REAL-TIME LOGIC SCHEDULING:
MODELLING AND SPECIFICATION
VERFICATION PROBLEMS
SCHEDULING ISSUES

A MOK

Rapporteurs:	**First Lecture**	R de Lemos A Saeed
	Second Lecture	A Saeed
	Third Lecture	A Saeed

Real Time Logic, Programming and Scheduling

Aloysius K. Mok†
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

1. Introduction

Many definitions have been proposed for real-time systems. However, what we really need to consider is why and how "time" enters into design considerations. This is a fair question since conventional wisdom dictates that programs should be designed to function "correctly" independent of hardware speed. To answer this question, consider the following reasons why timing constraints are necessary and/or desirable.

• Time is an explicit parameter in ensuring system integrity in some applications. For example, the control surfaces of some modern aircrafts must be adjusted at a high rate to prevent catastrophic destruction. This places an upper bound on the response time of the avionics software system. Lower bounds are also needed, as in the case of an operating system which requires a potential intruder to wait for some minimum time before retyping a password that has been entered incorrectly. In these cases, the "physics" of the application dictates the timing requirements.

• Time is an essential synchronization mechanism for solving certain task coordination problems. An example is the Byzantine Generals problem for which it has been proven that there is no asynchronous solution. However, a solution is possible if the generals adopt the synchronous protocol of voting in rounds. Each round of voting imposes a timing constraint on the good generals. Another example of time as an essential synchronization mechanism is the self-stabilization problem of state machines in a distributed environment. Here, a set of state machines are required to always reach a safe configuration in the global (joint) state space in a bounded number of steps from any initial configuration. The machines are allowed to communicate with one another by passing messages only. It has been proved that a solution to this problem is impossible unless some form of timeout is allowed. A solution to this problem requires a machine to be able to determine if a communication channel is empty, e.g., by asking the sender to stop sending and then examining the input buffer after a set period of time. This imposes a timing constraint on the communication network.

It should be noted that timing constraints are required in the above two examples to introduce synchrony so that the task coordination problems can be solved. Even though

† Supported by United States Office of Naval Research under contract number N00014-89-J-1472.

timing constraints are not identified explicitly as such (they are introduced by way of the control abstractions: voting in rounds, testing for an empty channel), they are nevertheless needed to solve the problems. It is possible that there is waiting to be discovered an interesting theory about how partial synchrony imposed by timing constraints can help solve otherwise unsolvable reliability problems.

• Time is a powerful control mechanism which can be exploited to solve problems more efficiently. An example can be found in communication protocols which use rate control to improve throughput, e.g., the NETBLT protocol proposed by Dave Clark's group at MIT. In these protocols, the receiver guarantees the sender that it will be able to process incoming packets at a certain rate, or alternatively, it will meet the deadline associated with each packet. Since the sender does not need to wait for an acknowledge from the receiver, network throughput can be significantly improved, especially for networks where the round-trip transmission time is long compared with the width of a packet. This is going to be very important for fiber optics communication systems. Another example of time as a powerful control mechanism is a solution to the distributed election problem in the case of an Archimedean ring. Here, a set of processors arranged in a ring wants to elect a leader by passing messages around. It has been shown that if the processors work at a pace that are not too different (at most a bounded ratio) from one another, the number of messages that are required to elect a leader can be reduced to less than the theoretical lower bound required in the asychronous case.

In reality, the world is often neither completely synchronous nor completely asynchronous. Timing constraints are a useful means to introduce partial synchrony to systems. How we can exploit the partial synchrony to gain efficiency is an open problem. It should be pointed out that in this case, we should design algorithms which must not cause catastrophic failure if a timing constraint is not met. Rather, an exception may be generated and system performance should be able to degrade gracefully. However, the important point remains that our design objective is to meet the timing constraints. If analysis shows that the timing constraints cannot be met, there is little hope to reap the performance benefits.

It is not our intent to advocate the imposition of arbitrary timing constraints; a discipline needs to be developed to allow us to use timing constraints as control mechanisms in a systematic way.

Having said all of the above, we now give an "academic" definition of a real-time system which does not have the word "time" in it:

A real-time system is one that must synchronize with processes whose progress it cannot directly control.

We hasten to caution the reader not to deduce from the above definition that real-time programming need not involve "time". Timing constraints are sometimes necessary and/or desirable, as the foregoing discussion shows.

1.1 Real-Time Programming Issues

Traditionally, real-time systems have been programmed at the level of assembly languages. This is certainly unsatisfactory from a maintenance point of view. There are two basic issues that must be resolved so that high level languages can be confidently used to program real-time applications:

(1) Expressibility: how should high level languages support the expression of the wide spectrum of absolute timing properties required of software running in the hard-real-time environment?

(2) Enforcement: what language constructs can help/hinder the enforcement of critical timing constraints?

In this paper, we shall discuss these two basic issues by exploiting RTL, a logic for expressing absolute timing properties and by relating results from real-time scheduling theory to language design. There are two major contributions in this paper. The first is a formal way to impose timing constraints on programs. The second is an investigation of the real-time scheduling problems that will allow us to make technical assessments of the efficacy of programming constructs for real-time programming.

Past work in language design for distributed real-time programming includes [Cornhill & Sha 87], [Lo 87], [Donner 87], [Shaw 87], [Volz and Mudge 87], [Lee & Gehlot 85]. In [Lin, Natarajan & Liu 87], a programming system for imprecise computations in real-time applications was introduced. In this paper, We shall discuss a novel approach for expressing timing constraints which can be superimposed on any block-structured language. Our emphasis is on the mechanization of the enforcement of timing constraints.

1.2 Organization of This Paper

To be concrete, our discussion will be centered on an Ada-like language.®. We emphasize that the discussion in this paper applies not only to Ada but also to block-structured languages in general.

The rest of this paper is organized as follows. Section 2 examines the problems in expressing timing constraints by using *ad hoc* time-related constructs in Ada. A formal system of annotating Ada-like programs will be introduced so as to make it possible to specify the absolute timing behavior of real-time Ada programs. Section 3 investigates the problems of scheduling time-critical Ada tasks and analyze the efficacy of the Ada

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

tasking facility in the real-time domain. Section 4 is the conclusion.

2. Expressibility of Timing Constraints

Since Ada is a general-purpose programming language, one may use a Turing universality argument to show that Ada can be used to simulate any desired real-time behavior. In a distributed environment, however, the issue is not so simple. Suppose our job is to program a computer on board a train which is going into a railway crossing. The train is to stop if, after 45 seconds, the controller at the crossing still fails to lower the gate. The following piece of Ada code might be used for this purpose. (The use of the **timed entry call** below follows the suggestion in the Ada Language Reference Manual [Ada Manual 83]. Also, comments in Ada starts with the delimiter "--".)

```
select
  GATE_CONTROLLER.REQUEST;
or
  delay 45.0;
  STOP_TRAIN; -- controller not responding, stop the train.
end select;
```

The **select** statement above has two alternatives. The first alternative is a **rendezvous** with the gate controller (the entry call GATE_CONTROLLER.REQUEST). The other alternative simultaneously starts a watchdog timer. The semantics of the *timed entry call* is given in the Ada Language Reference Manual: "If a **rendezvous** can be started within the specified duration (or immediately, as for a conditional entry call, for a negative or zero delay), it is performed and the optional sequence of statements after the entry call is then executed. Otherwise, the entry call is cancelled when the specified duration has expired and the optional sequence of statements of the delay alternative is executed." Thus if the controller does not respond within 45 seconds, the train will automatically be stopped. However, if the **rendezvous** with the controller starts within 45 seconds but takes a long time to complete (or never completes owing to a controller breakdown in the middle of the **rendezvous**), then the watchdog timer will not be able to take effect according to the Ada Language Reference Manual. The train will therefore not stop even though the intended timing constraint is to stop the train when the controller fails to lower the gate within 45 seconds, i.e., the train should stop if the **rendezvous** does not complete in bounded time. †.

