisor, Professor Michael Dertouzos.

My thesis committee (professors Ward, Dertouzos and Charles Leiserson) must be commended for the efficiency with which they read my thesis. Charles especially has been very gracious despite my unreasonable demands on his time at short notice. Let me hope that the threat, "You will regret it if you become a professor" Charles uttered on an occasion when I was pestering him will not come true.

The members of RTS must be thanked for their warm camaraderie at all hours of the day (and night). In them, especially Chris Cesar, I have found so many wonderful personal qualities that I would do well to emulate. Their company will be sorely missed.

And of course, my family has my deepest gratitude for their moral support. The thought that they are always behind me is sometimes the only thing that keeps me going.

Without the love and motivation of my wife Amy, I might never get out of graduate school. But then she is a psychologist.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

Table of Contents

Chapter 1 Introduction to Hard Real-Time Systems

1.1 Introduction 9
1.1.1 Elimination of Timing Related Bugs 11
1.1.2 More Reliable Man/Machine Interface 15
1.1.3 Improvement in System Throughput 17
1.2 Systems Issues 19
1.2.1 The Maintainability/Efficiency Dichotomy 19
1.2.2 System Integrability 21
1.2.3 Implementation Independence 23
1.3 Software Automation 24
1.4 Review of Past Work 25
1.4.1 Virtual Processor Methodology 26
1.4.2 Processor Sharing Methodology 28
1.5 Synopsis 31

Chapter 2 Design via Traditional Process Models

2.1 Process-Based Models of Computation for the Hard Real-Time Environment 32
2.2 The Scheduling Model and Timing Constraints 35
2.3 Real-Time Process Scheduling 37
2.3.1 Single Processor Scheduling 38
2.3.2 The Deterministic Rendezvous Model 50
2.3.3 The Kernelized Monitor Model 57
2.4 Implications on the Design of Real-Time Languages 66
2.4.1 Incorporation of Performance Objectives into a Real-Time Language 66
2.4.2 Choice of Concurrency Control Mechanisms 69
2.4.3 Scheduling of Indeterministic Constructs 72

Chapter 3 Design via a Graph-Based Model

3.1 Graph-Based Model of Computation for the Hard Real-Time Environment 75
3.2 Decomposition of Design Requirements: an Example 76
3.2.1 Statement of Design Requirements 76
3.2.2 Implementation Environment 78
3.2.3 Summary of Example Process-Based Real-Time Language 79
3.2.4 Decomposition by Timing Constraints 80
3.2.5 Decomposition by Minimizing Interprocess Communication 84
3.2.6 Decomposition by Maximizing Concurrent Processes 89
3.2.7 Comparison of Decomposition Strategies 92
3.2.8 Implications of the Semantic Gap on Software Automation 94
3.3 Definition of a Graph-Based Computation Model 96
3.3.1 Scheduling Problems with the Graph-Based Computation Model 102
3.3.2 Design Constraints on the Run-Time Scheduler 103
3.4 Latency Scheduling 105
3.4.1 Upper Bound on the Length of a Static Schedule 108
3.4.2 Computing Static Schedules for Asynchronous Timing Constraints 121
3.5 Computing Static Schedules for the Graph-Based Computation Model 131

Chapter 4 Design Issues of Distributed Systems

4.1 Resource Sharing in a Distributed System 136
4.2 The Processor Allocation Problem in the Hard Real-Time Environment 137
4.2.1 Another Time Bomb 138
4.2.2 A Robust Formulation of the Processor Allocation Problem 141
4.3 Scheduling Mechanism for the Broadcast Data Bus 147
4.3.1 The Earliest Deadline Algorithm for Bus Scheduling 147
4.3.2 The Distributed Arbitration Hardware 149
4.4 Complexity of the Processor Allocation Problem 153
4.5 Hierarchical Approach to Resource Allocation 157

Chapter 5 Automation of Software Design

5.1 Design System for Hard Real-Time Software 158
5.2 The CONSORT Design System 159
5.3 Implementation Ideas 159
5.3.1 The User Interface 160
5.3.2 Automatic Pipelining of Operations 161
5.3.3 Detection and Queueing of Activation Conditions 162
5.3.4 Dynamic Computation Requirements 164

Chapter 6 Conclusion

6.1 Summary of Thesis 167
6.2 The Domain Specific Model Approach to Software Automation 169
6.3 Avenues for Further Research 171

Appendix 173

Bibliography 180

List of Figures

Figure 2.1 Example of time slice swapping 40

Figure 2.2 Example for adversary argument in theorem 2.2 43

Figure 2.3 The form of any feasible schedule in theorem 2.4 48

Figure 2.4 Example of scheduling constraint imposed by interprocessor synchronization 52

Figure 2.5 Example of a forbidden region imposed by mutual exclusion constraints 59

Figure 3.1 Example control system function block diagram 77

Figure 3.2 Gantt chart for decomposition by timing constraint 83

Figure 3.3 Gantt chart for example of eliminating redundant function calls 83

Figure 3.4 Example of decomposition by minimizing communication and pipelining 86

Figure 3.5 Example Gantt chart for decomposition by maximizing concurrent processes 91

Figure 3.6 Specification of design example in graph-based model 99

Figure 3.7 Simulation of the execution trace "AABCDEE" 111

Figure 3.8 Reduction of SET PARTITION to latency scheduling problem 126

Figure 3.9 The form of any feasible schedule in theorem 3.11 133

Figure 4.1 Timing constraints augmented by transmission operations 143

Figure 4.2 Timing constraints on transmission operations induced by allocating a function to a processor 146

Figure 4.3 Example of distributed contention on a broadcast bus 151

Figure 5.1 The CONSORT design system 160

Index for Lemmas and Theorems

Theorem 2.1	39
Theorem 2.2	41
Lemma 2.3	43
Theorem 2.4	46
Lemma 2.5	54
Theorem 2.6	54
Theorem 2.7	62
Theorem 3.1	103
Lemma 3.2	112
Lemma 3.3	114
Lemma 3.4	114
Theorem 3.5	118
Theorem 3.6	119
Theorem 3.7	122
Theorem 3.8	122
Theorem 3.9	124
Theorem 3.10	127
Theorem 3.11	132
Theorem 4.1	155

Chapter 1

Introduction to Hard Real-Time Systems

1.1 Introduction

A major application of computers has been to control physical processes such as regulating a power plant or manipulating a robot arm. In these applications, the computation required for responding to external events is often repetitive and cannot be delayed beyond certain time limits which are determined by the nature of the physical processes under control. Failure to observe critical timing constraints might bring about castastrophic results with the destruction of physical plants and even loss of lives. Computer systems which must continuously observe critical timing constraints are said to function in a *hard real-time* environment. For these systems, it is not sufficient for the software to be logically correct, i.e., to implement the intended algorithms. The system must also respond in a timely fashion so as to meet stringent timing constraints such as maximum response time or minimum periodic computation rates.

There are many hurdles to cross in meeting this requirement. A serious difficulty is that the actual timing characteristics of software is determined not only by raw processor speed, but also by the sharing policy for scarce resources. For example, the real-time response of a time-shared system depends heavily on the processor scheduling strategy of its operating system. In most high level languages, this dependency is considered as non-essential detail that is to be hidden from the programmer. As a result, the performance of software implemented in these languages becomes sensitive to system resource allocation strategies and outside the control of individual programmers. System reliability is often contingent on a number of implicit assumptions about interface details between a programming language and the run-time system which supports it. More complex resources such as the communication subsystem of multiprocessor

distributed systems further accentuates the problem with the introduction of (sometimes distributed) resource allocation algorithms which are usually inaccessible to the application programmer.

Current practice in designing systems for the hard real-time environment is rather ad hoc. There are few tools to verify that timing constraint specifications can invariably be satisfied. In fact, systems are often built with little provision for guaranteeing that stringent timing specifications can be met. Performance evaluation is accomplished either by stochastic simulation or by actual measurements on prototype testbeds, and system performance is improved by fine tuning certain system parameters. When specifications cannot be met, structural modifications may become necessary so that at least the major performance objectives can be achieved. While this iterative approach seems to suffice for building the less critical data processing systems, it is not suitable for systems which must function in a hard real-time environment. Unless the interactions among different components of a system are well understood and taken into account in the design process, there is no easy way to simultaneously satisfy multiple stringent timing constraints by fine tuning. Furthermore, there is no absolute guarantee that a timing constraint will invariably be met by the use of stochastic simulation or by taking performance measurements for a limited set of load conditions. For this reason, most current systems are better categorized as *soft real-time* in that they lack hard guarantee on vital performance characteristics.

The objective of this thesis is to develop a methodology, i.e., a basis for mechanizing the design and maintenance of software which must operate in the hard real-time environment. While impressive systems have been built for many hard real-time applications, as witnessed by the progress in space exploration, problems in the reliability and maintainability of current systems, mostly soft real-time, are far from being solved. For example, the first flight of the Space Shuttle was delayed by a synchronization error

which was traced to an improbable race condition in the flight control software [GARM 81]. Indeed, the largely undisciplined approach that is current practice in designing real-time systems exacts a heavy price, e.g., a study by B.W. Boehm of TRW Corporation indicated that the life-cycle cost of real-time software products has been three times as much as analytical software products, not to mention the less readily quantifiable but important costs of safety risks to life and property. As a reflection on the state of the art, designers of one complicated system reportedly opted for retaining existing software and instead modified the hardware to accomodate design change requests.‡

In proposing an alternative methodology, however, it is incumbent upon us to justify that the ability to design truly hard real-time systems as pursued in this thesis offers significant advantages over the traditional soft real-time approach. The following points will hopefully convince the reader of the value of the hard real-time approach.

1.1.1 Elimination of Timing Related Bugs

There are two kinds of software bugs that are attributable to timing faults. One kind involves computational events that occur in an improper sequence and results in undesirable system behavior such as deadlocks, lack of safety, e.g., buffer overflow, or violating logical relations on data. These are *relative* timing faults and are identifiable solely by the relative order in which computational events occur, i.e., the absolute time values marking event occurrences are irrelevant. Relative timing faults can be avoided by prohibiting undesirable orderings of events. The other kind of bugs is caused by the violation of some specified stringent timing constraints such as missing a deadline,

‡ The following news item about the F18 aircraft is from *ACM SIGSOFT Engineering Notes, vol. 6, no. 2, pp.1.* "Apparently the effort that has gone into developing and testing the software is so extensive that, when changes are required, it is now preferable to modify the plane to fit the existing software, rather than to modify the software to match the plane."

thereby violating correctness assertions that involve the absolute timing of events. These are *absolute* timing faults and are germane to hard real-time systems. The following is an example of a program which is susceptible to an absolute timing fault.

EXAMPLE

```
/* COMMENT
   This program takes the derivative of a sensor input at nominally 5 seconds
   intervals. The timer has a long enough range so that it never overflows
   during the entire run of the program. The delay command suspends the
   process for an interval at least as long as the argument, but has no effect
   (i.e., a no-op) if the argument is non-positive.
*/

process time_bomb

begin
  variable next_period,current_time,last_time : second;
  variable current_sample,last_sample : sensor_data;
  variable derivative : sensor_data per second;

        do /*COMMENT initialization */
        {       next_period := 5;
                last_sample := 0;
                last_time := 0;
        } od;

loop:   delay(next_period-read_timer());

        do /* COMMENT update the derivative */
        {       current_sample := read_sensor();
                current_time := read_timer();
                derivative := (current_sample-last_sample)
                                / (current_time-last_time);
                last_sample := current_sample;
                last_time := current_time;
        } od;

        next_period := next_period + 5;
        goto loop;
end
```

The above program computes the derivative of a sensor input by taking the difference between successive samples and dividing it by the length of the interval between samples. Following the recommendation of the designers of Ada ® [ADA MAN

80], the program uses a variable delay between samples to prevent cumulative drift in the actual period which would occur had a constant been used for the delay argument. As in Ada, the basic resolution of the timer can be as large as a second and the *delay* command suspends a process for an interval at least as long as the argument, but has no effect (i.e., the command is treated as an no-op) if the argument is non-positive. The nominal sampling period is 5 seconds, but the actual interval between samples may vary because of fluctuations in multiprocessing load.

The hazard here is that a divide-by-zero exception may occur if at transient peak load, a *delay* command actually takes over 10 seconds to complete. The arithmetic exception occurs because at the next time, the derivative may be updated twice without at least 5 seconds in between. If it takes less than a second in real time to compute the derivative twice, then the measured interval between the two updates might be 0 second owing to the limited resolution of the timer.

A cautious programmer would of course provide a check to make sure that the time between two samples is at least one second. However, this program will not cause arithmetic exceptions if one makes the assumption that the program is run on a dedicated processor of uniform speed, i.e., a program with the same input always takes the same time to execute and this holds *even if the processor is too slow to compute a loop in 5 seconds*. This is an especially tempting assumption since it is consistent with the widely held programming principle that sequential processes should be programmed to run on virtual processors. (In light of our example, "real-time programmers" should be warned that a virtual processor may have a not only unknown but also *variable* processor speed.) The important point here is not that a check should be included to prevent an arithmetic overflow, but that a real-time program may depend on implicit as-

---

® Ada is a registered trade mark of the United States Defense Department.

sumptions about the absolute timing characteristics of the run-time system. These assumptions may hold at the time the program is designed, but may be violated later on, e.g., the timing fault in the example may not surface until the processor load reaches a critical point after more computational requirements have been added by maintenance programmers in the field.

To avoid the arithmetic exception, the system must guarantee that the sensor input is sampled once every 5 seconds. This is a hard real-time constraint which cannot be enforced by the **delay** command of **Ada** since there is no upper bound on the delay in real time. In general, soft real-time systems are concerned almost exclusively with relative timing faults which can be avoided by designing software to function without knowledge of actual hardware speed. They do not eliminate absolute timing faults which are left to be dealt with by recovery mechanisms. In contrast, hard real-time systems prevent absolute timing faults by enforcing hard real-time constraints and thereby maintaining correctness assertions that involve the absolute timing of events.

▸ *More coherent allocation of system resources*

The capability to specify and enforce stringent timing constraints provides a mechanism for the designer to control the allocation of system resources to achieve multiple performance goals. This mechanism is different from the run-time optimization mechanisms in traditional operating systems in two significant ways. First, the stringent timing constraints specified in hard real-time systems enables the designer to supply information directly from the application domain whereas the traditional source of information for system optimization is from the observed behavior of application programs. Second, it is possible to carry out deterministic analysis before run time to meet the performance goals of a hard real-time system whereas traditional operating systems use mostly adaptive feedback techniques which are necessarily limited by the predictive

value of the stochastic models employed. In particular, while current stochastic methods are relatively successful in analyzing aggregate system throughput, they are of limited value in estimating or controlling response times.

In practice, simple scheduling disciplines such as round robin or static priority list are often used for resource allocation in the hard real-time environment. While these scheduling mechanisms are simple to implement, they offer only limited and rather inflexible control over response times. Much better control can be achieved by making use of timing constraint specifications. Consider the following example.

A microprocessor is to be used for analyzing data from two sensors. The design requires that data must be collected from sensor A every 20 milliseconds and from sensor B, every 50 milliseconds. It takes 10 milliseconds to process each sample of data from sensor A and 25 milliseconds for data from sensor B. A real-time clock which runs continuously interrupts every 10 milliseconds so that scheduling decisions are made every 10 milliseconds or when the processor becomes idle. It is easy to verify that neither a round robin scheduler (which switches to the next ready task at every scheduling decision) nor a static priority scheduler (which selects the ready task with the highest priority) is sufficiently flexible to satisfy the performance requirements which can in fact be met by always selecting the task with the nearest deadline at every scheduling decision.

The important point here is not what type of scheduling algorithm to use, but that knowledge of the timing constraint specifications is essential to making proper scheduling decisions in the hard real-time environment. It is therefore useful to incorporate precise performance specifications in the design of the system resource scheduler.

1.1.2 More Reliable Man/Machine Interface

Consider a toggle button which a human operator uses to interface with a comput-

er controlling some physical device. In a soft real-time system, the time it takes the computer to respond to a request to turn on/off the device may show considerable variance, depending on the instantaneous work load. In particular, the response time may be much longer under emergency conditions when the work load on the computer is likely to be heavy. Fearing that the button may not have been pushed properly, an operator who has grown accustomed to a relatively fast response in ordinary times will be tempted to push the button again when the response is slow in coming, thus negating the original request which the computer may in fact be processing. This man/machine interface problem will be substantially alleviated if the operator can be assured that the system will invariably respond within a specified deadline. There is experimental evidence which seems to support that

"... increasing the variability of response time generates poorer performance and lower user satisfaction. Users may prefer a system which always responds in 4.0 seconds to one which varies from 1.0 to 6.0 seconds, even though the average in the second case is 3.5." [SHNE 79].

Although the cited experiments pertain to common interactive systems, it is not unreasonable to expect that similar conclusions can be drawn for life critical applications where it is especially important to reduce the probability of operator errors due to panic or confusion. † The capability to specify and enforce response time limits for different operator commands should be valuable for man/machine interface engineering.

† It is interesting to relate an observation that the author made while watching people play a home-brew version of the arcade video game PACMAN on a popular time-sharing system. In this version, the movement of the player through a maze on a CRT was controlled by pushing a set of keys on a standard keyboard. When the system was heavily used, the time it took to respond to a push became highly variable. Since the movement of the pursuing "monsters" and that of the player were not synchronized, it was very difficult to track the player's position in relation to the "monsters". This version of the game was never very popular.

1.1.3 Improvement in System Throughput

A potential bottleneck for distributed systems is the efficiency of interprocess communication. Substantial overhead is often incurred by the overhead in synchronizing the software running on different processors. This overhead can sometimes be reduced if there are guarantees on how fast messages are moved across the communication network. Consider, for example, a communication processor which multiplexes data packets from 10 satellite processors. The transmission delay between the communication and satellite processors is 10 milliseconds, and the communication processor automatically buffers one packet for each of the satellite processors. The transmission length of a packet and the time it takes the communication processor to process an input packet are both 100 microseconds.

If a satellite processor must wait for an acknowledge signal from the communication processor after sending each packet, then the effective bandwidth per satellite processor will be limited by the transmission delay to one packet per 10 milliseconds. This bandwidth can be substantially increased if the communication processor guarantees that a packet delivered to any input buffer in any order will be removed within a maximum time bound. In this case, the communication processor requires $10 \times 100 = 1000$ microseconds to process packets from all of its input buffers. Since the transmission length of a packet (100 microseconds) is shorter, the bandwidth limit per satellite processor is decided by the processing time of the communication processor, resulting in a tenfold increase in bandwidth. This improvement is possible only if the communication software can be tailored to enforce a timing constraint on the processing time of packets. With the anticipated wide use of fiber optics for data transmission, it is worth noting that this optimization technique will be increasingly significant as transmission delay dominates packet length.

In general, we can identify two extreme approaches to controlling distributed com-

putation. On one extreme, the distributed system is coordinated by a single system clock to which all system components are synchronized so that computation progresses by distinct steps as marked by the system clock and that communication is programmed to occur only at specific times. On another extreme, concurrent components of a distributed system are synchronized only when necessary and do so by executing appropriate hand-shake protocols. The former approach requires less communication overhead but is rigid and not very robust, since all system components are designed to progress in lock step at all times. The latter approach is flexible but may be costly in terms of communication overhead, since many acknowledge signals may be required to maintain proper synchronization. The use of stringent timing constraints to establish a weak form of synchronization among system components represents an alternative in the middle.

1.2 Systems Issues

Given that the capability to design hard real-time systems is desirable, our task is then to identify a suitable design methodology. The design problem can be tackled by many approaches. For example, one can select a programming language and provide tools for verifying any program against the timing constraint specfications when it is run on some target hardware configuration. Alternatively, one can dictate a set of software design rules and specify restrictions on resource usage so that there is an efficient algorithm to determine, for any hardware configuration, if some program written to conform with the rules can meet the specified timing constraints.

In general, a well defined methodology must have a model of computation in terms of which the computation requirements of the application domain can be expressed. The function of a methodology is to provide (meta)rules and algorithms for feasibility analysis and to suggest a solution when appropriate. For this purpose, the model of computation must be sufficiently precise so that software tools can be brought to bear to determine the feasibility of a design. A simple measure for evaluating the effectiveness of a design methodology is its efficiency, i.e., the range of stringent design requirements that can be met by adopting the design methodology. There are, however, other systems issues that ought to be considered for judging the effectiveness of a design methodology. These issues are not readily quantifiable but before we formalize our problems, they should be reviewed so that pertinent system objectives will be given proper consideration.

1.2.1 The Maintainability/Efficiency Dichotomy

In practice, there is often a need to modify complex software systems to accomodate hardware updates or, during development, changes in design specifications. Software modifications are especially error prone for hard real-time systems since they

may introduce subtle resource conflicts. For instance, the absolute timing fault shown earlier will not occur if the processor load is sufficiently low, a condition which may hold initially but which may be violated subsequently in the field as more computation is added. The fact that it is possible to invoke an exception with the same input data and no modification to the program itself attests to the hazardous nature of maintaining hard real-time systems. In general, when design changes are made, the instance of the computation model used by the design methodology must be modified to reflect changes in application domain specifications. To evaluate the effectiveness of a design methodology with respect to maintainability, we can examine the costs of modifying the computation model.

A easy solution is to start from scratch and generate a new model from the revised application domain specifications. However, this may not be desirable because of cost considerations, or it may not be possible at all if only incomplete knowledge of the entire system is available because of administrative reasons. For example, consider a system of parallel processes where the only form of interprocess communication permitted in the model is through the use of global variables and where every input device is modelled by a periodically scheduled system process. Two subsystems A and B periodically update their outputs from the input value of a sensor which is computed by a system process and stored in a global variable. Since A updates its output twice as often as B, the period of the sensor process is initially set to that of A. Later, it is found necessary to reduce the update rate of A fourfold because of numerical instabilities. Since the update rate of B is now twice that of A, the period of the sensor process should only be doubled instead of multiplied by four. Clearly, knowledge of subsystem A's specifications alone is insufficient to revise the period parameter of the sensor process. Futhermore, local modifications may have far reaching consequences, e.g., the output of A may be used by other processes, or the sensor process may refer-

ence the output of other processes to determine its own internal parameters such as filter coefficients for signal processing. In general, it may not be easy to determine all the necessary modifications to the computation model and certainly not without global knowledge of the system.

Relative to a fixed class of design changes and a suitably defined metric on the magnitude of a design change, one can presumably measure the effectiveness of a design methodology with respect to maintainability by studying the complexity of the computation required to determine the corresponding modifications to the computation model. However, intuition suggests that maintainability is not free and one must consider the potential loss in efficiency. For instance, the maintainability problem in the above example will be greatly eased if input devices are modelled as monitors [HOAR 74] so that processes A and B can individually request sensor input updates as they are needed. However, an efficiency price is paid in this model in that the sensor must now be sampled to field every individual request of all the calling processes, whereas only one update may be sufficient to simultaneously satisfy multiple requests. This represents redundant work and a waste in computing time.

While maintainability and the control of design complexity are best achieved by using a computation model close to the application domain, efficiency in resource allocation requires more direct access to the available physical resources; these are conflicting objectives. There are two keys to resolve this maintainabilty/efficiency dichotomy. First, the translation of application domain specifications into the computation model ought to be mechanizable. Second, the computation model should permit direct expression of performance objectives for resource allocation purposes.

1.2.2 System Integrability

Owing to the size and complexity of many real-time systems, it is unlikely that a

single designer will be able to attend to all the details of an entire system. Thus an effective design methodology should have provisions for integrating separately designed components into a system to meet global requirements. For hard real-time systems, this objective is complicated by the need to share resources for meeting timing constraints. If each component is itself a hard real-time system, then system integration is achievable only if it is possible to resolve the resource conflicts among the components, and this in turn requires an appropriate quantification, specialized to the hard real-time environment, of the demand for various kinds of system resources. For example, the maximum bandwidth utilization of a shared data bus alone is in general insufficient for characterizing the demand for bus access if transmission delay cannot be ignored. In other words, there is insufficient information to determine whether two hard real-time systems sharing a data bus can function properly if we are given only the bus access rates of both systems.

The effectiveness of a design methodology with respect to system integrability is indicated by the variety of system resources that can be adequately characterized, i.e., given the computational requirements expressed in an instance of the computation model, a decision procedure should exist which determines whether there is sufficient resource of every type to satisfy the requirements. For an ideal design methodology, the computation model should be able to characterize the demanded load on any kind of existing or to-be-invented resource by a hard real-time system. More realistically, we would like our design methodology to be able to integrate systems using conventional hardware, e.g., shared data busses, and the more esoteric VLSI devices such as systolic arrays. We shall attempt to accomplish this objective by characterizing resources uniformly as (possibly distributed) servers capable of meeting some types of stringent timing constraints.

1.2.3 Implementation Independence

For well defined design methodologies, implementation independence is achieved if the computation model does not introduce any artificiality which biases the implementation towards a particular system architecture, i.e., the computation model should not be based implicitly on the availability of certain kinds of hardware support. For instance, a model of parallel processes where interprocess communication is via global variables only presumes the existence of a shared memory or an efficient broadcast facility in general. A more subtle example is the use of acyclic data communication graphs to model industrial feedback controllers. This approach is predicated on the fact that feedback loops can always be broken by storing the feedback information in state variables which are implicitly available to the processing nodes that would have formed a cycle. Such assumptions are unrealistic when, for example, a sparsely connected network of processors with local memories is used for implementation. A parallel processes model which supports message passing is likely to be more appropriate in this case.

It is difficult to justify that a computation model is not biased against or in favor of any particular system architecture. We note, however, that the primary purpose of the computation model is to reduce the often informal application domain specifications into precisely stated computational events which are physically implementable. To this end, we can identify two primitive types of computational events: the functional transformation and transmission of information. A computation model which builds on these two types of primitive events alone is less likely to be biased. Moreover, there is always the possibility of transforming an unbiased model into one which is more amenable to optimization analysis that takes advantage of the special features of a given architecture.

1.3 Software Automation

The above discussion of systems issues leads to a better perspective on our work. Whereas the study of design methodologies has been the key activity in software engineering research and many useful principles have been introduced for managing design complexity, e.g., the use of layers of abstraction for vertical decomposition and the encapsulation of data and procedures in modules for horizontal decomposition, there has been only limited progress in providing a formal basis for the discipline which often resorts to philosophical arguments. (Some related theoretical work is being done, most notably research in abstract data types.) This lack of an encompassing formal framework should not be surprising since an important goal of software engineering is to improve programming productivity which is mostly a human enterprise and as such cannot be subject to formalization. Without formalization, however, it is very difficult to compare the merits of contending methodologies or to evaluate a new design technique. Incremental improvements are limited in scope and more importantly, they do not point to potential trouble spots or areas for further improvement.

A more profitable approach is to judge a design methodology by how much it contributes to software automation, i.e., the success in eliminating the intermediary programmer from the design loop. As the above discussion of systems issues illustrates, a lot of the limitations of a design methodology can be related to the imperfections of the underlying computation model. Thus the usefulness of any automation tool will ultimately depend on the propriety of the computation model on which it is built. Therefore, an important objective of our research is to identify an appropriate computation model for the hard real-time environment before we can tackle some of fundamental resource sharing problems in automating the design of hard real-time systems.

The rest of this chapter will review past research in real-time systems and give a synopsis of our work.

1.4 Review of Past Work

There has been a lot of research in the production of real-time software. We shall list only those that are intended for application in the hard real-time environment. Most work reported in this area falls into one or more of three categories: specification techniques, language concepts, design disciplines and related scheduling techniques. Specification techniques are concerned mostly with the functional completeness and consistency of the application domain descriptions of real-time systems. Examples are [HAM & ZEL 76], [ALFO 77]. The specification of stringent timing constraints is treated in some detail in [DEW & PRI 77], [COHE 78]. [HENI 80] describes a complete specification of the avionics software of the A-7 aircraft including timing specifications. Some commercial languages have been augmented by scheduling primitives to support real-time programming, e.g., PL/1 [BARN 79], Fortran [KNEI 81], and a number of new languages have been designed specifically for real-time applications, e.g., Tomal [HEN et al 75], Pearl [MART 78], Iliad [KRUL 81]. The best known language of this genre is probably Ada [ADA MAN 80]. Some authors have investigated special language concepts for concurrent processes to facilitate real-time processing, e.g., [HANS 78a], [ICH et al 79], [MAO & YEH 80].

For our purposes, the cited literature in specification falls short of defining a computation model which can be mechanically processed for feasibility analysis. Further translation is needed to associate timing constraints with the concrete computation that needs to be carried out, and it is not clear that the specification techniques provide complete and consistent information for this necessary step. However, the mentioned work, especially [HENI 80] provides useful examples of application domain specifications. The work in language concepts almost exclusively assumes a process-based model. While innovative concepts for process coordination have been invented, the use of processes as the syntactic unit for specifying performance requirements in-

troduces artificialities which will become clear when we examine the problem of translating application domain specifications into computation models.

The work that is most closely related to our research are the design methodologies that have been proposed for writing real-time programs. Some of these methodologies are of limited use for hard real-time applications in that they do not discuss the problem of verifying compliance to timing constraint specifications. Almost all are process-based and can be roughly categorized as adopting one of two approaches.

1.4.1 Virtual Processor Methodology

In this approach, each process is presumed to be running on a dedicated processor. The objective is to guarantee bounded completion time for all required computation. To this end, the designer needs to guarantee that no deadlock can result from the control structure of the program and that no part of the computation will be denied progress indefinitely, i.e. no live locks, provided that all resource schedulers are in some sense fair. The problem of verifying compliance to timing constraint specifications is deferred and must be solved for the actual scheduling strategy in an implementation, presumably by computing the worst case bounds for all completion times and comparing them against the specifications.

The viability of this approach is based on the availability of cheap computing power such as multiprocessors on a chip, an assumption buttressed by the promise of advances in VLSI technology, so that as long as all the computation can be carried out in finite time, then the timing constraint problems should be solved, if necessary, by adding another processor. By the same token, scheduling is necessary only for the purpose of process activation and suspension on individual processors which are not shared. This approach requires minor extensions to existing high level languages to support explicit process scheduling and some mechanism for interprocess communica-

tion and coordination. An example of this approach is [YAU et al 81] where the time bounds are computed by assuming round robin scheduling for all resources.

Wirth [WIRT 77] proposed a discipline for real-time programming which adopts the conceptual simplicity of the virtual processor methodology but relies on priority interrupts as an essential means for achieving response times for I/O devices. Cyclic timing constraints are imposed on device processes which are assigned static priorities. Furthermore, it is assumed that high priority devices have proportionately longer cycle times than low priority devices in order that all response times are met. The exact condition for verifying compliance to timing constraints is, however, not given.

The virtual processor methodology is viable only if the premise about resource availability is valid, and only up to the point that interprocessor communication does not become a bottleneck. Even if there are enough processors so that each process can be run on a dedicated processor, timing constraints may still be violated because of the communication delay between processors. In general, there is a tradeoff between communication (routing delay) and computation (scheduling and context switching overhead) costs. For more demanding applications, the simplistic approach of assigning one process to one processor may not work. The basic problem of balancing computing and communication costs cannot always be ignored.

The communication bottleneck of the virtual processor methodology has been recognized by Hansen [HANS 78b] who argued that a hierarchical organization of processors will reduce communication overhead and is also a natural organizational structure for the solution of many practical problems. While a hierarchical organization may be a good match for the solution strategy of many problems, its generality is limited. We note, in particular, that if we associate a process with each state variable in the standard state space formulation of control engineering problems, then the interprocess communication pattern in the solution to the control problem is in general closer to a

densely connected graph than to a tree.

Some attempts have been made to formulate the processor allocation problem in the presence of stringent timing constraints, e.g., [CHU et al 80], [MA et al 82]. However, none of the published formulations seems to be satisfactory since all of them are formulated with algebraic constraints on mean value parameters such as communication bandwidth and processor utilization factor. These parameters are in principle derivable from the process model, but unfortunately, they are more useful for average time rather than worst case analysis. The bursty nature of many real-time applications is such that there is no guarantee that individual timing constraint will be met even if the average load does not exceed either processor or communication capacity. In this thesis, we shall provide a formulation of the resource allocation problem that is more amenable to the requirements of the hard real-time environment.

1.4.2 Processor Sharing Methodology

In this approach, processes are expected to be sharing resources subject to known scheduling disciplines and usage restrictions which are part of the design methodology. Since the scheduling disciplines are fixed, the run-time behavior of the system is predictable and this offers potential for exploitation. First, the designer need not guarantee that a process system is inherently deadlock-free or may have other undesirable properties; it is sufficient that the system will function properly under the given resource scheduling disciplines. For example, the system scheduler might require that all shared resources be explicitly declared and apply banker's algorithm for resource allocation. Second, there may be (intentional) asymmetry in the resource scheduling disciplines which can be exploited to favor processes with tighter timing constraints. This can be facilitated by introducing scheduling attributes, e.g., deadlines

for interfacing with the system scheduler.

