Real-Time Scheduling Theory and Ada*

Lui Sha and John B. Goodenough
Software Engineering Institute, Carnegie Mellon University

In real-time applications, the correctness of computation depends on not only the results of computation but also the time at which outputs are generated. Examples of real-time applications include air traffic control, avionics, process control, and mission-critical computations.

The measures of merit in a real-time system include

- Predictably fast response to urgent events.
- High degree of schedulability. Schedulability is the degree of resource utilization at or below which the timing requirements of tasks can be ensured. You can think of it as a measure of the number of timely transactions per second.
- Stability under transient overload. When the system is overloaded by events and meeting all deadlines is impossible, we must still guarantee the deadlines of selected critical tasks.

Traditionally, many real-time systems use cyclical executives to schedule concurrent threads of execution. Under this approach, a programmer lays out an execution timeline by hand to serialize the execution of critical sections and to meet task deadlines.

While such an approach is manageable for simple systems, it quickly becomes unmanageable for large systems. It is a painful process to develop application code so that it fits the time slots of a cyclical executive while ensuring that the critical sections of different tasks do not interleave. Forcing programmers to schedule tasks by fitting code segments on a timeline is no better than the outdated approach of managing memory by manual memory overlay.

Rate monotonic scheduling theory puts real-time software engineering on a sound analytical footing. Here, we review the theory and its implications for Ada.

Under the cyclical executive approach, meeting the responsiveness, schedulability, and stability requirements has become such a difficult job that practitioners often sacrifice program structure to fit the code into the "right" time slots. This results in real-time programs that are difficult to understand and maintain.

The Ada tasking model represents a fundamental departure from the cyclical executive model. To reduce the complexity of developing a concurrent system, Ada tasking allows software engineers to manage concurrency at an abstract level divorced from the details of task executions. Indeed, the dynamic preemption of tasks at runtime generates nondeterministic timelines at odds with the very idea of a fixed execution timeline.

This nondeterminism seems to make it impossible to decide whether real-time deadlines will be met. However, Ada's tasking concepts are well-suited to the rate monotonic theory being considered in our project.

In essence, this theory ensures that as long as the CPU utilization of all tasks lies below a certain bound and appropriate scheduling algorithms are used, all tasks will meet their deadlines without the programmer knowing exactly when any given task will be running. Even if a transient

* This article is a modified version of a technical report, CMU/SEI-89-TR-14, sponsored by the US Dept. of Defense and bearing the same title.

0018-9162/90/0400-0053$01.00 © 1990 IEEE

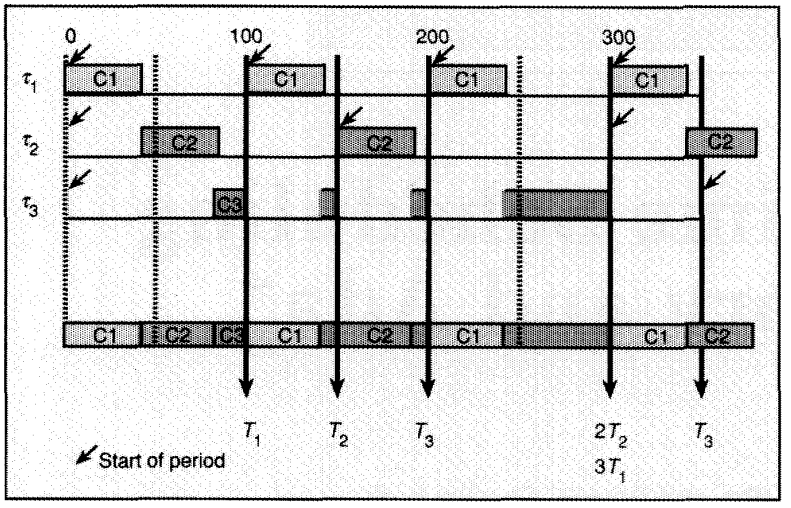

Figure 1. Application of the critical zone theorem to Task τ_3.

overload occurs, a fixed subset of critical tasks will still meet their deadlines as long as their CPU utilizations lie within the appropriate bound.

In short, the scheduling theory allows software engineers to reason about timing correctness at the same abstract level used by the Ada tasking model. Applying this theory to Ada makes Ada tasking truly useful for real-time applications while also putting the development and maintenance of real-time systems on an analytic, engineering basis, making these systems easier to develop and maintain.

The next section reviews some of the basic results in the rate monotonic scheduling theory, although the theory also deals with mode changes and multiprocessing.[1,2] In the final section, we review the Ada tasking model and scheduling policies, and suggest some workarounds that permit us to implement rate monotonic scheduling algorithms within the framework of existing Ada rules.

Scheduling real-time tasks

This section contains an overview of some of the important issues of real-time scheduling theory, beginning with the problem of ensuring that independent periodic tasks meet their deadlines. Then, we show how to ensure that critical tasks will meet their deadlines even when a system is temporarily overloaded, address the problem of scheduling both periodic and aperiodic tasks, and review real-time synchronization and communication issues.

Periodic tasks. Tasks are independent if their executions need not be synchronized. Given a set of independent periodic tasks, the rate monotonic scheduling algorithm gives each task a fixed priority and assigns higher priorities to tasks with shorter periods. A task set is said to be *schedulable* if all its deadlines are met, that is, if every periodic task finishes its execution before the end of its period. Any set of independent periodic tasks is schedulable by the rate monotonic algorithm if the condition of Theorem 1 is met.[3]

> *Theorem 1*: A set of n independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if
>
> $$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) = U(n)$$
>
> where C_i and T_i are the execution time and period of task τ_i, respectively.

Theorem 1 offers a sufficient (worst-case) condition that characterizes schedulability of a task set under the rate monotonic algorithm. This bound converges to 69 percent (ln 2) as the number of tasks approaches infinity. The values of the scheduling bounds for one to nine independent tasks are as follows: U(1) = 1.0, U(2) = 0.828, U(3) = 0.779, U(4) = 0.756, U(5) = 0.743, U(6) = 0.734, U(7) = 0.728, U(8) = 0.724, and U(9) = 0.720.

Example 1: Consider the case of three periodic tasks, where $U_i = C_i/T_i$.