The point of the above example is that in a distributed environment, a timing constraint may involve the execution of more than one task. Since tasks may synchronize and interact with one another in many ways, it is not obvious whether the constructs in a language like Ada are sufficient to express at least the timing constraints that are of

† There is no easy way in Ada to express timing constraints like this. The resolution of this semantic problem is being debated in the Ada community. The annotation system introduced later in this paper will provide a precise way to specify the intended timing property.

practical import. To answer this question, we need a way to specify timing constraints on computation that should be independent of the particular choice of synchronization mechanisms of the programming language. We can then examine a language to see if it is sufficiently powerful to enforce the timing constraints of interest. For this purpose, we shall provide a system of formal annotations for specifying timing constraints in any block-structured language later in this paper.

Given the absolute timing behavior that we want a program to satisfy, the obvious question to ask is whether we can derive the intended timing behavior from a program. To be effective for real-time applications, the timing behavior of a real-time program should be readily deducible from the program text, preferably by a syntactic analysis. For otherwise, there is insufficient information to allocate resources to meet the specified timing constraints. More importantly, a real-time program whose timing behavior is difficult to analyze is also hard to maintain, since it is all too easy to introduce subtle time-related errors when modifications are made. In the next section, we shall discuss the difficulties in deducing the timing behvior of real-time Ada programs.

2.1 Examining the Timing Behavior of an Ada Task

Consider the following skeleton of an Ada task:

```
task body T1 is

declare
 use CALENDAR;
 NEXT_TIME : TIME := CLOCK+INTERVAL;

begin
 loop
   delay NEXT_TIME - CLOCK;
   T2.SYNCHRONIZE; --synchronize with task T2
   CRITICAL_SECTION(); --execute a critical section
   NEXT_TIME := NEXT_TIME+INTERVAL;
 end loop;
end;
end T1
```

The above task, T1 is intended to be scheduled exactly once every INTERVAL time units. Every time T1 is run after the delay, it will synchronize with another task T2 before executing a critical section, and then compute the time at which it should be next scheduled. Suppose we want to write an analysis tool to mechanically determine the intended timing behavior of the task T1. Since the **delay** statement in Ada only puts a lower bound on when a task can be next scheduled, one might expect the semantics of

Ada to permit an analysis tool to suggest that the task T1 should be scheduled at most once every INTERVAL time units. A correct implementation can legally execute T1 far less frequently. Furthermore, if the actual parameter to the delay statement evaluates to a negative number, the **delay** statement will have no effect, and the task T1 might be executed more than once within a time interval shorter than INTERVAL time units. Thus the semantics of **delay** does not guarantee that the task T1 will be executed no more than *or* no less than once every INTERVAL time units.

One might attempt to prescribe the intended periodic timing constraint by raising an exception whenever the actual parameter of **delay** is negative as follows.

```
task body T1 is

declare
  use CALENDAR;
  NEXT_TIME : TIME := CLOCK+INTERVAL;

begin
  loop
    TIME_LEFT := NEXT_TIME - CLOCK;
    if TIME_LEFT<0 then
      raise ERROR;
    else
      delay TIME_LEFT;
    end if;
    T2.SYNCHRONIZE; --synchronize with task T2
    CRITICAL_SECTION(); --execute a critical section
    NEXT_TIME := NEXT_TIME+INTERVAL;
  end loop;
end;
end T1
```

While the use of an exception does force the run-time system to signal an error if a period is missed, it does not in itself lead to the mechanical derivation of the intended timing constraint which requires task T1 to be executed exactly once every INTERVAL time units. Specifically, a mechanical tool may not be able to ascribe the condition that causes the exception (TIME_LEFT<0) to the failure of this particular timing constraint. It may be the case that the programmer intends the variable TIME_LEFT to be always non-negative for some reason other than to enforce a timing constraint. For example, the TIME_LEFT variable may be used subsequently in an arithmetic calculation whose result is meaningless unless TIME_LEFT is non-negative. It is also possible that some other timing constraint which also makes use of the TIME_LEFT variable will fail if TIME_LEFT is negative. In other words, an analysis tool cannot simply assume that the

exception is meant to signal the violation of the intended periodic timing constraint.

In general, the **delay** statement gives the programmer only limited control over the timing behavior of the program. In fact, unless the programmer can take into account the time it takes to evaluate the actual parameter of a **delay** statement, the actual duration of the delay is bound to be bigger than that specified by the computed value. To be fair to the designers of the Ada language, however, one should understand their reluctance to adopt a powerful construct whose semantics effectively dictates an upper bound on how soon certain computation must be completed. To do so would require the compiler to guarantee that the generated code will indeed meet the specified timing constraints, a technical challenge beyond the technology of the time.

It is easy to give a formal proof to show that it is impossible to write an analysis tool which can mechanically deduce the intended timing behavior of an Ada program from its text alone; such a tool would have to be able to solve the halting problem which is of course undecidable. More importantly, the intent of the above discussion is to convince the reader that it is non-trivial even to deduce simple timing constraints even when they are expressed in a straightforward fashion.

It has been suggested by some authors that a **pragma** can be used to inform an Ada compiler of a program's intended timing behavior. For example, a simple pragma may be used to indicate that a task is cyclic and must be executed at a specified rate. This is a good approach, but there are two difficuties with it:

(1) A pragma is only a suggestion to the compiler. The actual timing behavior of the program is still determined by the **delay** statements in the program text. It is possible that the timing behavior as suggested by a pragma may conflict with the **delay** statements. Thus the compiler may end up having to check that no conflict exists as it takes advantage of the pragma information to meet the intended timing constraints. This consistency checking problem is just as hard as the previous one.

(2) The timing behavior of a complex real-time program may be quite involved, and the official repertoire of pragmas may not be sufficient to express the wide spectrum of timing constraints. For example, in every period, the task T1 must also synchronize with task T2 and to gain access to a critical section before the end of the period runs out. These requirements involve interactions among tasks and complicate the timing behavior of the program. In Ada, enforcement of both the synchronization and critical section constraints are usually carried out by using the **rendezvous** construct. Later in this paper, we shall show that in order to meet all the timing constraints, it is very important to be able to distinguish the places in the program text where a **rendezvous** is used for synchronization from where a **rendezvous** is used for implementing critical sections. The need to convey this type of information to the compiler calls for more pragmas. However, this tends to

encourage proliferation of implementation-defined pragmas, thus hurting program portability. Portability is, of course, a major design objective of Ada.

2.2 A Uniform System for Timing Behavior Annotation

We now propose a system for annotating Ada programs that will help us to deal with the two difficulties discussed above. For ease of understanding, the reader may regard our approach as introducing a uniform system (a language really) to define timing-related pragmas. Thus instead of establishing an official repertoire of timing-related pragmas which every validated Ada compiler must support, we advocate a facility for interpreting timing-related annotations (formalized pragmas). We shall explain our system by annotating the tasking program which is our running example.