Some examples of the processor sharing methodology are found in [WEI et al 80] and in [REG et al 78]. In both cases, processes are assigned deadline and period attributes and are run on a single processor. The scheduling disciplines used are variations of the *earliest deadline* algorithm which always runs the ready process with the nearest deadline, or the *rate monotonic static priority* algorithm which assigns higher (fixed) priorities to processes with higher repetition rates. In [LEIN 78], processes may also contain *device segments* (I/O delays) and *resource segments* (critical sections) which are scheduled on a FIFO basis.

An earlier methodology was proposed by Dertouzos [DERT 74] where process-like *daemons* are given boolean conditions for their activation. Conditions must be recognized within certain recognition times and the daemons must complete their computation within given deadlines. Daemons are supposed to be implemented on a multiprocessor architecture with global memory. The earliest deadline algorithm is used for daemon scheduling on each processor and (ignoring scheduling and context switch overheads) has been shown to be optimal [DERT 74] in the sense that it always results in a feasible schedule if one exists. However, it is no longer optimal when there are mutual exclusion restrictions or when there are two or more processors.

It should be noted that if we can find an optimal scheduling algorithm for every resource, then there is no reason why the processor sharing methodology should not be adopted with the optimal scheduling disciplines. However, it can be shown [MOK & DER 78] that when processes may request service with no *a priori* known request times, then not all feasible sets of timing constraints can be met by any one multiprocessor scheduling algorithm.

In general, the processor sharing methodology makes better use of system resources than the virtual processor methodology. However, there may not be any easy

way to systematically take advantage of the resource scheduling disciplines adopted by a processor sharing methodology. For example, while there is an efficient assignment of static priorities to processes when every timing constraint can be expressed in terms of a deadline on a periodic process †, there is a significant semantic gap between static process priorities and timing constraints that involve the cooperation of two or more processes to satisfy. However, the use of a static priority as a process scheduling attribute has been adopted by many "real-time languages", most notably **Ada.**

† The combinatorial problem of static priority assignment has been treated in [LIU & LAY 73]. The proof of theorem 1 in the cited paper is, however, not quite complete since it does not deal with the case when a task of higher priority is being executed at the request time of the task being analyzed for its *critical instant* [CESA 80], but the theorem and the priority assignment procedure given are correct.

1.5 A Synopsis

In this chapter, we have argued that there are significant advantages in building hard real-time systems and that we need a better understanding of the fundamental problems for automating the design and maintenance of these systems. Since the obvious way to organize the software for these systems is via straightforward extensions to process-based models, we shall formulate in chapter two the problems of designing hard real-time software in terms of process-based models and discuss some techniques for on-line scheduling. There is, however, a serious semantic gap between process-based models and hard real-time applications. After discussing the intrinsic weaknesses of process-based models, we shall introduce in chapter three an alternative graph-based model (a generalization of the CONSORT block diagram schemata [WARD 78]) on which stringent timing constraints can be naturally expressed. This model is machine independent and makes no assumption about the availability of physical resources. Relevant problems in resource scheduling with critical timing constraints can then be formulated and studied. A technique called latency scheduling which can be used to meet asynchronous timing constraints (spontaneous service requests) by shifting most of the work off-line will be formalized and studied in terms of the graph-based model. Chapter 4 presents a general approach to model the constraints imposed by communication delays in a distributed system so as to provide a precise formulation of the processor allocation problem for the hard real-time environment. We shall also show how communication delay constraints can be met by solving analogous deadline scheduling problems for a broadcast communication system such as a backplane bus. Chapter 5 describes the architecture of a design system and also some implementation ideas. Chapter 6 is the conclusion and remarks about avenues for further research.

Chapter 2

Design via Traditional Process Models

2.1 Process-Based Models of Computation for the Hard Real-Time Environment

Our objective in this chapter is twofold: to examine the use of process-based models for the design of hard real-time systems, and to introduce on-line scheduling techniques for meeting stringent timing constraints. There are two incentives for examining process-based models. First, they represent a conservative extension of the vast majority of sequential programming languages in providing a parallel processing capability. Design techniques for process-based models are therefore of considerable practical interest. Second, by investigating the problems with process-based models, we can gain a better appreciation of the computational requirements of the hard real-time environment. Our investigation will also provide justifications for an alternative computation model which is more suitable for expressing performance requirements in the hard real-time environment.

Informally, a (*sequential*) *process* is an abstraction of a single sequence computer which, at any point of a computation, can be characterized by its state information, namely, the program counter, the stack and the value of its static variables. A process interacts with another process by exchanging information only at specific points in its program and has only one thread of control, i.e., the program counter of a process is determined solely by the result of its own computation and cannot be set directly by asynchronous external events. (Thus interrupts are a processor-related concept and are logically transparent to a process.) A variety of process-based models for parallel processing have been proposed, e.g., communicating sequential processes [HOAR 78], distributed processes [HANS 78a], the tasking model of Ada [ICH et al 79]. These models differ primarily in the mechanisms they provide for interprocess communication

and in the amount of internal parallelism (coroutining) inside a process. We shall concern ourselves with only those problems about process-based models that are relevant to the design of hard real-time systems.

▸ *Decomposition of computational requirements*

Given the performance specifications of a hard real-time system, the problem is how to decompose the required computation into processes with the appropriate timing constraints so as to meet performance and other system objectives.

▸ *Process scheduling*

Given a set of processes with timing constraints, the problem is how to schedule them at run time so as to meet their timing constraints.

▸ *Adequacy for concurrency control*

Resource sharing in the hard real-time environment requires facilities for concurrency control. The problem is to determine the appropriate interprocess coordination mechanisms which support control structures commonly required for hard real-time systems.

It should be emphasized that the above problems are not independent of one other. In particular, many simple primitives for concurrency control are so powerful that their unstructured use may cause undesirable system behavior which is very difficult to analyze. The corresponding scheduling problem is likewise computationlly intractable. On the other hand, scheduling algorithms which are efficient to analyze and implement may permit so little concurrency control that they are adequate for only a very limited class of problems. We shall provide some answers to the above problems by analyzing a sequence of increasingly sophisticated process-based models. Our investigation will also reveal some semantic weaknesses and possible improvements to current process-

based "real-time languages" such as Ada.

2.2 The Scheduling Model and Timing Constraints

For scheduling purposes, a process T_i consists of a chain of *scheduling blocks*, $\{ T_{i,j}, j=1,n_i \}$ where $T_{i,j}$ is a piece of code to be executed after $T_{i,j-1}$. Each $T_{i,j}$ has a bound on computation time, $c_{i,j}$ which is known *a priori* and the sum of the $c_{i,j}$ is the total computation time, c_i of process T_i. Interprocess coordination is achieved through communication primitives which may appear only between scheduling blocks. These communication primitives are used to pass information among processes or for coordination purposes. Their semantics is important only to the extent that they impose certain scheduling restrictions which will be defined for each process model.

When a process is made ready to run, say at time t, it must be finished by a specified deadline, d_i relative to t, i.e., the last scheduling block of T_i must complete execution on or before $t+d_i$. There are two types of processes: *periodic* and *sporadic*. If T_i is periodic, it is requested (becomes ready to run) every p_i time units, starting from time 0. The deadlines of periodic processes are normally shorter than the corresponding periods. If it is sporadic, then it may be requested at any time, but consecutive requests of T_i are kept at least p_i time units apart, where p_i is a specified minimum period which is required to prevent a sporadic process from monopolizing system resources. (In practice, sporadic processes are often invoked by external events, e.g., device interrupts. The minimum period restriction may be enforced by keeping a queue of pending requests for T_i which is made ready every p_i time units until all pending requests are exhausted.)

Formally, an instance of a process model $M = M_p \cup M_s$ is a finite set of processes which is the union of two disjoint subsets: M_p (the periodic processes) and M_s (the sporadic processes). The i^{th} process, $T_i = (c_i,p_i,d_i)$ has three parameters: c_i (computation time), d_i (deadline), p_i (period) such that $c_i \leq d_i \leq p_i$. If a process $T_i \in M_p$, then it is requested at time $= kp_i$ for every non-negative integer k. If on the other hand, $T_i \in$

M_s, then it can be requested at any time instant t, but two successive requests must be at least p_i time units apart. All time parameters are non-negative integers. (In practice, time parameters are presumably given in integral multiples of a basic time unit, e.g., a processor instruction cycle.) Process preemptions are allowable only at integral time instants and may be subject to additional scheduling restrictions imposed by communication primitives placed between scheduling blocks of a process. A set of processes is schedulable if at a request-time t, the process (c,p,d) requested is executed completely on a processor for c units of computation time in the interval $[t,t+d]$.

2.3 Real-Time Process Scheduling

Although much progress has been made in deterministic scheduling theory in the last decade, the classical scheduling model studied by most authors (e.g., [LAG et al 81] contains a complexity classification of deterministic scheduling problems based on a parameterized model) deals with tasks that are to be performed only once, i.e., each task has a given request-time after which it can be scheduled, and a task is never considered again after it has been completed. These results are not directly applicable to our problem which differs from them in two essential aspects: our tasks (processes) may be invoked an infinite number of times, and the request-times of sporadic processes are not known *a priori*.

Nevertheless, we can use some of the techniques developed for the classical model if all processes are periodic in which case it is sufficient to examine schedules of length on the order of the least common multiple of the periods. Although algorithms following this approach are at best pseudo-polynomial in complexity, they suffice in many practical applications since the parameters involved (deadlines, periods) are neither expected to be very large integers nor relatively prime. However, it should be noted that complexity results derived for the classical model cannot be carried over directly to our model owing to the periodicity of processes. Specifically, a necessary condition for scheduling in our model is that the sum of the *utilization factors*, c_i/p_i of all the processes must not be greater than the number of available processors. Difficult problems in the classical model may not be computationally intractable when they are restricted to the subsets which meet the utilization constraint. To emphasize the difference between the classical deterministic scheduling problems and our problems, the problems of continuously meeting periodic and sporadic timing constraints in real-time will be called *real-time scheduling* problems. Algorithms which solve real-time

scheduling problems are real-time scheduling algorithms.

In general, a real-time scheduling problem involves two schedulers: an off-line scheduler and a run-time scheduler. The *off-line* scheduler examines the instance of the process model and creates a run-time scheduler together with a database for making scheduling decisions at run time. The run-time scheduler is the code for allocating resources in response to requests generated at run time, e.g., timer or external device interrupts. The purpose of a real-time scheduling algorithm is to create an off-line scheduler for a class of real-time scheduling problems. A run-time scheduler is *totally on-line* if its decisions do not depend on *a priori* knowledge of the future request-times of the process(es). A run-time scheduler is *clairvoyant* if it has an oracle which can predict with absolute certainty the future request-times of all processes. A run-time scheduler is *optimal* if it always produces a feasible schedule whenever it is possible for a clairvoyant scheduler to do so.

2.3.1 Single Processor Scheduling

For the case of a single processor, Dertouzos [DERT 74] showed the existence of a totally on-line, optimal run-time scheduler for the case where interprocess communication primitives do not impose any scheduling restrictions, i.e., the scheduler can choose to preempt a process by any other ready process at integral time instants. The algorithm embodied in the run-time scheduler is the *earliest deadline algorithm* which runs at every instant the ready process with the nearest deadline. It is interesting to note that there are more than one totally on-line optimal scheduler under the same assumption. Let us denote the remaining computation of a ready process at time t by c(t) and its current deadline by d(t) and define the *slack* of the process at time t by maximum{d(t)–t–c(t),0}, i.e., the slack is the maximum time the run-time scheduler can delay running the process before it is bound to miss the current deadline. Another to-

tally on-line optimal algorithm is the *least slack algorithm* which schedules at any time the ready process with the least slack, ties being broken arbitrarily. The optimality of the least slack algorithm can be proved by the same "time slice swapping" technique used in [DERT 74].

Theorem 2.1

The least slack algorithm can be used as a totally on-line optimal run-time scheduler under the assumption that the scheduler can choose to preempt a process by any other ready process at any integral time instants.

Proof:

By definition, the least slack algorithm is totally on-line. The key observation is that at any time t, we can always transform a feasible schedule for the interval [0,t] (say one that is produced by a clairvoyant scheduler) to one that is produced by the least slack algorithm without missing a deadline within that interval. This is trivially true at time $t=0$. Suppose the hypothesis holds for [0,t], and a process T_i is scheduled in the interval $[t,t+1]$ while there is another ready process T_j with a smaller slack at time t. Notice that the process T_j must be scheduled at least once before d(t), the *current* deadline of process T_i in the feasible schedule. Otherwise, the slack of process T_j must be at least as big as d(t)–t which is greater than the slack of process T_i, a contradiction. Thus, we can simply schedule process T_j in the interval $[t,t+1]$, and schedule process T_i in the first unit interval occupied by process T_j before the deadline d(t). The resulting schedule remains feasible. QED

Figure 2.1 gives an illustration of the "time slice swapping" technique involving two processes: T_1 with computation time $c_1=1$, deadline $d_1=4$, and T_2 with $c_2=3$, $d_2=5$. The earliest deadline algorithm schedules process T_1 first whereas the least slack algorithm would schedule process T_2 first.

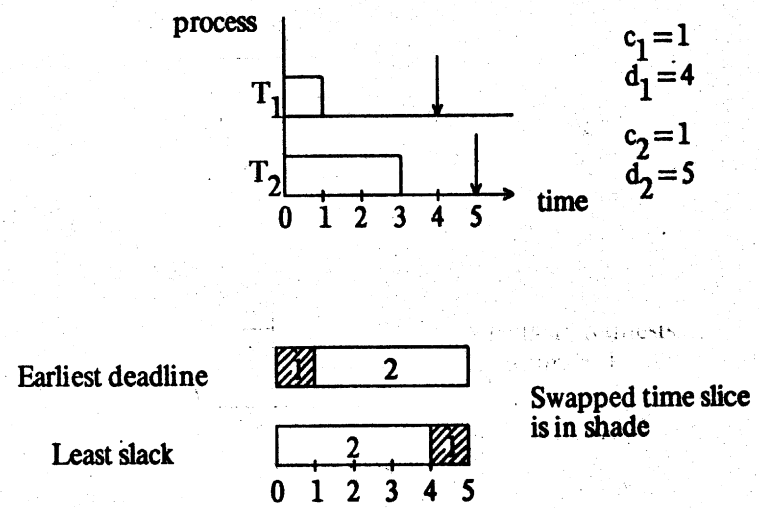


Figure 2.1
Example of time slice swapping

Corollary (Liu and Layland)

For the case where the deadline and period are equal for the same process, (i.e., $d_i = p_i$ for every process T_i), a necessary and sufficient condition for scheduling a set of all periodic processes on a single processor is that $\Sigma\, c_i/p_i \leq 1$ where c_i is the computation time of the process T_i.

Proof:

This result was first proved in [LIU & LAY 73]. The technique of "time slice swapping" used in the above theorem provides a simpler way to prove the same result. Specifically, we note that if a round robin scheduler allocates c_i/p_i of every time unit to process T_i, then T_i will be guaranteed to receive c_i units of processor time in every period of length p_i, thus meeting its deadline. Since preemptions are permitted only at integral instants of the basic time unit, we have to transform the schedule produced by the round robin scheduler to one in which process switching occurs only at integral instants of time. This is easily done by using an optimal scheduler such as the earliest

deadline or the least slack algorithm and not allowing the processor to stay idle whenever there is an unfinished ready process. Since all request-times, computation times and deadlines are integral, the "time slice swapping" technique will yield a schedule in which process switching occurs only at integral instants of time. QED

Remark

There are in fact an infinite number of totally on-line optimal schedulers, e.g., any combination of the earliest deadline and the least slack algorithm may conceivably be used in a run-time scheduler to minimize process switching overheads.

The assumption that any ready process can be freely selected to preempt another process poses a cubersome restriction on the design of some real-time software. For example, the position of an aircraft is updated by a periodic process which computes the X and Y coordinates from sensor measurements. A sporadic process may read the X value before being preempted by the tracking process and then reads the new Y value. This inconsistency can be prevented by enforcing a *mutual exclusion constraint* on the two processes, i.e., they are not allowed to preempt each other. With this restriction on the real-time scheduling problem, however, the earliest deadline algorithm is no longer optimal. In fact, we can show that in general, a run-time scheduler cannot be optimal unless it is clairvoyant.

Theorem 2.2

When there are mutual exclusion constraints, it is impossible to find a totally on-line optimal run-time scheduler.

Proof:

Consider the following instance of a process model with two mutually exclusive processes. T_p is a periodic process with computation time $c_p = 2$, deadline $d_p = 4$, and

period $p_p = 4$. Process T_s is a sporadic process with computation time $c_s = 1$, deadline $d_s = 1$, and minimum period $p_s = 4$. Let us examine the problem of scheduling them on a single processor. We can always meet the deadline specifications by using a clairvoyant scheduler as follows. At every instant t when process T_p is requested, (i.e., $t = 0 \mod 4$), schedule T_p for the interval $[t,t+2]$ if the oracle claims that process T_s will not be requested at $t+1$. Else defer running process T_p to the interval $[t+2,t+4]$. Schedule process T_s immediately whenever it is requested. (Figure 2.2 illustrates the situation where process T_p is scheduled at time 0 and the adversary requests process T_s at time 1.)

For any totally on-line scheduler, we can prove that it is not optimal by giving an adversary argument. Specifically, at any instant $t<4$, a decision must be made either to run process T_p in the interval $[t,t+2]$ or to defer it. If process T_p is scheduled at time t, then a request for process T_s at time $t+1$ cannot be met since T_p cannot be preempted at time $t+1$. If the decision is deferred, then the adversary will not request process T_s at time $t+1$, and the scheduler is forced to make the decision again. Since the scheduler cannot defer running process T_p past time $t=2$, it is bound to miss one of the deadlines. QED

Remark

This theorem can be generalized trivially to the case of multiprocessors by creating for each additional processor, a periodic process whose deadline and computation time are both equal to its period. More significantly, it can also be proved [MOK 76] that for the multiprocessor case, it is impossible to find a totally on-line optimal run-time scheduler even if any ready process is permitted to preempt any other process in progress.

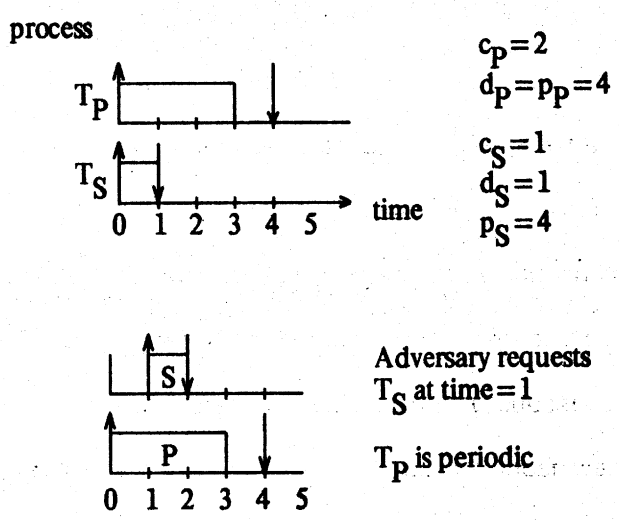


Figure 2.2
Example for adversary argument in theorem 2.2

The negative result above certainly does not imply that a clairvoyant scheduler is required for ALL problem instances. One might suspect that the process model concocted for the proof puts too heavy a load on the single available processor. However, it is easily seen that the proof holds even if we set the period of both processes to any integer bigger than 4. Thus an optimal scheduler may not exist even if the processor utilization factor is kept arbitrarily small. The crux of the problem lies in the relative urgency of the processes.

The following lemma shows that a clairvoyant scheduler is not necessary for scheduling a set of sporadic processes if they can be replaced by a set of "equivalent" (in the sense of the following lemma) periodic processes for which a feasible on-line scheduler exists. For this purpose, let us define the *nominal slack*, l_i of a process T_i to be $l_i = d_i - c_i$ where d_i and c_i are the deadline and computation time of process T_i.

Lemma 2.3

Let $M = M_p \cup M_s$ be an instance of a process model. Suppose we replace every

sporadic process $T_i = (c_i, p_i, d_i) \in M_s$ by a periodic process $T'_i = (c'_i, p'_i, d'_i)$ with $c'_i = c_i$, $p'_i = MIN(l_i + 1, p_i)$, and $d'_i = c_i$. If the resulting set of all periodic processes M' can be successfully scheduled, then the original set of processes M can be scheduled without *a priori* knowledge of the request-times of the sporadic processes in M_s.

Proof:

If the periodic processes in M' can be scheduled, then a run-time scheduler can repeat the finite schedule for the interval [0,L] *ad infinitum* to meet all future deadlines where L is the LCM (least common multiple) of the periods. We can modify this scheduler to schedule the original processes as follows. At a request-time of a process T_i, if $T_i \in M_p$, then schedule it by following the recurring schedule above. If the process $T_i \in M_s$, then schedule it at the earliest time when an instance of its equivalent periodic process is next run on the recurring schedule. This modified scheduler does not use *a priori* knowledge of the request-times of the sporadic processes and it remains for us to show that the sporadic processes will not miss their deadlines.

The worst case occurs when the request-time of a sporadic process T_i occurs one time unit after the latest instance of its equivalent periodic process has started running. By the definition of the transformation, the next request-time of process T'_i occurs at the most $l_i + 1 - 1 = l_i$ time units later, and the next instance of process T'_i is completed c_i time units after that. Hence, the sporadic process is completed at the most $l_i + c_i = d_i$ time units after its request time. QED

Remark

The above scheduler makes use of a database (the finite schedule) which is computed off-line by exploiting the fact that the request-times of periodic processes are known *a priori*. Thus it is a question of philosophy whether to call the scheduler totally on-line or not. We would choose to reserve the term *totally on-line* for schedulers

which do no make use of any *a priori* knowledge of request-times.

In general, we can replace a sporadic process T with computation time c, deadline d and minimum period p by a periodic process T' with computation time c', deadline d' and period p' as long as the following conditions are satisfied: (1) $d \geq d' \geq c$(2) $c' = c$;(3) $p' \leq d - d' + 1$. This is an example of a general technique called *latency scheduling* which is used to schedule sporadic computation by exploiting periodic computation and which will be investigated in detail in the next chapter.

Intuitively, a sporadic process is more demanding (i.e., it makes it more unlikely to find a run-time scheduler which is not clairvoyant for the problem instance) the shorter its nominal slack is, and this is reflected in the shorter length of the period, $p'_i = l_i$ of its "equivalent" periodic process. However, it is important to point out that the transformation used in the above lemma is not unique. In fact, there are many ways to transform a sporadic process into equivalent periodic process(es). We shall defer the scheduling problem of sporadic computation until the next chapter and deal with only periodic timing constraints in the rest of this chapter. We shall also concentrate on the single processor case since a discussion of the multiprocessor case is unrealistic without addressing the related problem of interprocessor communication which we shall address in a later chapter.

When there are no restrictions on selecting a process for preemption, we have already remarked that the feasibility problem of scheduling a set of periodic processes whose deadlines and periods are equal can be solved efficiently. In general, we can test for feasibility by using an optimal scheduler to simulate the execution of the periodic processes over a sufficiently long interval. (If the deadlines are all shorter than the periods, then the simulation interval needs no longer than the LCM of the periods.) We now turn to the issue of interprocess coordination which is to be supported by the

communication primitives. We have noted that the ability to enforce mutual exclusion constraints is important to some real-time applications. Our next task is then to select the communication primitives which are sufficiently powerful for our purposes and study their implications on the scheduling problem. In general, a communication primitive may be used to coordinate parallel activities of concurrent processes and thus put some restrictions on the run-time scheduler by disallowing a subset of of the schedules which the scheduler may otherwise generate. Perhaps the most well known mechanism for interprocess coordination is the use of *semaphores* which are known to have wide applications, e.g., it can be used to enforce mutual exclusion and precedence constraints. (The *spinning lock* implementation of the P operation is inappropriate in the hard real-time environment. One can assume that processes will be blocked and queued at a semaphore when they cannot proceed.) The natural question to ask is how difficult the scheduling problem becomes when P and V operations are permitted to delineate the scheduling blocks of a process. Unfortunately, the problem of scheduling a set of periodic processes to meet their deadlines is NP-hard even if semaphores are used to enforce mutual exclusion only (i.e., each P(x) operation must be followed by a V(x) operation and every semaphore is initialized to permit only one process to proceed.) Our proof is a straightforward modification of the NP-completeness proof of the SEQUENCEING WITHIN INTERVALS problem [GAR & JOH 79, pp. 102-103] which uses the well known NP-complete 3-PARTITION problem.

Theorem 2.4

The problem of deciding whether it is possible to schedule a set of periodic processes which use semaphores only to enforce mutual exclusion is NP-hard.

Proof:

We shall transform an instance of the 3-PARTITION problem to an instance of our

scheduling problem as follows. Let $A = \{ a_1, a_2, ..., a_{3m} \}$ be a set of 3m elements, B a positive integer, and $w_1, w_2, ..., w_{3m}$ be integral weights of the elements of A respectively such that $B/4 \leq w_i \leq B/2$ and $\Sigma w_i = mB$. The decision question is whether A can be partitioned into m disjoint sets each of which has weight (the sum of the weights of its elements) B.

The corresponding instance of our scheduling problem has $3m+1$ processes all of which have the following form.

```
Process T_i

Attribute period = p_i, deadline = d_i
P(x)
{ Scheduling block which together with the P and V operations takes c_i
time units }
V(x)

end T_i
```

For each element a_i in A, we create a process T_i with $p_i = d_i = mB + m$ and $c_i = w_i$. In addition, we create a process T_{3m+1} with $p_{3m+1} = B + 1$ and $d_{3m+1} = c_{3m+1} = 1$. This transformation obviously takes polynomial time. Notice that all feasible schedules must run process T_{3m+1} in the intervals $[t, t+1]$ where $t = 0$ (modulo $B+1$) since T_{3m+1} has 0 nominal slack. This leaves m separate slots of time each of which has length B in the interval $[0, mB+m]$. There can be no idle time in any of these slots since the m processes corresponding to the elements of A must be executed once before $mB+m$. Furthermore, none of these m processes can appear in more than one slot, otherwise process T_{3m+1} would have been blocked from running by the semaphore. (Figure 2.3 illustrates the form which any feasible schedule must take.) Thus a feasible schedule exists for the interval $[0, mB+m]$ iff the 3-PARTITION problem can be solved. The scheduling problem obviously cannot be solve if there is no feasible schedule for $[0, mB+m]$. If there is one, then a run-time scheduler can simply repeat

this finite schedule *ad infinitum* to meet the deadlines. Hence our scheduling problem is at least as hard as 3-PARTITION. QED

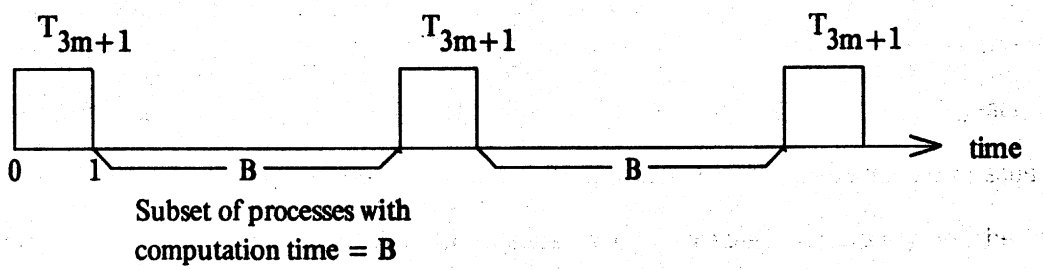


Figure 2.3
The form of any feasible schedule in theorem 2.4

Thus there is strong evidence to support that insisting on finding a run-time scheduler whenever one exists can be prohibitively expensive. One alternative is to use suboptimal algorithms. (It should be emphasized that the run-time scheduler must be guaranteed not to miss any deadline. Sub-optimal algorithms are permissible only for off-line computation.) Another alternative is to put as many restrictions on the use of the communication primitives as it is deemed reasonable for programming real-time systems and hope that the restricted scheduling problem can be efficiently solved. The simple form of the periodic processes in the NP-hardness proof above seems to suggest that not much more can be done by way of restricting the use of the P and V primitives. The reason for the "NP-hardness" of the above scheduling problem lies in the possibility that there are mutually exclusive scheduling blocks which have different computation times. (These mutually exclusive scheduling blocks are similar to the *critical sections* of Dijkstra who originally defined a critical section as a single code segment which is shared by two or more processes such that at most one process may be executing it at any time.) As we shall show later, the scheduling problem becomes a lot more tractible if mutually exclusive scheduling blocks must have the same computation

time.

In general, interprocess coordination by means of semaphores is far too unstructured for our analysis, e.g., the same semaphore may be used to enforce mutual exclusion sometimes and to enforce a precedence constraint at other times. The complexity of the corresponding scheduling problem will very easily get out of hand. While we can impose conventions to structure the use of semaphores and thus keep the analysis more manageable, the availability of a global memory implied by the use of semaphores imposes an architectural constraint which is hard to justify if we are to apply our results to distributed systems. For these reasons, we shall adopt a communication primitive which is closer to message passing among processes. Specifically, we shall allow a process to *rendezvous* (borrowing the terminology from Ada) with another process.

2.3.2 The Deterministic Rendezvous Model

The rendezvous communication primitive has the following syntax and may be used to delineate scheduling blocks within a process.

rendezvous (process_name).

For brevity, we define processes which have rendezvous primitives targetting each other to be *communicants*. (This definition induces a *communicant relation* on the set of processes.) When a process T_i attempts to execute a rendezvous primitive, it must wait until the corresponding communicant also executes a rendezvous primitive with T_i as its argument. Presumably, information may be exchanged by the two processes at a rendezvous, but the nature of the exchange is not of interest to us. The primary purpose of the rendezvous primitive is for synchronizing two processes. More specifically, a rendezvous establishes a precedence constraint which requires that all the computation before the rendezvous primitive in each process must precede all the computation after the corresponding rendezvous primitive in the other process.

For scheduling purposes, a rendezvous is assumed to take zero time. In practice, this can be justified by splitting the rendezvous overhead and including it in the scheduling blocks right before the rendezvous. This raises a fine point in that a rendezvous may be interrupted if the run-time scheduler preempts the process to which the rendezvous overhead has been charged. The rendezvous primitive by itself does not guarantee mutual exclusion, e.g., it should not be used to manipulate sets of variables for which some mutual consistency constraint must be maintained.

It should be pointed out that a rendezvous between a periodic process and a sporadic process is incompatible with the semantics of the timing constraints since a periodic process must be executed regularly while by definition, there is no guarantee that a sporadic process will request computation at all. If a periodic process must communicate with another process, then that other process must be made ready regularly

and is no longer sporadic. However, if two periodic processes with different periods need to communicate with the same process T_i, then there is the question of how to model T_i. We may specify T_i as a periodic process with appropriate parameters or we may treat T_i as a sporadic process with the provision that T_i is made ready whenever a periodic process wants to communicate with it. The second alternative suggests that these processes are likely to be "servers" which cater to the periodic processes on demand. These "pseudo sporadic" processes will be treated separately in the next model.

Two periodic processes are defined to be *compatible* if they have the same period or if one period is an exact multiple of the other. We require that if two processes are related via the transitive closure of the communicant relation, then they must be compatible. This requirement does not seem to be too restrictive since processes which must synchronize with one another are likely to have the same period; in any case, the scheduling problem is not significantly harder without this restriction. Also, two communicants are assumed execute the same number of rendezvous primitives targetting each other in every (the longer of the two) period in order not to miss any deadline.

The scheduling problem will now be tackled. The following example shows that the earliest deadline algorithm modified to run the ready process which has the nearest deadline and which is not blocked by a rendezvous primitive is not optimal.

Example

There are three periodic processes. T_1 consists of two scheduling blocks with $c_{11} = c_{12} = 1$, $d_1 = 3$, $p_1 = 5$. T_2 has two scheduling blocks with $c_{21} = 1$, $c_{22} = 3$, $d_2 = p_2 = 10$. T_3 has one scheduling block with $c_3 = 1$, $d_3 = 9$, $p_3 = 10$. T_1 must rendezvous with T_2 after the first scheduling block, and T_2 must rendezvous with T_1 after the

first and second scheduling block.

The earliest deadline algorithm fails because it does not make use of the information that T_2 is forced by the second rendezvous to finish before the second deadline of T_1, i.e., the real deadline for T_2 is at time 7 instead of at time 10 and is therefore nearer than the deadline of T_3 which is at time 9. Figure 2.4 illustrates the situation when the earliest deadline algorithm fails.