- Task τ_1: $C_1 = 20$; $T_1 = 100$; $U_1 = 0.2$
- Task τ_2: $C_2 = 40$; $T_2 = 150$; $U_2 = 0.267$
- Task τ_3: $C_3 = 100$; $T_3 = 350$; $U_3 = 0.286$

The total utilization of these three tasks is 0.753, which is below Theorem 1's bound for three tasks: $3(2^{1/3} - 1) = 0.779$. Hence, we know these three tasks are schedulable, that is, they will meet their deadlines if τ_1 is given the highest priority, τ_2 the next highest, and τ_3 the lowest.

The bound of Theorem 1 is very pessimistic because the worst-case task set is contrived and unlikely to be encountered in practice. For a randomly chosen task set, the likely bound is 88 percent.[4] To know if a set of given tasks with utilization greater than the bound of Theorem 1 can meet its deadlines, we can use an exact schedulability test based on the *critical zone* theorem (rephrased from Liu and Layland[3]):

> *Theorem 2*: For a set of independent periodic tasks, if each task meets its first deadline when all tasks are started at the same time, then the deadlines will always be met for any combination of start times.

This theorem provides the basis for an exact schedulability test for sets of independent periodic tasks under the rate monotonic algorithm. In effect, the theorem requires that we check to see if each task can complete its execution before its first deadline. To do so, we need to check the *scheduling points* for a task. The scheduling points for task τ are τ's first deadline and the ends of periods of higher priority tasks prior to τ's first deadline. The process is illustrated by Figure 1, which applies to the task set in the next example.

Example 2: Suppose we replace τ_1's algorithm in Example 1 with one that is more accurate and computationally intensive. Suppose the new algorithm doubles τ_1's computation time from 20 to 40, so the total processor utilization increases from 0.753 to 0.953. Since the utilization of the first two tasks is 0.667, which is below Theorem 1's bound for two tasks, $2(2^{1/2} - 1) = 0.828$, the first two tasks cannot miss their deadlines. For task τ_3, we use Theorem 2 to check whether the task set is schedulable. Figure 1 shows the scheduling points relevant to task τ_3, that is, the ends of periods of higher priority tasks for

times less than or equal to τ_3's period. To check against the critical zone theorem, we check whether all tasks have completed their execution at any of the scheduling points. For example, to see if all tasks have met their deadlines at time 350, we can see that task τ_1 will have had to start executing four times; task τ_2, three times; and task τ_3, once. So we check to see if all these executions can complete in 350 milliseconds:

$$4C_1 + 3C_2 + C_3 \le T_3$$
$$160 + 120 + 100 > 350$$

This test fails, but that doesn't mean the task set can't meet its deadlines; the theorem requires that all the scheduling points be checked. If all tasks meet their deadlines for at least one scheduling point, the task set is schedulable. The equations to be checked are shown below, for each scheduling point:

$$C_1 + C_2 + C_3 \le T_1$$
$$40 + 40 + 100 > 100$$

$$2C_1 + C_2 + C_3 \le T_2$$
$$80 + 40 + 100 > 150$$

$$2C_1 + 2C_2 + C_3 \le 2T_1$$
$$80 + 80 + 100 > 200$$

$$3C_1 + 2C_2 + C_3 \le 2T_3$$
$$120 + 80 + 100 = 300$$

$$4C_1 + 3C_2 + C_3 \le T_3$$
$$160 + 120 + 100 > 350$$

The equation for scheduling point $2T_2$ is satisfied. That is, after 300 units of time, τ_1 will have run three times, τ_2 will have run twice, and τ_3 will have run once. The required amount of computation just fits within the allowed time, so each task meets its deadline. Liu and Layland[3] showed that since the tasks meet their deadlines at least once within the period T_3, they will always meet their deadlines. Hence, we can double the utilization of the first task in Example 1 from 20 percent to 40 percent and still meet all the deadlines. Since task τ_3 just meets its deadline at time 300, we cannot add any tasks with a priority higher than that of task τ_3, although a task of lower priority could be added if its period is sufficiently long.

The checking required by Theorem 2 can be represented by an equivalent mathematical test[4]:

Theorem 3: A set of n independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasings, if and only if

$$\forall i,\ 1 \le i \le n,$$

$$\min_{(k,l)\ \varepsilon\ R_i} \sum_{j=1}^{i} C_j \frac{1}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil \le 1$$

where C_j and T_j are the execution time and period of task τ_j, respectively, and

$$R_i = \{(k,l) \mid 1 \le k \le i, l = 1, \ldots, \lfloor T_i / T_k \rfloor\}.$$

A major advantage of using the rate monotonic algorithm is that it allows us to follow the software engineering principle of separation of concerns. In this case, the theory allows systems to separate concerns for logical correctness from concerns for timing correctness.

Suppose a cyclical executive is used for this example. The major cycle must be the least common multiple of the task periods. In this example, the task periods are in the ratio 100:150:350 = 2:3:7. A minor cycle of 50 units would induce a major cycle of 42 minor cycles, which is an overly complex design.

To reduce the number of minor cycles, we can try to modify the periods. For example, it might be possible to reduce the period of the longest task, from 350 to 300. The total utilization is then exactly 100 percent, and the period ratios are 2:3:6; the major cycle can then be six minor cycles of 50 units.

To implement this approach and minimize the splitting of computations belonging to a single task, we could split task τ_1 into two parts of 20 units computation each. The computation of task τ_2 similarly could be split into at least two parts such that task τ_3 need only be split into four parts. A possible timeline indicating the amount of computation for each task in each minor cycle is shown in Table 1, where 20_1 on the first line indicates the first part of task τ_1's computation, which takes 20 units of time.

Table 1. Minor cycle timeline for Example 2. Each minor cycle is 50.

	1	2	3	4	5	6
τ_1	20_1	20_2	20_1	20_2	20_1	20_2
τ_2	30_1	10_2		30_1	10_2	
τ_3		20_1	30_2		20_3	30_4

When the processor utilization level is high and many tasks need to be performed, fitting code segments into time slots can be a time-consuming iterative process. In addition, a later modification of any task may overflow a particular minor cycle and require the entire timeline to be redone.

But more importantly, the cyclic executive approach has required us to modify the period of one of the tasks, increasing the utilization to 100 percent without in fact doing more useful work. Under the rate monotonic approach for this example, all deadlines are met, but total machine utilization must not exceed 95.3 percent instead of 100 percent.