Program Text::

```
task body T1 is
declare
 use CALENDAR;
 --NEXT_TIME : TIME := CLOCK+INTERVAL;
--v- RUN_T1
begin
 --delay NEXT_TIME - CLOCK;
 T2.SYNCHRONIZE; --synchronize with task T2
 --^- SYNC_WITH_T2
 CRITICAL_SECTION(); --execute a critical section
 --NEXT_TIME := NEXT_TIME+INTERVAL;
end;
--^- SUSPEND_T1
end T1

task body T2 is
begin
 --statements of T2 before synchronization
 accept SYNCHRONIZE; --synchronize with task T1
 --^- SYNC_WITH_T1
 --statements of T2 after synchronization
end T2;

procedure CRITICAL_SECTION() is
begin
--v- ENTER_CS
 --body of critical section
--^- EXIT_CS
end CRITICAL_SECTION;
```

Timing Behavior Specification::

$\forall i$ (i-1)*INTERVAL $\leq$ @(RUN_T1, i) $\wedge$
@(SUSPEND_T1, i) $\leq$ i*INTERVAL

$\forall i$ @(SYNC_WITH_T2, i) = @(SYNC_WITH_T1, i)

$\forall i$ @(ENTER_CS, i+1) $\geq$ @(EXIT_CS, i)

(Note: For simplicity of explanation, we have left out the axioms
for system initialization.)

The reader may notice that in task T1, we have commented out the **delay** statement and related time calculations. This is because the semantics of the **delay** construct in Ada does not lend itself to specifying stringent timing constraints which involve upper bounds on task suspension, as we have discussed earlier; the intended periodic timing constraint is being taken care of by the annotations. We have also included the source codes for task T2 and the critical section which is a procedure.

Our annotation system has two parts: event marker definitions and timing behavior specification. Event markers are stylized comments that are placed strategically in the program text. There are two syntactic forms:

--v- <event name>

--^- <event name>

A event marker can be thought of as a time-keeper of computational activity. Every time a CPU executes a statement on one side of and right next to an event marker, the event marker records the time instant at which it happens. More precisely, if a CPU *initiates* the execution of a statement which is right *below* a event marker, say "--v- E" at time t, then we say that an instance of the event E occurs at time t. If a CPU *completes* the execution of a statement which is right *above* a event marker, say "--^- E'" at time t, then we say that an instance of the event E' occurs at time t. For example, if the first time the scheduler starts running the task T1 is at 8:10 a.m., then the first occurrence of the event RUN_T1 is at 8:10 a.m.

A timing behavior specification is a set of assertions that relate the time of occurrences of different events to one another. The notation @(<event name>,<index>) denotes an application of the function "@" (the occurrence function) to an event and an integer argument. The "@" function can be thought of as the master time-keeper who can interrogate an event marker for the time at which some instance (specified by the <index> argument) of the event occurs. The three assertions in our running example

should be interpreted as follows.

The first assertion states that the i^{th} time at which the task T1 is run must be after the i^{th} period has started, and T1 must be suspended, having completed its execution before the end of the i^{th} period. (For ease of explanation, we assume that the system starts executing at time=0.)

The second assertion states that the i^{th} time the task T1 completes the issue of an entry call to T2 must be at the same time at which T2 completes the acceptance of the same entry call.

The third assertion states that the $i+1^{th}$ time the critical section is entered by some task must not occur before the i^{th} time the critical section is exited by some task.

The timing behavior specifications should be regarded as obligations that any correct implementation should honor. However, if a compiler (or an analysis tool working in collaboration with the compiler) determines that an implementation might miss a timing constraint, then some of the timing behavior assertions cannot be regarded as axioms. If this is acceptable, an exception can be raised to indicate which assertion has been violated. Assertions in the form of implications can be added to the program to catch the exceptions and force recovery actions to happen. For instance, we can define event markers around the entry call in the first alternative of the select statement in our train crossing example, and add an assertion to put a time bound, say 10.0 seconds on the duration of the **rendezvous**:

```
--v- REQUEST_TO_LOWER_GATE
GATE_CONTROLLER.REQUEST;
--^- GATE_LOWERED
```

$$\forall i \quad @(\text{GATE\_LOWERED}, i) - @(\text{REQUEST\_TO\_LOWER\_GATE}, i) \leq 10.0$$

This system of annotation helps to avoid the two difficulties with pragmas because unlike the latter, our annotation system is based on formal logic (actually an extension of Presburger Arithmetic) and so does not permit ambiguity in expressing timing behavior. Our uniform syntax for describing timing properties does away with implemenation-dependent pragmas which may take many special forms.

2.3 Relation with RTL (Real Time Logic)

The approach of annotating Ada programs described here in fact makes use of RTL (Real Time Logic) which is a formal system for reasoning about timing behavior. Details of RTL can be found in [Jahanian & Mok 86]. Briefly, RTL is invented to describe systems for which the absolute timing of events and not only their relative ordering is important. RTL reasons about occurrences of events. We distinguish between four

classes of events: (1) External event, e.g., device interrupts, (2) Start event which marks the initiation of an action (an action in this case is the execution of an Ada statement), (3) Stop event which marks the completion of an action (executing an Ada statement), and (4) Transition event which marks a change in the system state. (Transition events have not been discussed in this paper. They can be used, however, to keep track of the results of expression evaluations, but we shall not develop this idea here.)

In RTL, time is captured by the occurrence function, denoted by the character "@", which assigns time values to event occurrences. The occurrence function is a mapping from the space (E,W) to W where E, W are respectively the set of events and non-negative integers.

Definition:

@(e,i) ≡ time of the i^{th} occurrence of event e; where e is a start, stop, external or transition event, and i is an interger constant/variable.

The notion of the occurrence function is central to RTL. In particular, timing requirements imposed by the system specifications are restrictions on the "@" function. A system satisfies a timing property P if there is no mapping of event occurrences to time values which is consistent with the negation of the property P in conjunction with the system specification. RTL formulas are constructed using the equality/inequality predicates, universal and existential quantifiers, and the first order logic connectives.

In the foregoing discussion, we have examined the deficiencies of Ada in the specification of critical timing constraints. We have proposed a system of formal annotations which can be used to remedy these deficiencies. The rest of this paper will address the issues of timing constraint enforcement in Ada-like languages.

3. Enforcement of Timing Constraints

The formal annotation system that we have presented in the previous section is a very rich language with which a wide spectrum of timing constraints can be specified. It is impossible for a run-time system to meet the wide spectrum of timing constraints with a resource allocation policy that is independent of the class of timing constraint to be met. The ability of the run-time system to exploit the semantics of a programming language to optimize resource allocation is thus crucial for meeting critical timing constraints. This observation suggests a rigorous approach to evaluate the efficacy of programming language constructs for real-time programming: For any given class of timing constraints, we first investigate the related real-time scheduling theory and determine what type of information is crucial for making good resource allocation decisions. A programming language can then be analyzed to see if its interface with the run-time system is sufficiently powerful to convey the crucial information.

In section 2, we have used as a running example a task that has a periodic timing constraint, synchronizes with another task and also enters a critical section. The corresponding timing constraints were formalized by RTL formulas given in the timing behavior specification. We shall investigate the the efficacy of the Ada-like tasking facility for meeting specifically these types of timing constraints. For this purpose, we shall augment the tasking model with timing constraints to formalize the corresponding real-time scheduling problems. This model† is sufficiently sophisticated to include task synchronization and critical sections as they are implemented by the Ada **rendezvous**. Through our analysis of the related real-time scheduling problems, we shall be able to evaluate task scheduling support in Ada and the use of the Ada **rendezvous** for concurrency control in the real-time environment.

3.1 A Tasking Model with Timing Constraints

There are two classes of tasks in our model: user tasks and semaphore tasks. We shall refer to a user task simply as a task for brevity; semaphore tasks will be explicitly stated. For scheduling purposes, the computation of a task 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 task T_i. Concurrency control is achieved through communication primitives which may appear only between scheduling blocks. These communication primitives are used to pass information among tasks for coordination purposes. Their semantics is important only to the extent that they impose certain scheduling restrictions corresponding to task synchronization or mutual exclusion requirements. The precise semantics will be given later when we discuss the related scheduling problems.

When a task 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 $t+d_i$. In general, tasks can be either *periodic* or *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 tasks 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 task from monopolizing system resources. For the purpose of this paper, we shall limit ourselves to periodic tasks. (A technique exists which transforms periodic tasks to "equivalent" sporadic tasks and can be found in [Mok 83].)