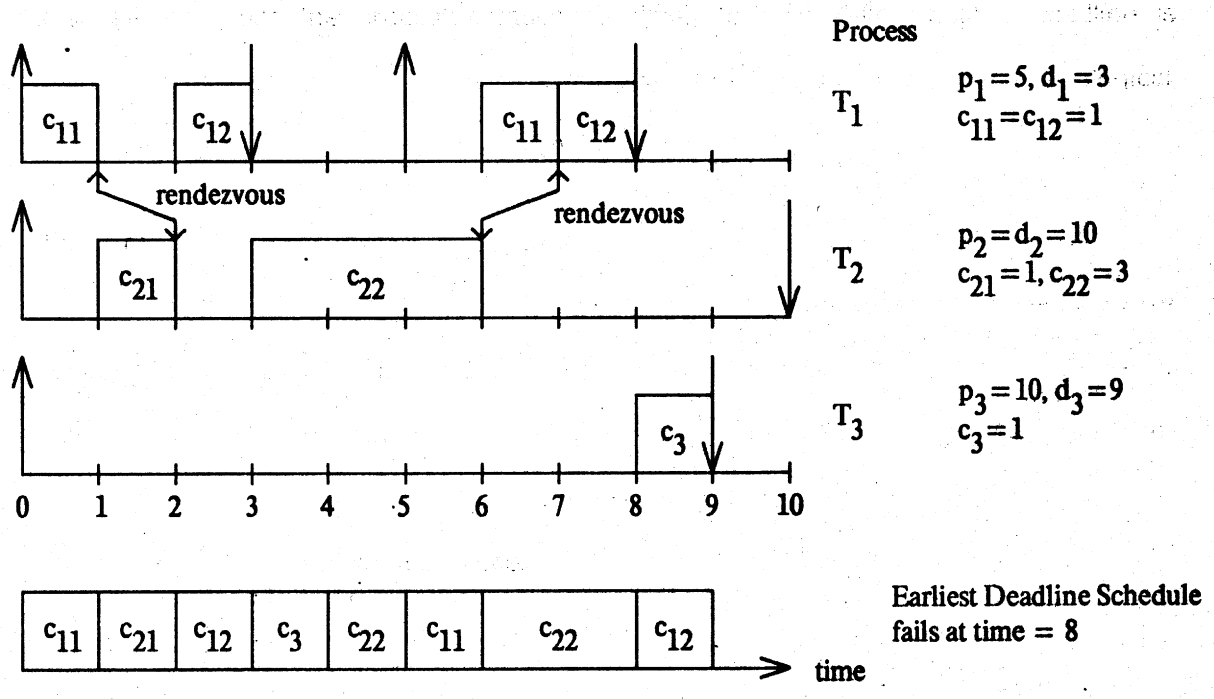


Figure 2.4
Example of scheduling constraint imposed by interprocess synchronization

This problem can be easily fixed by adopting a technique for revising deadlines to eliminate precedence constraints in the classical model of scheduling e.g., [BLAZ 76]. We shall apply this technique to build a database for the run-time scheduler so that the earliest deadline algorithm can again be used with dynamically assigned process deadlines. Let us consider the computation that must be performed for process T_i in

the interval [0,L] where L is the longest period among the processes which belong to the same equivalence class (induced by the communicant relation) as T_i. Denote the chain of scheduling blocks generated in chronological order for T_i in [0,L] by $T_i(1)$, $T_i(2)$, ..., $T_i(n_i)$. These scheduling blocks must also obey additional precedence constraints introduced by the rendezvous primitives. Specifically, suppose process T_i targets process T_j for a rendezvous between the scheduling blocks $T_i(k)$ and $T_i(k+1)$, and the corresponding rendezvous primitive occurs between the scheduling blocks $T_j(l)$ and $T_j(l+1)$. Then $T_i(k) \rightarrow T_j(l+1)$, and $T_j(l) \rightarrow T_i(k+1)$. Having thus defined the precedence constraints, we proceed to assign a deadline to each of the scheduling blocks generated in [0,L].

(1) Sort the scheduling blocks generated in [0,L] in reverse topological order.

(2) Initialize the deadline of the kth instance of $T_{i,j}$ to $(k-1)*p_i + d_i$.

(3) Revise the deadlines in reverse topological order by the formula: $d_S = MIN(d_S, \{d_{S'} - c_{S'} : S \rightarrow S'\})$ where S and S' are scheduling blocks and c_S, d_S are respectively the computation time and current deadline of S.

The purpose of the above procedure is to move up the deadline of a scheduling block if it must precede another scheduling block which has a nearer deadline but which is not yet ready to run. These revised deadlines can now be used to update the current deadlines of the processes at run-time by recycling them every L time units. Specifically, if a process T_i is executing the k^{th} (modulo L/p_i) instance of its jth scheduling block, then it must be assigned a deadline equal to the revised deadline (relative to the k^{th} request-time of T_i in [0,L]) of the k^{th} instance of the jth scheduling block generated in [0,L]. Since all the revised deadlines have been moved up, a successful schedule obeying the revised deadlines certainly meet the old deadlines. On the other hand, the amount of time by which a deadline has been moved up is sufficiently

tight (consider the chain along which the MIN function returns its value.) so that any schedule which violates a revised deadline must also miss one of the original deadlines. We summarize the above arguments in

Lemma 2.5

Suppose M is a process model and all of the processes in M are periodic and may have *rendezvous* communication primitives. Then the feasibility of the process model M is not affected by dynamically updating the process deadlines as described by the procedure above. Furthermore, whenever the dynamic deadline of a ready process T_i is nearer than that of another ready process T_j, then scheduling T_i ahead of T_j will not violate any precedence constraints involving the two processes.

Proof:

Let S and S' be two scheduling blocks in [0,L]. It follows directly from the formula for revising deadlines that $d_S < d_{S'}$ if $S \rightarrow S'$. The claim in the lemma simply states the contrapositive of this statement. QED

Theorem 2.6

If a feasible schedule exists for an instance of a process model restricted by rendezvous scheduling constraints, then it can be scheduled by the earliest deadline algorithm modified to schedule the ready process which is not blocked by a rendezvous and which has the nearest dynamic deadline.

Proof:

The "time slice swapping" technique can be applied to transform any good schedule to one produced by the modified earliest deadline algorithm. Lemma 2.5 guarantees that swapping will not violate any precedence constraint imposed by *rendezvosu* primitives as long as the process with the nearest dynamic deadline is

scheduled first. QED

Remark

Obviously, scheduling is feasible if the modified earliest deadline algorithm produces a feasible schedule for the interval [0,L]. In practice, it may not be necessary for the earliest deadline scheduler to observe all the dynamic deadlines if the timing constraints are not too demanding. A simple procedure for minimizing the size of the database is to use the process (static) deadlines for scheduling until a deadline is missed. In that case, insert dynamic deadline(s) one by one in reverse chronological order until the missed deadline can again be met.

Monitors

To deal with the "pseudo periodic" processes that we alluded to earlier, we now introduce a special type of process called a *monitor* which performs some service for ordinary processes on demand. Our *monitors* are a simplified version of Hoare's concept [HOAR 74] and have the following syntax.

```
Process <monitor_name>

Attribute monitor
rendezvous(ANY_PROCESS Tj)
{ A single scheduling block with no communication primitives }
rendezvous(Tj)

end <monitor_name>
```

The body of a monitor consists of a single scheduling block which is prefixed by a rendezvous primitive with *any process* (a wild card) as the target. An ordinary process requests service from a monitor by attempting to rendezvous with the monitor. If two or more processes are requesting service, the system scheduler is free to choose (in accordance with some scheduling policy) a single process to rendezvous with the monitor. The wild-card parameter is needed to avoid deadlocks which might result if the

monitor must rendezvous with user processes in a fixed order. After the scheduling block has been executed, the monitor attempts to rendezvous with the same process a second time. Even though a monitor does not have an explicit timing constraint attribute, it must meet the current deadline of the process for which it is performing a service.

It is obvious that a monitor realizes a critical section and so can be used to enforce mutual exclusion constraints. For example, a binary semaphore may be implemented by using the first rendezvous as a P operation and the second rendezvous as a V operation. The scheduling problem with monitors is therefore NP-hard (in the strong sense) by Theorem 2.4. As we have mentioned earlier, the problem becomes a lot more manageable if mutually exclusive scheduling blocks must have the same computation time. This can be enforced by requiring processes to execute the second rendezvous with a monitor immediately after the first one, i.e., the two rendezvous primitives targetting the same monitor must occur one right after the other in the code. This has the same effect as using the two rendezvous primitives as a macro for inserting the scheduling block of the monitor into a process with the guarantee that no more than one process can be executing the monitor code at any one time (i.e., a critical section). For scheduling purposes, every monitor will be treated as a critical section. The monitor concept is brought in so that we may look at the related scheduling problem (which is NP-hard in general), and also to be consistent with our goal of not relying on a global memory for process coordination in our process model.

2.3.3 The Kernelized Monitor Model

In this model, the operating system kernel enforces mutual exclusion by allocating processor time only in uninterruptible quantums, say of size q which is chosen to be bigger than the longest monitor. For simplicity, we shall require the computation times of all scheduling blocks to be exact multiples of q so that each scheduling block takes an integral number of quantums to execute. This restriction seems reasonable if critical sections are kept very short, e.g., for accessing a small set of variables which must be kept mutually consistent. (In the next model, even this restriction will be relaxed.) In fact, it will become obvious that the shorter the time quantum is, the better is the chance to design a run-time scheduler with a small database. Notice that with this processor allocation discipline, critical sections no longer impose any more restrictions on the scheduler. As far as the scheduling problem is concerned, the only difference between the kernelized monitor model and the previous one is that a process may be interrupted only after it has been allocated an integral number of time quantums.

We shall adopt a scheduling technique [GAR et al 81] involving a concept called "forbidden regions" which has been invented to yield a necessary and sufficient condition for generating a (finite) schedule for a set of (one-time) unit-length tasks with real number request-times and deadlines in the classical scheduling model. For simplicity, we shall assume that all (periodic) processes to be compatible and let L be the longest period. Relaxing this constraint does not make the scheduling problem significantly harder, but increases the size (from O(L) to O(LCM$\{p_i\}$)) of the database for the run-time scheduler. The following example shows why a simplistic earliest deadline scheduler might fail.

Example

There are two periodic processes T_1, T_2. T_1 consists of a single critical section of

length $c_1 = 2$ and has a deadline $d_1 = 2$ and period $p_1 = 5$. T_2 has two scheduling blocks with the following parameters: $c_{21} = 2$, $c_{22} = 2$, $p_2 = d_2 = 10$. The second scheduling block of T_2 is the same critical section as T_1. The preemption time quantum is set to be 2.

The second deadline of T_1 will be missed if the second scheduling block of T_2 is scheduled at time 4, since the second instance of T_1 must be scheduled as soon as it is requested at time 5, and T_2 cannot be preempted before it uses up the second quantum of processor time allocated to it at time 4. A cleverer scheduler will leave the processor idle in the interval [4,5] and execute T_{22} in [7,9]. The (open) interval (3,5) is an example of a forbidden region in which a scheduler must not allocate a new quantum of processor time to any process so that a future deadline may be met. Figure 2.5 illustrates the situation when the second instance of T_1 misses its deadline because T_{22} is started in the forbidden region (2,3).

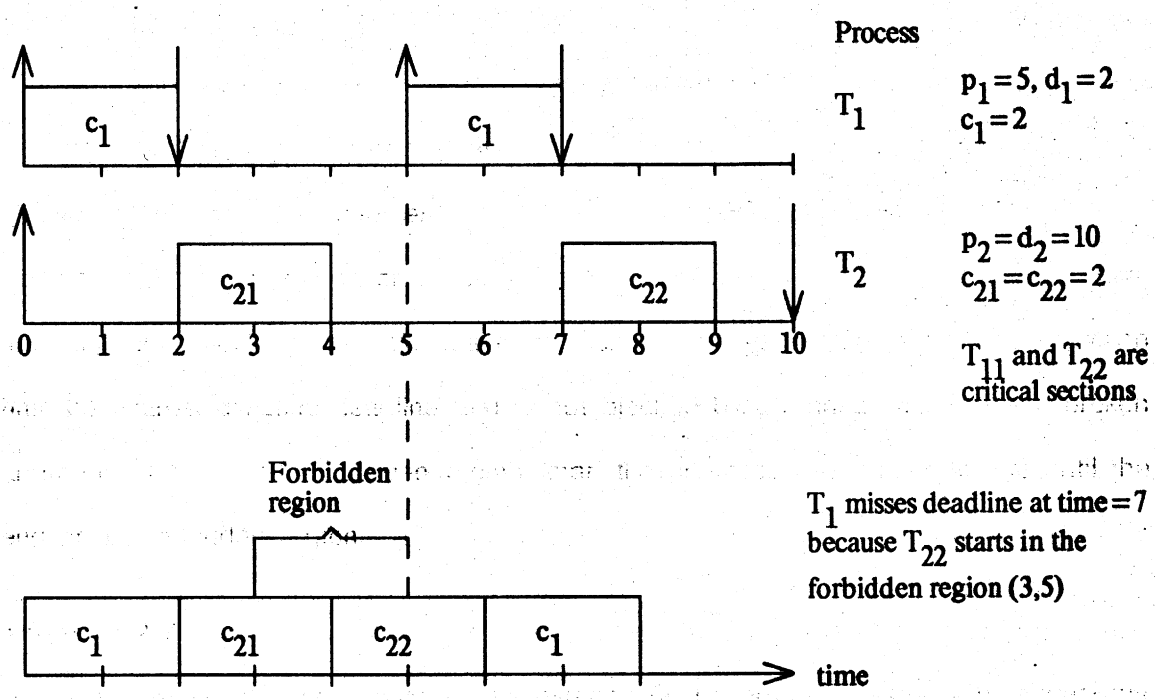


Figure 2.5

Example of a forbidden region imposed by mutual exclusion constraints

The database to be used by the run-time scheduler contains a collection of forbidden regions in the interval [0,L] where L is the longest period among the (compatible) processes. The run-time scheduler recycles the database every L time units to locate forbidden regions at all future times t, and allocates a quantum of processor time to the ready process which has the nearest dynamic deadline and which is not blocked by a rendezvous iff t does not lie on the within a forbidden region. To compute the set of forbidden regions, each process is considered to be a chain of *mini scheduling blocks* each of which is a quantum (the basic time unit of processor allocation). Consider all the mini scheduling blocks generated in the interval [0,L]. As in the previous section, these mini scheduling blocks form a partial order imposed by the (intra and interprocess) precedence constraints, and each of these mini scheduling blocks can be given a request-time and deadline consistent with the partial order as follows:

(1) Sort the mini scheduling blocks generated in [0,L] in forward topological order.

(2) Initialize the request-time of the kth instance of each mini scheduling block of T_i in [0,L] to $(k-1)*p_i$.

(3) Revise the request-times in forward topological order by the formula: $r_S = MAX(r_S, \{r_{S'} + q : S' \rightarrow S\})$ where S and S' are mini scheduling blocks in [0,L], and r_S, q are respectively the current request-time and the computation time of S (i.e. a quantum).

(4) Sort the mini scheduling blocks generated in [0,L] in reverse topological order.

(5) Initialize the deadline of the kth instance of each mini scheduling block of T_i in [0,L] to $(k-1)*p_i + d_i$.

(6) Revise the deadlines in reverse topological order by the formula: $d_S = MIN(d_S, \{d_{S'} - q : S \rightarrow S'\})$ where S and S' are mini scheduling blocks; d_S and $d_{S'}$ are respectively the current deadlines of S and S'.

The effect of the above procedure is to assign to each mini scheduling block S in [0,L] a request-time r_S which is the earliest time at which it can be scheduled, and a deadline d_S which is the latest time by which it must be completed if their partial ordering is to be maintained. The request-times are optimistic for two reasons. First, two mini scheduling blocks, S and S' may be assigned the same request-times if they are unrelated, i.e., if it is not true that S→S'or S'→S. Second, the processor may have to be be kept idle at time t in anticipation of more urgent computation that will not be made ready until a short while after t. For each request-time r_S, we can declare the interval: (x_S, r_S), $q > r_S - x_S \geq 0$ to be a *forbidden region* if we cannot delay scheduling the scheduling block S beyond $x_S + q$. If the processor is unwisely allocated to some process after time = x_S, then it cannot be released until after time = $x_S + q$ whence it will be too late to schedule the scheduling block S. The set of forbidden regions in the interval [0,L] is computed recursively by the following algorithm.

(1) Sort the request-times in reverse chronological order and determine the forbidden region associated with each request-time as follows. Initially, there are no forbidden regions.

(2) For each request-time, r_S and *any* deadline d for which $L \geq d \geq d_S$, let $n_{r,d}$ be the number of mini scheduling blocks which must be scheduled in the interval $[r_S,d]$, i.e., count the number of all scheduling blocks S' for which $r_{S'} \geq r_S$ and $d_{S'} \leq d$. Given a set of forbidden regions in the interval $[r_S,d]$, schedule $n_{r,d}$ mini scheduling blocks (time slices of length q) in $[r_S,d]$ so that none of them starts in a forbidden region. Let $s_{r,d}$ be the latest time at which the first mini scheduling block must be so scheduled. (There may be more than one way to fit $n_{r,d}$ time slices in the interval $[r_S,d]$ without violating any forbidden regions. A easy way to find $s_{r,d}$ is to work backwards in time and place each time slice as close as possible to the left of the previous one, starting from time = d. If placing a time slice right next to the previous one will result in the left boundary being inside a forbidden region, align the left boundary of the time slice with the left limit of the forbidden region.) If $s_{r,d} < r_S$, (i.e., there is no way to fit $n_{r,d}$ time slices of size q each in the interval $[r_S,d]$ without putting the left boundary of at least one of them in a forbidden region), then declare failure. Otherwise, declare $(s_{r,d} - q, r_S)$ to be a forbidden region if $s_{r,d} < r_S + q$.

The above algorithm constructs a set of O(n) forbidden regions:

$W = \{(x_i,y_i) : y_i$ is the revised request-time of some mini scheduling block in [0,L], and no process should start past x_i before $y_i\}$

in time $O(n^2)$ where n is the number of mini scheduling blocks generated in [0,L]. The set W can be used to locate forbidden regions at run time as follows: At any time t, t lies in a forbidden region iff $x_i < t \pmod L < y_i$ for some $(x_i,y_i) \in W$.

We give a simple proof (different from the approach in [GAR et al 81]) that a run-

time scheduler can use the following modified earliest deadline algorithm to produce a feasible schedule whenever one exists.

The *kernelized monitor scheduler*:

At any time t when the processor is free, and t does not lie in a forbidden region, the scheduler allocates the next quantum of processor time to the ready process which has the nearest dynamic deadline and is not blocked by a rendezvous. Ties are broken arbitrarily. If t lies in a forbidden region, then the processor is allowed to idle until the end of the forbidden region.

Theorem 2.7

If a feasible schedule exists for an instance of the process model with *rendezvous* and *monitor* communication primitives, then the kernelized monitor scheduler can be used to produce a feasible schedule.

Proof:

By the construction of the kernelized monitor scheduler, we only need to concentrate on the interval [0,L]. First, we show by induction that if the algorithm for finding forbidden regions fails, then no feasible schedule can exist; furthermore, in any feasible schedule, no mini scheduling block can start in a forbidden region.

If the algorithm declares failure when it is processing a request-time, y_j before any forbidden region is declared, then there must be a deadline d such that the computation to be scheduled in the interval $[y_j,d]$ exceeds $d-y_j$, a clearly hopeless situation. For a forbidden region (x_j,y_j), any mini scheduling block starting in it will not finish until after time $= x_j + q$. But by the construction of forbidden regions, y_j is the request-time of some mini scheduling block which must be started no later than $x_j + q$. Hence starting a mini scheduling block in a forbidden region will cause a deadline to be missed.

Suppose the hypothesis is true for all forbidden regions associated with request-

times later than some request-time y_i and the algorithm declares failure when it is processing y_i. Then there must be a deadline d such that the computation required in $[y_i,d]$ cannot be fit into that interval without starting some mini scheduling blocks in a forbidden region after y_i. By the induction hypothesis, no feasible schedule can exist. Consider the forbidden region (x_i,y_i) such that the hypothesis holds for all forbidden regions associated with request-times after time $= y_i$. From the algorithm for finding forbidden regions, there is a deadline d such that one of the mini scheduling blocks that must be executed in the interval $[y_i,d]$ must start running by time $= x_i + q$ in order not to violate any of the forbidden regions in the interval $[y_i,d]$. But any mini scheduling block starting in the forbidden region (x_i,y_i) will not finish until after time $= x_i + q$, so no mini scheduling blocks with request-times $\geq y_i$ can be executed before time $= x_i + q$, and the induction step holds.

If a feasible schedule exists for the interval [0,L], there must be one whose completion time (i.e., the time at which the last mini scheduling block in it finishes) is the earliest. If there is any processor idle time in this schedule, then it must be the case either (i) the idle time lies in a forbidden region or (ii) all the processes are either not ready or are waiting to rendezvous with some not yet ready process. Otherwise, the next mini scheduling block in the schedule can be started earlier, and we can repeat the argument to postpone the idle time to after the last mini scheduling block and obtain another feasible schedule with a shorter completion time, a contradiction. We can now apply the "time slice swapping" technique (with q as the size of a time slice) to transform any feasible schedule with the shortest completion time to one that is produced by the kernelized monitor scheduler. Again, no precedence constraints will be violated by swapping since the deadlines have been revised so that mini scheduling blocks with earlier deadlines cannot be preceded by ones with later deadlines. Furthermore, the resulting schedule will have idle time if and only if the kernelized monitor

scheduler also idles the processor. QED

Corollary

If a feasible schedule exists for the interval [0,L], then the kernelized monitor scheduler always generates one with the earliest completion time. Furthermore, none of the start-times in this schedule can be pushed to an earlier time without causing a failure.

Proof:

Any idle time before the start-time of a mini scheduling block in the schedule generated by the above algorithm must be either (i) inside a forbidden region or (ii) when every process is either not ready or is waiting for a rendezvous. In both cases, the mini scheduling block cannot be started any earlier.

Remark

If the kernelized monitor scheduling algorithm generates a feasible schedule for [0,L], then a run-time scheduler which repeats this schedule at run time is guaranteed not to miss any deadline. It is actually unnecessary to simulate the kernelized monitor algorithm for the interval [0,L] since if the algorithm for computing forbidden regions does not declare failure, then we can show that the kernelized monitor scheduler must succeed at all times. To see this, assume the contrary and let S be the first mini scheduling block to miss its deadline, d_S and let r_S be its request-time. Notice that the kernelized monitor scheduler guarantees us that all the mini scheduling blocks that have been allocated processing time in the interval $[r_S,d_S]$ must have deadlines no later than d_S. If there is no idle time in the interval $[r_S,d_S]$, then the forbidden region algorithm would have failed in trying to fit the mini scheduling block S and all the other mini scheduling blocks in $[r_S,d_S]$. On the other hand, the kernelized monitor scheduler

guarantees us that the only processor idle time in the interval $[r_S, d_S]$ must lie inside a forbidden region since the process which contains the mini scheduling block S is ready and cannot be waiting for a rendezvous after time = r_S. Again, the algorithm for constructing forbidden region would have declared failure in computing the forbidden region associated with the request-time r_S.

The kernelized monitor model imposes two restrictions on the scheduler. First, it disallows preemption of critical sections by ordinary scheduling blocks. Second, the scheduler cannot take advantage of the semantic difference between different critical sections since they are not permitted to preempt one another. Both restrictions are tolerable if all critical sections are relatively short. This seems to be a plausible assumption since in practice, an operating system usually assigns a process a minimum quantum of computation time in order to keep process switching overheads within reasonable bounds. As long as the critical sections are short relative to the minimum quantum, the scheduler can execute only an integral number of scheduling blocks within a quantum without incurring unacceptable waste.

2.4 Implications on the Design of Real-Time Languages

The term "real-time languages" has been used loosely to denote a class of high-level languages which are designed to support real-time applications† There is, however, little consensus on what qualifies as a real-time language. Some of the language features that are often cited in relation to real-time applications are:

(1) Parallel processing capability

(2) Access to a timer

(3) Direct interface with I/O devices

(4) Fast execution

The last feature is primarily an implementation issue although it may be argued that the selection of language primitives, especially facilities for concurrency control can have significant influence on implementation efficiency. Direct access to a timer and I/O devices are important for control applications, but they can usually be implemented with an appropriate function package and therefore do not require conceptual innovations to conventional sequential languages. The capability to coordinate parallel activities indeed raises many interesting issues, and many language constructs have been proposed for concurrency control, e.g., Dijkstra's *semaphore*, Hoare's *monitor*, Hansen's *distributed process*, the *rendezvous* concept in Ada, etc. While the later proposals are generally "cleaner", comparison among them often tends to be ad hoc. Having examined the process scheduling problem in some detail, we are in a position to shed some light on the subject by examining the implications of real-time scheduling requirements on the design of process-based real-time languages.

2.4.1 Incorporation of Performance Objectives into a Real-Time Language

Since the basic unit for scheduling computation is the process, compliance with

† In military jargon, they are also known as *embedded systems*.

the stringent timing constraints required by an application must be achieved by proper scheduling of processes. An obvious approach is to augment high-level languages with a set of constructs, e.g., *delay* <time>, *start* <process> *at* <time>, so that processes can be explicitly scheduled. Unless every process runs on a dedicated processor, these explicit scheduling commands cannot always be used to guarantee that a process will be started on time. For example, if a process is delayed to a time at which another process is scheduled to run and there is only one processor, then the resource conflict is usually resolved according to some priority assignment so that except maybe for the process with the highest priority, explicit scheduling commands can guarantee only minimum but not maximum bounds on response times. Thus explicit scheduling commands are convenient for building soft but not necessarily hard real-time systems. More importantly, most explicit scheduling commands in current real-time languages are too restrictive for specifying timing constraints since they usually leave little room for resolving resource conflicts.

Conceivably, this problem can be solved by modifying these commands to take additional arguments so as to allow for margins in their timing parameters. A more serious pitfall with using explicit scheduling commands is that processes are scheduled with respect to the "current" value of time which must be read from a timer, e.g., computing a start-time for a process to be used as an argument in a *start* command. In a multiprocessing environment, a process may be interrupted for an indefinite amount of time immediately after it reads the timer, thus invalidating the timer value. For this reason, explicit scheduling commands must be executed without preemption so that only up-to-date time values are used. Unless time-valued arguments are restricted to be simple expressions, explicit scheduling commands represent non-preemptible scheduling blocks of arbitrary length, thus greatly increasing the complexity of the scheduling

problem as we have seen.

In general, a penalty in efficiency is incurred if a programming language permits or even requires individual processes to allocate system resources with an authority that is normally delegated to the operating system. This is so because scheduling decisions are best made with global information about system demand. The efficiency problem in both carrying out the scheduling function and the resulting allocation of resources suggests that

the function of scheduling constructs should not be as much to directly allocate computational resources as to implement a protocol between the system scheduler and the user processes requesting for resources.

The assignment of static priorities to processes can be viewed as an example of such a protocol which is widely used in practice. This protocol is not the most efficient since it offers only limited control over response times. For example, consider the problem of scheduling a set of periodic processes whose deadlines are the same as their periods. When there are no restrictions on the selection of processes for preemption, it has been shown [LIU & LAY 73] that there are sets of processes with a processor utilization factor $\simeq 0.7$ for which no static priority assignment can meet the timing constraints, whereas full processor utilization is always achievable by using the simple earliest deadline scheduling algorithm.

In practice, the assignment of priorities is often based on the relative importance of the computation performed by a process, timing constraint specifications being only secondary considerations. For example, periodic processes that are essential to the continuous operation of a system are often assigned high priorities regardless of their specified repetition rates. This precautionary approach is appropriate for soft real-time systems where there need not be any absolute guarantee on response times; in such cases, a conservative scheduling policy is in order. The penalty is that the processor

may not be fast enough to also meet the timing specifications of less essential processes which are assigned lower priorities even if all the timing specifications can in fact be met by an appropriate assignment of priorities. The ability to design truly hard real-time systems offers a more effective alternative since essential processes are not unnecessarily given higher priority at the expense of less essential ones. In fact, hard real-time systems are made more robust since by design, monopolizing system resources should be deliberately forbidden. To support this assertion, we must also deal with the contingency when the actual workload exceeds the specifications; the robustness aspects of hard real-time systems against unfaithful usage specifications will be discussed later. We shall only note that

the use of static priorities as a protocol for resource allocation should be considered primarily for robustness rather than efficiency reasons.

Given that the process is the atomic unit for scheduling, a straightforward protocol for allocating computing resources is to add scheduling attributes (e.g., deadline, period) to a process. (Static process priorities could be used to resolve conflicts when there is insufficient computing resources to meet all of the timing constraint specifications.) In practice, however, the specification of timing constraints is usually complicated by the need for processes to communicate with one another. In the next section, we shall discuss some implications of timing constraint requirements on the use of concurrency control mechanisms.

2.4.2 Choice of Concurrency Control Mechanisms

Concurrency control mechanisms have traditionally been designed to meet two important needs: synchronization between processes and the enforcement of mutual exclusion constraints. It is well known that both types of interprocess coordination can be implemented by the use of semaphores or simple message passing constructs (i.e.,

send, *receive*). These constructs have been considered to be too unstructured by recent designers of real-time languages and a number of alternative concurrency control mechanisms have been proposed. It is difficult, however, to make an objective evaluation of the different proposals, since many of the pros and cons for one construct or another is often based on conflicting language design principles. In the following, we shall first review some of these principles and then attempt to evaluate their applicability in the context of the hard real-time environment.

▸ *Localization of control information*:

Programs tend to be difficult to maintain if the control information associated with a single type of interprocess coordination is allowed to be scattered over different places in a program. This is especially true with the use of semaphores or simple message passing constructs for which additional programming rules must be observed so as not to subvert the intended use of a construct, e.g., a process may inadvertently exit out of a critical section after tripping an *exception handler* before the proper exit protocol has been performed. It has been suggested that the maintainability of a program can be improved by keeping control information (both code and data) close together.

▸ *Minimality in language constructs*:

While it is conceivable to achieve concurrency control by defining a language construct for every major type of interprocess coordination, e.g., the *monitor* construct for enforcing mutual exclusion, the size of the resulting language may be too unwieldy as to be practical. The opposite view is to strive for simplicity by minimizing the number of distinct control structures that are built into a language. For example, the designers of Ada have sought to unify mutual exclusion and synchronization between processes by providing a single interprocess communication facility (the *rendezvous* construct of

Ada) which can be used for both purposes.

▸ *Implementation efficiency*:

Informally, the run-time efficiency of a concurrency control mechanism can be measured by the run-time overhead it incurs in realizing interprocess coordination (i.e., by the difference in length between the actual schedule which takes into account the execution time of the concurrency control mechanism and the shortest *ideal schedule* which miraculously meets all the concurrency constraints without any concurrency control mechanism at all.) There are two important reasons why the run-time efficiency of a concurrency control mechanism may not approach the ideal. First, a mechanism may be too restrictive as to exclude some schedules which would otherwise be acceptable, i.e., the semantics of the mechanism may not permit the maximal amount of parallelism. Second, the inherent cost of the coordination mechanism may be unacceptably high, either in the amount of interprocess communication or the amount of compile-time analysis needed to optimize the translation of the concurrency control mechanism into executable code. For example, it has been reported [ROB et al 81] that a straightforward implementation of the *rendezvous* mechanism in Ada incurs, even for the simple operation of transferring one byte of data from a sender task to a receiver task, substantial context switching overhead (between the run-time system scheduler and the sender and receiver) whereas an alternative solution using semaphores requires essentially no context switching at all (provided that no acknowledgement signal is required from the receiving task.) The problem of optimally implementing the *rendezvous* mechanism by means of semaphores is, however, non-trivial in general.

It should be obvious that the above design principles are not necessarily compatible with one another. Whereas localizing control information and keeping control constructs to a minimal are generally considered a plus to the programmer, they often in-

cur an efficiency penalty which may not be negligible for real-time applications. These conflicts have been alluded to in the previous chapter as manifestations of the maintainability/efficiency dichotomy whose resolution ultimately depends on the degree to which we can automate the process of generating efficient software. It follows that *a more objective criterion for evaluating a concurrency control mechanism is by its impact on software automation; in this case, how does it impede or facilitate the construction of scheduling tools to satisfy stringent timing constraints.*

By this criterion, we have concrete evidence that the undisciplined use of semaphores is undesirable; the related scheduling problem quickly becomes NP-hard. Another lesson from our study of scheduling problems is that there is substantial benefit in making the enforcement of interprocess synchronization and mutual exclusion syntactically distinct since this piece of information is crucial to the solution of the related scheduling problems; and there may not be any easy way to deduce whether a control construct is being used to enforce a synchronization or a mutual exclusion constraint. It should be mentioned that the syntactic distinction need not be built into the programming language and a purist who feels strongly about minimizing the number of language constructs may prefer to annotate each instance of a control construct instead. However, stylized annotations of code in effect introduce subclasses of control constructs and this extra information must be supplied to the code generator.