This doesn't mean the rate monotonic approach is less efficient. The capacity not needed to service real-time tasks in the rate monotonic approach can be used by background tasks, for example, for built-in-test purposes. With the cyclic executive approach, no such additional work can be done in this example.

Control applications typically require that signals be sent and received at fixed intervals with minimal *jitter*, that is, with precise timing in sending or receiving data. Jitter requirements are a special kind of deadline requirement.

One way of meeting an output jitter requirement using rate monotonic scheduling is to have a normal periodic task compute a result and place it in a memory-mapped I/O buffer before it is time to send the value. Rate monotonic scheduling theorems can be used to ensure the value is computed in time. A hardware timer or the operating system clock interrupt handler routine can then initiate the device I/O precisely at the required time. A similar approach can be used for input jitter requirements.

Stability under transient overload. So far, we have assumed that the computation time of a task is constant. However, in many applications, task execution times

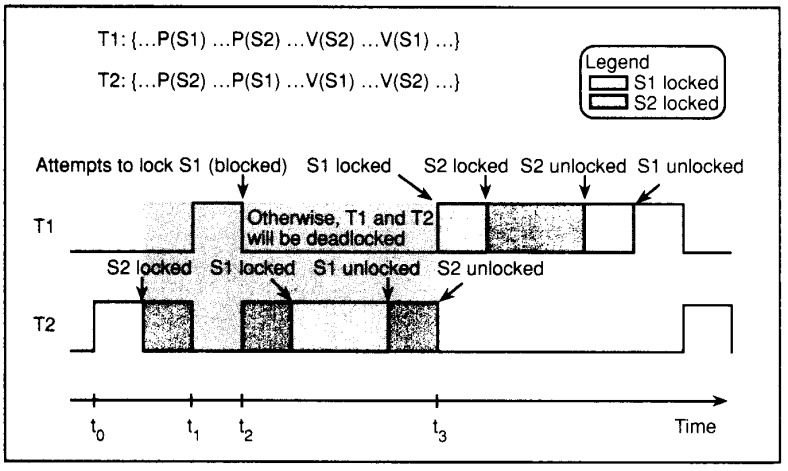

Figure 2. Example of deadlock prevention. The gray section shows the interval of time when the priority ceiling protocol prevents task T1 from locking semaphore S1 or semaphore S2.

are often stochastic, and the worst-case execution time can be significantly larger than the average execution time.

To have a reasonably high average processor utilization, we must deal with the problem of transient overload. We consider a scheduling algorithm to be *stable* if a set of tasks exists such that all tasks in the set will meet their deadlines even if the processor is overloaded.

This means that, under worst-case conditions, tasks outside the set may miss their deadlines. The rate monotonic algorithm is stable — the tasks guaranteed to meet their deadlines under worst-case execution times are always the n highest priority tasks. These tasks form the schedulable task set. Software designers, therefore, must be sure that all tasks whose deadlines must be met under overload conditions are in the schedulable set.

Since task priorities are assigned according to the period of each task, a critically important task might not be in the schedulable set. A task with a longer period could be more critical to an application than a task with a shorter period. You might try to ensure that such a task always meets its deadline simply by increasing its priority to reflect its importance. However, this approach makes it more likely that other tasks will miss their deadlines, that is, the schedulability bound is lowered.

The *period transformation* technique can be used to ensure high utilization while meeting the deadline of an important, long-period task. For example, suppose task τ with a long period T is not guaranteed to meet its deadline under transient overload conditions, but nonetheless, the work performed by the task is critical, and it must never miss its deadline.

We need to ensure that task τ belongs to the schedulable task set. Since the rate monotonic priority of a task is a function of its period, we can raise task τ's priority only by having it act like a task with a shorter period. We can do so by giving it a period of $T/2$ and suspending it after it executes half its worst-case execution time, $C/2$.

The task is then resumed and finishes its work in the next execution period. It still completes its total computation before the end of period T. From the viewpoint of the rate monotonic theory, the transformed task has the same utilization but a shorter period, $T/2$, and its priority is raised accordingly. Note that the most important task need not have the shortest period. We only need to make sure that it is in the schedulable set. A systematic procedure for period transformation with minimal task partitioning can be found in Sha, Lehoczky, and Rajkumar.[5]

Period transformation resembles the task slicing found in cyclic executives. The difference here is that we don't need to adjust the code segment sizes so different code segments fit into shared time slots. Instead, τ simply requests suspension after performing $C/2$ amount of work. Alternatively, the runtime scheduler can be instructed to suspend the task after $C/2$ work has been done, without affecting the application code.

Scheduling both aperiodic and periodic tasks. It is important to meet both the regular deadlines of periodic tasks and the response time requirements of aperiodic requests. Aperiodic servers are tasks used to process such requests. Here is a simple example.

Example 3: Suppose we have two tasks. Let τ_1 be a periodic task with period 100 and execution time 99. Let τ_2 be a server for an aperiodic request that randomly arrives once within a period of 100. Suppose one unit of time is required to service one request.

If we let the aperiodic server execute only in the background, that is, only after completing the periodic task, the average response time is about 50 units. The same can be said for a polling server that provides one unit of service time in a period of 100.

On the other hand, we can deposit one unit of service time in a "ticket box" every 100 units of time; when a new "ticket" is deposited, the unused old tickets, if any, are discarded. With this approach, no matter when the aperiodic request arrives during a period of 100, it will find a ticket for one unit of execution time at the ticket box. That is, τ_2 can use the ticket to preempt τ_1 and execute immediately when the request occurs. In this case, τ_2's response time is precisely one unit and the deadlines of τ_1 are still guaranteed.

This is the idea behind a class of aperiodic server algorithms[6] that can reduce aperiodic response time by a large factor (a factor of 50 in this example). We allow the aperiodic servers to preempt the periodic tasks for a limited duration allowed by the rate monotonic scheduling formula.

A good aperiodic server algorithm is known as the *sporadic* server.[7] Instead of refreshing the server's budget periodically, at fixed points in time, replenishment is determined by when requests are serviced. In the simplest approach, the budget is refreshed T units of time after it has been exhausted, but earlier refreshing is also possible.

A sporadic server is only allowed to preempt the execution of periodic tasks as long as its computation budget is not exhausted. When the budget is used up, the server can continue to execute at background priority if time is available. When the server's budget is refreshed, its execution can resume at the server's assigned priority.