An instance of our tasking model is a pair (M, S). M is a finite set of tasks $\{ T_i \}$. The i^{th} task, $T_i = (C_i,p_i,d_i)$ has three parameters: C_i (a chain of scheduling blocks with

† The inclusion of task synchronization and critical sections in our model extends previous models such as [Liu and Layland 73], [Leung & Merrill 80] that treat independent tasks only.

communication primitives in between blocks), d_i (deadline), p_i (period) In this model, we assume that $c_i \leq d_i \leq p_i$. The i^{th} task T_i is requested at time=$(k-1)p_i$ for every positive integer k and the i^{th} execution of T_i must be completed no later than time=$(k-1)p_i+d_i$. S is a finite set of semaphore tasks { S_i } whose sole function is to communicate with (to enforce mutual exclusion among) the tasks in M.

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.) Preemption of a task by another is allowable only at integral time instants and may be subject to additional scheduling restrictions imposed by communication primitives placed between scheduling blocks of a task. A task set is feasible if and only if there is a schedule in which all its deadlines can be met.

3.2 Real-Time Scheduling

Unlike classical scheduling problems, real-time scheduling deals with the problems of continually meeting periodic and sporadic timing constraints. 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 task 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 task(s). A run-time scheduler is *clairvoyant* if it has an oracle which can predict with absolute certainty the future request-times of all tasks. A run-time scheduler is *optimal* if it always produces a feasible schedule whenever it is possible for a clairvoyant scheduler to do so.†

In the following, we shall present some relevant real-time scheduling results that will shed light on the design of concurrency control facilities in real-time programming languages, particularly Ada. For clarity of presentation, we shall formulate the real-time scheduling problems, state the results and discuss their implications; proofs are relegated to the appendix.

3.3 Dynamic versus Static Priority Scheduling for Independent Tasks

A set of tasks are said to be independent if they do not synchronize with one another and do not execute critical sections. (In our tasking model, this corresponds to the absence of any communication primitives in the tasks.) Independent tasks are allowed to

† It can be proved that knowledge about request times is needed for optimal scheduling for *sporadic* tasks with critical sections [Mok 83]. The rationale for the above definitions will be clear after the following discussion.

preempt one another at any integral time instants. In the case the set of tasks is independent and the deadline of each task is equal to its deadline, a well known result in [Liu & Layland 73] showed that a task set is feasible if and only if the utilization factor of the task set does not exceed 1. (The utilization factor of the i^{th} task T_i is c_i/p_i. The utilization factor of a task set is the sum of the utilization factors of the tasks in it.) The run-time scheduler used by Liu to achieve 100% utilization is the *earliest deadline algorithm* which executes at every instant the ready task with the nearest deadline. The *earliest deadline* scheduler is a totally on-line optimal scheduler. (There are in fact infinitely many totally on-line optimal schedulers if tasks are independent [Mok 83]. Let us denote the remaining computation of a ready task at time t by c(t) and its current deadline by d(t) and define the *slack* of the task 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 task before it is bound to miss the current deadline. Another optimal scheduler is the *least slack scheduler* which runs at any time the ready task with the least slack, ties being broken arbitrarily.)

Both the earliest deadline and the least slack schedulers are dynamic priority schedulers, i.e., the priority of a task does not remain fixed throughout a schedule. Unfortunately, Ada does not support dynamic priority scheduling.† In Ada, the only way to control task response times is by assigning static (fixed) priorities to tasks. However, a static priority scheduler is theoretically not as efficient as dynamic priority schedulers, since it has been shown ([Liu & Layland 73]) that the achievable utilization factor of the best static priority scheduler is about 70%. (A scheduler is said to have an achievable utilization factor f if it can always successfully schedule a set of tasks whenever the utilization factor of the task set does not exceed f.) More importantly, the impact of task scheduling overheads on the achievable utilization factor is likely to be worse for static priority schedulers since the lowest priority task can be preempted by every one of the other tasks. Since the best static priority scheduler (the *rate monotonic algorithm* [Liu & Layland 73]) assigns the lowest priority to the task with the longest period, it is likely that the task with the lowest priority will be preempted more than once by each of the other tasks. The worst-case response time of the lowest priority task must therefore account for the scheduling overheads of many more preemptions. This is not so for the earliest deadline scheduler.

On the other hand, a static priority scheduler is easier and cheaper to implement and may thus offset the theoretical advantages of dynamic priority schedulers. In practice, a balance may be struck by the use of a hybrid scheduler, e.g., one that adjusts priorities less frequently than the earliest deadline scheduler. In any case, the lack of support for dynamic priority scheduling is an important weakness in a programming language for real-time applications such as Ada.

† It has been claimed that there are ways to "simulate" dynamic priority assignment by using entry families in Ada. In addition to being cumbersome and impractical, it is unlikely to be the intent of the Ada designers to support dynamic priority scheuling this way.

3.3.1 The Priority Inversion Problem

In [Cornhill & Sha 87], L. Sha and D. Cornhill first discussed a problem called priority inversion that can have a serious impact on the response time of high priority tasks in Ada programs. Priority inversion is the phenomenon where a higher priority task is forced to wait for the execution of a lower priority task. This happens if for example, the highest priority task must **rendezvous** with a low priority task before its execution can proceed any further and the low priority task is preempted by all the other tasks with a higher priority. In [Sha, Rajkumar & Lehoczky 87], protocols were introduced to allow a lower priority task to temporarily adopt the highest priority of the tasks that are waiting for it. These protocols were named priority inheritance protocols. While priority inheritance protocols can alleviate the impact of priority inversion, it is unlikely that the achievable utilization factor can be maintained even at 70%, as in the case of independent tasks. In the next section, we shall investigate the impact of task synchronization on dynamic priority schedulers. We shall show that under certain conditions, the achievable utilization factor can still be maintained at 100% in our tasking model.

3.4 Synchronization with the Ada Rendezvous

In our tasking model, the computation of a task consists of a chain of scheduling blocks which are separated by communication primitives. We now consider the case where a communication primitive is used solely for enforcing synchronization between two tasks, i.e., a task may try to **rendezvous** with another task by executing a **rendezvous** command (i.e., an **entry call** or an **accept statement**).

We call two tasks which synchronize (by means of the **rendezvous**) with each other *communicants*. (This definition defines a *communicant relation* on the set of tasks.) When a task T_i attempts to execute a **rendezvous** (**entry call/accept**) command, it must wait until the corresponding communicant is also executing the corresponding **rendezvous** (**accept/entry call**) command. Information may be exchanged by two tasks 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 tasks. More specifically, a **rendezvous** establishes a precedence constraint which requires that all the computation before the **rendezvous** command in each task must precede all the computation after the corresponding **rendezvous** command in the other task. 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**.

If two user tasks are communicants, then we require that either they have the same period, or one period is an exact multiple of the other. This requirement does not seem to be overly restrictive since in practice tasks which synchronize with one another are likely to perform related periodic application functions; in any case, the scheduling problem is

not significantly harder without this restriction. Also, two communicants are assumed to 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.

3.4.1 The Impact of Synchronization on Task Scheduling

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

Example

There are three periodic tasks. 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. (See figure 1.)

The earliest deadline algorithm will execute the scheduling block T_{11} at time=0 since task T_1 has the nearest deadline. At time=1, T_1 is blocked by a command to synchronize with task T_2. However, T_2 cannot be executed immediately because task T_3 has a nearer deadline. Thus the scheduling block T_{21} will not be executed until time=2 and the **rendezvous** with task T_1 will not be completed until time=3. Thus task T_1 will not be able to make its deadline at time=3.