2.4.3 Scheduling of Indeterministic Constructs

Another design issue which is closely related to the choice of concurrency control mechanisms and the incorporation of performance objectives and which is not very well understood is the scheduling of indeterministic constructs. Other than process scheduling, a system scheduler is also needed to make a choice among alternative paths of execution when the real-time language has indeterministic constructs, e.g., the

select statement in Ada. There is, however, little concensus on how the scheduler should behave other than that it ought to be in some sense "fair" (for which the common interpretation is round-robin scheduling.) This approach is problematical since a straightforward implementation of a "fair" scheduler may not guarantee that the computation will make progress.

For example, consider two variables x, y which are guarded by individual semaphores and are both updated by two processes T_1, T_2 with the provision that they must be kept mutually consistent (i.e., if x is updated by T_1 before T_2, then y must also be updated by T_1 before T_2 and vice versa.) In order to avoid a deadlock, the order in which x and y are accessed may be arbitrarily fixed, (say x before y) by administrative decree. However, this solution is deemed unacceptable since it puts an intolerable constraint on how future programs can be written. As a compromise, the order in which x and y are accessed at run-time is left to be decided by the execution of an indeterministic construct in both T_1 and T_2, and both processes release the semaphores that they are holding whenever they are blocked by the other one. It is easy to see that both processes may never make any progress if the execution of the indeterministic construct which updates the variables follows the round-robin discipline in both processes (e.g., T_1 may access x first and T_2 may access y first and so on.) The particular problem encountered here stems from the fact that fairness is a global property and may not be achieved by schedulers which are only locally fair. While the above admittedly academic problem can be solved by randomizing the scheduler, it serves to illustrate the problems of specifying the semantics of the scheduler in general.

The stringent timing constraints of the hard real-time environment suggest an approach for resolving the above issue. The key observation is that indeterministic constructs need not be stochastic but are better regarded as providing a margin of freedom to the scheduler for achieving performance objectives. Instead of (over)specifying

the behavior of the scheduler, it is more profitable to devise language mechanisms with which the scheduler can be manipulated to achieve desired performance objectives. In other words, the behavior of the scheduler should not be defined by the language but by the application. In fact, there is no reason why the scheduler should be "fair" if a particular set of performance objectives does not require some execution path to be exercised at all. The default behavior of the scheduler may be decreed as part of the specification of a real-time language, and for that purpose, the adoption of just about any scheduling strategy, e.g., round-robin is defensible. The important point is for a real-time language not to unnecessarily usurp the scheduling function but to provide the programmer (or more importantly, software automation tools) with sufficient handle for improving system performance.

For example, if the indeterministic *rendezvous* in our version of the *monitor* construct is restricted by the language to select a user process by random, then it will not be possible to make the best use of available processing power by applying clever scheduling algorithms.

We have already discussed the language mechanisms for incorporating performance objectives of hard real-time systems into programs. It is an interesting problem to design an appropriate protocol between the system scheduler and user processes for soft real-time applications. For example, we might permit the programmer the option of fine tuning the system scheduler in terms of a *policy function* which selects an execution path when an indeterministic construct is encountered. The selection may be made according to the current values of a set of scheduling parameters which are modifiable by the policy function at appropriate moments in real time. The definition of the policy function is of course dependent on the target performance objectives. This is a potentially rich research topic but is, however, outside the scope of this thesis.

Chapter 3

Design via a Graph-Based Model

3.1 Graph-Based Model of Computation for the Hard Real-Time Environment

In adopting a process-based computation model for studying resource scheduling problems, there is an implicit assumption that the computational requirements of an application have been somehow translated into a set of processes with the appropriate scheduling attributes. Owing to a semantic gap, this translation can be a serious source of inefficiencies during system design and substantially complicates software maintenance later on. As such, a process-based model is less than desirable for defining computational requirements in the hard real-time environment. However, the process abstraction has been the basis of the computation model for the majority of software designs, and prudence requires us to present concrete evidence in order to justify an alternative.†

To this end, we shall examine three general strategies for decomposing the computational requirements of a design problem into a set of concurrent processes. It will be demonstrated that in the presence of stringent timing constraints, efficient decomposition of the required computation into processes is inherently implementation dependent, and that a set of processes resulting from a highly efficient decomposition is likely to be unstructured and difficult for human programmers to maintain. The design example will also illustrate the concept of *latency scheduling* for meeting asynchronous timing constraints (i.e., computation performed in response to sporadic external events), sometimes by exploiting the periodic computation required for satisfying periodic timing

† Whereas English-like languages have been used to describe system requirements, a process-based model is almost invariably used for software design and resource allocation. From a practical point of view, learning a new language for software design is usually a major undertaking that most people are justifiably reluctant to pursue.

constraints. We shall then introduce a graph-based computation model which is more amenable to representing design requirements in the hard real-time environment. The latency scheduling technique will be formalized in terms of the new model and the computational problems of latency scheduling will be investigated.

3.2 Decomposition of Design Requirements: an Example

In this section, we are going to examine the systems issues that are involved in decomposing the computation of a hard real-time system into processes by examining the relative merits of three different decomposition strategies. In a narrow sense, it can be shown formally that there is no unique best decomposition since the most efficient implementation depends on system parameters such as interprocess communication costs. This observation is hardly surprising and is only one of the concerns arising from the semantic gap between the hard real-time environment and a process-based language. The broader purpose of this discussion is to bring focus on the important but less readily quantifiable systems issues, e.g., maintainability, implementation independence which we have identified in the first chapter.

3.2.1 Statement of Design Requirements

Figure 3.1 is the block diagram of an automatic control system which is the design problem to be considered. This control system has three inputs x, y, z and a single output u. There are five function blocks: f_X, f_Y, f_Z, f_S and f_K. The function block f_S has two outputs one of which is fed back via f_K to itself so that u is a function of x', y', z' and its own previous value. The other output is to the external environment and has the same value. For brevity, we use the same name, u to denote the two outputs. The computation times of the five functions are assumed to be bounded and their maximum values are respectively c_X, c_Y, c_Z, c_S and c_K.

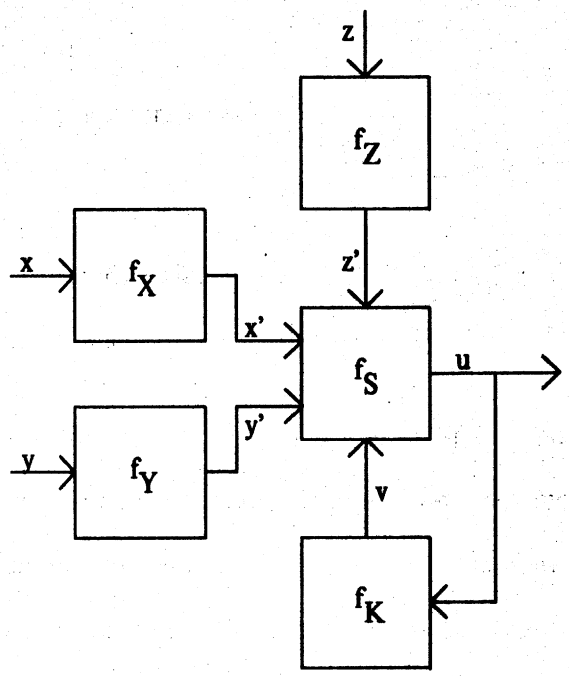


Figure 3.1
Example control system function block diagram

The design objectives of this system can be stated in terms of the computation required by two periodic and one asynchronous timing constraints as follows. The input x is to be sampled at the regular rate of $1/p_x$ cycles per second. (Sampling rates are determined by the dynamics of the physical process under control.) The output u must be recomputed by executing the function f_S with the new value of x' and recent values of y', z' and v (to be determined by their individual update rates). The internal state v must then be updated by executing f_K with the new value of u. The input y is to be sampled at a rate of $1/p_y$ cycles per second and the variables u and v must be likewise recomputed. The input z is a boolean signal, i.e., $z \in \{0,1\}$, and can change state asynchronously. When a state transition occurs, the new value of z must be detected and a new z' computed by executing f_Z. The output signal u must also be recomputed by f_S within d_z time units. The input z is assumed to change state very infrequently compared with p_x and p_y.

A physical interpretation of this block diagram is to regard f_X and f_Y as the

preprocessors of signals from two sensors measuring the physical quantities x and y one of which changes much more slowly than the other, hence the different sampling rates. The signal z can be regarded as the output from a toggle switch and u is the control signal to an actuator. The signal u is also used to compute an internal state to be used in subsequent calculations, e.g., f_K may be a state estimator in a compensator. f_S is used to determine the output from the inputs x and y and the internal state. The variable z' may be a parameter which selects a different mapping for f_S depending on the operating regime selected by a human operator via the toggle switch z.

3.2.2 Implementation Environment

The example system will be implemented by a set of concurrent processes running on one or more processors with common access to a shared memory. For concreteness, each process will be programmed in an Algol-type language augmented by the *rendezvous* and *monitor* constructs of the previous chapter for concurrency control. A library of programs will supply the code for the functions f_X, f_Y, f_S, f_K, f_Z. The signals x', y', v, z' will be stored as global variables in the shared memory. The input signals x, y, z are read from the external environment by f_X, f_Y, f_Z respectively.

It should be noted that a shared memory is not essential to a process-based model since each global variable may be implemented by the private variable of a *monitor* whose sole function is to serialize access to the global variable, and in general, the implementation environment is inconsequential to the validity of our observations about process-based models. The crucial assumption is that the basic object to be scheduled is the process.

All functions have a nominal (constant) execution time of 10 ms (milliseconds). The nominal sampling periods of x and y are respectively, 80 and 160 ms, and u must be recomputed within 80 ms after z has changed state. A timer is accessible to all pro-

cessors and initiates periodic interrupts as required by the periodic processes. When an interrupt occurs, one or more processes are made ready to run, but are not necessarily allocated processor time immediately so that the scheduler will not be unduly restricted. Hence, a sporadic process does not have *a priori* priority over periodic processes. For ease of reading, we give a summary description of the pertinent language features.

3.2.3 Summary of Example Process-Based Real-Time Language

A process is declared by:

process <process_name>
activated by <signal_name> | *timer*
attribute <attribute_name> = <attribute_value>
<code body>
end <process_name>

A process may be either periodic or sporadic and may have a *period* and/or *deadline* attribute. Periodic processes are activated by timer interrupts and a sporadic process is activated when a signal variable changes value in response to external interrupts. The period of a sporadic process is the minimum time between two successive activations. (In practice, external interrupts may be queued to maintain a specified minimum period and an overload condition may be declared if the queue overflows. For this example, we need not worry about the period attribute of a sporadic process since the external signal z, e.g., a toggle switch is assumed to change very infrequently compared with the periodic signals.) The deadline attribute will be defined as *default* in which case the system will set it to the smallest feasible value.

A process may communicate (synchronize) with another process by executing:

rendezvous <process_name>

A process in a rendezvous is suspended until the target process also executes the corresponding rendezvous statement. To enforce mutual exclusion, a process may in-

voke a *monitor* by executing:

rendezvous <monitor_name>

A monitor is declared by:

monitor <monitor_name>
<code body>
end <monitor_name>

A rendezvous with a monitor is completed by executing the body of the monitor. In general, monitors embody critical sections and may be implemented in various ways, e.g., by a process which is activated by any process attempting to rendezvous with it, or by expansion of macros augmented with appropriate scheduling mechanisms.

For accounting simplicity, the computation time of a process will be the sum of each function call to f_X, f_Y, f_S, f_K, f_Z plus the cost of executing rendezvous statements. Initially, we shall ignore the overhead incurred by the operating system kernel and the communication network and let c_{sys} (nominal cost of a rendezvous) be zero. The effects of these overheads will be considered when they are significant in determining the relative merits of decomposition strategies.

In the following, we shall describe three decomposition strategies which represent extreme approaches spanning the design space. The results of applying different strategies to the design problem will then be compared.

3.2.4 Decomposition by Timing Constraints

In this strategy, a process is created to perform the computation required by each and every specified timing constraint. A process is usually made up of a sequence of function calls representing the operations (signal processing steps) on the data path between input and output devices, but a process may also be created to update a variable which holds some internal state information of the physical plant under control.

Since a function may be called in more than one process, some of the arguments may not be variables local to the process. To preserve data integrity, a monitor is created to enforce mutual exclusion on the execution of every function called by two or more processes. The scheduling attributes of a process are set according to the associated timing constraint in the obvious way. The following program implements a solution to the design problem.

```
/* COMMENT
   This program uses a process for each timing constraint. The name of each
   process is denoted in capital letters by the names of the functions called
   by the process. The processes XSK, YSK are for meeting the two periodic
   timing constraints. The asynchronous timing constraint is met by the
   sporadic process ZS which is invoked when a change in the sensor input
   (read into the variable z) is detected.
*/

process XSK
activated by timer;
attribute period = 80, deadline = 80;

x := sensor_x();
x' := f_X(x);
rendezvous S;
rendezvous K;
end XSK


process YSK
activated by timer;
attribute period = 160, deadline = 160;

y := sensor_y();
y' := f_Y(y);
rendezvous S;
rendezvous K;
end YSK


process ZS
activated by z;
attribute deadline = 80, period = default

z := sensor_z();
z' := f_Z(z);
rendezvous S;
end ZS


monitor S

u := f_S(x',y',z',v);
end S


monitor K

v := f_K(x',y',z',v);
end K
```

The nominal timing constraints have been specified so that the above program will work on a single processor with any process scheduling discipline which does not idle the processor when there is one or more ready processes. The timing diagram (commonly known as a Gantt chart) in figure 3.2 shows an example execution sequence of the function calls.

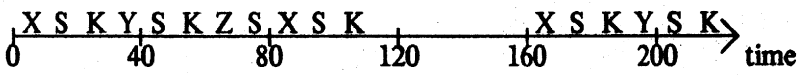


Figure 3.2
Gantt chart for decomposition by timing constraint

This is perhaps the most straightforward way to decompose the computation and the resulting design is very easy to understand. Insofar as maintainability can be quantified, this decomposition strategy should yield a highly maintainable design solution. However, the gain in maintainability may be offset by a loss in efficiency owing to the unnecessary duplication of some computation in two or more processes with compatible timing constraints. In the above program, the functions f_S and f_K are executed in both XSK and YSK while in fact, it suffices to execute these functions only once after **both** x' and y' have been updated. With this saving, it is possible to sample x and y at the tighter specifications of 60 and 120 ms respectively. The Gantt chart in figure 3.3 shows the required execution sequence.

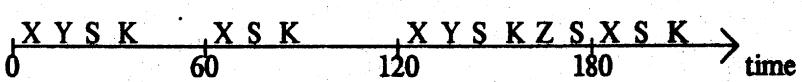


Figure 3.3
Gantt chart for example of eliminating redundant function calls

It should be emphasized that it is not always possible or desirable to eliminate the unnecessary duplication of computation. In particular, the saving noted above will not be as easy to achieve if p_x and p_y are relatively prime and in general, decomposition by timing constraints is conducive to designs which are stable against changes in parameter values. However, the loss in efficiency may be significant since it incurs not only extra processor time but also communication costs for enforcing mutual exclusion. This efficiency issue may be alleviated by a variety of decomposition strategies which may be viewed as tradeoffs between two diametrically opposite approaches.

3.2.5 Decomposition by Minimizing Interprocess Communication

Whereas the previous approach assigns the computation required by one timing constraint to one process, the objective of this decomposition strategy is to minimize interprocess communication by clustering as many timing constraints as possible into each process. This is done by partitioning the computation required by the timing constraints into sets such that (i) only compatible timing constraints are assigned to the same set (Two periodic timing constraints are compatible iff they have the same period or one period divides the other), and (ii) two compatible timing constraints are assigned to the same set if some of the operations (function calls) required by them are the same. The computation in each set is assigned to a periodic process whose period attribute is set to the highest common factor of the periods in the set. Each asynchronous timing constraint is assigned to a sporadic process as before. (In fact, we may want to satisfy an asynchronous timing constraint by means of an "equivalent" periodic process and do away with sporadic processes altogether.) Under this decomposition strategy, the design solution now requires two instead of three processes.

```
/* COMMENT
   The process XYSK replaces the two processes XSK, YSK in the previous
   solution. Since YSK needs to be executed only every 160 ms, a boolean
```

```
   procedure skip_Y is used every 80 ms to determine if f_Y need to be
   executed. (This procedure may be implemented by using the real-time clock
   or simply by toggling a static boolean variable.) The sporadic process
   ZS associated with the signal z and the monitor S are the same as before.
*/

process XYSK

activated by timer;
attribute period = 80, deadline = 80;

x := sensor_x();
x' := f_X(x);

if skip_Y() = FALSE then { y := sensor_y(); y' := f_Y(y); }
rendezvous S;
v := f_K(u);

end XYSK
```

In this solution, f_S and f_K are executed only once every 80 ms instead of three times every 160 ms. With this improvement, we can in fact sample x and y at respectively 60 and 120 ms and guarantee to meet a 60 ms deadline for responding to a change in the signal z. These tighter parameters are impossible to meet with the previous decomposition strategy. Figure 3.4a shows an execution sequence of this program.

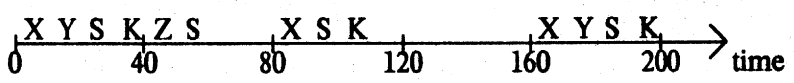


Figure 3.4a
Example Gantt chart for decomposition by minimizing communication

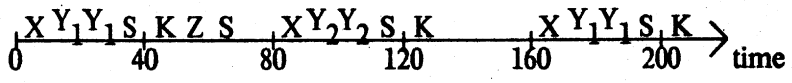


Figure 3.4b
Example Gantt chart for two-stage pipeline implementation
(Y takes 40 msec and executes in two stages)

Figure 3.4
Example of decomposition by minimizing communication and pipelining

In general, this decomposition strategy improves efficieny in two respects. First, it may eliminate substantial redundant computation among compatible timing constraints. Second, since there are fewer processes and the processes tend to be independent (they have fewer common operations), less interprocess communication is required for concurrency control. Whereas in principle this approach can be pushed to the extreme by clustering all the required computation into a single process, it defeats the whole purpose of a process as a structural unit for software design. Efficiency improvement gained in this way must be paid for by an increase in design complexity which in turn makes maintenance more difficult.

Because the nominal parameters of our design example are rather lax, process XYSK is only slightly more complicated than the two processes it combines. Nevertheless, some scheduling decisions must now be programmed inside a process (calling skip_Y so as to execute f_Y only in alternate cycles). In general, the control logic which implements these internal scheduling decisions is likely to be sensitive to system parameters and can be quite ad hoc when resource allocation is highly optimized. For example, if the function f_Y is replaced by a new version which requires 40 ms instead

of 10 ms computation time (field trials suggest that the value y' must be computed with double precision!), then the above program for XYSK will not work. For in the worst case, there is only 80 ms to execute both XYSK which now requires a maximum of 70 ms, and ZS which requires 20 ms. However, if the computation in f_Y is split into two stages of 20 ms each, then we can compute one stage of f_Y in each 80 ms and thereby meet all the timing constraints. The modified program for the process XYSK is shown below. An example execution sequence is shown in figure 3.4b.

```
/* COMMENT
   This is a modification of the previously shown version of a periodic
   process for meeting the two timing constraints for x and y.
   The computation y' := f_Y(y) is performed in stages by two assignments:
   (1) temp := f_Y1(y) to be executed in the first half of a 160 ms cycle
   (2) y' := f_Y2(temp) to be executed in the second half of the cycle
*/

process XYSK

activated by timer;
attribute period = 80, deadline = 80;

x := sensor_x();
x' := f_X(x);

if skip_Y() = FALSE then { y := sensor_y(); temp := f_Y1(y); }
else y := f_Y2(temp);

rendezvous S;
v := f_K(u);

end XYSK
```

By implementing f_Y as a two-stage pipeline, two successive samples of x are processed in 160 ms while the same sample of y is processed in both halves of a 160 ms cycle. We would like to draw attention to the fact that it is impossible for any process scheduler to automatically simulate this "pipelining" technique with the previous version of XYSK since at most one sample of x can be processed in each activation of XYSK and a process can have only one thread of control at a time!

As might be expected, the optimized design is more difficult to understand since there is no logical necessity for explicitly splitting a function into stages. Modifications are harder to make since a local change in parameter value may bring about substantial reorganization in the control logic elsewhere, e.g., if f_Z requires 35 instead of 10 ms to execute while all other parameters retain their nominal values, then it is still necessary to pipeline f_Y into two stages of 5 ms each. In general, optimization measures which minimize interprocess communication can easily create maintenance night-

mares for human programmers. However, when interprocess communication costs are high and if "spaghetti control logic" is tolerable, then this decomposition strategy may be preferable.

3.2.6 Decomposition by Maximizing Concurrent Processes

The objective of this decomposition strategy is to partition the required computation into as many processes as possible so as to maximize parallelism. The decomposition procedure is best explained in terms of the data flow graph of the given problem such as the function block diagram of figure 3.1. Specifically, a periodic process is created for each node in the data flow graph. We stress the distinction between a node which represents an operation on some data flow path and the function which is called to process the data passing through the node. Thus it is possible for two or more processes to call the same function in which case a monitor is needed to enforce mutual exclusion. In general, a node may be involved in the computation required by one or more periodic timing constraints, and the process assigned to the node is given a period attribute equal to the highest common factor of the periods of the relevant timing constraints. Each asynchronous timing constraint is assigned a sporadic process which contains the appropriate function calls. (Again, there is the possibility of satisfying all the asynchronous timing constraints by means of "equivalent" periodic processes, in which case no process will need to call more than one function.)

When a periodic process is activated, it must synchronize with an appropriate set of processes which precede it, call the function to perform the operation associated with it, and then synchronize with the set of processes which it precedes. Intuitively, the predecessors of a process P at time t correspond to the operations before P required by some timing constraint whose period divides t. (The definition of a predecessor relation will be made clear in our graph-based computation model later.) Under this

decomposition strategy, the design solution now has five processes. The following program shows the four periodic processes. The sporadic process for z and the monitor S are the same as before.

```
/* COMMENT
   This program uses a process for each node in the data flow graph of the
   design problem (figure 3.1). The period attribute of a process is set to
   the highest common factor of the periods of the periodic timing constraints
   that require the execution of the corresponding operation.
*/

process X
activated by timer;
attribute period = 80, deadline = 80;

x := sensor_x();
x' := f_X(x);
rendezvous S;
end X


process Y
activated by timer;
attribute period = 160, deadline = 160;

y := sensor_y();
y' := f_Y;
rendezvous S;.
end Y


process XYS
activated by timer;
attribute period = 80, deadline = 80;

rendezvous X;
if skip_Y() = FALSE then rendezvous Y;
rendezvous S
end XYS


process K
activated by timer;
attribute period = 80, deadline = 80;

rendezvous S;
v := f_K(u);
end K
```

Since the nominal parameters have been set so that computation time predominates the costs for communication and concurrency control (e.g., the nominal cost of a rendezvous, c_{sys} is 0), a wider range of timing constraints can be enforced when each process is run on a separate processor. The timing diagram of Figure 3.5 shows an example execution sequence with five processors such that the input signals x and y are sampled at respestively 30 and 60 ms, and the deadline for responding to a change in z can be as tight as 30 ms.

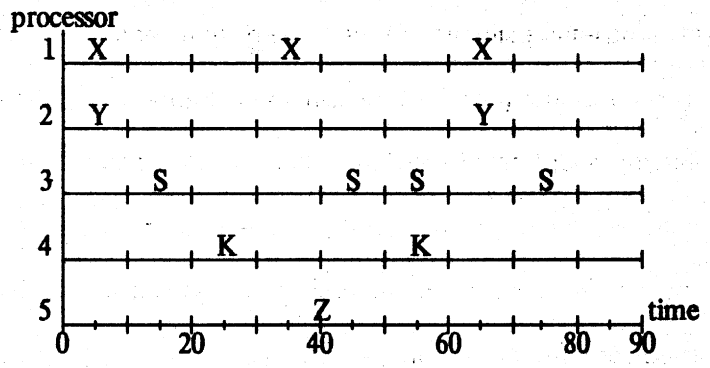


Figure 3.5
Example Gantt chart for decomposition by maximizing concurrent processes

By assigning a separate process to each function block, this decomposition strategy maximizes the computation that can be performed in parallel. Redundant computation is reduced since timing constraints that require the same operation to be performed in compatible time intervals are recognized in the construction of the synchronization code for each periodic process. If as many processors are available as there are processes and computation time indeed dominates c_{sys}, then this decomposition strategy will generally tolerate a wider range of timing constraints than the others. Like the previous strategy, however, there is also a price to be paid in the increased complexity of the design solution.

3.2.7 Comparison of Decomposition Strategies

The semantic gap between a process-based computation model and the hard real-time environment is best illustrated by a comparison of the three decomposition strategies presented above. If feasibility is the only concern, then decomposition by timing constraints would be the least preferable. In general, a feasible solution will depend on the tradeoff between computation and communication costs. When interprocess communication costs are relatively low, decomposition by maximizing concurrent processes is more likely to succeed, e.g., the tightest timing constraints for the design example are achieved by this approach. This will not be the case when interprocess communication costs become significant.

As an illustration, let each rendezvous statement add an overhead of 10 ms (which is the nominal computation time of a function call) to the computation time of either participating process. Specifically, $c_{sys} = 10$ ms if the participating processes reside on separate processors and $c_{sys} = 5$ ms if they are on the same processor. If the five processes resulting from the last decomposition strategy are run on separate processors, it is easy to check that p_x and p_y cannot be simultaneously shorter than 60, 120 ms respectively while at these sampling rates, d_z cannot exceed 50 ms. On the other hand, if we use two processors to implement the decomposition by minimizing communication strategy, then p_x, p_y can be held to 50 and 100 ms respectively without adversely affecting d_z. More surprisingly, we need only one processor to achieve the same periodic sampling rates if a longer d_z (70 ms) can be tolerated. Specifically, the the computation required by all the timing constraints can be clustered into a single periodic process as follows.

```
/* COMMENT
   The process XYZSK replaces the two processes XYSK, ZS in a previous
   solution. Since ZS needs to be executed only when z changes value,
   a boolean procedure skip_Z is used to determine if f_Z need to be
```

```
   executed. (This procedure may be implemented by comparing the new
   reading from the sensor for z with its previous value which is kept in
   a static variable.)
*/

process XYZSK

activated by timer;
attribute period = 50, deadline = 50;

x := sensor_x();
x' := f_X(x);

if skip_Y() = FALSE then { y := sensor_y(); y' := f_Y(y); }

if skip_Z() = FALSE then { z := sensor_z(); z' := f_Z(z); }

u := f_S(x',y',z',v);

v := f_K(u);
end XYSK
```

The technique employed above is an example of *latency scheduling* which will be formalized and studied later. In general, it works by (more or less) periodically performing the computation required by asynchronous timing constraints while taking advantage of the computation that is already required by the periodic timing constraints. For now, it suffices to note that the strategy of minimizing interprocess communication may be pushed to the extreme by clustering all the computation in a single periodic process. When interprocess communication overhead is predominant, there may be only one process in a feasible decomposition whereas many processes may be needed in the case where it is crucial to maximize concurrent processes. In between the two extremes lie a wide range of alternatives.

Aside from efficiency, an important criterion for comparing decomposition strategies is the maintainability of the resultant design. Intuition suggests that decomposition by timing constraints should rank highest among the three. But unfortunately, there is currently no concensus on how maintainability should be defined and hence, the validi-

ty of our evaluation is necessarily a matter of judgement. However, we believe that a reasonable measure of maintainability is the stability of a design against changes in the problem specification. Specifically, let us consider the minimum adjustments that must be made when (a) a period or deadline assumes a different value; (b) a new timing constraint is added or an old one is deleted.

If the decomposition is along timing constraints, then (a) will simply require a scheduling attribute to be updated and (b) will require the creation of a new process or the deletion of an old one. Both adjustments are straightforward and involve only one process. If the decomposition has been to minimize interprocess communication, then both (a) and (b) may require updating a scheduling attribute and/or modifying the control logic in one process; (b) may also require creating or deleting a process. The adjustments involve only one process but may now require the maintainer to modify some "spaghetti control logic". If the decomposition has been to maximize concurrent processes, then both (a) and (b) may require substantial modifications to the scheduling attributes and control logic of a number of processes.

Although decomposition by timing constraints seems to require the least adjustment in response to specification changes, it should be said that maintainability and efficiency are not entirely separate issues. If a change in the specifications causes some deadline to be missed after the straightforward design adjustments have been made, then either faster hardware must be bought or a different decomposition strategy must be pursued. This type of major overhaul is less likely with a more efficient decomposition strategy.

3.2.8 Implications of The Semantic Gap on Software Automation

In order to automate the design and maintenance of software to run in the hard real-time environment, we need a computation model with which to express the compu-

tational requirements of a system. Ideally, appropriate software tools can then be built to translate an instance of the model all the way to executable code. In the traditional approach, a process-based model is almost invariably chosen. Unfortunately, methodologies which use a process-based model to define design requirements are necessarily limited in their usefulness inasmuch as the first precise problem representation that an automation tool can work on is a set of processes.

As the above discussion suggests, the semantic gap between a process-based model and the hard real-time environment has serious implications with regard to the systems issues raised in an earlier chapter. To wit: (1) The *maintainability/efficiency dichotomy* has been amply demonstrated. (2) *System integrability* suffers since a process is essentially an abstraction of the traditional von Neumann computer architecture and may be difficult to map directly into other types of machine architecture, e.g., VLSI systolic arrays. A uniform way is lacking for determining the feasibility of a set of processes to be implemented on a combination of current computers and other types of computing resources. (3) *Implementation independence* is limited by the natural bias for implementing processes on the traditional architecture because of the obvious efficiency advantages.

We contend that these are sufficiently strong reasons for finding an alternative model of computation which is semantically closer to the hard real-time environment. It ought to be admitted, however, that the validity of our contention is necessarily a matter of judgement in the absence of more discriminating metrics.

3.3 Definition of a Graph-Based Computation Model

The purpose of the computation model underlying a design methodology for hard real-time systems is to provide an abstract representation of a design problem with sufficient precision to allow the specification of stringent performance requirements so that automation tools can be built for resource allocation and feasibility analysis. As such, the model should be as close as possible to the problem representation familiar to control system designers. To this end, we adopt a graph based model which is intended to capture the data flow and computational requirements that control engineers often describe via a block diagram.

Our model is a tuple (G,T) where G is a *communication graph* describing the data dependency among the operations (functional elements) of the system, and T is a set of timing constraints. Specifically, $G = (V,E,W_V)$ is a digraph (which may contain cycles) where V and E are respectively the set of nodes and edges, and W_V is a function which assigns a non-negative integer weight to each node in V. The nodes denote functional elements which take their inputs from the incoming edges and produce outputs on the outgoing edges. Edges denote data paths connecting functional elements and two nodes may be connected by more than one edge. The weight of a node is the computation time of the corresponding functional element. Edges may be labelled by the names of the variables whose values are transmitted along the corresponding edges. To simplify drawing, some edges in a communication graph may not have an originating or destination node in which case the omitted node is understood to denote the external environment.

T is the union of two finite sets of timing constraints: T_p (periodic timing constraints) and T_a (asynchronous timing constraints). Each timing constraint is a tuple (C,p,d) where p, d are respectively the *period* and *deadline* which are non-negative integers and C (the timing constraint graph) is an **acyclic** graph compatible with the

communication graph G. We say that the graph C is *compatible* with the graph G if there is a mapping h such that: (1) If v is a node in C, then h(v) $\in$ V; and (2) If e is an edge from a node u to another node v in C, then h(e) is an edge from h(u) to h(v) in E. A timing constraint graph C is meant to define the precedence relation of the computational events that must occur to satisfy a timing constraint. A node in the timing constraint graph C denotes an operation (an execution of the corresponding functional element in the communication graph). An edge in the graph C denotes the transmission of some output value from one functional element to another. We do not rule out multiple instances of the same node or edge of the graph G in the graph C so as to allow for limited iteration. The computation time w_i of the i^{th} timing constraint T_i is the sum of the weights of the nodes in the timing constraint graph C. If the timing constraint is periodic, i.e., (C,p,d) $\in T_p$, then it is activated every p time units, starting from time = 0. If it is asynchronous, i.e., (C,p,d) $\in T_a$, then it can be activated at any time t for any non-negative integer t with the provision that two successive activations must be at least p time units apart.