There is no overhead if there are no

requests. Therefore, the sporadic server is especially suitable for handling emergency aperiodic events that occur rarely but must be serviced quickly.

From a schedulability point of view, a sporadic server is equivalent to a periodic task that performs polling. That is, we can place sporadic servers at various priority levels and use only Theorems 1 and 2 to perform a schedulability analysis. With this approach, both periodic and aperiodic task modules can be modified at will, as long as the rate monotonic utilization bound is observed.

When the average aperiodic workload is no more than 70 percent of the CPU capacity allocated for the sporadic server, randomly arriving requests are likely to find the server available and can successfully preempt the periodic tasks. This results in about 6 times faster response than polling and 10 times faster response than background service.[7]

When the aperiodic workload is about equal to the server's budget, the likelihood of server availability decreases and the resulting performance advantage also decreases. The performance of such an overloaded server is essentially that of a polling server.

Finally, sporadic servers can be used to service hard-deadline aperiodic requests. An example will be given in the section "An example of application of the theory."

Task synchronization. In previous sections, we discussed the scheduling of independent tasks. Tasks, however, do interact. In this section, we discuss how the rate monotonic scheduling theory can be applied to real-time tasks that must interact. The discussion is limited in this article to scheduling within a uniprocessor. Readers interested in the multiprocessor synchronization problem should see Rajkumar, Sha, and Lehoczky.[1]

Common synchronization primitives include semaphores, locks, monitors, and Ada rendezvous. Although the use of these or equivalent methods is necessary to ensure the consistency of shared data or to guarantee the proper use of nonpreemptable resources, their use may jeopardize the ability of the system to meet its timing requirements. In fact, a direct application of these synchronization mechanisms may lead to an indefinite period of *priority inversion*, which occurs when a high-priority task is prevented from executing by a low-priority task. Unbounded priority inversion can occur as shown in the following example:

Example 4: Suppose periodic tasks T1, T2, and T3 are arranged in descending order of priority (high, medium, and low). Suppose tasks T1 and T3 share a data structure guarded by semaphore S1. During the execution of the critical section of task T3, high-priority task T1 starts to execute, preempts T3, and later attempts to use the shared data. However, T1 is blocked on the semaphore S1.

We would prefer that T1, being the highest priority task, be blocked no longer than the time it takes for T3 to complete its critical section. However, the duration of blocking is, in fact, unpredictable. This is because T3 can be preempted by the medium priority task T2. T1 will be blocked, that is, prevented from executing, until T2 and any other pending tasks of intermediate priority are completed.

The problem of unbounded priority inversion can be partially remedied in this case if a task is not allowed to be preempted while executing a critical section. However, this solution is only appropriate for very short critical sections because it creates unnecessary priority inversion. For instance, once a low-priority job enters a long critical section, a high priority job that does not access the shared data structure may be needlessly blocked.

The *priority ceiling protocol* is a real-time synchronization protocol with two important properties:

(1) freedom from mutual deadlock and
(2) bounded priority inversion. Namely, at most one lower priority task can block a higher priority task.[8,9]

There are two ideas in the design of this protocol. First is the concept of priority inheritance: when a task τ blocks the execution of higher priority tasks, task τ executes at the highest priority level of all the tasks blocked by τ. Second, we must guarantee that a critical section is allowed to start execution only if the section will always execute at a priority level higher than the (inherited) priority levels of any preempted critical sections.

It was shown in Sha, Rajkumar, and Lehoczky[9] that such a prioritized total ordering in the execution of critical sections leads to the two desired properties. To achieve such a prioritized total ordering, we define the *priority ceiling* of a binary semaphore S to be the highest priority of all tasks that may lock S.

When a task τ attempts to execute one of its critical sections, it will be suspended unless its priority is higher than the priority ceilings of all semaphores currently locked by tasks other than τ. If task τ is unable to enter its critical section for this reason, the task that holds the lock on the semaphore with the highest priority ceiling is said to be blocking τ and, hence, inherits the priority of τ. As long as a task τ is not attempting to enter one of its critical sections, it will preempt every task that has a lower priority.

The priority ceiling protocol guarantees that a high-priority task will be blocked by at most one critical region of any lower priority task. Moreover, the protocol prevents mutual deadlock, as shown in the following example:

Example 5: Suppose we have two tasks T1 and T2 (see Figure 2). In addition, there are two shared data structures protected by binary semaphores S1 and S2, respectively. Suppose task T1 locks the semaphores in the order S1, S2, while T2 locks them in the reverse order. Further, assume that T1 has a higher priority than T2.

Since both T1 and T2 use semaphores S1 and S2, the priority ceilings of both semaphores are equal to the priority of task T1. Suppose that at time t_0, T2 begins execution and then locks semaphore S2. At time t_1, task T1 is initiated and preempts task T2, and at time t_2, task T1 tries to enter its critical section by attempting to lock semaphore S1. However, the priority of T1 is not higher than the priority ceiling of locked semaphore S2. Hence, task T1 must be suspended without locking S1. Task T2 now inherits the priority of task T1 and resumes execution. Note that T1 is blocked outside its critical section. Since T1 is not given the lock on S1 but instead is suspended, the potential deadlock involving T1 and T2 is prevented. At time t_3, T2 exits its critical section; it will return to its assigned priority and immediately be preempted by task T1. From this point on, T1 will execute to completion, and then T2 will resume its execution until its completion.