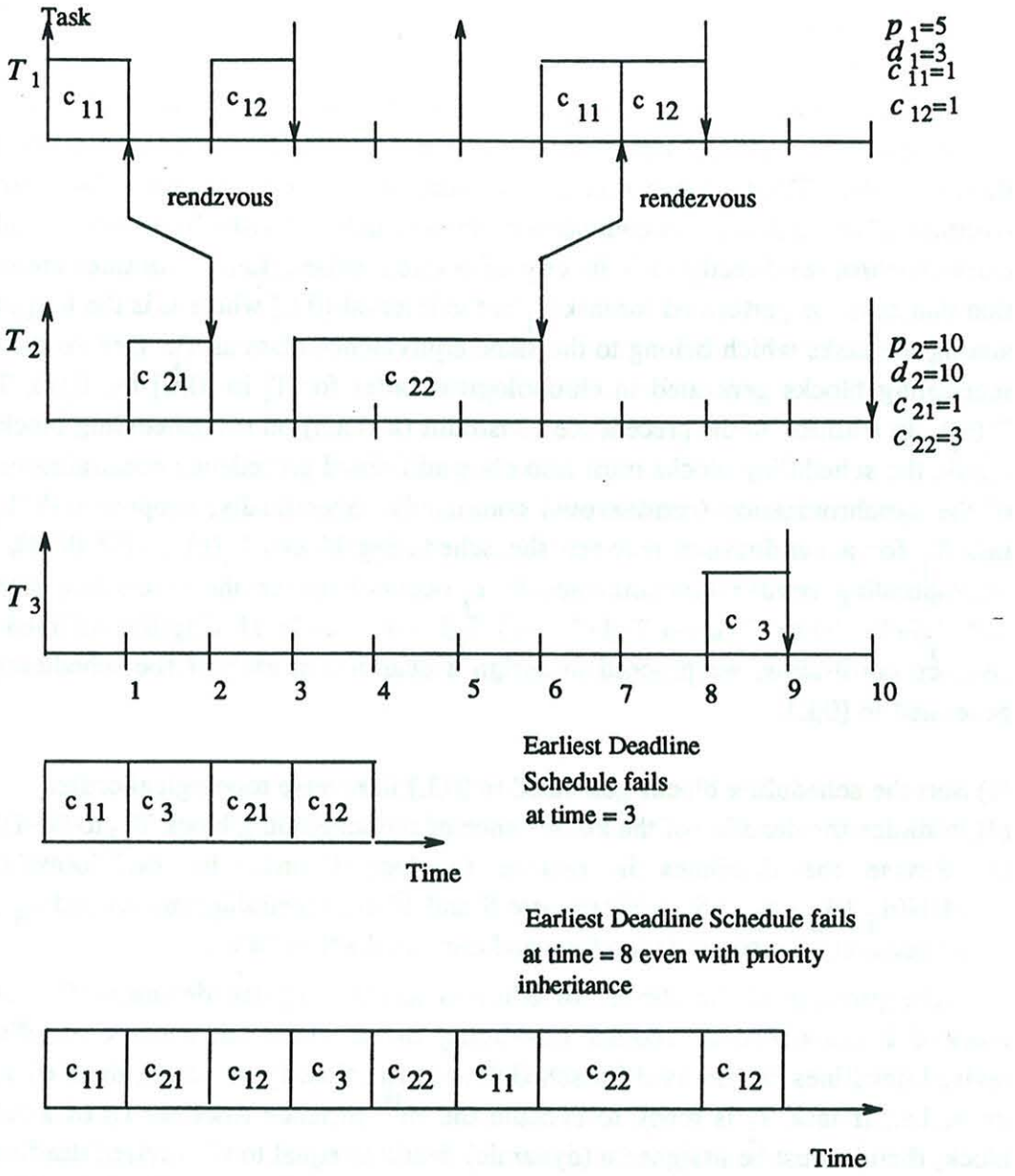

Figure 1. Example of scheduling constraint imposed by task synchronization

This example is the analog of the priority inversion problem for the earliest deadline scheduler. The earliest deadline scheduler fails because the highest priority (nearest deadline) task must wait for a low priority task which can be blocked by other tasks with higher priority. This problem can be solved by adopting a technique for revising deadlines to eliminate precedence constraints in the classical model of scheduling (e.g.,

[Blazewicz 76]). In real-time scheduling, an off-line scheduler is needed to compute a set of *dynamic deadlines* which can then be used for scheduling at run time by a earliest deadline scheduler.

We recall that tasks T_i and T_j are said to be related by a communicant relation if there is a matching pair of sychronization (**rendezvous**) commands in the computation of the two tasks. The reflexive transitive closure of this communicant relation induces a partition of the task set into equivalence classes such that tasks in the same equivalence class synchronize directly or indirectly with one another. Let us consider the computation that must be performed for task T_i in the interval [0,L] where L is the longest period among the tasks which belong to the same equivalence class as T_i. Denote the chain of scheduling blocks generated in chronological order for T_i in [0,L] by $T_i(1)$, $T_i(2)$, ..., $T_i(n_i)$. In addition to the precedence constraint (a chain) on the scheduling blocks within a task, the scheduling blocks must also obey additional precedence constraints as a result of the synchronization (**rendezvous**) commands. Specifically, suppose task T_i targets task T_j for a **rendezvous** between the scheduling blocks $T_i(k)$ and $T_i(k+1)$, and the corresponding **rendezvous** command in T_j 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 the scheduling block $T_{i,j}$ 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. The revised deadlines can be used for scheduling at run time by recycling them every L time units, i.e., if task T_i is ready to execute the m^{th} instance (modulo L) of a scheduling block, then it must be assigned a (dynamic) deadline equal to the revised deadline of that scheduling block relative to time=(m-1)*L. The optimality of the Deadline Update procedure is stated in:

Lemma 1

Suppose (M,S) is a tasking model where all the communication primitives are **rendezvous** commands for task synchronization. Then the feasibility of the model is not affected by using the dynamic deadlines as given by the Deadline Update procedure above. Furthermore, whenever the dynamic deadline of a ready task T_i is nearer than that of another ready task T_j, then scheduling T_i ahead of T_j will not violate any

precedence constraints involving the two tasks.

Theorem 2

If a feasible schedule exists for an instance of a tasking model where all the communication primitives are **rendezvous** commands for task synchronization, then the task set can be scheduled by modifying the earliest deadline algorithm to schedule the ready task which is not blocked by a **rendezvous** and which has the nearest dynamic deadline.

To achieve optimal scheduling, the compiler (or some preprocessor) must therefore collect information about the synchronization commands among tasks and prepare an appropriate database for the run-time scheduler. The concept of an *off-line* scheduler is currently not in the language Ada.

It could be argued that the database of dynamic deadlines may require too much memory space and that a totally on-line scheduler based on the idea of priority inheritance ([Sha, Rajkumar & Lehoczky 87]) might be sufficient for optimal scheduling. Intuitively, if a high priority task tries to **rendezvous** with a low priority task, then the low priority task should be temporarily given the priority of the high priority task so that it can compete for the CPU with the rest of the tasks. To see if priority inheritance is sufficient, we now consider an analog of the priority inheritance protocol for the earliest deadline scheduler which we call the ED-PI (*Earliest Deadline- Priority Inheritance*) scheduler.

The ED-PI scheduler works in the following way. In addition to running at every instant the ready task with the nearest deadline, the ED-PI scheduler also assigns temporary deadlines as follows: If task T_1 executes a **rendezvous** command to try to synchronize with task T_2, then the deadlines of both T_1 and T_2 will be set to the smaller of their two deadlines. After the **rendezvous** has completed, the deadlines of the two tasks will be restored to their previous values.