A timing constraint graph C is said to be executed in a time interval I if a subset of the (multi)set of operations that have been executed in I forms a partial order such that: (1) There is a bijective mapping between the operations in the partial order and C. (2) Under this mapping, the partial order is consistent with C. (3) In the case the operations are distributed, an execution of C must also include the transmissions of data that are denoted by the edges of C. More precisely, if the graph C contains an edge from the node u to the node v, then an execution of C must include the transmission of the last output of u to v before the output of v can be computed. When a timing constraint is activated at time t, the corresponding timing constraint graph must be executed once in [t, t+d].

Intuitively, the set T defines all the computation that the system is required to per-

form in real time. The purpose of the compatibility condition is to make explicit any communication that may be required for synchronization purposes. (Another way to interprete compatibility is that one operation need to precede another only if the output of the former is an input to the latter.) We also allow the same operation to appear more than once in a timing constraint graph so as not to rule out bounded iteration.

(As an option, a timing constraint C may also have an non-negatrive integer *release time* attribute, r in which case there must be an execution of C in $[t+r,t+d]$ whenever the timing constraint is activated at time t. In general, the addition of a release time attribute will not affect the complexity of the related scheduling problems and we shall assume that $r=0$ to simplify our discussion.)

Asynchronous timing constraints are usually activated by the occurrence of an external event or when some predicate on the state variables of the physical process under control is satisfied. In either case, we assume that mechanisms exist for an activation condition to be automatically detected, e.g., interrupt detection hardware. However, if an activation must be detected by explicitly evaluating a predicate, then the computation involved must be included in the graph and the timing constraint activated whenever the variables in the predicate change value. (For scheduling purposes, we must assume that in the worst case, the predicate is satisfied every time it is evaluated.) Alternatively, the designer may specify a periodic timing constraint to evaluate the predicate and to perform the required computation at a chosen rate.

In addition to the critical time parameters, there are other scheduling constraints that must also be observed in performing real-time computation so that data integrity is preserved. As a motivation for these data integrity constraints, we shall first illustrate the use of the graph-based model by defining the computation requirements of the previous example design problem in figure 3.6

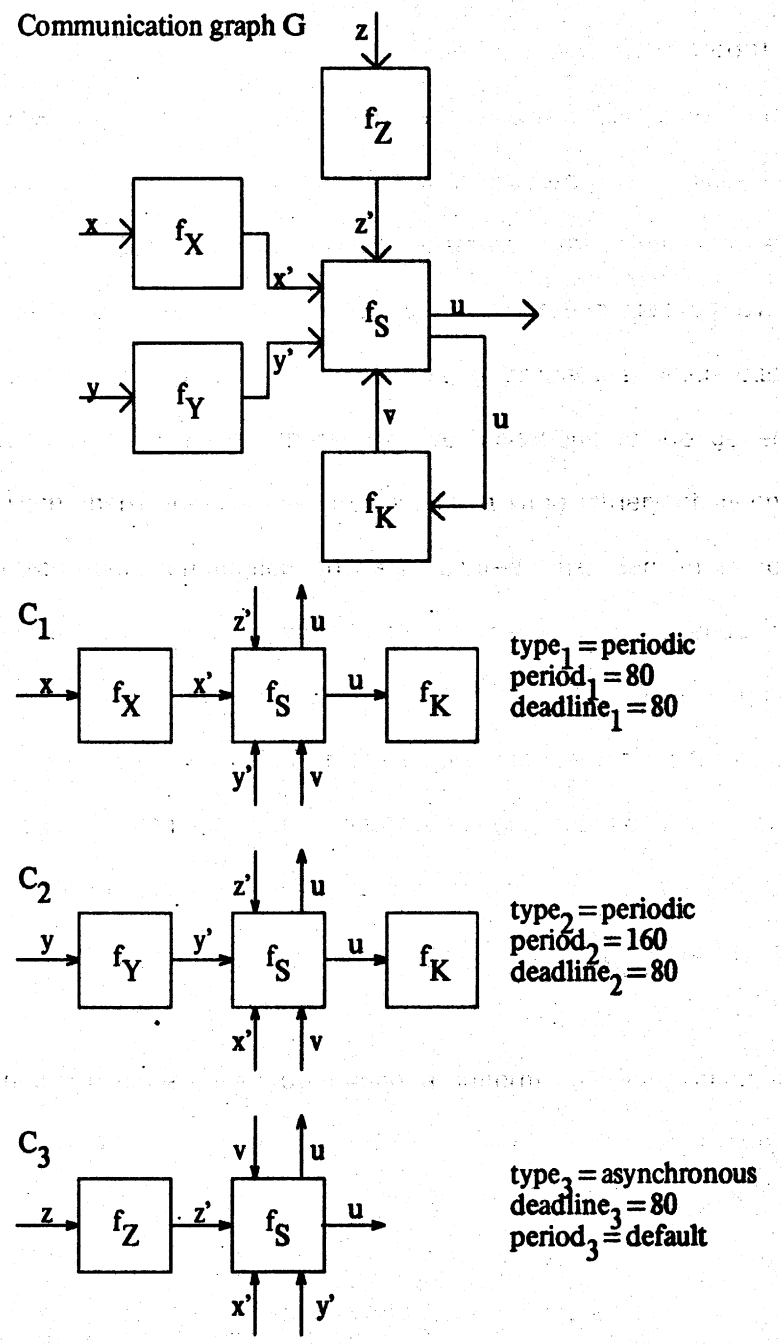


Figure 3.6

Specification of Design Example in Graph-based Model

Notice that the graphs for both periodic and asynchronous timing contraints contain the edge (labelled z') from the operation f_Z to the operation f_S but only the asynchronous timing constraint requires the operation f_Z. Suppose that the two operations f_Z and f_S are implemented on separate processors and that the input z changes value when the last value of z' is being transmitted to f_S as part of the execution of a periodic timing constraint. In response to the asynchronous timing constraint for the signal z, the function f_Z is executed and a new value of z' is also transmitted to f_S. An anomaly may occur if the new value of z' arrives at f_S before the old one does since in this case, the output of f_S will be based on the old value of z after the function f_S is executed for a second time and will remain so until the next execution of the periodic timing constraint. In order to eliminate this kind of anomaly, we require real-time computation to be *pipeline-ordered*.

(1) Two executions of the same functional element in G are pipeline-ordered if they have distinct start-times and that the execution which has an earlier start-time must also have an earlier finish-time than the other.

(2) Two data transmissions along an edge connecting the functional element u to the functional element v in G are pipeline-ordered if they are sent at distinct instants at the site of u and that the earlier transmission must also be received earlier at the iste of v.

When all executions of operations and data transmissions are pipeline-ordered, it can be easily shown by induction that if the output of each operation is given an upward counting version number, then the input used in the execution of every operation must show only increasing version numbers. (We do not require the version number of an input used in an execution to be exactly one bigger than that used in the previous execution.)

The graph-based model is an attempt to abstract the computational events in real-

time feedback control problems, e.g., avionics systems, industrial processes. While we do not claim universality for our model, we note that the usual state space formulation familiar to control engineers can be naturally translated into our model. From a computational point of view, the graph-based model is also capable of simulating a restricted version of the well known data flow model of computation. In fact, any data flow schema which has only bounded iteration constructs (with fixed limits) can be simulated by a single asynchronous timing constraint with appropriate time parameters. (While simple conditionals can be readily incorporated into our model, unbounded iteration is fundamentally inconsistent with real-time computation.)

3.3.1 Scheduling Problems with the Graph-Based Computation Model

The graph-based computation model provides us with a language which does not suffer from the semantic-gap problems confronting process-based models in defining the computation requirements of hard real-time systems. An obvious strategy for automation is to design algorithms to decompose the computation defined by an instance of the graph-based model into an appropriate set of processes which can be run on the available processor(s). However, there are two limitations to the generality of this approach. First, when there are multiple processors, a wide variety of mechanisms may be used for interprocessor communication, e.g., shared bus, Banyan switch, multiport memory. The optimality of a decomposition algorithm necessarily depends on the peculiarities of the interconnection devices so that the applicability of any one decomposition algorithm is limited. Second, some of the operations performed by a real-time system may be best implemented by special hardware, e.g., systolic arrays for signal processing. Thus an optimal decomposition algorithm must also be able to make efficient use of special devices which may be awkward to model in terms of the process abstraction.

The first limitation can be eased by finding a uniform characterization of communication resources for the hard real-time environment and will be dealt with in the next chapter. It suffices to mention here that the decomposition problem is in general computationally intractable (NP-hard). The second limitation is a more fundamental difficulty with the process abstraction. In practice, the decomposition of the computation required by a real-time system may also be subject to artificial constraints which stipulate that certain operations must be executed on the same processor. In any case, a decomposition algorithm must be able to decide if the computation assigned to a processor can indeed be scheduled. In this chapter, we shall examine the single processor scheduling problem for the graph-based computation model, and in particular,

the technique of *latency scheduling*.

3.3.2 Design Constraints on the Run-Time Scheduler

Given a problem instance in terms of the graph-based model (G, T), our objective is to design a run-time scheduler which will execute the operations in G in an appropriate sequence so that the timing constraints in T are satisfied. In addition, the run-time scheduler must also guarantee that the data integrity constraints are observed. The following theorem guarantees that the computation performed on a single processor implementation will be pipeline-ordered by introducing two implementation assumptions.

Theorem 3.1

If the following two implementation assumptions are satisfied, then the computation performed on a single processor will be pipeline-ordered.

(1) The outputs of a functional element u are stored in unique variables which are updated as the last action in an execution of u. The inputs of a functional element v are parameters which are passed by value to v when v is executed.

(2) The run-time scheduler will permit only one execution of an operation to be in progress at any time (i.e., an operation is not permitted to preempt itself), but we do not preclude the preeemption of an operation by a different one.

Proof: follows directly from the definition of pipeline-orderedness.

We shall be primarily interested in exploiting the class of run-time schedulers that can be simulated by a round-robin scheduler which makes scheduling decisions by repeating a precomputed (possibly very long) schedule. If all the timing constraints are periodic, then this class of schedulers is as powerful as any run-time scheduler. The techniques developed for the process-based models can be used to help solve the

scheduling problem. For dealing with asynchronous constraints, some advantages of our approach are: (1) It opens up the possibility of exploiting the computation that is already required to satisfy the periodic timing constraints. (2) The on-line computation required to make a scheduling decision at run time may be shifted off-line. (3) It is relatively easy to trace back a segment of the immediate history of a computation at any point during run time, a significant help for debugging systems which must cope with asynchronous events.

Given a round-robin scheduler, the scheduling problem of interest is to decide whether there is a finite schedule which the scheduler can repeat to meet all the asynchronous timing constraints. To study this problem, we now formalize the concept of *latency scheduling* (first introduced in [WARD 78].)

3.4 Latency Scheduling

We shall first introduce some terminology.

An *execution trace* of a processor is a mapping F from the non-negative integers to the set of nodes (functional elements) in a communication graph G plus a null symbol φ such that F(i) = u if the scheduler executes u the time interval [i,i+1] and F(i) = φ if the processor idles in that interval. (The null symbol φ may be subscripted with an integer to indicate the length of the idle interval.)

An execution trace can be represented by a semi-infinite string of φ symbols and node labels with the interpretation that the scheduler performs the first operation v in $[0, c_v]$ where c_v is the computation time of v, and then sequence through the rest of the operations in the string. For any pair of non-negative integers s and t, the function F induces a natural mapping which assigns to every time interval [s,t], the finite string of operations which are **completely** executed in that interval, i.e., (with a slight abuse of notation) the first operation of F([s,t]) starts no earlier than time = s and the last operation completes no later than time = t. Using our favorite design example (with all functions having a nominal computation time of 10 ms), the run-time scheduler which repeats the string:

"f_X f_Y f_Z f_S f_K"

will generate an execution trace F where F([0,20]) = "f_X f_Y" and F([15,25]) = ω, the null string.

For brevity, a timing constraint (C,p,d) will be denoted simply by C whenever there is no confusion. An execution trace F is said to contain an execution of the timing constraint C in the time interval [s,t] iff the sequence of operations in F([s,t]) contains a subsequence S such that (1) There is an isomorphic mapping between S and the nodes of the timing constraint graph C; and (2) The linear order of S is consistent with the precedence relation defined by the acyclic graph C. The execution of the timing

constraint C starts at the start-time of the first operation in the subsequence S and finishes at the time when the last operation in S completes execution.

An execution trace F is said to have a *latency* of l time units with respect to the timing constraint C iff for every non-negative integer s, F([s,s+l]) contains an execution of C.

We remark that the above definition implies that if an execution trace $\alpha = \delta\gamma$ (where δ is a finite suffix of α) has a latency of l with respect to the timing constraint C, then γ is also an execution trace with a latency of l with respect to C.

A *static schedule* L (a finite string of operation symbols) is said to have a latency l with respect to the timing constraint C iff the execution trace F which a round-robin scheduler generates by repeating L *ad infinitum* has a latency of l with respect to C.

If a static schedule L has a latency l with respect to an asynchronous timing constraint C such that $l \leq d$ (the deadline of C), then a round-robin scheduler which repeats L *ad infinitum* obviously satisfies C. A static schedule L is said to be *feasible* with respective to a set of asynchronous timing constraints T_a if for every $C \epsilon T_a$, L has a latency $\leq d$, the deadline of C. For our design example, the static schedule:

"$f_X\ f_Y\ f_Z\ f_S\ f_K\ \varphi_{10}$"

has a latency of 80 ms with respect to the asynchronous constraint which requires the functions f_Z and f_S to be executed within 80 ms of an activation. This is in fact a feasible schedule which meets all the nominal timing constraints of the design problem. We shall use the terms latency and deadline interchangeably as long as it causes no confusion.

Since external events may not occur at integer points on the time axis, the physical response time guaranteed by our definition of latency may be late by a fraction of our chosen time unit. However, we can allow latency to be measured on the real line

by making the following observation: If an integer latency d must be satisfied for any interval on the real line, a static schedule is feasible if and only if it has a latency of d-1 when measurements are taken at integer points only. We shall be concerned with discrete-time time systems only.

3.4.1 Upper Bound on the Length of a Static Schedule

Given a finite set of n asynchronous timing constraints { C_i }, i = 1,...,n, our problem is to determine if there is a feasible static schedule and how hard it is to compute it. Notice that there are two potential reasons why a static schedule may not exist: (1) The deadline specifications are too tight to be met. (2) Any feasible schedule might be necessarily infinite in length, i.e., any execution trace which has latencies $\leq$ the deadlines of the respective timing constraints must be aperiodic. (Consider the proposition that there is a set of asynchronous timing constraints involving 10 operations labelled from 0 to 9 such that the only feasible schedule is represented by the fractional expansion of π.) We shall first give an upper bound on the length of a static schedule if one exists and show that (2) is impossible.†

The basic idea of our proof is to demonstrate that any feasible execution trace (one that meets all the latency requirements) can always be simulated by a program with a finite number of states. Without loss of generality, we shall assume that all operation symbols in an execution trace have unit computation time. (We can always replace every node v in a communication graph G by a chain of c_v uniquely labelled nodes where c_v is the computation time of v.) The simulation is done by moving *timers* around the timing constraint graphs each of which is augmented by a new node and extra edges as follows. For the i^{th} timing constraint C_i, we add a node called *sink* and we add an edge to *sink* from every node in C_i which does not have any successor. Each *sink* node also has an *alarm* which increments with time and is reset to a new value whenever an execution of the timing constraint C_i completes. (When there is no

† Teixeira [TEXI 78] has also proved that (2) is impossible for a computation model pertinent to the monitoring of bandwidth-limited analog signals which propagate through an acyclic network of function elements.

confusion, C_i will be used to denote the corresponding augmented graph.)

Intuitively, a timer records the "age" of an execution of the timing constraint C_i that is in progress. Specifically, a timer of value t on the output edge of a node v indicates that there is an execution of v and all its predecessors starting exactly t time units ago. During the simulation of a feasible execution trace, a bounded number of timers may exist on an edge, but no two timers on the same edge will have the same value. The *alarm* records the time that has elapsed since the start-time of the latest completed execution of C_i. Given an execution trace, the simulation proceeds by sequentially scanning the string of operation symbols and follows the simulation procedure below. Each *round* of the simulation starts at step (2) and ends when the next operation symbol in the execution trace is to be scanned at step (2) or (6). An example of the simulation procedure is shown in figure 3.7.

Simulation Procedure

(1) Initialize the *alarms* to 0. The graph is initially clear of timers.

Apply the following procedure to every augemented graph C_l.

(2) Scan the next symbol v in the execution trace. Increment all the timers on the graph and the alarm by one. If v is φ, then go to (2). (The current round of the simulation ends.) Else go to (3).

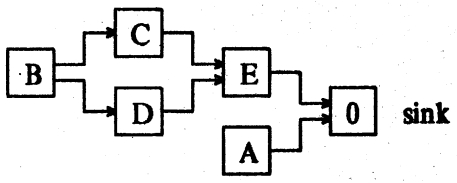
(3) Let S be the set of nodes with label v such that each node in S either has no input edge or has at least one timer on each of its input edge(s). For every node x in S which does not have any input edge, remove x from S and create a new timer (initialized to 1) to be added to every output edge of x after S becomes empty.

(4) If S is empty, update the appropriate output edges and go to (6). Else go to (5).

(5) For every node x in S, repeat the following until at least one of the input edge(s) of x becomes empty and then remove x from S: Let the maximum value of the timers on the input edge(s) of x be t; remove all the timer(s) with value t from the input edge(s) of x and create a timer with value t to be added to every output edge of the node after S becomes empty. When S is empty, update the appropriate output edges and go to (6).

(6) If the *sink* has an empty input edge, then go to (2). (The current round of the simulation ends.) Else go to (7).

(7) Repeat the following until at least one input edge of the *sink* becomes empty: Let the maximum value of the timers on the input edge of the *sink* be t; remove every timer with value t from the input edge(s) of the *sink* and reset the *alarm* to t.



Augmented timing constraint graph

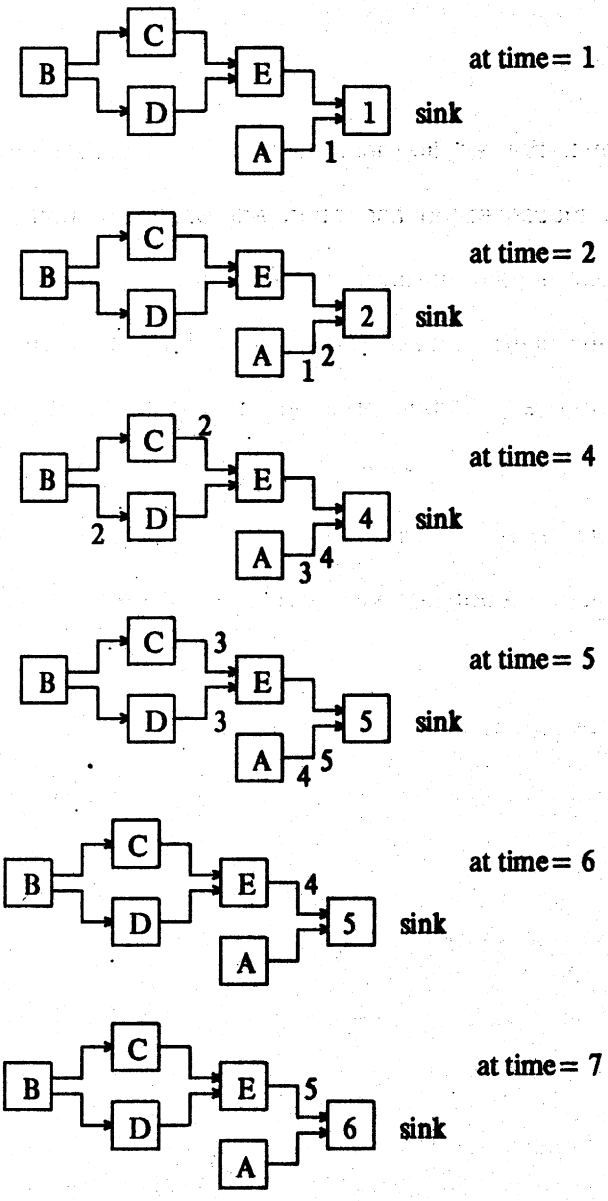


Figure 3.7
Simulation of the execution trace "AABCDEE"

The simulation procedure is designed to ensure that an execution trace satisfies all the latency constraints if and only if none of the *alarms* exceeds the corresponding deadline at the end of every round during a simulation. The distribution of timers and their values on the edges of the augmented graphs together with the values of the *alarms* at the end of a round of simulation constitutes a state of the simulation. The upper bound of a static schedule is obtained by considering the number of states any feasible simulation can be in. In what follows, the simulation clock always starts at time 0 and is incremented at the end of each round. Without confusion, the letter F will be used to denote the execution trace under consideration.

Lemma 3.2

At the end of every round of a simulation, the timer(s) on the output edge(s) of every node are strictly greater than the timer(s) on the input edge(s) of the same node.

Proof

Suppose that the lemma holds after k symbols of an execution trace have been scanned. If the next symbol v is φ, or if all the nodes with label v have either no input edge or at least one empty input edge, then all the timers are incremented by the same amount and the lemma holds. If a node with label v has no input edge, then step (3) is performed and a new timer (initialized to 1) is added to each of its output edges. Since all the timers already on the graph have been incremented at step (2), the new timers created at step (3) must be strictly smaller than the ones already on the graph. So the lemma holds. If a node with label v has at least one timer on every input edge, then step (5) is performed and the new timer(s) added to the output edge(s) of v are initialized to the maximum, t of the timer values on the input edge(s). But all the timers with value t are removed from the input edge(s) in the same step, and so the new timer(s) on the output edge(s) must be strictly greater than the remaining ones on the

input edge(s). Hence the lemma holds at the end of processing the $k+1^{th}$ symbol. Since the lemma is trivially true for k = 0 when the graph does not have any timer, it must hold at every step of the simulation. QED

Corollary 1

During a simulation, the values of new timers added to an output edge must be strictly smaller than the ones already on the edge and hence no two timers on the same edge of an augmented graph can have the same value at the end of every simulation round.

Proof

New timers created at step (3) are initialized to a value strictly smaller than the timer(s) already on the edge. Timers created at step (5) are to be added to an output edge but are initialized to the value of a timer on an input edge. By the lemma, they must be strictly smaller than the ones already on the (output) edge. QED

Corollary 2

At the end of every round of a simulation, the value of the *alarm* is at least as big as any of the timers on the corresponding graph.

Proof

Since the *alarm* has been initialized to 0, its value must be at least as big as any timer on the graph before it is reset for the first time. Step (7) of the simulation procedure always resets the *alarm* to the maximum of the timer value(s) that are on the input edge(s) of the *sink*. The result follows from the acyclicity of the augmented timing constraint graphs. QED

Lemma 3.3

If F([0,t]) contains an execution of a node and all its predecessors, then at least one timer will be added to the output edge(s) of the node before the end of the simulation round which ends at time t.

Proof

If the node has no predecessor, then a timer will be added to its output edge(s) the first time it is executed. Suppose the node has predecessors and that the lemma holds for all of its predecessors. Let r be the smallest integer such that F([0,r]) contains an execution of the node and all its predecessors. Then F([0,r–1]) must contain an execution of all the predecessors of the node and by the induction hypothesis, at least one timer must have been added to every input edge of the node before the end of the simulation round which ends at time r–1. If any of these timers is removed, then a new timer will be added to the output edge(s) of the node. If none of them has been removed before time r–1, a timer will be added to the output edge of the node by step (5) in the simulation round which ends at time r, thus completing the induction step. QED

Lemma 3.4

During the simulation round ending at time t, if a timer of value d is added to the output edge(s) of a node by step (3) or (5) of the simulation procedure, then F([t–d,t]) contains an execution of the node and all its predecessors. If the last timer added to the output edge has value x, then t–x is the start-time of the latest execution of the node and its predecessors up to time t, i.e., t–x is the largest integer such that F([t–x,t]) contains an execution of the node and all its predecessors. If no new timer is added to the output edge(s) of a node in this round, then the start-time of the latest execution of the node and all its predecessor has not changed since the last round,

i.e., either (i) F([0,t]) does not contain an execution of the node and all its predecessors; or (ii) if d is the smallest integer such that F([t-1-d,t-1] contains an execution of the node and its predecessors, then d+1 is the smallest integer such that F([t-(d+1),t]) contains an execution of the node and its predecessors.

Proof

If the node has no predecessor, then a timer with value 1 is added to its output edge(s) by step (3) of the simulation procedure if and only if the node is executed in [t-1,t]. Thus the lemma holds for all nodes which do not have any predecessor.

Suppose the node has predecessors and the lemma is true for all the predecessors of the node. If a timer with value d is added to its output edge(s), then every input edge of the node must have at least one timer with value $\geq$d before the new timer is created by step (5). Any one of these timers must have been added to the corresponding output edge of an immediate predecessor of the node at an earlier round which ends, say, at time r. If this timer was initialized to a value y, then by the induction hypothesis, F([r-y,r]) must contain an execution of this immediate predecessor and all its predecessors. In the interval [r,t], this timer increases its value from y to d, i.e., t-r = d-y and so F([r-y,r]) = F([t-(d-y)-y,r]) = F([t-d,r]). In other words, every immediate predecessor of the node and all their predecessors must have been executed in [t-d,r] for some r < t. Therefore F([t-d,t]) must contain an execution of the node and all its predecessors which starts at time s-y, i.e., at t-d.

When the last timer is added to the output edge of the node, an output edge of one of its immediate predecessors must have become empty. If this timer has value x, then the smallest timer on the corresponding output edge of this immediate predecessor must have value x-1 at the end of the simulation round which ends at t-1. By the first corollary of lemma 3.2, this timer must be the last one added to the output edge

of the immediate predecessor and hence by the induction hypothesis, x is the smallest integer such that F([t–1–x,t–1]) has an execution of this immediate predecessor and its predecessors. Hence, x is the smallest integer such that F([t–x,t]) contains an execution of the node and its predecessors.

If no new timer is created to be added to the output edge(s) of the node, then there are two cases. In the first case, there has never been an execution of the node and its predecessors in [0,t] and so the start-time of the latest execution of the node and its predecessors must not have changed since the last round. In the second case, F([0,t]) contains an execution of the node and it predecessors, but at least one of the immediate predecessor(s) of the node must have an empty output edge at the end of the round ending at time t–1. By lemma 3.3, at least one timer must have been added to the this edge before the simulation round which ends at time t. Consider the last timer removed from the output edge of this immediate predecessor. Let this timer be removed from the input edge of the node during the simulation round ending at time r and let its value be x. By the induction hypothesis, the latest execution of this immediate predecessor and its predecessors must have started at time r–x. Hence, the latest execution of the node and its predecessors must have started at time $\leq$r–x. Since a timer with value x is added to the output edge(s) of the node at time r, there is an execution of the node and all its predecessors which starts at time r–x. Therefore, the start-time of the latest execution of the node and its predecessors has not changed since the last simulation round. QED

Corollary

At the end of the simulation round which ends at time t, (i) if F([0,t]) does not contain an execution of the augmented graph C_i, then the *alarm* on C_i will have value t; (ii) if F([0,t]) contains an execution of the augmented graph C_i, then the *alarm* on C_i

will have as its value, say d, the time that has elapsed since the start-time of the latest execution of C_i, i.e., d is the largest integer such that F([t–d,t]) contains an execution of C_i.

Proof

If F([0,t]) contains an execution of the timing constraint graph C_i, then by lemma 3.3, at least one timer will be added to each one of the input edges of the *sink* in [0,t]. Hence if F([0,t]) does not contain an execution of the timing constraint graph C_i, then at least one of the input edge(s) of the *sink* will be empty in [0,t] so that step (7) of the simulation procedure is never performed in [0,t] and the *alarm* which is initialized to 0 is incremented after every simulation round and therefore has value t at time t.

If F([0,t]) contains an execution of the timing constraint graph C_i, there are two cases. In the first case, every input edge of the *sink* has at least one timer and step (7) of the simulation procedure is performed. At the end of step (7), at least one of the input edges of the *sink* must be empty. Let d be the value of the last timer removed from one of these empty edges. Since timers with larger values are removed first, all except for the last timer removed from an output edge of an immediate predecessor of the *sink* an edge which is empty after step (7) must have value $\geq$d. By the above lemma, there must be an execution of every immediate predecessor of the *sink* and its predecessors starting at time $\geq$t–d. Hence there is an execution of the timing constraint graph C_i in [t–d,t]. Since timers are removed in the same order they are added to an edge, the latest execution of the immediate predecessor of the *sink* whose output edge has been emptied by step (7) must have started at time t–d. Hence the latest execution of the timing constraint C_i cannot have started after t–d. Hence t–d is the latest start-time of an execution of the timing constraint C_i. Since the *alarm* is updated to d, the corollary holds. QED

Theorem 3.5

An execution trace has latency d_i with respect to the timing constraint C_i if and only if the value of its *alarm* never exceeds d_i at the end of every round of its simulation.

Proof

Consider any interval I which ends at time t and has length d_i, i.e., I = [t–d,t]. At the end of the simulation round which end at t, let the value of the *alarm* be equal to x. By the corollary of lemma 3.4, there must be an execution of the timing constraint C_i in [t–x,t]. If $x \leq d_i$, then there must be an execution of the timing constraint C_i in I. Hence if the *alarm* never exceeds d_i, the execution trace must have latency d_i with respect to the timing constraint C_i.

If the execution trace has latency d_i with respect to the timing constraint C_i, then there is an execution of C_i in any interval of length $\geq d_i$. Hence at any time $t \geq d_i$, there must be an execution of the timing constraint C_i in $[t-d_i,t]$. Let the start-time of the latest execution of the timing constraint C_i in this interval be at t–d where $d \leq d_i$. By the corollary of lemma 3.4, the *alarm* of the augmented graph C_i must have value equal to d at the end of the round which ends at time t. In the interval $[0,d_i]$, the maximum value of the *alarm* is d_i. Hence the value of the *alarm* at any time cannot exceed d_i. QED

If a feasible execution trace exists, then by theorem 3.5, the value of the *alarm* in the i^{th} augmented graph C_j is bounded from above by d_i. By the second corollary of lemma 3.2, the value of any timer on the augmented graph C_i must be smaller than that of the *alarm* and hence the value of the timers on any edge in C_i cannot exceed d_i–1. By the first corollary of lemma 3.2, all the timers on the same edge must have unique values. Hence there are at most d_i–1 timers on any edge. Let us define the

state of an edge in the i^{th} augmented graph C_i be the finite set of timers on it and define the *state* of C_i to be the value of its *alarm* together with the set of states of its edges. The *state* of a simulation at the end of a simulation round is the set of states of the augmented graphs.

Theorem 3.6

If an execution trace exists which meets the latency requirements of a set of n asynchronous timing constraints { C_i }, i = 1,...,n, then there must be a (finite) feasible static schedule for the set of asynchronous timing constraints.

Proof

If an execution trace exists, then there are only a finite number of states which the i^{th} augmented graph C_i can be in at the end of every simulation round. Hence the simulation of the execution trace can be in only a finite number of states. Thus after a finite number of operation symbols in the execution trace have been scanned, the simulation must reenter a previous state. Let $\delta\, x_1 \ldots x_k\, \gamma$ be the feasible execution trace so that the state of the simulation program before the symbol x_1 is scanned is the same as that immediately after the simulation program scans the symbol x_k. Let β be the string $x_1 \ldots x_k$. The simulation of the execution trace $\delta\, \beta^*$ produces a sequence of states. By theorem 3.5, the *alarm* of the augmented graph C_i cannot exceed d_i in any of these states, and Theorem 3.5 in turn implies that the execution trace $\delta\, \beta^*$ is a feasible one and hence β^* is a feasible execution trace. Thus β must be a feasible static schedule. QED

We can now give an upper bound on the length of a feasible static schedule which has latency d_i with respect to the i^{th} timing constraint C_i. Let E_i be the number of edges in the augmented graph C_i. The maximum number of states that an edge can be in is 2 ** d_i and the *alarm* can have value from 0 to d_i. Hence the number of

states the augmented graph C_i can be in is $E_i(d_i + 1)(2 ** d_i)$. The maximum number of states a simulation can be in is:

$$\Pi_i E_i(d_i + 1)(2 ** d_i).$$

If a communication graph has V nodes and E edges and if each node and edge of the communication graph appears only once in each augmented graph C_i, then e_i is bounded by E + V. If there are n timing constraints and d_{max} is the maximum of the deadlines, then the maximum number of states is:

$$(d_{max} + 1)^n (E + V)^n \ (2 ** nd_{max})$$

which is an upper bound on the length of a feasible static schedule where every operation need to be executed at most once in each timing constraint.