Let B_i be the longest duration of blocking that can be experienced by task τ_i. The following theorem determines whether the deadlines of a set of periodic tasks can be met if the priority ceiling protocol is used:

Theorem 4: A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate monotonic algorithm for all task phasings, if the

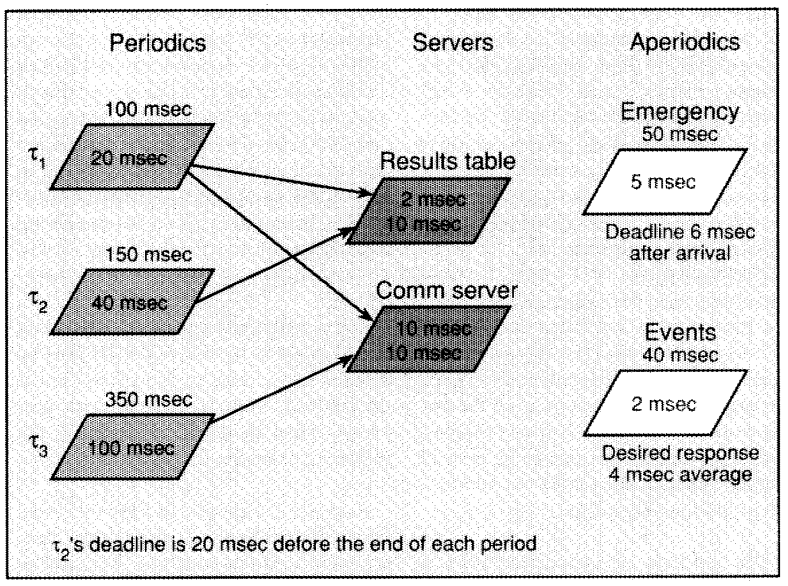

Figure 3. Task interactions for Example 6.

following condition is satisfied[9]:

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} + max\left(\frac{B_1}{T_1}, \ldots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n(2^{1/n} - 1)$$

This theorem generalizes Theorem 1 by taking blocking into consideration. A similar extension can be given for Theorem 3. The B_i's in Theorem 4 can be used to account for any delay caused by resource sharing. Note that B_n is always zero, since the lowest priority task cannot, by definition, be blocked by a task of lower priority, and hence, B_n is not included in the theorem's formula.

In the priority ceiling protocol, a task can be blocked even if it does not lock any semaphores. For example, in Figure 2, any task, τ, with a priority between that of T1 and T2 could be prevented from executing while T2 has semaphore S2 locked. This can happen because S2's priority ceiling is greater than the priority of τ. Therefore, if T1 tries to lock S1 while T2 has locked S2, T2 will inherit T1's priority and will prevent any tasks of intermediate priority from executing. Since the low priority task T2 prevents the execution of the intermediate priority tasks, these tasks suffer priority inversion, or blocking. This source of blocking is the cost of avoiding unbounded priority inversion. It must be taken into account when applying Theorem 4.

An example application of the theory. A simple example illustrates how to apply the scheduling theory.

Example 6: Consider the following task set (see Figure 3):

(1) Emergency handling task: execution time = 5 milliseconds; worst case interarrival time = 50 milliseconds; deadline is 6 milliseconds after arrival.

(2) Aperiodic event handling tasks: average execution time = 2 milliseconds; average interarrival time = 40 milliseconds; fast response time is desirable, but there are no hard deadlines.

(3) Periodic task τ_1: worst-case execution time = 20 milliseconds (which includes service time for accessing a shared data object and a shared communication server); period = 100 milliseconds; deadline is at the end of each period. In addition, τ_3 may block τ_1 for 10 milliseconds by using a shared communication server, and task τ_2 may block τ_1 for 20 milliseconds by using a shared data object.

(4) Periodic task τ_2: execution time = 40 milliseconds (including 20 milliseconds spent accessing the shared data object); period = 150 milliseconds; deadline is 20 milliseconds before the end of each period.

(5) Periodic task τ_3: execution time = 100 milliseconds (including 10 milliseconds for accessing the communications server); period = 350 milliseconds; deadline is at the end of each period.

Solution: First, we create a sporadic server for the emergency task, with a period of 50 milliseconds and a service time of 5 milliseconds. Since the server has the shortest period, the rate monotonic algorithm will give this server the highest priority. It follows that the emergency task can meet its deadline.

Since the aperiodic event handling activities have no deadlines, they can be assigned a low priority. However, since fast response time is desirable, we create a sporadic server executing at the second highest priority. The size of the server is a design issue. A larger server (that is, a server with higher utilization) needs more processor cycles but will give better response time. In this example, we choose a large server with a period of 100 milliseconds and a service time of 10 milliseconds. We now have two tasks with a period of 100 milliseconds — the aperiodic event server and periodic task τ_1. The rate monotonic algorithm allows us to break the tie arbitrarily, and we let the server have the higher priority.

We now have to check if the three periodic tasks can meet their deadlines. Since, under the priority ceiling protocol, a task can be blocked at most once by lower priority tasks, the maximal blocking time for task τ_1 is B_1 = maximum (10 milliseconds, 20 milliseconds) = 20 milliseconds. Since τ_3 may lock the semaphore S_c associated with the communication server and the priority ceiling of S_c is higher than that of task τ_2, task τ_2 can be blocked by task τ_3 for 10 milliseconds. (This may occur if τ_3 blocks τ_1 and inherits the priority of τ_1.) Finally, task τ_2 has to finish 20 milliseconds earlier than the nominal deadline for a periodic task. This is equivalent to saying that τ_2 will always be blocked for an additional 20 milliseconds, but its deadline is at the end of the period. Hence, $B_2 = (10 + 20)$ milliseconds = 30 milliseconds.* At this point, we can directly apply the appropriate theorems. However, we can also reduce the number of steps in the analysis by noting that period 50 and 100 are harmonics, so we can treat the emergency server

* Note that the blocked-at-most-once result does not apply here. It only applies to blocking caused by task synchronization using the priority ceiling protocol.

mathematically as if it had a period of 100 milliseconds and a service time of 10 milliseconds. We now have three tasks with a period of 100 milliseconds and an execution time of 20 milliseconds, 10 milliseconds, and 10 milliseconds, respectively. For the purpose of analysis, these three tasks can be replaced by a single periodic task with a period of 100 milliseconds and an execution time of 40 milliseconds (20 + 10 + 10). We now have the following three equivalent periodic tasks for analysis:

- Task τ_1: $C_1 = 40$; $T_1 = 100$; $B_1 = 20$; $U_1 = 0.4$
- Task τ_2: $C_2 = 40$; $T_2 = 150$; $B_2 = 30$; $U_2 = 0.267$
- Task τ_3: $C_3 = 100$; $T_3 = 350$; $B_3 = 0$; $U_3 = 0.286$

Note that B_3 is zero, since a task can only be blocked by tasks of lower priority. Since τ_3 is the lowest priority task, it can't be blocked.

When Theorem 3 is extended to account for blocking, we simply check to be sure that each task will meet its deadline if it is blocked for the maximal amount of time:

(1) Task τ_1: Check $C_1 + B_1 \leq T_1$. If this inequality is satisfied, τ_1 will always meet its deadline. Since $40 + 20 < 100$, task τ_1 is schedulable.