Unfortunately, the ED-PI scheduler is not optimal, as the example in figure 1 shows. (The bottom schedule in figure 1 illustrates how the ED-PI scheduler fails.) The ED-PI scheduler fails because it does not make use of the information that task 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.

Fortunately, the ED-PI scheduler does work under fairly non-restrictive conditions. Specifically, we can prove the following theorem.

Theorem 3

If (1) tasks that synchronize with one another have the same period, and (2) the period of every task is the same as its deadline, then a necessary and sufficient condition

for scheduling a task set with synchronization constraints is that the utilization factor of the task set does not exceed 1.

If the conditions of theorem 3 are satisfied, then the ED-PI scheduler is indeed optimal. Again, a dynamic priority scheduler can be used to maintain a 100% achievable utilization factor, thus giving further credence to the need to support dynamic priority scheduling in a real-time programming language such as Ada.

3.5 Critical Sections

In our formulation, an instance of a task model is given by a pair (M, S). M is a set of periodic tasks and the computation of a periodic task is a chain of scheduling blocks separated by communication primitives (**rendezvous** commands). We say that two scheduling blocks in different tasks are mutually exclusive if their executions are not allowed to overlap. We shall call such scheduling blocks critical sections. The set S in our model is a set of semaphore tasks that enforce mutual exclusion on critical sections. Before executing a critical section, a periodic task must execute a **rendezvous** command with some task in S. Upon leaving the critical section, the periodic task must again **rendezvous** with the same task in S. The two **rendezvous** correspond to a P and V action on a semaphore. We call S the guard of the critical section. In general, a semaphore task S may be the guard of more than one critical section. For scheduling purposes, the execution time of a **rendezvous** with a task in S can be considered to be 0. This can be justified by charging the **rendezvous** overhead to the computation time of the critical section.

3.5.1 The Impact of Critical Sections on Scheduling

The need to share critical sections among tasks is another source for the priority inversion problem. Suppose a low priority task succeeds in entering a critical section before a high priority task becomes active, and the high priority task also wants to execute the same critical section. Then the high priority task will have to wait until the low priority task has exited from the critical section. The priority inheritance protocols proposed in [Sha, Rajkumar & Lehoczky 87] can alleviate the problem, but again it is unlikely that the achievable utilization factor can be maintained at 70% for static priority schedulers.

It is not difficult to see that with arbitrarily long critical sections, the earliest deadline algorithm is no longer optimal. In fact, we can prove the following theorem.

Theorem 4[‡]

The problem of deciding whether an instance of the tasking model (M, S) has a feasible schedule or not is NP-hard even for the case where the deadline of each task is

the same as its period.

In general, the achievable utilization factor can be arbitrarily low even for dynamic priority schedulers if there is no restriction on the size of critical sections. To see this, we can add to any task set an additional task whose computation time is longer than any of the deadlines in the task set and whose period is so long that its utilization factor is negligible. If this new task is mutually exclusive with all the other tasks, then the task set will be infeasible regardless of its utilization factor. In practice, the size of critical sections are usually kept small. If we restrict the lengths of critical sections to be no bigger than a certain size, say q, then the run-time system can enforce mutual exclusion on critical sections by not permitting a task to be preempted unless it has received at least q units of CPU time. This is indeed a common strategy in operating system kernels for protecting system state information, and is compatible with the technique of using non-preemptible "code strips" for real-time programming (the TOMAL language [Hennessy 77]). However, the earliest deadline scheduler is still not optimal even if all critical sections are of the same length, as shown by the following example.

Example

There are two periodic tasks 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 q 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 CPU time allocated to it at time 4. A cleverer scheduler would have left the CPU *idle* in the interval [4,5] and execute T_{22} in the interval [7,9]. The earliest deadline scheduler fails because it never leaves the CPU idle when there is a task ready to run, whereas in this case an optimal scheduler must not allocate a new quantum of CPU time to any task after time=3 and before time=5 so that a future deadline may be met. Figure 2 illustrates the situation when the second instance of T_1 misses its deadline because T_{22} is started in the interval (3,5).

‡ In [Mok 83], we proved that the problem of deciding whether it is possible to schedule a set of periodic tasks which use semaphores only to enforce mutual exclusion is NP-hard. However, the construction in that proof requires tasks whose deadlines are not the same as their periods. Theorem 4 is a stronger result.

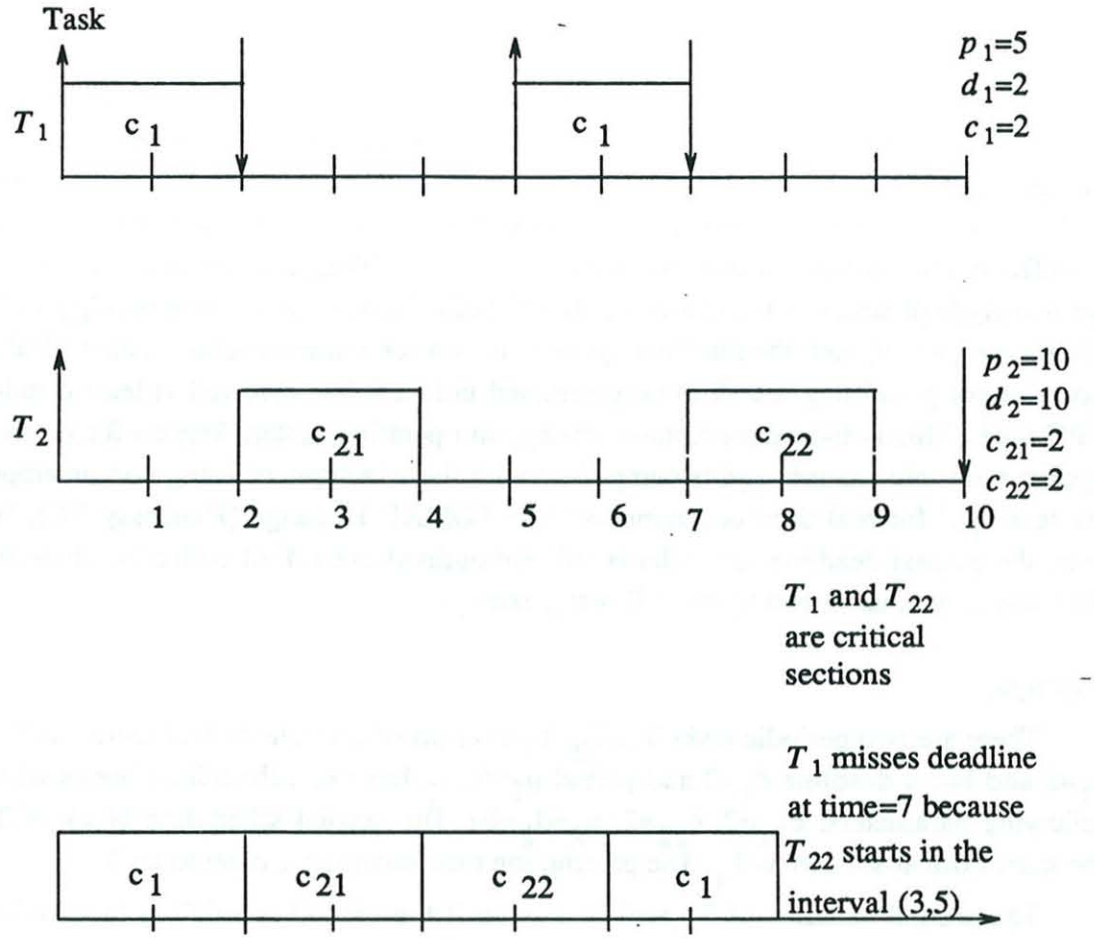

Figure 2. Example of scheduling constraint imposed by critical sections

Notice that in the above example, the more urgent task T_1 misses its deadline because it is blocked by a less urgent task T_2 which has entered a critical section before T_1 becomes ready to run again. This illustrates the analog of the priority inversion problem for the earliest deadline scheduler even under the restriction that the size of all critical sections is bounded by a constant. Unlike task synchronization, the achievable utilization factor when tasks have critical sections is no longer 100%, even if all critical sections have the same size. However, we can put a lower bound on the achievable utilization factor as a function of the parameter q, the upper bound on the size of critical sections as follows. We define the augmented utilization factor of a task T_i by $(c_i+q)/p_i$ where c_i, p_i are respectively the computation time and period of task T_i. The augmented utilization factor of a task set is the sum of the augmented utilization factors of the tasks in it.