3.4.2 Computing Static Schedules for Asynchronous Timing Constraints

In the last section, we have obtained an upper bound on the length of a feasible static schedule. Thus the existence of a feasible static schedule can always be decided in finite time. When the computation load is heavy, however, the length of any feasible static schedule can indeed get very long. For example, given any integer n, consider the following latency scheduling problem:

The communication graph consists of $n+1$ unconnected nodes each with unit computation time, and there are $n+1$ asynchronous timing constraints. Except for the $n+1^{th}$ constraint, the i^{th} timing constraint consists of executing the i^{th} node with *deadline* $d_i = 2**i+1$. The $n+1^{th}$ constraint consists of executing the $n+1^{th}$ node with *deadline* $d_{n+1} = 2**n+1$.

It is easy to check that the only static feasible schedule must execute the i^{th} node exactly once every $2**i$ time units. The processor utilization factor is given by $1/2+1/4+ \ldots +1/(2**n)+1/(2**n) = 1$. The length of this schedule is $2**n$ which grows exponentially with the size of the problem (where integers are given as binary numbers).

In general, it is impractical to find a feasible schedule by generating candidate schedules in increasing length and testing for feasibility until all schedules of length $\leq$ the upper bound have been exhausted, especially if the timing constraints require heavy processor utilization. However, it may not be necessary to generate a complete schedule in order to decide whether a feasible static schedule exists (witness the example above). Unfortunately, the computation required for answering the decision problem alone is likely to be prohibitive in the worst case, as evidenced by the fact that two rather restricted versions of the latency scheduling decision problem are NP-hard.

We now state a known NP-complete problem by Galil and Megiddo which we shall reduce to another restricted version of the latency scheduling problem. First, we need

a definition.

Given a circular arrangement C of a set of symbols from some alphabet so that each symbol on the circle appears only once, we say that an ordered triple (a b c) is consistent with C iff these three symbols appear in that order when the circle is traversed in the clockwise direction starting at the symbol a. A set of ordered triples is consistent with a circular arrangement C iff every one of them is consistent with C.

The CYCLIC ORDERING problem is: Given a finite set of ordered triples of symbols from some alphabet, decide whether there exists a cyclic arrangement of symbols such that every symbol which appears in the set of ordered triples appears one and only once in the cyclic arrangement.

Theorem 3.7

The CYCLIC ORDERING problem is NP-complete (in the strong sense).

Proof

This problem is reducible from 3-satisfiability and can be found in [GAL & MEG 77].

Theorem 3.8

The cyclic ordering problem can be reduced to the latency scheduling problem restricted to instances where all operations have unit computation time and all the timing constraint graphs are chains of length 1 or 3.

Proof

Given a set of ordered triples, we construct the corresponding latency scheduling problem as follows. The communication graph has a node corresponding to each distinct symbol which appears in the set of ordered triples. Let the number of nodes be n. All nodes denote operations of unit computation time. An edge is added from the node

x to the node y and another edge from y to the node z in the communication graph iff there is an ordered triple (x y z). Each ordered triple (x y z) specifies an asynchronous timing constraint graph in the natural way, i.e., an instance of x precedes y which precedes z. Each of these asynchronous timing constraints has a deadline and minimum period equal to 2n–1. Also, an asynchronous timing constraint consisting of a single operation is created for each node in the communication graph. Each of these n single-operation timing constraints has a deadline and minimum period equal to n.

If a consistent cyclic ordering exists, it is easy to check that it can be used as a feasible static schedule. Conversely, assume that a feasible static schedule exists. The latency constraints imposed by the single-operation precedence graphs require that there must be exactly one instance of every operation in any interval of length n in the execution trace. Thus the execution trace must be generated by the periodic execution of some permutation of the n operations. Hence if a feasible static schedule exists, there must be one which is a permutation of the n operations. We claim that the circular placement defined by this permutation is a consistent cyclic ordering. To see this, take any ordered triple (x y z) and consider an interval of length 2n–1 starting immediately after an instance of x. The next instance of x starts exactly n–1 time units later. So in order to meet the specified latency of 2n–1, the operations x, y and z must occur in that order in the next n time units which is the length of the permutation. Thus they must occur in the correct order in the circular placement. QED

The above result has very harsh implications. First, it is noted that the NP-hardness of the above problem is independent of whether different operations are allowed to preempt each other. Most processors allow interrupts to be processed only after the current instruction cycle has been finished. In general, there is a smallest quantum of computation time between successive preemptions that a processor or the operating system is prepared to tolerate. This is taken as the unit computation time

above. Second, the length of a feasible static schedule for the above problem is a linear function of the problem size (since it is a permutation of the operation symbols). Hence the complexity of the latency scheduling problem remains intractable even if the problem is restricted to finding static schedules whose length is bounded by a linear function of the problem size.

The difficulty of the problem above is rooted in the fact that the precedence orderings of the timing constraint graphs are inconsistent with one another. Thus there is hope that the scheduling problem would be easier if the timing constraint graphs are consistent with one another. However, this is not the case if we do not allow different operations to preempt one another. We shall first describe the SET PARTITION problem which is known to be NP-complete [GAR & JOH 79].

The SET PARTITION problem is: Given a finite set of positive integers $\{ w_i \}$ where $\Sigma w_i = 2q$ for some integer q, the decision question is whether the set can be partitioned into two subsets such that the two sums of all the integers in each partition are equal (to q).

Theorem 3.9

If different operations are not allowed to preempt one another, then the set partition problem can be reduced to the latency scheduling problem restricted to instances where every timing constraint graph consists of a single operation and that all but one of the specified deadlines are the same.

Proof

Given a set of integers $\{ w_i \}$, let $q = 1/2 * \Sigma w_i$ and construct a latency scheduling problem as follows. The communication graph consists of $n+1$ unconnected nodes. For $i = 1$ to n, let the i^{th} node has computation time w_i. The $n+1^{th}$ node has a computation time of q. There are $n+1$ asynchronous timing constraints. For $i =$

1 to n, the i^{th} timing constraint consists of the i^{th} node alone and has a deadline and minimum period of $w_i + 4q-1$. The $n+1^{th}$ constraint consists of the $n+1^{th}$ node alone and has a deadline and minimum period of $3q-1$.

Suppose a partition $\{ S_1, S_2 \}$ exists such that $| S_1| = | S_2| = q$. Then we can list a feasible static schedule as shown in figure 3.8a. Notice that any two successive executions of the i^{th} operation, $i = 1$ to n spans an interval of length $w_i + 4q$ and that for the $n+1^{th}$ operation, the interval between two complete executions is 3q. Thus this static schedule has the required latencies by the problem.

To prove the converse, we first claim that for any execution trace which meets all the latency constraints, each instance of the $n+1^{th}$ operation must start exactly q time units after the completion of the previous instance. Any further delay causes the two instances to span an interval of length $> 3q$ and a latency violation will result. If any two successive instances of the $n+1^{th}$ operation spans an interval $< 3q$, then there must be three successive instances of the $n+1^{th}$ operation which spans an interval of length $d < 5q$. Let the first of these three instances starts at t_1, i.e., the third instance completes at $t_1 + d$. Let $g = w_{max} + 4q-1$ where w_{max} is the maximum of the set $\{ w_i, n \geq i \geq 1 \}$. We can assume that $w_{max} \leq q$ for otherwise the partition problem is trivial. This in turn implies $g < 5q$. Now there are two cases depending on whether $d \geq g$ or $d < g$.

If $d \geq g$, then in the interval $[t_1, t_1 + d]$, at least one instance of each of the i^{th} operation, ($i = 1$ to n) must occur in addition to to the three instances of the $n+1^{th}$ operator. The total computation required in this interval is therefore at least 5q. Since $d < 5q$, some latency violation must result. See figure 3.8b for an illustration.

If $d < g$, then we can apply the same argument to the interval $[t_1, t_1 + g]$ and conclude that some latency violation must occur in that interval. See figure 3.8c.

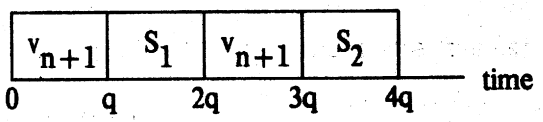


Figure 3.8a A feasible schedule

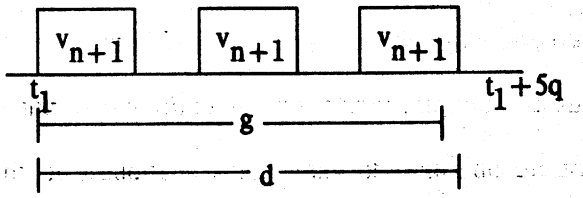


Figure 3.8b

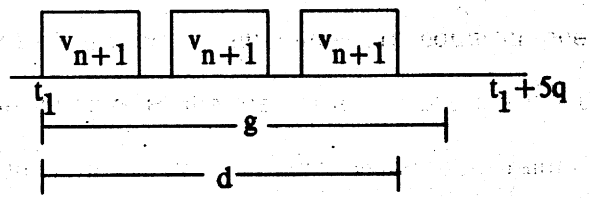


Figure 3.8c

Figure 3.8
Reduction of SET PARTITION to latency scheduling problem

Thus any feasible static schedule must generate an execution trace which has instances of the $n+1^{th}$ operation spaced exactly q time units apart. Now take any interval spanning three complete instances of the $n+1^{th}$ operation. This interval must have length 5q, and so must contain at least one instance of every operation and the processor cannot idle in this interval. This is possible only if the set $\{ w_i, i = 1 \text{ to } n \}$ can be partitioned into two subsets of equal weight. Thus the existence of a feasible static schedule implies a feasible partition. QED

Other well known NP-complete problems, e.g., 3-PARTITION [GAR & JOH 79] can also be transformed in a similar fashion to this restricted version of the latency scheduling problem which is in general not amenable to attack even by pseudo-polynomial time algorithms. However, it may have been noticed that in both theorem 3.8 and 3.9 above, the equivalent scheduling problems derived from the NP-complete prob-

lems all require full processor utilization in order to meet the deadlines. Intuitively, one may expect the decision problem to be easier if the processor utilization required to meet the timing constraints is low. The following theorem gives sufficient conditions for a feasible static schedule to exist for a set of asynchronous timing constraints when the demand for processor time is sufficiently low.

Theorem 3.10

Let (G,T) be an instance of the graph-based model and suppose all the timing constraints in T are asynchronous. Then a feasible static schedule always exists if the following three conditions hold.

(1) $\Sigma\, w_i/d_i \leq 1/2$ where w_i, d_i are respectively the computation time and deadline specification of the i^{th} timing constraint.

(2) For the i^{th} timing constraint, $[d_i/2] \geq w_i$ where [x] is the integer part of x.

(3) Every node in the communication graph can be pipelined into a chain of operations each of which takes one unit of computation time. The function performed by the node must be expressible as a composition of a number of sub-functions with no shared variables except for the input/output variables at the interface between successive stages.

Proof

Translate each asynchronous timing constraint C_i into an "equivalent" periodic process as follows. The body of the periodic process consists of a straight-line program which is any topological sort of the operations in the timing constraint graph C_i. The *deadline* and *period* attributes of the periodic process are both set to $[d_i/2]$, the integer part of $d_i/2$. Condition (2) above guarantees that the computation time w_i of the process is smaller than or equal to the corresponding deadline, and condition (3) allows the processes to preempt one another at unit time intervals, i.e., the processes

are independent. By the corollary of theorem 2.1, a sufficient condition for scheduling these periodic processes is that $\Sigma\, w_i/[d_i/2]) \leq 1$. Thus condition (1) guarantees that a feasible execution trace must exist. Since the period attributes are chosen so that the maximum interval spanned by two consecutive executions of a timing constraint never exceeds d_i, the execution trace is a feasible one and by theorem 3.6, a feasible static schedule must exist. QED

Remark

In practice, it is not necessary to pipeline a function element with computation time c into c unit-time stages. The length of each stage can be chosen to be the basic quantum q of processor time used by the operating system kernel as long as all deadlines are integral multiples of q.

The first two conditions of theorem 3.10 indicate that if preemption is allowed, the latency scheduling problem is essentially trivial as long as maximum processor utilization does not exceed 50% and that all of the timing constraints have sufficient slack. (Specifically, the slack should be at least as large as the computation time). On the other hand, Theorem 3.8 and 3.9 indicate that this problem is likely to be computationally intractable if the processor must be kept very busy in order to meet all the timing constraints. Between these two extremes, there is a wide system load gap where efficient algorithms may exist. A general strategy for attacking the problem is to replace every specified deadline by a (hopefully not too much) shorter one such that there is a convenient relationship among all the modified deadlines that makes the decision problem easier to solve. In particular, it may be possible to derive tighter upper bounds on the system load below which there is an efficient algorithm for solving instances of the decision problem which have the modified deadlines only. Then by considering the maximum deviation between any set of deadlines with the corresponding

modified deadlines, we can hopefully further close the system load gap. However, a tighter bound has eluded us so far.

In the following, we shall describe a family of heuristic algorithms which might prove useful for further closing this gap when all the timing constraint graphs are consistent with one another and every node in the communication graph can be pipelined into a chain of unit length operations. This family of heuristic algorithms is inspired by the success of theorem 3.10 and is parameterized by a set $P = \{ x_i \}$ of compatible integers. (Two integers are *compatible* if either they are equal or one divides the other.) Given a set of compatible integers P, the idea is to replace the i^{th} asynchronous timing constraint (C_i,p_i,d_i) by an "equivalent" periodic timing constraint (C'_i,p'_i,d'_i) with the same graphs, i.e., $C'_i = C_i$. Let the computation time of the i^{th} timing constraint (the sum of the computation times of the operations in the graph C_i) be w_i. P is a heuristically chosen set of compatible integers. The choice of the period p'_i and the deadline d'_i of the periodic constraint replacing the i^{th} asynchronous timing constraint are determined heuristically subject to the conditions:

(1) $d_i \geq d'_i \geq w_i$

(2) $p'_i \in P$

(3) $p'_i \leq d_i - d'_i + 1$.

These two conditions together guarantee that there is an execution of the timing constraint C_i in every interval of length $\geq d_i$ if the equivalent periodic timing constraints can be successfully scheduled. Specifically, for each asynchronous constraint with deadline d_i, we assign a periodic timing constraint with the same graph $C'_i = C_i$. The period p'_i of the replacement periodic timing constraint is set to the largest member of P such that $2p'_i - 1 \leq d_i$. The deadline d'_i is set to the largest integer such that $d'_i + p'_i - 1 \leq d_i$. (This is a *greedy* algorithm.) The following algorithm attempts to construct a feasible static schedule of length $p_{max} = \text{MAXIMUM} \{ p'_i \}$ such that there is

an execution of the timing constraint C'_i in the interval $[kp'_i, (k+1)p'_i]$ for $k = 0, \ldots, p_{max}/p'_i$. Let p_{min} = MINIMUM { p'_i }.

(1) Sort the periodic timing constraints in increasing order of their deadlines and incorporate them one by one into a partial schedule for the interval $[0,p_{max}]$. A feasible static schedule is obtained when all the timing constraints have been successfully incorporated.

(2) Let the shortest deadline be d. At time $t = kp_{min}$ for each $k \leq p_{max}/p_{min}$, let C be the union of all the timing constraint graphs which have deadline d and whose periods divide t, i.e., an execution of each of these timing constraints must occur in $[t,t+d]$. Let w_C be the computation time of C and execute C in the interval $[t+d-w_C, t+d]$ if $w_C \leq d$. Declare failure if $w_C > d$.

(3) Suppose all the timing constraints with deadline $\leq d$ have been scheduled. The timing constraints with the next larger deadline d' will be scheduled as follows. At time $t = kp_{min}$ for each $k \leq p_{max}/p_{min}$, let C be the union of all the timing constraint graphs which have deadline d' and whose periods divide t, i.e., an execution of each of these timing constraints must occur in the interval $I = [t,t+d']$. Let w_C be the computation time of C. If I is idle and has length $\geq w_C$, then execute C in $[t+d'-w_C, t+d']$. In general, I will consist of a sequence of busy intervals I_j separated by idle intervals. Let the first busy interval I_1 start at time s_1 and end at time f_1. Some of the operations in C may have been executed in I_1 already. Hence if the predecessors of these operations are executed before I_1, then they do not have to be executed again. This is possible if all these predecessors can be executed in $[t,s_1]$. If not, some of the operations already executed in I_1 will have to be repeated after f_1. The next step attempts to minimize the number of operations in I_1 that have to be repeated after time = f_1.

(4) Let $V_1 = \{ v_1, \ldots v_m \}$ be the set of operations in I_1 which are also in C. For each

v_j in V_1, assign a *weight* w_j which is the sum of the computation time of all its predecessors in C which are not in V_1. Assign to v_j a *value* r_j according to one of two cases: (i) If all the predecessors of v_j in C already precede v_j in I_1, then v_j has *value* = 0; (ii) If at least one of the predecessors of v_j in C does not precede v_j, then the *value* of v_j is equal to the sum of the computation time of v_j and all its successors in V_1 which have the same predecessors in C. Thus, if all the predecessors of v_j in C are executed before I_1, then we have a potential saving of r_j. We can select a subset of V_1 to maximize Σr_j by executing all their predecessors in C in $[t,s_1]$ subject to the condition $\Sigma w_j \leq s_1 - t$. This is a 0-1 KNAPSACK problem which can be solved by dynamic programming in time $O(s_1 - t)$.

(5) Repeat step (4) to schedule the remainder of C in each of the idle intervals in I. If some operations of C with computation time w have to be executed in the last idle interval in I, schedule them so that they finish at the end of I, i.e., in the interval $[t + d' - w, t + d']$.

The success of the above heuristic algorithm depends on the choice of the set P of compatible integers. Intuitively, we want to perform an execution of an asynchronous timing constraint as infrequently as possible. As an initial guess, we can choose P to be the set $\{ p^n : n \geq 1 \}$ for some integer p such that MAXIMUM $\{ d_i - 2p^n$: n is the largest integer for which $d_i - 2p^n \geq 0 \}$ is minimized.

3.5 Computing Static Schedules for the Graph-based Computation Model

In practice, both periodic and asynchronous timing constraints may be specified for a hard real-time system. Since the problem of computing a static schedule for asynchronous timing constraints alone is NP-hard, the scheduling problem is computationally intractable in general. In fact, the problem of computing a static schedule for periodic timing constraints alone is also NP-hard as can be shown by reducing the well

known KNAPSACK problem to a restricted version of our scheduling problem with periodic timing constraints.†

An instance of the KNAPSACK problem is given by two integers L, K and a finite set $S = \{ s_i \}$ of N elements each of which has an integer weight w_i and an integer value v_i. The decision problem is to decide whether there is a subset S' of S such that $\Sigma v_i \geq K$ and $w_i \leq L$ where the sums are over the elements of S'.

Theorem 3.11

The KNAPSACK problem can be reduced to the following restricted scheduling problem: The communication graph consists of a set of chains of two nodes each. Every timing constraint is periodic with the same period, say p, and consists of either a single operation or a chain of two operations. All timing constraints with a single operation have the same deadline $d < p$, while all the timing constraints with a chain of two operations have the same deadline = p.

Proof

Given an instance of the KNAPSACK problem (using the above symbols), the corresponding communication graph contains a chain of two nodes for each element s_i in S such that the first node has computation time w_i and the second node has computation time v_i. let V be the sum of the v_is and W be the sum of the w_is. There are N periodic timing constraints each of which consists of one of these chains and has period and deadline = W+2V–K. The second nodes in the chains (i.e., the nodes with computation time v_i) also form the graphs of N periodic timing constraints each of

† The KNAPSACK problem can be solved in pseudo-polynomial time by dynamic programming techniques, so we do not rule out acceptably fast solutions to the scheduling problem for periodic timing constraints alone in practice.

which has deadline = L + V and period = W + 2V–K.

If there is a subset S' of S with weight $\leq$L and value $\geq$K, then we can execute the first nodes in the corresponding chains in [0,L]. All the nodes with computation time v_i can be executed in [L,L + V]. The set of the first operations in the chains corresponding to S–S' has weight W–L and can be executed in [L + V,W + V]. The successors of these operations have weight $\leq$V–K and can thus be executed in [W + V, W + 2V–K], and a feasible static schedule has been constructed covering the interval [0,W + 2V–K]. The form of the feasible schedule is shown in figure 3.9.

Conversely, a feasible schedule must contain an execution of each of the timing constraints in the first period, i.e., in [0,W + 2V–K]. Since the total computation time of all the timing constraints is W + 2V, there must be a saving of K time units which can only be obtained by executing the first nodes of a subset of the chains in [0,L] such that their successors have computation time $\geq$K. QED

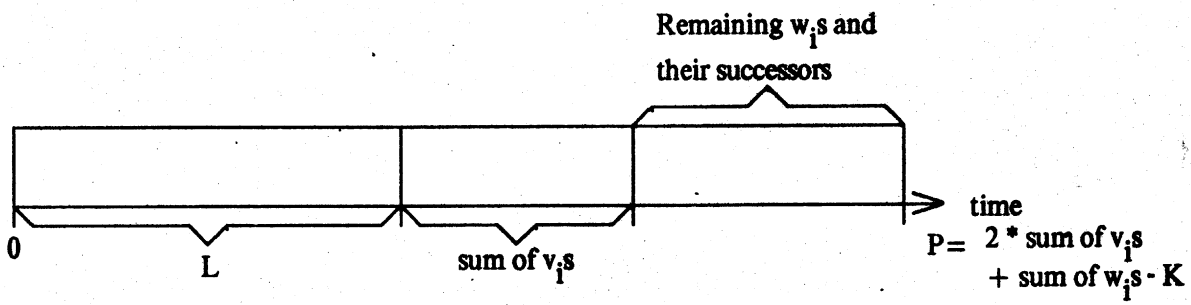


Figure 3.9
The form of any feasible schedule in theorem 3.11

The computational complexity of the scheduling problems for the graph-based model should not be interpreted as a deterrent for exploiting static schedules. On the contrary, the NP-hardness results may be taken as evidence that as much of the scheduling work as possible should be done off-line. In particular, much efficiency may

be gained by using the latency scheduling technique to reduce the redundant computation of the asynchronous timing constraints which is already required by the periodic timing constraints.

Given a set of timing constraints in the graph-based model, we can in general proceed as follows. (The heuristics described below are all *greedy*.)

(1) Convert heuristically all the asynchronous timing constraints into "equivalent" periodic timing constraints with periods compatible with those of the periodic timing constraints. A greedy algorithm is to select the smallest period among the periodic timing constraints which is closest to but smaller than the deadline of the asynchronous timing constraint.

(2) Merge all the periodic timing constraints with the same deadline and period into a single periodic constraint and eliminate all the redundant computation by taking the union of their graphs. If two periodic timing constraints C_1 and C_2 have the same period but different deadlines with $d_1 < d_2$, then a greedy algorithm is to eliminate the operations in C_2 that are also in C_1. Specifically, if the timing constraints C_1 and C_2 are consistent with each other and they both contain the operation v_i, then eliminate v_i from C_2 and make all the predecessors of v_i in C_2 also the predecessors of the corresponding instance of v_i in C_1. Likewise, make all the successors of the operation v_i in the timing constraint C_2 to be the successors of the corresponding instance of v_i in the timing constraint C_1. This technique may also be applied to timing constraints with different periods and deadlines and is especially useful if the periods are compatible. Specifically, if the timing constraints C_1 and C_2 have compatible periods with $p_1 \leq d_2 \leq p_2$, then eliminate all the operations from C_2 that are also in C_1 and make the predecessors/successors of these operations in C_2 to be the predecessors/successors of the corresponding instances of the operations in C_1 that must be executed in the period right before the dead-

line d_2. (In the parlance of the process-based model, we are simply doing a decomposition by minimizing interprocess intercommunication since the goal of our greedy algorithm is to merge all the required computation into one single timing constraint.)

(3) Apply the scheduling algorithm for the kernelized monitor model in the last chapter to schedule the resulting timing constraints by identifying each operation as a chain of scheduling blocks and the precedence constraints as being imposed by *rendez-vous* commands executed at the end of of the operations.

The interested reader may want to verify that the above heuristics will yield the efficient but delightfully simple schedule in section 3.17. Finally, it should be noted that it is not necessary to store the complete static schedule in the database for the run-time scheduler. For example, the kernelized monitor scheduler of the previous chapter may be used as the run-time scheduler in which case only the relevant dynamic deadlines and forbidden regions for the interval spanned by the static schedule need to be stored on-line, the penalty being that more scheduling overhead may be incurred at run time by the more complicated run-time scheduler. In general, there is a tradeoff between memory space and scheduling overhead that is available to the off-line scheduler.

Chapter 4

Design Issues of Distributed Systems

4.1 Resource Sharing in a Distributed System

In the previous chapters, we have motivated and developed a graph-based computation model for the hard real-time environment and investigated the related scheduling problems on a general purpose computer. Many real-time applications, however, share a family of signal processing algorithms which are best implemented by specialized computing devices, e.g., systolic arrays. With the rapid advance in VLSI technology, specialized hardware is becoming increasingly available and most future real-time systems are likely to be implemented by an interconnection of general purpose computers and special devices. Before we can make any headway in automating the production and maintenance of the software for these distributed systems, we must first resolve two difficulties. First, the delay introduced by the communication subsystem must be taken into account if stringent timing constraints are to be met. Second, the performance characteristics of a wide variety of special computing devices must be represented in such a way that they can be exploited by software automation tools to meet stringent timing constraints. Both difficulties can be greatly eased if we can adopt a uniform model for different system resources that is precise enough for the requirements of the hard real-time environment.

In general, we shall attempt to model a system resource as a (composite) node in the communication graph together with a scheduling algorithm for determining whether the resource is sufficient for meeting a given set of timing constraints. It should be noted that our modelling approach will not necessarily result in the most efficient use of resources. However, it meets the important objective of guaranteeing that stringent timing constraints are being met and may also improve system reliability by providing an

error margin for synchronizing the operation of physically distributed subsystems.

In the following, we shall first give an example to exaggerate the potential hazards of a distributed system where processors communicate with one another by transmitting information strictly in pre-computed time slots. In contrast, we shall give a formulation of the processor allocation problem for the hard real-time environment that is conducive to more robust implementation. A broadcast data bus will be used to illustrate our modelling approach and an efficient hardware implementation of a communication scheduler for the broadcast data bus will be described. The general processor allocation problem is, however, computationally intractable even if efficient algorithms are available for scheduling the computation allocated to each processor.

4.2 The Processor Allocation Problem in the Hard Real-Time Environment

Given a set of timing constraints expressed as an instance of our graph-based computation model and an interconnection of computing devices, the processor allocation problem is to partition the required computation and allocate it to the computing devices so that all the timing constraints can be met. Before this problem can be formulated properly, the delay introduced by the communication subsystem must first be characterized with respect to the generated traffic so that the designer can verify that no deadline is being missed because of communication delays. Traditionally, a queueing model is usually used to analyze the expected values of point-to-point delays. The constraints resulting from these delays are expressed as algebraic inequalities which must be satisfied by a candidate partition of the required computation. However, this type of stochastic analysis is inadequate for the hard real-time environment since the traffic generated by a partition of the required computation may be very bursty. Nevertheless, the majority of the published formulations of the processor allocation problem for real-time systems, e.g., [CHU et al 80], [MA et al 82] have chosen to either

ignore communication delay altogether or replace the timing constraints by load-balancing constraints. Their analysis is more pertinent to the time-average behavior of the system and is applicable only to soft real-time systems.

On the other extreme, a designer may choose to do a detailed control flow analysis of the programs running on every processor and identify all the time instants at which the processor will send or receive data. A detailed communication schedule can then be worked out according to which every processor will be scheduled to send or receive data at certain predetermined time slots so that all the timing constraints can be met with certainty.

In general, the communication subsystem can be used more effectively for meeting timing constraints if it is possible to monitor and control the state of the communication network at every instant. Unlike a processor, however, the relative timing of the events in a communication system is generally more susceptible to physical perturbations and may not always be controllable. Hence, a real-time system which has little tolerance for deviations in the timing behavior of a communication network is likely to be less robust. As an illustration of the potential hazard of such systems, consider the following scenerio.†

4.2.1 Another Time Bomb

An engine is controlled by a distributed system of processors which communicate with one another via a broadcast data bus to which all the processors are connected. Processor A has a direct interface to an auxiliary switch and processor B is interfaced to an ignition unit which fires up the motor. There are certain safety rules that must be observed at all times. (1) The auxiliary switch must be turned on before the motor can

† This example is meant to be reminiscent of the widely publicized synchronization error which preceded the first successful flight of the Space Shuttle.

be fired up. (2) The motor must be fired up within 100 milliseconds after turning on the auxiliary switch. Otherwise, the auxiliary switch must be turned off immediately. (3) The auxiliary switch must remain on as long as the motor is firing.

The interface of the motor ignition unit provides the function FIRE_UP() to fire up the motor. From time to time, however, the motor may refuse to fire up immediately in which case the FIRE_UP function will automatically abort after trying for 50 milliseconds and return FALSE. The procedures TURN_ON_SWITCH and TURN_OFF_SWITCH turn on and off the auxiliary switch respectively. Our Ada programmer has written the following program to start the engine.

```
/* COMMENT
   There are two tasks: Task AUXILIARY_SWITCH controls the auxiliary switch
   and task MOTOR_IGNITION controls the fire-up unit. AUXILIARY_SWITCH will
   turn off the switch if the motor has not been fired within 100 milliseconds
   after the switch has been turned on. This is accomplished by a timed
   entry call with the second alternative of the select statement
   being executed if the first alternative has not made any progress after
   the specified delay expires.
*/

task body AUXILIARY_SWITCH is
   RESULT: boolean;
begin
   TURN_ON_SWITCH();
   select
      MOTOR_IGNITION.START_FIRING(RESULT);
      if RESULT = FALSE then
         TURN_OFF_SWITCH();
      end if;
   or
      delay 100;
      TURN_OFF_SWITCH();
   end select;
end AUXILIARY_SWITCH;


task MOTOR_IGNITION is
   entry START_FIRING(RESULT:out boolean);
end

task body MOTOR_IGNITION is
```

```
begin
  accept START_FIRING(RESULT:out boolean) do
    RESULT := FIRE_UP();
  end START_FIRING;
end MOTOR_IGNITION;
```

There are two tasks: AUXILIARY_SWITCH and MOTOR_IGNITION which are to be run on processors A and B respectively. The task AUXILIARY_SWITCH turns on the auxiliary switch and **rendezvous** with the MOTOR_IGNITION task which then starts firing up the motor. If the engine is successfully started, the MOTOR_IGNITION task will return TRUE to confirm a successful firing. Otherwise, the FIRE_UP function will abort after 50 milliseconds and the MOTOR_IGNITION task will return FALSE to the AUXILIARY_SWITCH task which will then turn off the switch. Meanwhile, the AUXILIARY_SWITCH task executes a timed entry call and selects one of two alternative courses of action. If the **rendezvous** with the MOTOR_IGNITION task is unsuccessful after 100 milliseconds, the AUXILIARY_SWITCH task will turn the switch off. Otherwise, it will wait for the result of the firing from the MOTOR_IGNITION task.

This program will work if the MOTOR_IGNITION task can always confirm a successful firing by completing a **rendezvous** with the AUXILIARY_SWITCH task within 100 milliseconds after it has received the signal to start firing up. Now suppose that the broadcast data bus is time multiplexed so that each processor can broadcast data to other processors in fixed time slots accordingly to a predetermined schedule. Ordinarily, processor B will be able to access the bus and transmit a message to processor A before the AUXILIARY_SWITCH task on processor A times out. Once in a long while, however, the timing circuit in a processor may get out of synchronization with the arbitration circuit of the bus† and in particular, processor B may miss its turn to access the bus at the time it is supposed to send a message to processor A to confirm that the motor is being fired up. If processor B does not get another turn to transmit until

† For example, this may be caused by an unusually long metastable state in a syn-

the 100 milliseconds runs out, then the AUXILIARY_SWITCH task will assume that the motor has not been fired and turn off the switch, and a safety rule will be violated if the motor has actually been started.

The above example brings to focus a dilemma. On the one hand, we have rejected the use of time-average parameters for characterizing the communication delay constraints on the processor allocation problem because they are not precise enough for meeting stringent timing constraints. On the other hand, a completely deterministic approach which requires physically distributed subsystems to operate in lockstep is likely to be too inflexible for the system to be robust even if it is not too tedious to analyze. From a practical point of view, the last approach may not even be feasible since the state of a communication system may not always be predictable in the real world. In the next section, we shall strike a compromise by giving a formulation of the processor allocation problem which is precise enough for meeting stringent timing constraints and yet allows some degree of freedom in implementation for improved robustness.