(2) Task t_2: Check each of the following:

$$C_1 + C_2 + B_2 \leq T_1$$
$$40 + 40 + 30 > 100$$

$$2C_1 + C_2 + B_2 \leq T_2$$
$$80 + 40 + 30 = 150$$

These equations reflect the two scheduling points applicable to task τ_2, and take into account the fact that task τ_2 will be blocked at most B_2 milliseconds. Moreover, we don't need to consider the blocking time for task τ_1 because this time only affects the completion time for τ_1; it cannot affect τ_2's completion time. Hence, task τ_2 is schedulable and in the worst-case phasing will meet its deadline exactly at time 150.

(3) Task τ_3: The analysis here is identical to the analysis for Task 3 of Example 2. Hence, task τ_3 is also schedulable and in the worst-case phasing will meet its deadline exactly at time 300. It follows that all three periodic tasks can meet their deadlines.

Next, we determine the response time for the aperiodics. The server capacity is 10 percent and the average aperiodic workload is 5 percent (2/40). Because most of the aperiodic arrivals can find tickets, we would expect a good response time. Indeed, using a M/M/1 approximation[10] for the lightly loaded server, the expected response time for the aperiodics is $W = E[S]/(1 - \rho) = 2/(1 - (0.05/0.10)) = 4$ milliseconds where $E[S]$ is the average execution time of aperiodic tasks and ρ is the average server utilization. Finally, although the worst-case total periodic and server workload is 95 percent, we can still do quite a bit of background processing, since the soft deadline aperiodics and the emergency task are unlikely to fully utilize the servers.

The results derived for this example show how the scheduling theory puts real-time programming on an analytic engineering basis.

Real-time scheduling in Ada

Although Ada was intended for use in building real-time systems, its suitability for real-time programming has been widely questioned. Many of these questions concern practical issues, such as fast rendezvous and interrupt handling.

These problems are being addressed by compiler vendors aiming at the real-time market. More important are concerns about the suitability of Ada's tasking model for dealing with real-time programming. For example, tasks in Ada run nondeterministically, making it hard for traditional real-time programmers to decide whether any tasks will meet their deadlines.

In addition, the scheduling rules of Ada don't seem to support priority scheduling well. Prioritized tasks are queued in FIFO order rather than by priority; high priority tasks can be delayed indefinitely when calling low priority tasks (due to priority inversion); and task priorities cannot be changed when application demands change at runtime.

Fortunately, solutions exist within the current language framework, although some language changes would be helpful to ensure uniform implementation support.

Whether or not language changes are made, we must be careful to use Ada constructs in ways that are consistent with rate monotonic principles. The Real-Time Scheduling in Ada Project at the Carnegie Mellon University Software Engineering Institute is a cooperative effort between CMU, the SEI, system developers in industry, Ada vendors, and government agencies. It aims to specify coding guidelines and runtime system support needed to use rate monotonic scheduling theory in Ada programs. The guidelines are still evolving and being evaluated, but so far it seems likely they will meet the needs of a useful range of systems.

The remainder of this section summarizes the approach being taken by the project and shows how Ada's scheduling rules can be interpreted to support the requirements of rate monotonic scheduling algorithms.

Ada real-time design guidelines. Since Ada was not designed with rate monotonic scheduling algorithms in mind, it is not surprising to find some difficulties in using these algorithms. There are basically two ways of using rate monotonic algorithms in Ada:

(1) follow coding guidelines that take into account the scheduling policy that must be supported by every real-time Ada implementation or

(2) take advantage of the fact that Ada allows rate-monotonic scheduling algorithms to be supported directly by a special-purpose runtime system.

There are several Ada usage issues that must be addressed:

(1) how to code Ada tasks that must share data,
(2) how to use sporadic server concepts for aperiodic tasks, and
(3) how to ensure precise scheduling of periodic tasks.

We discuss the first two issues in this article. For data sharing among periodic Ada tasks (client tasks), our first guideline is that these tasks must not call each other directly to exchange data. Instead, whenever they must read or write shared data or send a message, they call a monitor task. Each monitor task has a simple structure — an endless loop containing a single select statement with no guards (see Figure 4). (The prohibition against guards simplifies the schedulability analysis and the runtime system implementation, but otherwise is not essential.)

This structure models the notion of critical regions guarded by a semaphore; each entry is a critical region for the task calling the entry. By using this coding style, we

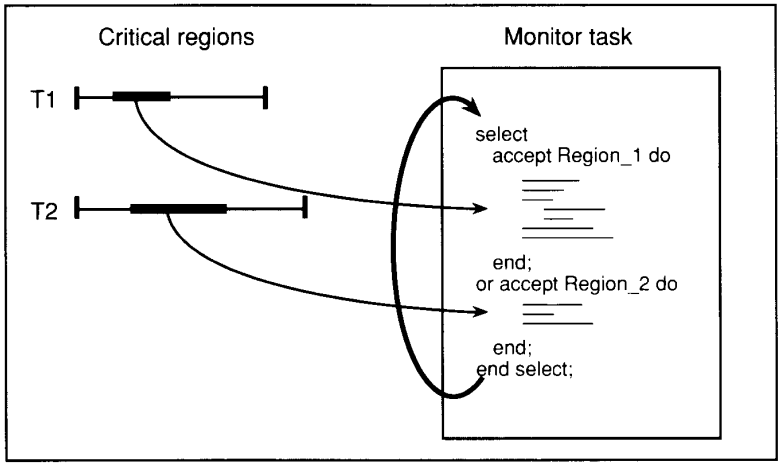

Figure 4. Modeling critical regions with an Ada task. T1 and T2 are two periodic tasks with critical regions guarded by a single semaphore. The corresponding monitor task acts as the semaphore guarding the critical regions. Each entry of the monitor task is the critical region for one of the periodic tasks.

have a program structure that can be analyzed directly in rate monotonic terms.

Client tasks are assigned priorities according to the rate monotonic algorithm. There are two options when assigning a priority to a monitor. If the Ada runtime system supports the priority ceiling protocol directly, then give the monitor task an undefined priority. In addition, tell the runtime system the priority ceiling of the monitor — that is, the highest priority of all its clients — using an implementation-defined pragma or system call.