Theorem 5

A task set is feasible if the following conditions are satisified: (1) The period of every task is the same as its deadline and is at least as long as its computation time plus q. (i.e., $c_i+q \leq p_i = d_i$ for every task T_i). (2) All scheduling blocks that are critical sections have size less than q. (3) The augmented utilization factor of the task set does not exceed 1.

The earliest deadline scheduler can be modified to generate a feasible schedule for a task set which satisfies the above conditions as follows. At any instant, the scheduler runs the task with the nearest deadline unless the task being executed is in a critical section, in which case no preemption is allowed until the critical section is exited. We note that theorem 5 still holds if tasks also synchronize with one another in addition to executing critical sections, as long as tasks that synchronize with one another have the same period. The ED-PI scheduler can likewise be modified to enforce mutual exclusion on critical sections.

Theorem 5 gives a technical justification for keeping critical sections small. Obviously, a high achieveable utilization factor can be obtained if the value of $q*\Sigma(1/p_i)$ is small.

3.6 Efficacy of Ada-like Tasking in Real-Time Programming

In order to meet stringent timing constraints, it is important to be able to exploit the semantics of the programming language to make good resource allocation decisions. For the class of timing constraints captured by our tasking model, we can now draw some conclusions about the efficacy of the Ada tasking facilities.

Firstly, as Cornhill and Sha have observed, the priority inversion problem can be detrimental to the achievable utilization factor whenever tasks need to synchronize or share critical sections. Our investigation indicates that a dynamic priority scheduler can be very useful in maintaining a high achievable utilization factor even with task synchronization and critical sections. Even though implementation overhead tradeoffs may call for schedulers that do not change task priorities as frequently as for example, the earliest deadline algorithm, there is a real need to support dynamic priority scheduling for real-time programming. Currently, dynamic priority schedulers are not supported by Ada.

Secondly, there is substantial benefit in making the enforcement of task synchronization and mutual exclusion distinct to the run-time scheduler since this piece of information is crucial to efficient scheduling. In the case of task synchronization, the online scheduler should allow the task with the more distant deadline to inherit the shorter deadline of the other task until synchronization is achieved. In the case of mutual exclusion, the scheduler can disallow preemption for q time units once a task has entered a critical section. In Ada, the same construct (the **rendezvous**) is being used for both purposes. As we have seen in the section 2, it is very difficult to derive the timing behavior of an Ada

program from its text. More importantly, it is in general impossible to automate the deduction of timing constraints from Ada programs. **Without knowledge of the type of timing constraints involved, it is impossible for an analysis tool to determine whether timing constraints can be guaranteed or not.** In contrast, this type of information can be easily obtained from annotated programs. Our scheduling results thus reinforce the utility of the formal annotation system that we have introduced in this regard.

Thirdly, when two or more tasks attempt to **rendezvous** with a semaphore task, a choice must be made to select one of the tasks for completion of the **rendezvous**. In Ada, this is determined by the implementation of the **select** statement and the FIFO discipline is usually adopted. Clearly, the FIFO discipline is non-optimal when stringent timing constraints must be considered. In general, nondeterministic constructs such as the **select** statement 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 application constraints. Our formal annotation again provides such a mechanism.

4. Conclusion

Programming language designers have traditionally abstracted away the notion of real time from high level languages, common wisdom being that programs are more robust if their correctness does not depend on execution speed. In real-time programs, however, the absolute timing of events may be crucial to the safe functioning of the system because of performance requirements, and because there are task coordination problems whose solutions depend on the satisfaction of stringent timing constraints. Herein lies the challenge: how do we design programming languages that allow us to reason about and enforce real time properties without tying the language to specific systems details? The approach taken by current real-time programming languages such as Ada is to add constructs, e.g., the **delay** command of Ada to explicitly schedule computation. We have shown that there are two serious problems with this approach. Firstly, the semantics of *ad hoc* time-related commands and concurrency control mechanisms may not be sufficient to express the wide spectrum of timing constraints. Secondly, it may not be possible to derive the necessary information about timing constraints for efficient scheduling from the concurrency control constructs in the language.

To help resolve the first difficulty, we have presented a formal system of annotating Ada-like programs to express timing constraints. This system of annotation is independent of language-specific concurrency control mechanisms and can be applied to block-

structured languages. Since it is based on a logic (RTL), it does not suffer from the imprecise semantics of the time-related constructs in current real-time programming languages. To help resolve the second difficulty, we have examined the real-time scheduling problem by augmenting the tasking model with timing constraints. We have investigated the theoretical efficiency of dynamic priority schedulers in solving the priority inversion problem, both for the case of task synchronization and critical sections. Our results indicate that there is technical justification for making task synchronization and mutual exclusion syntactically distinguishable since this piece of information is crucial to the construction of efficient real-time schedulers.

There are many other important issues in real-time systems research. For example, how do we recover from faults which violate design assumptions in the hard-real-time approach? This is often a source of misunderstanding of what real-time system design is all about. In the hard-real-time approach, we strive to design systems which are *guaranteed* to meet certain timing constraints. This does not mean that systems built to such specifications will not fail. There are fallible assumptions underlying the design of hard-real-time systems, e.g., underestimation of resource requirements, worst-case workload etc. However, the hard-real-time approach gives us a way to separate concerns. We can now estimate the reliability of a system by considering how likely the assumptions are to fail (hypothesis coverage analysis), independent of resource allocation details. Furthermore, a resource scheduler designed to meet a given set of hard-real-time constraints is likely to be more amenable to detecting when a design assumption has failed, e.g., a task is using up more CPU time than is specified. Consequently, it may be easier to adjust resource allocation policies to avoid catastrophic failure. Indeed, it may not be too farfetched to think of the hard-real-time approach as the closed-loop control approach (as opposed to the open-looped traditional approach) to achieve system reliability.

Bibliography

[Ada Manual 83]
United States Department of Defense (1983) "Reference Manual for the Ada Programming Language", *ANSI/MIL-STD-1815A, 1983.*

[Blazewicz 76]
Blazewicz, J. (1976) "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.*

[Cornhill & Sha 87]
Cornhill, D. and Sha L. (1987) "Priority Inversion in Ada", *Ada Letters, vol. 7, no. 7, November-December, 1987.*

[Donner 87]
Donner, M. (1987) "Language and Operating System Integration for Real-Time Systems", *Proceedings of the Fourth Workshop on Real-Time Operating Systems, Cambridge, MA., July 1987*

[Hennessy 77]
Hennessy, J. L. (1977) "A Real-Time Language for Small Processors: Design, Definition, and Implementation", *PhD thesis, State University of New York at Stony Brook, 1975.*

[Jahanian & Mok 86]
Jahanian F. and Mok, A. K. (1985) "Safety Analysis of Timing Properties in Real-Time Systems", *IEEE Transactions on Software Engineering, Vol. SE-12, No.9, September 1986.*

[Lee & Gehlot 85]
Lee, I. and Gehlot, V (1985) "Language Constructs for Distributed Real-Time Programming", *Proceedings of the IEEE Real Time Systems Symposium, December 1985, San Diego, CA, pp. 57-66.*

[Leung & Merrill 80]
Leung, Joseph Y. T. and Merrill, M. L. (1980) "A Note on Preemptive Scheduling of Periodic, Real-time Tasks" *Information Processing Letters* 11 115-118.