4.2.2 A Robust Formulation of the Processor Allocation Problem

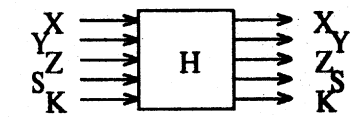
We shall treat a communication network connecting a set of m processors { P_1, ..., P_m } as a node, H which has m input edges and m output edges. The node H may be thought of as implementing a *transmission operation*, H_{ij} (an identity mapping with a finite delay) from the i^{th} input edge to the j^{th} output edge if and only if there is a path from the i^{th} processor to the j^{th} processor. Two or more transmission operations may be executed concurrently, subject to the internal pipelining and other constraints on the communication network denoted by the node H.

Given the computation requirements of a hard real-time application in terms of the

---

chronization circuit, an inherent hazard of any system which needs to synchronize two or more asynchronous subsystems.

graph-based model (G,T), partition the communication graph G into m disjoint subgraphs. Let e be an edge in the communication graph G which connects a node u in one subgraph to a node v in another subgraph in the partition. Replace every instance of e in a timing constraint by an instance of the transmission operatoin H_{ij} and the two edges from u to H_{ij} and from H_{ij} to v. The processor allocation problem is then to find a partition of the communication graph G such that the set of timing constraints augmented by the appropriate transmission operations can be successfully scheduled. Figure 4.1 illustrates the augmented timing constraints resulting from a straightforward partition of the communication graph of our favorite design example where each of the five nodes is assigned to a separate processor.



Communication Network Node

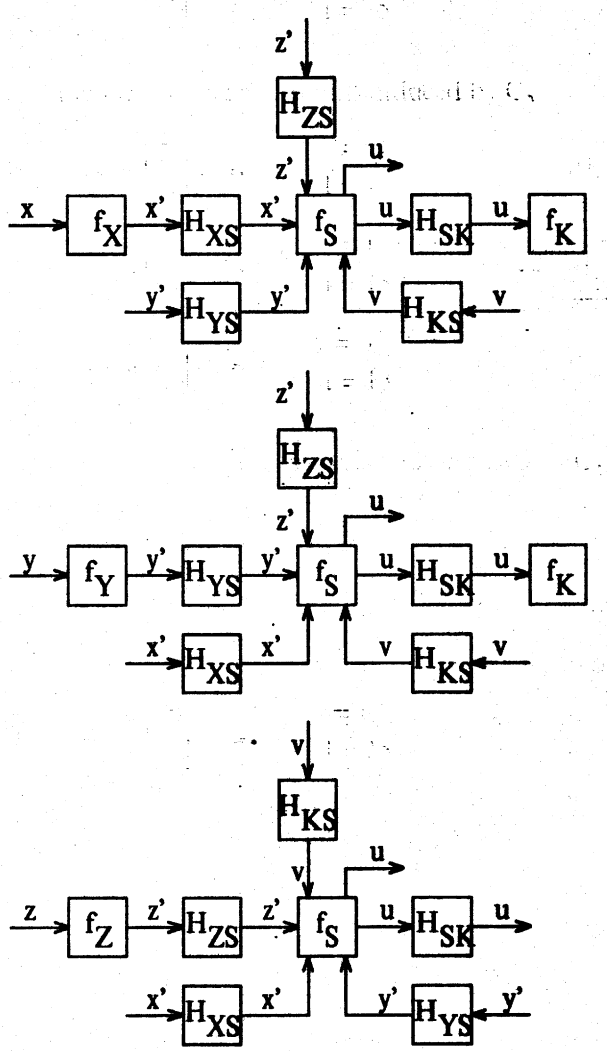


Figure 4.1

Timing constraints augmented by transmission operations

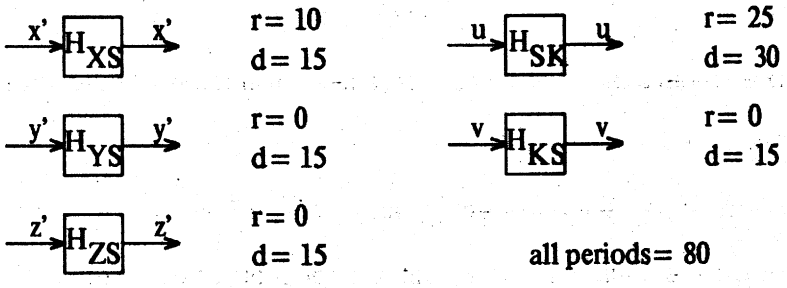
A completely deterministic solution to the processor allocation problem is given by a feasible static schedule for each of the m processors together with a schedule for

executing the required transmission operations in appropriate time slots. As the example in the previous section shows, the success of this approach may depend heavily on the assumption that the processors and the communication network can be kept in tight synchronization. To alleviate the impact of this assumption, the processor allocation problem can be constrained to explicitly provide some slack for the execution of each transmission operation. This can be done by specifying a *release time* and *deadline* for each instance of a transmission operation in all the timing constraints so that the communication network can schedule the required transmission operations dynamically as long as every instance of a transmission operation is executed after its specified *release time* and before its *deadline*. Specifically, let the transmission operation H connect a node u to another node v in a timing constraint graph C. Assign a *release time*, r and a *deadline*, d to H so that whenever C is activated at time t, then the operation u and all its predecessors must be executed in the interval [t,t+r], H must be executed in the interval [t+r,t+d], and v and its successors must be executed in the interval $[t+d,t+d_k]$. Given m processors, the processor allocation problem is now to partition the communication graph into m subgraphs and to find an appropriate set of *release-time* and *deadline* attributes for the transmission operations such that the set of timing constraints augmented by the transmission operations can be scheduled. (This formulation imposes individual release times and deadlines on some of the operations in a timing constraint graph. However, the complexity of the single processor scheduling problem is not affected by these additional parameters and in particular, the kernelized monitor scheduler can again be used simply by initializing the request-times of the scheduling blocks to the specified release-times instead of at the beginning of a period.)

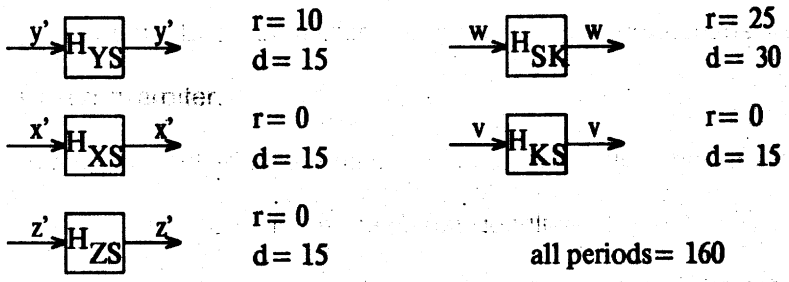
An important benefit of the above approach is that the traffic load generated by a partition of the computation can also be expressed as a set of periodic and/or asyn-

chronous timing constraints in terms of our graph-based model. If an instance of a transmission operation H with release time r and deadline d occurs in a timing constraint C which has period p, then the scheduler of the communication network must satisfy the corresponding timing constraint which consists of an instance of H and has release time, deadline and period attributes equal to r, d and p respectively. To be precise, a distinction should be made between two instances of a transmission operation if the two transmissions involve different variables. If the outputs of two or more operations need to be transmitted from processor i to processor j in the same time interval; then as many instances of the transmission operation H_{ij} must be scheduled in that interval. Figure 4.2 illustrates the timing constraints imposed on the communication network by the partition shown in figure 4.1 where each of the five nodes of the design example is assigned to a separate processor. The release times and deadlines on individual transmission operations are chosen to allow at least 5 milliseconds for each transmission operation to complete.

Periodic timing constraints induced by C_1



Periodic timing constraints induced by C_2



Asynchronous timing constraints induced by C_3

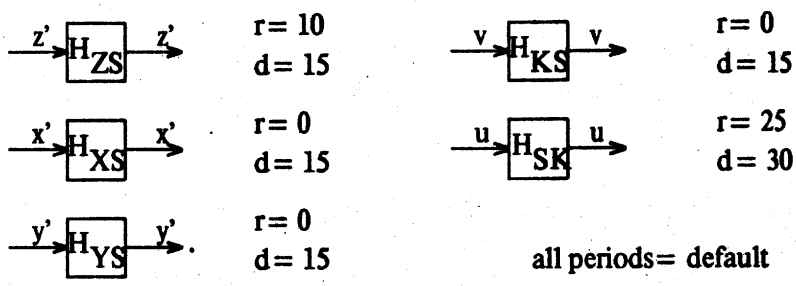


Figure 4.2
Timing constraints on transmission operations
induced by allocating a function to a processor
(Legend: r - release time; d - deadline)

Instead of using simple parameters such as throughput and delay, we can now characterize more precisely the capacity of a communication network for the hard real-time environment by the range of timing constraints that it can accomodate. In the next section, we shall illustrate our modelling approach by investigating the use of a broadcast data bus as an interconnection mechanism for multiple processors.

4.3 Scheduling Mechanism for The Broadcast Data Bus

Inasmuch as the broadcast data bus is a very common interconnection mechanism in computer systems, it is important to investigate in some detail its use in the hard real-time environment. Electrically, a bus may be viewed as an OR-gate which has an input from each processor connected to the bus and the output is monitored by all the processors. A transmission operation from the i^{th} processor to the j^{th} processor is accomplished by designating the i^{th} processor as the sole transmitter on the bus for an interval of time during which the i^{th} processor broadcasts data on the bus and names the j^{th} processor as the recipient. Obviously, only one transmission operation can be executed at any one time. Given a set of timing constraints, i.e., a set of transmission operations with the associated release time, deadline and period attributes, our problem is to come up with a scheduling mechanism and decide if all the timing constraints can be met.

Until recently, data busses have been designed to support data transfer between a single processor, the main memory, and various peripheral equipments. Access to the bus is usually decided by a fixed set of priorities preassigned to the devices connected to the bus. While this arrangement has been adequate when there is only one processor, it is not suitable for a multiprocessor system in the hard real-time environment where bus access should be decided by the relative urgency of the computation being performed by the processors at the moment. In particular, a mechanism is required to enforce a bus access policy that is optimal with respect to the timing constraints resulting from a partition of the real-time computation. Such a mechanism is described below.

4.3.1 The Earliest Deadline Algorithm for Bus Scheduling

The basic idea is to permit the transmission operation with the nearest deadline to

be executed first. To achieve this, the current bus access priority of the i^{th} processor is set inversely proportional to the time remaining before the nearest transmission operation, H_{ij} is due. The processor with the highest priority will then be granted access to the bus for the next time slot. This is of course the *earliest deadline* algorithm for single processor scheduling which has been shown to be optimal. To verify that a set of timing constraints can be met, we can simply run a simulation of the bus requests for a time interval at least as long as the LCM of the periods of the timing constraints. (All the asynchronous timing constraints are assumed to have been replaced by "equivalent" periodic constraints when they are scheduled on individual processors.)

The adoption of a dynamic scheduling algorithm such as the *earliest deadline* algorithm renders a multiprocessor system more robust. For example, if a processor is slightly behind and misses a turn to contend for the bus, a less urgent transmission operation may be executed instead, but the processor with the highest priority will be able to transmit in the next time slot and thus the more urgent transmission operation will not be delayed for an unduly long time.

To enforce the sharing policy, there must be a mechanism which permits only the processor with the highest priority to gain access to the bus. This can be achieved by a central arbiter which monitors the current priorities of all the contending processors. Since the number of priorities needed to encode the values of deadlines at run time is likely to be high, either the priorities must be transmitted to the arbiter serially or else the number of bus control lines required may be unacceptably high. Fortunately, only a reasonable amount of hardware is necessary to implement the *earliest deadline* algorithm as shown below.

4.3.2 The Distributed Arbitration Hardware

This bus arbitration mechanism seems to have been discovered and rediscovered by a number of researchers (including the author) and a variation of it has been proposed as a standard for the IEEE S-100 bus. However, none of the previous proposals has suggested the use of dynamic priorities for the hard real-time environment. The arbitration mechanism has the following nice properties:

(1) Control is distributed. Bus arbitration is achieved by concensus and does not depend on a central arbiter.

(2) The time required to resolve bus conflicts is bounded. Furthermore, the time bound can be a logarithmic function of the maximum deadline.

(3) It is not necessary to have a global clock. Local clocks are sufficient provided that they can maintain a certain degree of accuracy within a short time period; long term drift is automatically corrected by the feedback mechanism in the scheme.

For ease of explanation, we shall assume that all the processors are OR-wired to a number of control lines that are run parallel to the data bus. The number of control lines needed is the smallest integer equal to or bigger than $\log_2 (d_{max}-1)$ where d_{max} is the maximum value of the deadlines, although it will be seen that only one control line is sufficient. For the moment, let us assume that there are n control lines which are all accessible to some control logic on each processor board whose function will become apparent. The control logic can assert a logical "1" on a control line. If none of the processors asserts a logical "1" , the control line is in logical state "0". In practice, this can be implemented by means of open collector gating or tri-state logic. Each priority is represented by its binary representation $p_n p_{n-1} \ldots p_1$ where each p_i is either a 1 or a 0 and p_n is the most significant bit. (We can use the one's complement of the nearest deadline as the priority.) In general, bus conflict resolution takes n

"clock cycles". For ease of explanation, we shall assume that a global clock is available. This restriction will be removed by local synchronization mechanisms to be described later. At the beginning of the first clock cycle, all the contending processors must assert their own priorities on the n control lines. If a processor does not assert its priority on the control lines in the first clock cycle, then it must not participate in the rest of the conflict resolution process. At the end of each clock cycle, every contending processor makes its own decision to drop out of the contention process by comparing its own priority with the state of the control lines: $c_n c_{n-1} \ldots c_1$. A contending processor withdraws if and only if it can be inferred that its priority is lower than that of at least one of the contending processors. A processor signals its withdrawal by ceasing to assert its priority on the control lines. Specifically, every contending processor executes the following piece of program in synchronization with all other contending processors.

```
/*COMMENT
  Withdraw if the next significant priority bit is 0 and the corresponding
  control line is 1;
*/

      for i:= n step -1 until 1 do
      begin
            if ( p(i)=0 and c(i)=1 ) then withdraw
            else wait for next "clock cycle"
      end ;
```

As an example, consider the case of three control lines and five processors are contending (The priorities are in ascending order: 001 010 011 100 101). After the first clock cycle, 001 010 and 011 withdraw. After the second clock cycle, both 100 and 101 remain in contention. At the end of the third clock cycle, 100 withdraws and 101 emerges as the winner. The successive states of the control lines are shown in Fig. 4.3

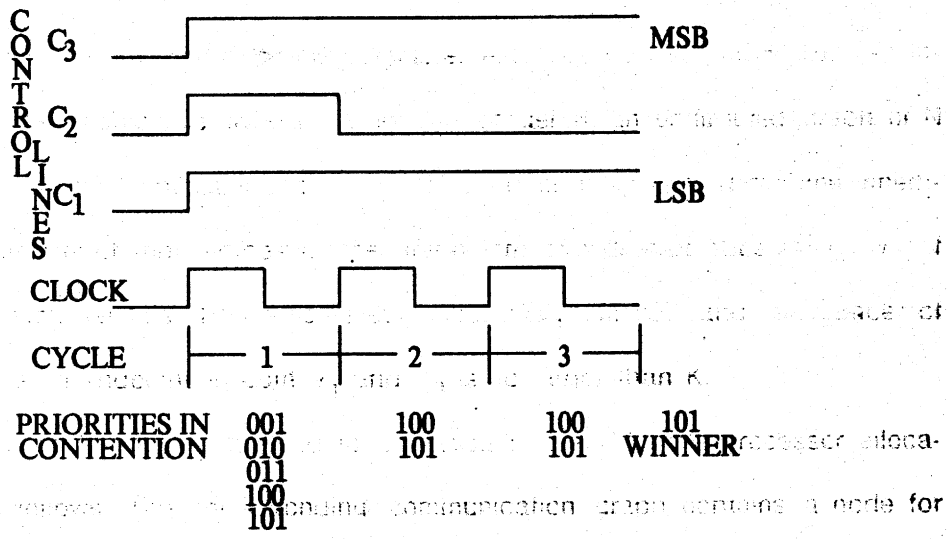


Figure 4.3
Example of distributed contention on a broadcast bus

It is easy to see that this algorithm works by simple binary elimination and therefore requires time proportional to the number of control lines. In the case two or more processors have the same priority, i.e., they have equally urgent transmission operations to execute, the conflict can be resolved by another round of distributed arbitration in which each remaining processor can use a unique preassigned priority. (It is in fact possible to break a tie by enforcing a fair sharing policy which does not rely on preassigned priorities. See [MOK & WAR 79].) Since the priorities are compared one bit at a time, we can in fact use only one control line if the contending processors are sufficiently synchronized to make the decision to withdraw from contention or not at about the same time.

In the implementation described above, the actions of the contending processors are synchronized by means of *clock cycles*. In a network environment, the use of a single global clock may not be desirable since the propagation delay of the bus may introduce significant signal skew. However, propagation delay is a transient phenomenon which will not pose any problems if the *clock cycles* are long enough so that the con-

trol lines achieve a steady state throughout the system. This minimum clock cycle is a function of the propagation delay τ (i.e., the time it takes a electrical signal to travel from one extreme to another extreme of the physical bus). In particular, if the bus has been idle for some time, then the first clock cycle will be established by the first processor to assert its priority on the control lines. However, the new state of the control lines may not be observable by another processor at the other extreme of the bus until τ time units later. The signals from the latter processor will take another τ time units to reach the first processor which must therefore wait for at least 2τ time units before making the first decision to withdraw or not. To allow for a margin for error, we may require each processor to wait for $\delta + 2\tau$ time units before making a decision. For n lines, the maximum time for bus conflict resolution is therefore $n(\delta + 2\tau)$ time units. It should be noted that since the processors are synchronized at the first clock cycle of each contention phase, drifting of the local clocks of individual processors will not accumulate with time. However, all local clocks must not be allowed to drift more than δ time units within the entire bus conflict resolution phase.

If extra reliability is required, we can simulate, by means of a pair of clock lines a "global clock" which is jointly maintained by all the contending processors. Ordinarily, the two clock lines are in opposite logic states (i.e., either 10 or 01). At the beginning of a contention phase, the processor which establishes the first clock cycle by asserting its priority on the control lines also attempts to complement the clock lines. The clock lines will therefore momentarily stay in the 11 state. The transition to the 11 state is a signal for all bus contenders to change the state of the control lines. All processors signal ready by attempting to complement the clock lines which will remain in the 11 state as long as one processor is not yet ready. Synchronization is completed when all the contending processors have complemented the clock lines which will then

be in a different 10 or 01 state.

The decision to withdraw from contention can then be made when the clock lines have settled down to a new (either 01or 10) state. Since it takes τ time units for the 11 state to propagate from one extreme of the bus to the other and another τ time units for the new state to settle, synchronization can be achieved in 2τ time units. In a noisy environment, spurious signals may cause the clock lines to be in the transitory 11 state for very brief moments. Noise immunity can be improved by requiring the contending processors to maintain the stable (01 and 10) states for δ time units where δ is some reasonable system parameter. The maximum time for bus conflict resolution is the same as before. If two contending processors attempt to set the clock lines to opposite logic states, a deadlock in the transitory state 11 will occur. This situation is in theory impossible because one of the processors must have failed to observe on the clock lines a 11 state followed by one which is *opposite* to the state the processor is supposedly maintaining on the same clock lines. In any case, permanent deadlocks can be avoided by a time-out restriction on the 11 state. More robust mechanisms using multiple clock lines and better encoding schemes are also possible.

4.4 Complexity of the Processor Allocation Problem

Since the processor scheduling problem is already NP-hard, it is unlikely that the processor allocation problem for the hard real-time environment will have an efficient solution. In fact, the problem of finding a feasible partition is in itself a hard problem in the sense that it is difficult even if the scheduling problem can be restricted to simple cases for which trivial solutions are available. As evidence, we shall show that the processor allocation problem is NP-complete even for the case where only two processors are available and the processor scheduling problem resulting from any partition is easy.

We shall make use of a restricted version of the MINIMUM CUT INTO BOUNDED

SETS problem which is known to be NP-complete, e.g., see [GAR & JOH 79]. An instance of this graph problem is given by a positive integer K, an undirected graph of N vertices with two special vertices s and t. The decision problem is to determine whether there is a partition of the vertices of the graph into two disjoint subsets V_1, V_2 of equal size (i.e., each set has N/2 vertices) such that $s \in V_1$ and $t \in V_2$ and the number of edges which have an endpoint in both V_1 and V_2 is no larger than K.

This graph problem can be reduced to a restricted version of the processor allocation problem as follows. The corresponding communication graph contains a node for each vertex in the graph problem plus a special node v'. All nodes have unit computation time. Every node v in the communication graph is supposed to denote a function which computes n different output variables where n is the number of edges connected to the vertex v in the graph problem. The edge set of the processor allocation problem contains n edges from node v to the special node v'.

The timing constraints are all periodic and have the same period equal to 1+N/2. For each node except v', we create a timing constraint which consists of a single instance of the node. The deadlines of these timing constraints are set to N/2 except for the nodes s and t for which the deadlines are set to 1. For each edge connecting the vertices v_i, v_j in the graph problem, we create a timing constraint whose deadline is set to 1+N/2. Each of the timing constraint graphs has three nodes, v_i, v_j, v', and two edges, one from each of v_i, v_j to v'. An edge from a node u to the node v' denotes the transmission of a unique output variable from u to v'.

There are two processors which are connected by a bus. If two nodes v_i, v_j appear in a timing constraint and they are allocated to separate processors, then a transmission operation must be performed over the bus to send the appropriate output variable of one of the two nodes to the other processor so that the operation v' can be executed. Otherwise, the output variable is transmitted via a shared variable on the

same processor and no additional communication overhead is incurred. Each processor is assumed to conduct a number of bus transactions in between processing data and the maximum capacity of the bus is such that K or fewer transmission operations from one processor to another can be completed in a time interval of length N/2–1. The decision question of the processor allocation problem is whether there is a partition of the communication graph such that all timing constraints can be met.

Theorem 4.1

The restricted version of the MINIMUM CUT INTO BOUNDED SETS problem can be reduced to the above processor allocation problem.

Proof

Notice that the timing constraints are set up such that both the operations s and t must be executed simultaneously right at the beginning of each period. Hence, they must be allocated to separate processors. Also, an instance of every operation must be executed in the first N/2 time units of every period so that the nodes must be partitioned into two sets of equal size and executed on separate processors. The operation v' must be executed in the last time unit of every period on either one of the processors.

If the instance of the MINIMUM CUT INTO BOUNDED SETS problem has a solution, then we can allocate the two subsets of the corresponding partition to separate processors. By our construction, two nodes appear in the same timing constraint if and only if they are connected by an edge in the graph problem. Hence, a solution partition can split up at most K pairs of nodes which appear in the same timing constraint and thus at most K transmission operations need to be performed over the bus In a period starting say, at time r, all the timing constraints can be met by executing the operations s and t in the interval $[r,r+1]$ on separate processors, the operation v' in

$[r+N/2, r+N/2+1]$ on either processor and the rest of the N–2 operations in the interval $[r+1, r+N/2]$. Since the bus has the capacity for K transmission operations in the interval $[r+1, r+N/2]$, the communication requirements can also be met and the partition is a feasible solution to the processor allocation problem.

Conversely, if the processor allocation problem has a solution, then no more than K transmission operations are required during each period. Since each transmission operation sends a different output variable of a node from one processor to the other, at most K edges can connect two nodes which have been allocated to different processors. Hence the two sets of nodes on the two processors correspond to a feasible partition for the graph problem. QED

In practice, there are probably additional constraints on the processor allocation problem (e.g., certain operations must be performed on specific processors) so that acceptable suboptimal partitions can be found by search algorithms which take advantage of the additional constraints to narrow the search space. While a considerable amount of research has been done in developing heuristic graph partitioning algorithms, e.g., [KER & LIN 70], these techniques are not immediately applicable to the processor allocation problem since we have shown that communication constraints in the hard real-time environment are not exactly expressible in the form of algebraic inequalities involving the edge weights of a graph. However, they may be used to guide a search algorithm if every edge of the communication graph can be assigned one or more parameters which reflect the multiple constraints on the traffic generated along an edge, e.g., one parameter for the volume and another for the urgency of the traffic. Graph partitioning techniques may then be used to satisfy multiple objectives simultaneously and eliminate candidate partitions which are unlikely to meet all the timing constraints.

4.5 Hierarchical Approach to Resource Allocation

For a large system consisting of a variety of special processors, it is unrealistic to distribute the required computation by performing a global optimization involving all system resources. It should be noted that the graph-based model can be used to model the global resource allocation problem hierarchically. Specifically, a node in the communication graph need not represent a simple function element but may be used to denote the communication graph of a subsystem. Similarly, an edge in a communication graph may denote the data path for transmitting complex data structures. More importantly, the formulation of the processor allocation problem discussed in this chapter can be applied to every level of a hierarchy of resource allocation problems. The advantages of a uniform problem representation are obvious.

Chapter 5

Automation of Software Design

5.1 Design System for Hard Real-Time Software

The goal of our research is to provide a methodology and associated tools to automate the design and maintenance of hard real-time system software. In particular, our work has been largely motivated by the development of the experimental software design system CONSORT (CONtrol Structure Optimized for Real Time) which has been implemented† at MIT [WARD 78]. CONSORT has a graphics interface which allows a user to compose the block diagram of a control system by connecting appropriate input/output ports of function blocks that are instantiated from a library of software modules. The user can specify latency (asynchronous) timing constraints on port-to-port paths in the block diagram and the CONSORT compiler will attempt to generate object code which meets all the specifications. If CONSORT is unable to guarantee that all the timing constraints can be met, the user will be duly notified. As a "toy" experiment, this design system has been successfully used to generate a software-implemented controller which drives a pair of motors to balance an inverted pendulum. The inverted pendulum is held by a gimballed holder on a cart which is free to move within the boundary of a square frame; the software controller is a microprocessor assembly program which has been generated automatically from the block-diagram representation of the control problem.

In this chapter, we shall review some implementation aspects of CONSORT and

† The design team of CONSORT was under the leadership of Professor Stephen Ward and included John Pershing, Tom Teixeira and the author, with able assistance from Chris Cesar who helped baby the inverted pendulum balancer and from Jay Wahid who contributed his expertise in control engineering. A video tape of the "toy" experiment (balancing an inverted pendulum with CONSORT) has also been made.

discuss various software/hardware mechanisms which will facilitate the construction of hard real-time systems.

5.2 The CONSORT Design System

There are three main components in the CONSORT design system: the graphics frontend editor, the off-line scheduler and the code generator. The graphics frontend editor provides the graphics facilities for a user to enter a block diagram and specify timing constraints. It also performs syntactic and semantic checks on the functional specifications in the graphics input. The off-line scheduler attempts to find a feasible static schedule which a round-robin run-time scheduler will repeat *ad infinitum* to satisfy the timing constraints. The code generator embeds the static schedule in an assembly language program for the target microprocessor.

The main database for the design system is a software library which contains a number of parameterized functions together with their timing and other interface information. New functions are manually programmed and entered into the library by running an administrative program which maintains the software library. The organization of the CONSORT design system is shown in figure 5.1.

5.3 Implementation Ideas

The CONSORT project has largely fulfilled its objectives and has brought forth a number of ideas for future implementations of software design systems. It has also led to some ideas for improving the performance of hard real-time systems. We shall sketch some of these ideas below.

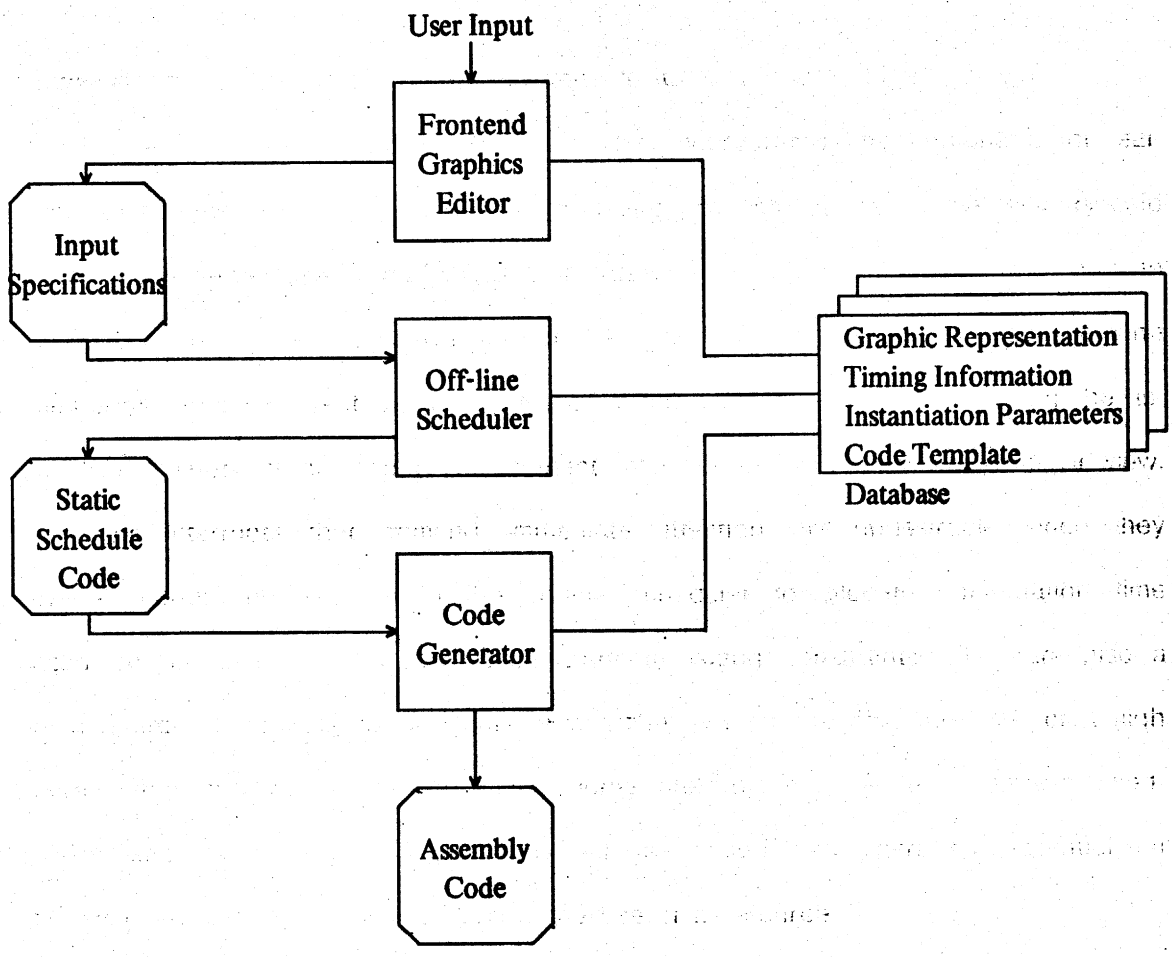


Figure 5.1
The CONSORT Design System

5.3.1 The User Interface

Although a graphics interface seems to be an effective means for entering the description of a control application, it is not very compact for describing large systems. Furthermore, the specification of timing constraints will not be as easy if there are cycles in the communication graph. This is not a problem with CONSORT since it allows only acyclic graphs so that the port-to-port timing constraints of CONSORT are uniquely specified by a pair of input/output ports. (All the function blocks on a path between the two ports are automatically included in the timing constraint.) However, this restricts the use of CONSORT for implementing distributed systems since the absence of cycles

may require some state information to be stored in a shared memory. In a distributed system, a shared memory may not be available.

In a practical system, a textual representation seems to be a more realistic choice. Instead of an Algol-like real-time language, however, the input should be in a higher level specification language which is as close as possible to the relevant application domain, e.g., avionics software may be specified in a language like the one developed for the flight program specification of the A-7 aircraft by Parnas and Heninger [HENI 80]. An application engineer may then specify system requirements in the language of his application domain and the domain specific language can in turn be converted into an instance of our graph-based model by an appropriate translation tool for resource allocation purposes.

5.3.2 Automatic Pipelining of Operations

The off-line scheduler of CONSORT generates static schedules which are lists of operations (procedure calls) and effectively disallows preemption among different operations. Without further information about shared variables, this conservative approach is justifiable on the grounds that different operations may access the same shared variables and their execution must be kept mutually exclusive so that integrity constraints will not be violated. (Since CONSORT block diagrams are acyclic, it is not always possible to make all shared variables explicit by passing their values from port to port. In general, CONSORT makes no assumption about how information is shared by the different operations at all.) When there are operations with long computation times, this may impose a severe response time penalty.

A clean solution to this problem is to remove the acyclicity restriction on the input graph so that every shared variable can be associated with a port. A long operation can then be "pipelined" into a chain of shorter ones such that each stage of a chain

receives as input the intermediate state of an execution (e.g., in the form of state variables) from the previous stage. Since there are now no hidden shared varaibles, integrity constraints will be maintained as long as only one execution of any stage is allowed to be in progress at any time. Ideally, this pipelining of software should be done automatically by the code generator in conjunction with the scheduler.