If the ceiling protocol is not supported directly, the same schedulability effect can be emulated by giving the monitor task a priority that is just higher than the priority of any client task. This artificial priority assignment is consistent with the priority ceiling protocol.

To see this, note that once a high-priority client task starts to execute, it cannot be blocked by any lower priority client that has called an entry of the monitor; if the low-priority client is in rendezvous with the monitor before the high-priority client is ready to execute, the low-priority client will complete its rendezvous before the higher priority client is allowed to execute. If the low-priority task is not in rendezvous with a monitor task and the high-priority client becomes eligible to run, the low-priority task will be preempted and won't be allowed to execute until the high-priority-client task has finished its work.

Therefore, a high-priority client task can be blocked by at most one lower priority client task. The effect on system performance of giving the monitor task a high priority is simply that the worst-case blocking time is more frequently experienced; when rendezvous times are short, this effect is negligible.

Giving monitor tasks a high priority is not an acceptable implementation of the ceiling protocol if the monitor task suspends itself while in rendezvous, for example, while waiting for an I/O device to complete. Suspension allows other tasks to execute, meaning first-in, first-out (that is, unprioritized) entry queues could build up or critical sections could be entered under conditions inconsistent with the requirements of the priority ceiling protocol. If such a suspension is a possibility, the ceiling protocol should be supported directly in the Ada runtime implementation.

We have shown how the effect of the priority ceiling protocol can be provided at the application code level. A simple version of the sporadic server algorithm can also be supported at the application code level by creating a server task with the following structure:

```
task body Server is
   Tickets := Initial_Budget;
   -- set initial execution time budget
begin
   loop
      accept Aperiodic_Request;
      Set_Replenishment_Time;
      -- do work
      Decrement_Ticket_Count;
      -- one ticket per request
      if No_More_Tickets then
   Delay_Until_Replenishment_Time;
         Replenish_Tickets;
      end if;
   end loop;
end Server;
```

In essence, this task refuses to accept work unless it has a ticket to complete a request. The algorithm for deciding when to replenish tickets can be considerably more complicated, and a single server can be structured to accept a variety of service requests with different execution times.

If the runtime system supports the sporadic server algorithm directly, then the budget and replenishment period are specified by system calls or by implementation-dependent pragmas, and the loop simply accepts an entry call and does the work. Execution time management and task suspension is then handled directly by the runtime.

On Ada scheduling rules. The most general solution to the problem of using rate monotonic scheduling algorithms in Ada, within the constraints of the language, is simply to not use pragma PRIORITY at all.

If all tasks in a program have no assigned priority, then the runtime system is free to use any convenient algorithm for deciding which eligible task to run. An implementation-dependent pragma could then be used to give "Rate_Monotonic_Priorities" to tasks.

This approach would even allow "priorities" to be changed dynamically at runtime because, in a legalistic sense, there are no Ada priorities at all. The only problem with this approach is Ada's requirement that calls be queued in order of arrival, independent of priority.

However, even this problem can be solved efficiently. To get the effect of prioritized queueing, the runtime scheduler should ensure that no calls "arrive" until they can be accepted. This can be achieved by a legalistic trick — keeping the Count attribute zero (because logically, no entry calls are queued), and correspondingly suspending tasks making entry calls at the point of the call if the priority ceiling protocol requires that the call be blocked.

The elimination of entry queues also

makes the runtime more efficient. However, we must emphasize here that this treatment of entry calls is only allowed if no tasks have a priority specified by Ada's pragma PRIORITY.

Of course, telling programmers to assign "Rate_Monotonic_Priorities" to tasks but not to use pragma PRIORITY surely says we are fighting the language rather than taking advantage of it. But, the important point is that no official revisions to the language are needed.

Now, let us consider in more detail the specific Ada scheduling rules that cause difficulties and the appropriate ways of getting around the problems. When we describe these workarounds, we don't intend to suggest that Ada is therefore completely satisfactory for use with rate monotonic theory, but rather, to show that software designers need not wait for a revision to the Ada standard before trying out these ideas.

• *CPU allocation: Priorities must be observed.* Ada requires that, in a uniprocessor environment, the highest priority task eligible to run be given the CPU. For applications using the rate monotonic theory, this scheduling rule is correct as far as it goes. The only problem arises when determining the priority at which a task rendezvous should be executed. This is discussed below.

• *Hardware task priority: Always higher than software task priorities.* This Ada rule reflects current hardware designs, but hardware interrupts should not always have the highest priority from the viewpoint of the rate monotonic theory.

Solution: When handling an interrupt that, according to the rate monotonic theory, should have a lower priority than the priority of some application task, keep the interrupt handling actions short (which is already a common practice) and include the interrupt handling duration as blocking time in the rate monotonic analysis. In other words, use the scheduling theory to take into account the effect of this source of priority inversion.

• *Priority rules for task rendezvous.* The priority of a task can only be increased while executing a rendezvous, and any increase is based only on the priorities of the rendezvousing tasks. In particular, the priority is not increased based on the priority of blocked tasks, as required by the priority ceiling protocol.

Solution: We have already discussed the two solutions to this problem: (1) give monitor tasks a high priority, or (2) implement the priority ceiling protocol directly in the runtime system. If a monitor task is given a priority higher than that of its callers, then no increase in priority is ever necessary, so the problem is side-stepped. Alternatively, give the monitor task no priority at all. An implementation can then use priority ceiling rules to increase the priority of the called task to the necessary level since, in this case, the Ada rules say a rendezvous is executed with "at least" the priority of the caller, that is, the rendezvous can be executed at the priority of a blocked task.

• *FIFO entry queues.* Ada requires that the priority of calling tasks be ignored; calls must be serviced in their order of arrival, not in order of priority. Using FIFO queues rather than prioritized queues usually has a serious negative effect on real-time schedulability. FIFO queues must be avoided.

Solution: As noted earlier, it is possible to avoid FIFO queueing by giving monitor tasks a high priority or by using an implementation-defined scheduler to ensure that calls do not arrive unless they can be accepted. In either case, no queues are formed.

• *Task priorities: Fixed.* This rule is inappropriate when task priorities need to be changed at runtime. For example, when a new mode is initiated, the frequency of a task and/or its criticality may change, implying its priority may change. In addition, the scheduling rules for a certain class of aperiodic servers demand that the priority of such a server be lowered when it is about to exceed the maximum execution time allowed for a certain interval of time and be raised when its service capacity has been restored.