[Lin, Natarajan & Liu 87]
Lin, K. J., Natarajan, S. and Liu, J. (1987) "Imprecise Results: Utilizing Partial Computations in Real-Time Systems", *Proceedings of the IEEE Real Time Systems Symposium, December 1987, San Jose, CA, pp. 210-217.*

[Liu & Layland 73]
Liu, C. L. and Layland, James W. (1793) "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment" *JACM 20, January 1973, 46-61.*

[Lo 87]
Lo, Virginia (1987) "Distributed Scheduling Calendars for Scheduling under Real-Time and Synchronization Constraints", *Proceedings of the Fourth Workshop on Real-Time Operating Systems, Cambridge, MA., July 1987*

[Mok 83]
Mok, Aloysius K. (1983) "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment" *Ph.D Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts.*

[Mok 85]
Mok, A. K. (1985) "SARTOR-a Design Environment for Real-Time Systems", *IEEE Proceedings of the 9th COMPSAC Conference, October, 1985, Chicago, Illinois, pp. 174-181.*

[Mok et al 87]
Mok, A. K., Amerasinghe, P., Chen, M., Sutanthavibul, S., Tantisirivat, K. (1987) "Synthesis of a Real-Time Message Processing System with Data-Driven Timing Constraints", *Proceedings of the IEEE Real-Time Systems Symposium, December 1987, San Jose, California, pp. 133-143.*

[Sha, Rajkumar & Lehoczky 87]
Sha, L, Rajkumar, R. and Lehoczky, J. (1987) "Priority Inheritance Protocols: an Approach to Real-Time Synchronization", *unpublished manuscript, Departments of CS, ECE and Statistics, Carnegie Mellon University, November, 1987.*

[Shaw 87]
Shaw, A. (1987) "Reasoning About Time in Higher-Level Language Software", *Department of Computer Science Technical Report 87-08-05 University of Washington, Seattle, WA 98195, August 1987*

[Volz & Mudge 87]
Volz, R. A. and Mudge, T. N. (1987) "Instruction Level Timing Mechanisms for Accurate Real-Time Task Scheduling", *IEEE Transactions on Computers, vol. C-36, no. 4, pp. 449-459, April 1987.*

DISCUSSION

First Lecture

Rapporteur: Rogério de Lemos
Amer Saeed

In one of the first examples presented which concerned two concurrent systems A and B, members of the audience questioned how the system A could detect a property of system B. Professor Mok answered that this is an implementation issue not a specification issue. Professor Turski asked if all considerations must be taken at the implementation level what was meant by "at the same time" at the specification level. Professor Mok answered that he could explained "at the same time" in terms of logic.

Another example presented, made up of a natural language specification and the respective formal specification in terms of RTL, members of the audience questioned some ambiguities found in the english specification. Professor Mok argued that an english description does not give a precise specification to which Professor Turski disagrees and said that the given specification was very precise. At this point Professor Shrivastava noted that the natural language specification was precise enough in the sense that a system could be built which could satisfy the specification.

Professor Turski questioned the notation of the operator "@". Professor Mok answered that the operator "@" considers absolute time and maps the occurrence of events to the integers (discrete time).

Professor Turski argued if the verification used ($SP \wedge \neg SA$) checks if the safety assertion is satisfiable or provable. Professor Mok replied that was provable if not satisfiable in the context of the model.

After the lecture Professor Nehmer asked how will the annotation of a program help in the analysis of the implementation in languages like ADA which have many nondeterministic properties; in his opinion languages should be modified to remove all nondeterminism. Professor Mok replied that the nondeterminism gives an important degree of flexibility which allows the scheduler to adapt to certain levels of performance, and instead of proposing a new language the choice was to use an annotation system, based on the claim that an engineer should not have to learn an entirely new language to benefit from the presented approach.

Second Lecture

Rapporteur: Amer Saeed

During the lecture

Professor Randell asked if the reduction process over the graphical representation preserves the cycles of the graph. Professor Mok replied that the reduction process preserves the number of positive cycles, and these are the only cycles he needs to determine satisfiability.

Professor Bron asked what conclusions can be declared after node reductions, in particular can you determine satisfiability or not. Professor Mok replied that

if all edges in a positive cycle (when reduced) correspond to unit clauses of the safety assertion then the assertion must be unsatisfiable. However, if some edges correspond to disjunctive clauses then further analysis will be required.

Professor Kopetz asked to what extent is analysis automated, and how practical is the approach. Professor Mok stated that a program which implements some of the algorithms has been constructed, and he hopes that in the future an automated tool to aid the analysis will be available. Professor Mok also said that at the moment he does not know how practical the approach is for timing analysis, but was aware of the fact that timing analysis is very hard.

At end of lecture

Professor Randell asked what part does he envisage for the approach in the construction of real-time systems. Professor Mok replied that two main objectives must be met before the approach could become engineering practice, these where the availability of tools to automate much of the analysis and a means to structure the system to minimise the components over which real-time constraints are imposed.

Professor Randell asked if their where any equivalent techniques or approaches which could be used to express and analyse the safety assertions. Professor Mok replied by stating that there were some alternative approaches, perhaps the most well known was temporal logic. Professor Mok then stated that several problems are encountered when temporal logic is used to express real-time constraints, in particular care must be taken on how time is modeled otherwise the usefulness of the logic can be limited. Finally, Professor Mok stated that he did not claim that his approach was the best, but that some approach which explicitly states the timing constraints is required.

Mr A. Waterworth stated that if the verification of the safety assertion can be exponential, how useful is the approach for very large systems. Professor Mok replied that the analysis of a complicated system would always be a difficult task. But at the moment he did not know how practical the approach would be for very large systems, the practicability of the approach could only be assessed after detailed experiments - which have not yet been performed. Professor Mok also stated that he felt that the verification problem could be simplified if the approach was used in conjunction with a structuring technique.

Third Lecture

Rapporteur: Amer Saeed

During the lecture

Professor Randell stated that if a system is constructed with enough processors to ensure that the system is heavily under loaded (as suggested by Professor Bron) then, surely, the importance of scheduling is reduced. Professor Mok agreed that if enough processors are available to ensure that a system will be heavily under loaded the problem of scheduling is no longer critical, and conceded the point that there may exist many (industrial) systems in which this is possible. But he also stressed that there are many (avionics and military) systems where it will not be possible, and it was these he was primarily concerned with.

Dr Holt pointed out to Professor Mok that in the implementation of any scheduler the action of swapping processes will take some finite time, and asked how this would effect his scheduling strategy. Professor Mok stated that if for periodic processes an upper bound can be placed on the number of preemptions then an upper bound can be placed on the total time required to swap the processes. If such an upper bound can be computed it can be easily encoporated into the scheduler.

At end of lecture

Professor Bron stated that he felt that the algorithm used for the earliest deadline scheduler was an application of the bankers algorithm in which money is replaced by time. Professor Mok conceded the point that there are certain similarities with the bankers algorithm (and others), but – as far as he was aware– it was not the same. However he did not claim that the algorithm was entirely original.

Professor Randell asked to what extent are the scheduling problems in computing science comparable to those in operations research, and how much use can computer scientists make of the work done in operations research. Professor Mok replied that there is a definite overlap between the scheduling issues in operations research and computing science scheduling issues. But operations research usually deals with factory level scheduling, in which there are fixed tasks and no (or minimal) mutual exclusion problems. Professor Mok also stated that in a recent and supposedly complete classification of scheduling problems (in operations research), many of the problems encountered in computing science were not covered.