5.3.3 Detection and Queueing of Activation Conditions

One of the objectives of CONSORT is to demonstrate that external interrupts are not essential to a model of real-time computation and we have taken care that the implementation of CONSORT is faithful to its goal, e.g., keyboard I/O at the operator terminal is treated just as another asynchronous timing constraint and scheduled accordingly. If an external signal (e.g., from a sensor) changes value, then the timing constraints that require sampling the signal will be automatically activated. CONSORT assumes that every external signal is continuously changing and an activation of an asynchronous timing constraint can occur as soon as the deadline of the previous activation expires. Hence, for analog signal processing, timing constraints are activated at their specified maximum rate.

In general, a user may want to perform certain computation when some event occurs. The occurrence of an event may be recorded by the hardware or it may correspond to the output of some boolean operation being evaluated to true. In the former case, the conventional approach for activating the relevant timing constraint is to notify the processor by setting a hardware interrupt flag. In our approach, the processor need not respond to the interrupt immediately since the on-line scheduler will schedule an execution of the computation activated by the event before the specified deadline. The occurrence of an event, however, must still be detected and held by the

hardware until the processor is free to attend to it.

A useful architectural concept that may be incorporated into computers for real-time applications is to augment the interrupt circuitry with an associative memory chip so that an external device may request an interrupt by writing its own address into an empty location in the associative memory. The on-line scheduler can then determine whether a particular event or group of events has occurred by simply querying the associative memory at an opportune moment later.† From a scheduling point of view, processor interrupts that demand immediate attention are undesirable since they severely curtail the freedom of the on-line scheduler to allocate computation time based on an analysis of the specified stringent timing constraints. They are also a prime source of robustness problems in practice, e.g., if the hardware flag of a high priority interrupt is stuck, then all lower priority interrupts may be permanently blocked. This is especially disastrous for the hard real-time environment where it is essential not to permit any one device to monopolize the use of a resource.

In the graph-based model, a minimum period must be specified for every asynchronous timing constraint so that two or more activations of the constraint cannot occur arbitraily close to one another. In practice, it may not be possible to specify the minimum period for an asynchronous timing constraint accurately. For example, while it seems reasonable to specify a minimum period of 100 milliseconds for reading an input character from a human typist (10 key strokes per second), an agile operator may make two or more key strokes in 100 milliseconds every now and then. A solution to this problem is to provide a hardware buffer to queue up the input (as is done in most

† For soft real-time systems, a simple FIFO memory is adequate for storing pending interrupts. From an architectural point of view, the use of an interrupt buffer is also consistent with the pipeline structure of high performance processors since saving the instantaneous state of a complex piplined execution unit in response to an interrupt is likely to be an engineering nightmare that is best postponed to more convenient moments at the choice of the computer architect.

UART chips). In general, a buffer of size n permits n activations of an asynchronous constraint with period p to occur arbitrarily close together in an interval of length $\geq$np.

In the case where a user wants to perform certain computation whenever the output of some boolean operation is evaluated to TRUE, either a periodic or asynchronous timing constraint may be used. Specifically, a periodic timing constraint may be specified to execute the relevant operations that affect the output of the boolean operation and perform the required computation if the boolean output evaluates to TRUE. As far as the scheduling problem is concerned, the value of the boolean output is irrelevant since in the worst case, the required computation must be performed in every period. Alternatively, the user may specify an asynchronous timing constraint which is activated if the execution of the boolean operation yields TRUE. There is a semantic ambiguity here since the boolean operation may never be executed at all if it does not occur in any other timing constraint. The natural interpretation of such a timing constraint is that its graph must include all the operations that may affect the boolean condition plus the required computation if the condition is TRUE, and this asynchronous timing constraint is automatically activated as soon as the specified minimum period expires. To determine whether the "activation condition" has occurred (i.e., whether it is necessary to perform the rest of the computation), a flag may be set whenever the boolean operation evaluates to TRUE and reset after the required computation has been executed.

5.3.4 Dynamic Computation Requirements

In the formulation of the graph-based model, the computation requirements of a system have been assumed to be static, i.e., the same set of timing constraints are to be satisfied throughout the operation of the system. While this is a fair assumption for control applications with a static structure, there are many systems whose computation

requirements often vary with time. For example, the inverted pendulum balancer mentioned earlier may start from a rest position with the pendulum leaning at an angle against a mechanical support. To bring the pendulum to the vertical position, a jerk must be applied to push the cart carrying the pendulum toward the mechanical support. Thus two different sets of control laws are needed to bring the pendulum to the vertical and to keep the pendulum in balance after it has attained the vertical position. For this purpose, CONSORT allows different timing constraints to be enacted by allowing the user to define different *phases* for a real-time application.

A *phase* is a set of timing constraints which must be satisfied when the system is operating in a certain region of its control space where a unique set of control laws must be implemented. There is an initial phase in which a system is started. A phase transition is triggered when the output of some selected boolean operation evaluates to TRUE. When this occurs, all remaining operations of activated timing constraints are then completed and the on-line scheduler starts to satisfy a new set of timing constraints after executing an initialization procedure. During a phase transition, the internal states of all function blocks will normally carry over to the new phase unless they are explicitly modified by the initialization procedure. Since the off-line scheduler can compute a different static schedule for each phase, no new scheduling problem is introduced. The inverted pendulum was successfully balanced starting from a rest position by using CONSORT to implement two sets of control laws in two phases.

In a more general setting, there might be a need to *enable* and *disable* an individual timing constraint at any time during the operation of a system, e.g., in an air traffic control system, a timing constraint may be enabled to monitor an airplane which has just come under the jurisdiction of the system or a timing constraint may be disabled after the corresponding airplane has left. In such cases, a limit must be set on the maximum number of timing constraints that the system can be guaranteed to satisfy

at any time and the scheduling problem for the worst case can then be solved.

Chapter 6

Conclusion

6.1 Summary of Thesis

This thesis can be roughly divided into two parts. The first part (chapters 1-3) considers two models for expressing the computational requirements of hard real-time systems: the process models and the graph-based model. The related real-time scheduling problems for a single processor have been examined in detail to delineate the boundary between the efficiently solvable and computationally intractable problems. The graph-based model which we have proposed has been shown to be more amenable to the development and maintenance of hard real-time systems. The second part of the thesis (chapter 4-6) deals with the more pragmatic aspects of implementing hard real-time systems. A robust formulation of the processor allocation problem for distributed systems has been given and various aspects of CONSORT, a design system for hard real-time software have been discussed. The following is a summary of the salient results in each chapter.

Chapter one emphasizes the advantages of hard real-time systems over the conventional soft real-time approach. The implications of stringent timing constraints on system design and maintenance are discussed, and past research in real-time system design methodologies is reviewed.

Chapter two formalizes the real-time scheduling problems for process-based models. The unrestricted use of semaphores for coordinating concurrent processes is shown to quickly lead to NP-hard scheduling problems. Scheduling algorithms are given for the construction of on-line schedulers which allow processes to synchronize with one another (by using the deterministic rendezvous) and also to implement mutual exclusion when the critical sections are of the same length (by using kernelized moni-

tors). The implications of the scheduling results on the design of real-time programming languages are also discussed.

Chapter three starts with an examination of the semantic gap between process-based models and the computational requirements of the hard real-time environment. It is seen that the decomposition of the required computation into processes is likely to cause substantial maintenance problems when design parameters are modified unless an inefficient decomposition can be tolerated. We then introduce a graph-based model which is semantically closer to the hard real-time environment. The latency scheduling technique for meeting asynchronous timing constraints is formalized within the graph-based model. An upper bound on the length of feasible static schedule if one exists is derived. The problem of computing a static schedule is, however, NP-hard even for very restricted cases but is efficiently solvable if the computational demand is sufficiently light. A heuristic algorithm for scheduling both periodic and asynchronous timing constraints is also given.

Chapter four demonstrates the practical hazards of implementing a distributed hard real-time system with the assumption that the operations of all subsystems can be completely synchronized. We then present a robust formulation of the processor allocation problem which is to a certain degree resiliant to the indeterministic internal behavior of the communication subsystem. The related scheduling problem for a broadcast bus is solved as an example to illustrate our approach. The general processor allocation problem is shown to NP-complete even if the related single processor scheduling problems can be trivially solved. We note, however, that the subproblems resulting from a processor allocation can all be expressed in terms of the graph-based model that we have introduced. Moreover, the graph-based model can be used to give a hierarchical formulation of resource allocation problems in the hard real-time environment.

Chapter five reviews the CONSORT design system which allows a user to specify

the computational requirements of a hard real-time application in terms of a block diagram which is a restricted version of our graph-based model. Various software and hardware techniques that can improve the design of hard real-time systems are discussed.

Chapter six is the conclusion and explores avenues for further research. The potential of the domain specific model approach for software automation is stressed.

6.2 The Domain Specific Model Approach to Software Automation

While past research in software engineering has emphasized the disciplined use of vertical (hierarchical) decompositon and horizontal (modular) decomposition techniques to structure large software system, the resulting performance issues have been largely ignored in the development process. This is usually justified by the overiding need to control the complexity of a design. (It has also been argued that the apparent complexity of software systems is dictated by the need to use resources efficiently, i.e., the "kludges" are there for performance reasons, and not only because of a lack of discipline.) Unfortunately, a lot of useful information which can be used to guide optimization efforts may be irretrivably lost by the time a design has gone through layers of abstractions so that subsequent optimization can only be applied to remove bottlenecks when they are located. This is usually accomplished by analyzing the frequency of primitive operations which are at the bottom layer of a hierarchical design and hence little optimization is done at the higher levels of the design. The performance penalty for failing to exploit these opportunities may not be negligible for some systems, espeically hard real-time applications.

The introduction of the graph-based model in this thesis illustrates a potentially powerful approach to building software systems that also allows performance issues to be considered at every level of the development process. In general, we want to find a

model that is *proper* for the application domain of interest. A model is *proper* with respect to an application domain if it has the following properties:

(1) The model must be *expressive* in the sense that there is an algorithm for translating every instance of a design problem in the application domain into an instance of the model.

(2) The model must be *natural* in the sense that it does not introduce any artificialities which would prevent an instance of the model from being implemented because of unnecessary constraints imposed by the model on resource allocation.

(3) The model must be *implementable* in the sense that every instance of the model induces a set of well defined resource allocation problems the solution of which will yield a feasible solution of the design problem.

For hard real-time applications, we have seen that process-based models do not satisfy the second condition inasmuch as it is impossible for a process decomposition algorithm to translate a design problem into a feasible set of processes without knowing the precise performance characteristics of available system resources. This is indicative of a loss in information that is useful for optimization purposes when a design problem is translated into a process model. In constrast, the graph-based model does not suffer from the same problems.

In a hierarchical design methodology, substantial useful information which may be exploited to guide optimization efforts can be retained if a proper model is constructed at every level of a top-down design so that demand on resources can be characterized at every level in terms of the more primitive computational events at the next lower level. More importantly, it may be easier to meet system specifications by designing appropriate allocation strategies for (abstract) resources at every design level so that performance objectives imposed from a higher level can be met.

6.3 Avenues for Further Research

There are two directions of research which need to be pursued to further the state of the art in designing hard real-time systems. First, although a feasible run-time scheduler exists if the demand on processor time is not too heavy (< 50%), better heuristic algorithms are needed to solve the scheduling problems of the graph-based model for both periodic and asynchronous timing constraints. While we have shown that the processor allocation problem can be factored into a number of subproblems which are all expressible in terms of the graph-based model, specific communication networks need to be characterized for their capacity to meet stringent timing constraints and this immediately spawns a large number of interesting problems.

The second direction of research is to experiment with various approaches for supporting the iterative design cycle which an application engineer usually goes through to arrive at a final design. More specifically, we have formulated the scheduling problems so that either a feasible on-line scheduler is found or the user is told that a feasible implemenation cannot be found. It would be much nicer if we can give the user more feedback information about the tradeoffs that can be made to arrive at a feasible solution. Obviously, there is a lot of work that needs to be done in this area.

Finally, we have not addressed the problems of system failure and recovery which must be dealt with in any real-time system, e.g., how should we design hard real-time software so that a system can resynchronize in due time after the hardware has been interrupted by a brief power failure. This is a complex issue for which a suitable formal framework is especially lacking in the case of the hard real-time environment. It would be interesting to see if system reliability problems can be approached from the top by using a computation model similar to the graph-based model in this thesis. However, we have only a few tentative ideas about how system reliability can be assessed by considering interruptions to software functions and relate their significance to hardware

design.

Appendix

A Summary of Scheduling Terminology and Results

For ease of reference, we provide a summary description of the two major real-time computation models discussed in this thesis and their related scheduling problems. Although deterministic scheduling problems have been widely studied by workers in diverse fields such as computer science, operations research, industrial engineering (see [LAG et al 81] for a computerized classification and [BRU et al 75] for an extensive survey of machine sequencing complexity results), the usual formulation of machine scheduling problems is not directly applicable to the real-time scheduling problems that are of interest to us. The scheduling results are for the *single processor case* only. While multiprocessor scheduling problems are of considerable theoretical interest, they are somewhat unrealistic unless communication costs and robustness considerations can be conveniently ignored. In general, the processor allocation problem for distributed systems is NP-hard (ref. Theorem 4.1).

Process-based Model

A process model $M = M_p \cup M_s$ is a finite set of processes. A process $T = (c,p,d)$ has three non-negative integer parameters: c (computation time), p (period), d (deadline) with $c \leq d \leq d$. If $T \in M_p$, then it is automatically requested at time = kp for every non-negative integer k (T is a periodic process). If $T \in T_s$, then it can be requested (by an external event) at any integral time instant t, but two successive requests must be at least p time units apart (T is a sporadic process).

A process $T = (c,p,d)$ is schedulable if at a request-time t, T is executed on a processor for c time units in the interval $[t, t+d]$. Process premmptions are allowable only at integral time instants and may be subject to additional scheduling restrictions imposed by communication primitives as follows.

The basic communication primitive for interprocess coordination is the rendezvous command which takes the name of a process or a monitor as its argument. When a process T executes a rendezvous with another process T' as argument, T is suspended until T' also executes a rendezvous with T as argument (This is a synchronization constraint). On the other hand, when a process T executes a rendezvous with a monitor as argument, then the process is suspended until the critical section, i.e., the program of the monitor has been executed in response to the rendezvous (This is a mutual exclusion constraint). A critical section may be executed in response to a rendezvous from a process at any time, but only one instance of the same monitor can be executed at any one time.

The following are the key scheduling results:

(1) In the absence of any scheduling constraint imposed by communication primitives, a process model can be scheduled optimally at run time by any combination of the least slack and earliest deadline algorithms. (ref. Theorem 2.1)

(2) When there are mutual exclusion constraints, *a priori* knowledge of the request-

times of the sporadic processes is necessary for optimal scheduling. However, a process model $M = M_p \cup M_s$ is schedulable without *a priori* knowledge of request-times if $M' = M_p \cup M'_p$ is schedulable where M'_p is an appropriate set of periodic processes derived M_s. (ref. Lemma 2.3)

The following results apply to a process model $M = M_p \cup M_s$ where $M_s = \{\}$ (the null set) or where M_s has been replaced by an appropriate set of periodic processes.

(3) When there are mutual exclusion constraints and the critical sections have different computation times, then the problem of finding a feasible schedule is NP-hard even for a very restricted version of the scheduling problem. (ref. Theorem 2.4)

(4) When there are synchronization constraints but no mutual exclusion constraints, then the earliest deadline algorithm can be modified to schedule the processes on-line by using a precomputed database. (ref. Theorem 2.6)

(5) When there are both synchronization and mutual exclusion constraints, and the computation times of all the critical sections are the same, then the kernelized monitor scheduler may be used to schedule the processes on-line by using a precomputed database. (ref. Theorem 2.7)

There is a semantic gap between process-based models and the computational requirements of the hard real-time environment and this causes substantial efficiency and maintainability problems. The graph-based model eliminates these problems.

Graph-based Model

A graph-based model M is an ordered pair (G,T) where G is a communication graph and T is a set of timing constraints. Specifically, $G = (V,E,W_V)$ is a digraph (which may contain cycles) where V and E are respectively the node and edge sets and W_V is a function which assigns a non-negative weight to each node $\in V$. $T = T_p \cup T_a$ is a finite set of timing constraints each of which is a tuple (C,p,d) where C is a timing constraint graph, p and d are respectively the non-negative integer period and deadline of the timing constraint.

A timing constraint graph C is an acyclic digraph which is compatible with the communication graph G. We say that the graph C is compatible with the graph G iff there is a mapping h such that: (1) if v is a node in C, then $h(v) \in V$; and (2) If e is an edge from a node u to another node v in C, then h(e) is an edge from h(u) to h(v) in E. (A timing constraint graph is meant to define the precedence relation of the computational events that must occur in order to satisfy a timing constraint. The nodes and edges of a timing constraint graph denote respectively the execution of the corresponding functional elements and transmission of data in the communication graph.) The computation time of a timing constraint (C,p,d) is the sum of all the weights of the nodes in C.

If a timing constraint $(C,p,d) \in T_p$, then it is activated automatically every p time units, starting from time = 0 (The timing constraint is periodic). If $(C,p,d) \in T_a$, then it can be activated at any integral time instant t with the provision that two successive activations of the same timing constraint must be at least p time units apart (The timing constraint is asynchronous). If a timing constraint (C,p,d) is activated at time = t, then the timing constraint graph C must be executed in the interval [t, t+d]. A timing constraint graph C is said to be executed in a time interval I if a subset S of the (multi)set of functional elements that have been executed in I forms a partial order such that:

(1) There is a bijective mapping between the functional elements in S and C; (2) Under this mapping, the partial order S is consistent with the acyclic graph C; (3) In the case where the functional elements are distributed, and if the graph C has an edge from a node u to another node v, then an execution of C must include the transmission of the latest output of the functional element u to the functional element v before the corresponding instance of v is executed in the time interval I. Furthermore, we require real-time computation to be pipeline-ordered in the sense that: (1) Two executions of a functional element must have distinct start-times and that the execution which has an earlier start-time must also finish earlier than the other. (2) Two data transmissions from a functional element u to another funtional element v must be sent at distinct instants at the site of u and the earlier transmission must also be received earlier at the site of v.

A periodic/asynchronous timing constraint (C,p,d) may be mapped into a periodic/sporadic process T' = (c,p,d) where the body of T' consists of a straight-line program which is any topological sort of the operations in the timing constraint graph C. The computation time c of the process T' is then the computation time of C. In order to enforce pipeline ordering, we create a monitor for each functional element that occurs in two or more timing constraints. To improve efficiency, we can reduce the size of critical sections by software pipelining, i.e., decomposing a functional element into a chain of sub-functions each of which has the same computation time. (One of the virtues of the graph-based model is that all the data dependencies are made explicit and hence software pipelining is easier.) The scheduling results for the process-based model can now be applied to the graph-based model by mapping each timing constraint into an equivalent process. However, this approach is inefficient since it does not take advantage of operations that are common to two or more timing constraints. The latency scheduling technique for meeting asynchronous timing constraints takes

advantage of common operations and is formulated in terms of the graph-based model as follows.

Let M = (G,T) be a graph-based model. An execution trace of a processor is a mapping F from the non-negative integers to V $\cup\{\varphi\}$ where V is the set of functional elements in G and the symbol φ denotes an idle interval. F(i) = u if the functional element u is being executed on the processor in the time interval [i,i+1]; F(i) = φ if the processor idles in [i,i+1]. An execution trace F is said to have a latency of k time units with respect to a timing constraint (C,p,d) iff F contains an execution of C in any time interval of length $\geq$k.

A static schedule is a finite string of symbols in V $\cup\{\varphi\}$. A static schedule L is said to have a latency of k time units with respect to the timing constraint (C,p,d) iff the execution trace which a round-robin scheduler generates by repeating L *ad infinitum* has a latency of k time units with respect to (C,p,d). A static schedule L is said to be feasible with respect to a set of asynchronous timing constraints T_a iff L has a latency of d time units with respect to every timing constraint (C,p,d) $\in T_a$.

The following are the key results for computing a feasible static schedule for a graph-based model (G,T) where T_p = { }, i.e., all the timing constraints are asynchronous. The same results hold if $T_p \neq$ { }.

(1) Suppose there is an execution trace F which has latency d with respect to every asynchronous timing constraint (C,p,d) in a graph-based model (G,T), then there must be a feasible static schedule (finite by definition) with respect to T. (ref. Theorem 3.6) This result shows that feasible static schedules can always be computed in finite time.

(2) The problem of determining whether a feasible static schedule exists for a graph-based model (G,T) is NP-hard in the strong sense even for the following two restricted cases:

(i) All the functional elements in G have unit computation time and all the timing constraint graphs in T are chains of length 1 or 3. (ref Theorem 3.8)

(ii) Every timing constraint graph in T consists of a single operation; all but one of the deadlines are the same and the functional elements cannot be pipelined into chains of sub-functions each of which has the same computation time. (ref. Theorem 3.9)

In general, we propose a heuristic algorithm which first computes a static schedule to satisfy the periodic timing constraints and then incorporates additional operations to satisfy the asynchronous timing constraints. We can also provide a lower bound on the performance of heuristic algorithms.

(3) If (i) $\Sigma\, w_i/d_i \leq 1/2$ where w_i, d_i are the comutation time and deadline of the i^{th} timing constraint; (ii) $[d_i/2] \geq w_i$; (iii) all the functional elements can be pipelined, then a feasible static schedule always exists. (ref. Theorem 3.10)

Unfortunately, static schedules are hard to compute even for periodic timing constraints alone since we can prove:

(4) The problem of computing a feasible static schedule for a graph-based model (G,T) where $T_a = \{\ \}$, (i.e., all timing constraints are periodic) is NP-hard. (ref. Theorem 3.11).

The negative results are disappointing in the sense that it is unlikely that we can find polynomial-time off-line schedulers. Once we have a feasible static schedule, however, the run-time scheduler can use it efficiently on-line.

Bibliography

[ADA MAN 80] "Reference Manual for the Ada Programming Language", Proposed Standard Document, United States Department of Defense, Jul. 1980.

[ALFO 77] M.W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements", *IEEE Transactions on Software Engineering*, vol. 3, no. 1, Jan 1977, pp. 60-69.

[BARN 79] R. Barnes, "A Working Definition of the Proposed Extensions for PL/1 Real-Time Applications", *SIGPLAN Notices*, vol. 14, no. 10, Oct. 1979, pp. 77-99.

[BLAZ 76] J. Blazewicz, "Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines", *Modelling and Performance Evaluation of Computer Systems*, E. Gelenbe, ed., North-Holland Publishing Company, 1976, pp. 57-65.

[BRU et al 75] P. Brucker, J. Lenstra and A.H.G. Kan, "Complexity of machine scheduling problems", Report BW 43/75, Mathematisch Centrum, Amsterdam, 1975.

[CESA 80] C. Cesar, private communication.

[CHU et al 80] W.W. Chu, L.J. Holloway, M.T. Lan and K. Efe, "Task Allocation in Distributed Data Processing", IEEE *Computer*, Nov. 1980, pp. 57-69.

[COHE 78] R.M. Cohen, "Formal Specifications of Real-Time Systems", *Proceedings of the 7th Texas Conference on Computing Systems*, Houston, Oct. 1978, pp. 1.1-1.8.

[DERT 74] M. Dertouzos, "Control Robotics: the procedural control of physical processes", *Proceedings of the IFIP Congress, 1974*, pp. 807-813.

[DEW & PRI 77] J.B. DeWolf and R.N. Principato, "A Methodology for Requirements Specification and Preliminary Design of Real-Time Systems", Report C-4923, Charles Stark Draper Laboratory, Inc., Jul. 1977.

[GAL & MEG 77] Z. Galil, N. Megiddo, "Cyclic Ordering is NP-complete", *Theoretical Computer Science*, vol. 5, no. 2, Oct 1977, pp. 179-182.

[GAR & JOH 79] M. Garey and D. Johnson, *Computers and Intractibility: a Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, California, 1979.

[GAR et al 81] M. Garey, D. Johnson, B. Simons and R Tarjan, "Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadlines", *SIAM J. Comput.*, vol. 10, no. 2, May 1981, pp. 256-269.

[GARM 81] J.R. Garman, "The Bug Heard 'Round the World", *Software Engineering Notes*, vol. 6, no. 5, Oct. 1981, pp. 3-10.

[HAM & ZEL 76] M. Hamilton and S. Zeldin, "Higher Order Software - a Methodology for Defining Software", *IEEE Transactions on Software Engineering*, vol. 2, no. 1, Mar. 1976, pp. 9-32.

[HANS 78a] P.B. Hansen, "Distributed Processes: a Concurrent Programming Concept", *CACM*, vol. 21, no. 11, Nov. 1978, pp. 934-941.

[HANS 78b] P.B. Hansen, "Multiprocessor Architectures for Concurrent Programs", *SIGARCH News*, vol. 7, no. 4, Dec. 1978, pp. 4-23.

[HENI 80] K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, Jan. 1980.

[HEN et al 75] J.L. Hennessy, R.B. Kieburtz, and D.R. Smith, "TOMAL: A Task-Oriented Microprocessor Applications Language", *IEEE Transactions on Industrial Electronics and Constrol Instrumentation*, vol. 22, no. 3, Aug. 1975, pp. 283-289.

[HOAR 74] C.A.R. Hoare, "Monitors: an Operating System Structuring Concept", *CACM*, vol. 17, no. 10, Oct. 1974, pp. 549-557.

[HOAR 78] C.A.R. Hoare, "Communicating Sequential Processes", *CACM*, vol. 21, no. 8, Aug. 1978, pp. 666-677.

[ICH et al 79] J.D. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine and B.A. Wichmann, "Rationale for the Design of the ADA Programming Language", *SIGPLAN Notices*, vol. 14, no. 6, part B, Jun. 1979.

[KER & LIN 70] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *The Bell System Technical Journal*, Feb. 1970, pp. 291-307.

[KNEI 81] W. Kneis et al., "Draft Standard, Industrial Real-Time FORTRAN, Definition of Procedures for the Application of FORTRAN for the Control of Industrial Processes", proposed by Technical Committee 1 of the International Purdue Workshop on Industrial Computer Systems and of the European Workshop on Industrial Computer Systems (EWICS), *ACM SIGPLAN Notices*, vol. 16, no. 7, July 1981, pp. 45-60.

[KRUL 81] F.N. Krull, "Experience with ILIAD: a High-Level Process Control Language", *CACM* vol. 24, no. 2, Feb. 1981, pp. 66-72.

[LAG et al 81] B.J. Lageweg, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, "Computer Aided Complexity Classification of Deterministic Scheduling Problems", Report BW 138/81, *Mathematisch Centrum*,

Amsterdam, 1981.

[LEIN 78] D. Leinbaugh, "Guaranteed Response Times in a Hard-Real-Time Environment", *Proceedings of the 7th Texas Conference on Computing Systems*, Houston, Oct. 1978, pp. 3.24-3.36.

[LIU & LAY 73] C.L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment", *JACM*, vol. 20, no. 1, January 1973.

[MA et al 82] P.-Y. R. Ma, E.Y.S. Lee and M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems", *IEEE Transactions on Computers*, vol. C-31, no. 1, Jan 1982, pp. 41-47.

[MAO & YEH 80] T.W. Mao, R.T. Yeh, "Communication Port: A Language Concept for Concurrent Programming", *IEEE Transactions on Software Engineering*, vol. 6, no. 2, Mar. 1980, pp. 194-204.

[MART 78] T. Martin, "Realtime Programming Language PEARL - Concepts and Characteristics", *Proceedings of the 2nd Computer Software and Applications Conference*, Chicago, 1978, pp. 301-306.

[MOK 76] A.K. Mok, "Task Scheduling in the Control Robotics Environment", TM-77 Sept. 1976, Laboratory for Computer Science, Mass. Inst. of Tech.

[MOK & DER 78] A.K. Mok and M.L. Dertouzos "Multiprocessor Scheduling in a Hard Real-time Environment", *Proceedings of the Seventh Texas Conference on Computing Systems*, Houston, Oct. 1978, pp. 5.1-5.12.

[MOK & WAR 79] A.K. Mok and S. Ward, "Distributed Broadcast Channel Access", *Computer Networks*, vol. 3, no. 5, Nov. 1979, pp. 327-335.

[REG et al 78] H.K. Reghbati, F.F.L. Chow and V.C. Hamacher, "Some Implementation Results in Real-Time Operating Systems", *Proceedings of the 1978 Canadian Computer Conference*, May 1978, pp. 124-128.

[ROB et al 81] E.S. Roberts, A. Evans Jr., C.R. Morgan and E.M. Clark, "Task Management in Ada - a Critical Evaluation for Real-Time Multiprocessor", *Software Practice & Experience*, vol. 11, no. 10, Oct. 1981, pp. 1019-1052.

[SHNE 79] B. Shneiderman, "Human Factors Experiments in Designing Interactive Systems", *IEEE Computer*, vol. 12, no. 12, Dec. 1979, pp. 9-19.

[TEIX 78] T.J. Teixeira, "Real-time Control Structures for Block Diagram Schemata", MIT Laboratory for Computer Science Technical Report TR-204, Aug. 1978.

[WARD 78] S. Ward, "An Approach to Real-Time Computation", *Proceedings of the Seventh Texas Conference on Computing Systems*, Houston,

Oct., 1978, pp. 5.26-5.34.

[WEI et al 80] A.Y. Wei, K. Hiraishi, R. Cheng and R.H. Campbell, "Application of the Fault-Tolerant Deadline Mechanism to a Satellite On-Board Computer System", *Digest of Papers, The 10th International Symposium on Fault-Tolerant Computing*, Kyoto, Japan, Oct. 1980, pp. 107-109.

[WIRT 77] N. Wirth, "Toward a Discipline of Real-Time Programming", *CACM*, vol. 20, no. 8, Aug. 1977, pp. 577-583.

[YAU et al 81] S.S. Yau, C.C. Yang, S.M. Shatz, "An Approach to Distributed Computing System Software Design", *IEEE Transactions on Software Engineering*, vol. SE-7, no. 4, July 1981.

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 7/14/95

Report # LCS-TR-197

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 183 (192-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:
☒ DOD Form(2) ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description :	Page Number:
(A) IMAGE MAP : (1) UNTITLED TITLE PAGE	
(2-183) PAGES #'ED 2-183	
(184-189) SCANCONTROL, COVER, SPINE, PRINTER'S NOTES, DOD(2)	
(190-192) TRGT'S (3)	

(B) SOME PAGES DOTTED WITH WHITEOUT. (C) MANY PAGES HAVE LATENT IMAGES OF WORDS NOT BELONGING TO IMAGE.

Scanning Agent Signoff:

Date Received: 7/14/95 Date Scanned: 7/21/95 Date Returned: 7/27/95

Scanning Agent Signature: Michael N. Cobb

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MIT/LCS/TR-297	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment		5. TYPE OF REPORT & PERIOD COVERED Ph.D dissertation May 1983 6. PERFORMING ORG. REPORT NUMBER MIT/LCS/TR-297
7. AUTHOR*(s)* Aloysius Ka-Lau Mok		8. CONTRACT OR GRANT NUMBER*(s)* DARPA/DOD N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS MIT Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/DOD 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE Janaury 1984 13. NUMBER OF PAGES 183
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* ONR/Department of the Navy Information Systems Program Arlington, VA 22217		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for Public Release, distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Hard real-time systems, real-time scheduling, distributed systems, software engineering, design automation, embedded systems

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Software designed to function in a hard-real-time environment where strict timing constraints must be met often entails implicit assumptions about a programming language and the underlying system which supports it. Programs which are logically correct, i.e., they implement the intended algorithms, may not function correctly if their assumed timing characteristics are not met. This can occur if the programming language is not expressive

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. continued

enough to permit an adequate specification of the desired timing characteristics of the software or if the expressible timing characteristics cannot be verified before run time. For distributed in particular, the software must be tailored to a myraid of implementation parameters, e.g., communication bandwidth, thus rendering subsequent modifications hazardous.

Our research investigates the basic problems in automating the design and maintenance of hard real-time software. After examining the limitations of the traditional approach to real time software design via process-based models, we shall provide a graph-based computation model which is more suitable for expressing the computational requirements of the hard real-time environment. This model is an extension of CONSORT (control structure optimized for real-time), an experimental software design system which has been implemented to generate process control application programs from block diagram schemata. While our graph-based model is abstract, it can serve as a useful intermediate representation between textual requirements specifications and target application programs. Using the graph-based model, the complexity of the relevant resource allocation problems for meeting stringent timing constraints is investigated.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned
Date: 7/21/1995
M.I.T. Libraries
Document Services