Solution: When an application needs to adjust the priority of a task at runtime, this task should be declared as having no Ada priority. The runtime system can then be given a way of scheduling the task appropriately by, in effect, changing its priority.

• *Selective wait.* Priority can be ignored. That is, the scheduler is allowed, but not required, to take priorities into account when tasks of different priorities are waiting at open select alternatives.

Solution: If a monitor task is given a priority higher than that of any of its callers and is not suspended during a rendezvous (for example, by executing a delay statement), this rule doesn't matter because there will never be more than one caller at any given time. Otherwise, the implementation should select the highest priority waiting task.

Summing up. From what our project has learned so far, it seems to be possible in practice to support analytic scheduling algorithms in Ada by using an enlightened interpretation of Ada's scheduling rules together with a combination of runtime system modifications and appropriate coding guidelines. Of course, it would be better if the language did not get in the way of rate monotonic scheduling principles. The future revision of Ada should reword some of these rules so that support for rate monotonic scheduling is more clearly allowed.

In this article, we have reviewed some important results of rate monotonic scheduling theory. The theory allows programmers to reason with confidence about timing correctness at the tasking level of abstraction. As long as analytic scheduling algorithms are supported by the runtime system and resource utilization bounds on CPU, I/O drivers, and communication media are observed, the timing constraints will be satisfied. Even if there is a transient overload, the tasks missing deadlines will be in a predefined order.

Rate monotonic scheduling theory puts real-time system design on a firmer, analytical, engineering basis than has been possible before now. Moreover, the theory is not just an academic curiosity. It has been used in the design of several production systems. Interestingly, none of the systems actually supported the theory in an ideal way.

The theory has proven to be quite robust in giving insight into the behavior of existing systems that are failing to meet their performance requirements and to suggest remedies. It promises to be even more useful when used at the outset in designing real-time systems.

The ideas have been applied to uniprocessor systems and to a real-time local area network. Currently, work is actively underway to apply the ideas to distributed systems in a more integrated fashion.

Although the treatment of priorities by the current Ada tasking model can and should be improved, the scheduling algorithms can be used today within the existing Ada rules if an appropriate coding and design approach is taken and if runtime

systems are written to take full advantage of certain coding styles and the existing flexibility in the scheduling rules. DDC-I, Verdix, Telesoft, and Ready Systems are among the Ada compiler vendors working on supporting rate monotonic algorithms. In particular, DDC-I and Verdix currently support the priority inheritance protocol,[9] whose performance for practical purposes is quite similar to that of the priority ceiling protocol. Verdix also provides special optimizations for monitor tasks. The SEI is providing technical reports showing how to implement the algorithms, and we will be providing test programs to check the correctness and the performance of the implementations. ■

Acknowledgments

The authors thank Mark Klein and Mark Borger for pointing out why the priority ceiling protocol cannot be correctly simulated simply by giving a monitor task a high priority when the monitor can suspend itself during a rendezvous. In addition, we thank Anthony Gargaro for catching an error in an early draft of Example 2. Finally, we thank the referees for their helpful comments.

References

1. R. Rajkumar, L. Sha, and J.P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," *Proc. IEEE Real-Time Systems Symp.*, CS Press, Los Alamitos, Calif., Order No. 894, 1988, pp. 259-269.

2. L. Sha et al., "Mode Change Protocols for Priority-Driven Preemptive Scheduling," *J. Real-Time Systems*, Vol. 1, 1989, pp. 243-264.

3. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," *J. ACM*, Vol. 20, No. 1, 1973, pp. 46-61.

4. J.P. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm — Exact Characterization and Average Case Behavior," *Proc. IEEE Real-Time System Symp.*, CS Press, Los Alamitos, Calif., Order No. 2004, 1989, pp. 166-171.

5. L. Sha, J.P. Lehoczky, and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *Proc. IEEE Real-Time Systems Symp.*, CS Press, Los Alamitos, Calif., Order No. 749, 1986, pp. 181-191.

6. J.P. Lehoczky, L. Sha, and J. Strosnider, "Enhancing Aperiodic Responsiveness in a Hard Real-Time Environment," *Proc. IEEE Real-Time Systems Symp.*, CS Press, Los Alamitos, Calif., Order No. 815, 1987, pp. 261-270.

7. B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *J. Real-Time Systems*, Vol. 1, No. 1, 1989, pp. 27-60

8. J.B. Goodenough and L. Sha, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks," *Ada Letters, Special Issue: Proc. 2nd Int'l Workshop on Real-Time Ada Issues VIII*, Vol. 7, Fall 1988, pp. 20-31.

9. L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," tech. report, Dept. of Computer Science, Carnegie Mellon Univ., 1987. To appear in *IEEE Trans. Computers*.

10. L. Kleinrock, *Queueing Systems*, Vol. I, John Wiley and Sons, 1975.

Lui Sha is a senior member of the technical staff at the Software Engineering Institute, Carnegie Mellon University, and a member of the research faculty at the School of Computer Science at CMU. His interests lie in real-time computing systems. He has authored and coauthored a number of articles in this area.

Sha co-chaired the 1988 IEEE and Usenix Workshop on Real-Time Software and Operating Systems; chairs the Real-Time Task Group on the IEEE Futurebus+ and is the architect of the Futurebus+ Real-Time System Configuration guide; is vice chair of the 1990 10th International Conference on Distributed Computing Systems and a reviewer of NASA Space Station Freedom's data management system design; and is a member of the IEEE Computer Society. He received his BSEE from McGill University in 1978, and his MSEE and PhD from Carnegie Mellon University in 1979 and 1985, respectively.

John B. Goodenough is a senior member of the technical staff at the Software Engineering Institute at Carnegie Mellon University. His technical interests include programming languages, with special emphasis on Ada, real-time methods, software engineering, and software reuse. He was heavily involved in the design of Ada as a distinguished reviewer and led the Ada compiler validation test effort.

Goodenough is chair of the group responsible for providing interpretations of the Ada standard and has the title of distinguished reviewer for the Ada revision effort. He has published papers on software testing, exception handling, and software engineering principles. He earned his MA and PhD in applied mathematics at Harvard University in 1962 and 1970, respectively. He is a member of the IEEE Computer Society.

The authors can be contacted at the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890.