An Object-Oriented Real-Time Programming Language

Yutaka Ishikawa, MITI Electrotechnical Laboratory

Hideyuki Tokuda and Clifford W. Mercer, Carnegie Mellon University

The demand for real-time systems increases with the demand for time-critical applications such as multimedia, robotics, factory automation, telecommunication, and air traffic control. Traditional programming languages do not support real-time systems development. They have neither the analytical techniques for modeling systems accurately nor the explicit specifications for timing constraints.

By providing high-level abstractions of program modules, the object-oriented paradigm makes it easier to design and develop applications. However, the object-oriented model and its implementing languages typically offer no more support for real-time programming than traditional languages do.

We have developed an extended object-oriented model — the real-time object model. It encapsulates rigid timing constraints in an object. We have also designed and implemented RTC++, a programming language that extends C++ on the basis of the real-time object model.

The real-time object model is a methodology for describing real-time systems. RTC++ is a programming language that extends C++ based on this model.

Basic issues

Schedulability analysis. A system is said to be *schedulable* if it meets all deadlines of a task set. One major difficulty in building a real-time system is the lack of good techniques for analyzing schedulability.

Schedulability analysis lets a program designer predict (under certain conditions) whether given real-time tasks can meet their timing constraints. It requires a bound on the execution time of each task. To meet this requirement, the system must avoid priority inversion problems that occur when a higher priority task must wait while a lower priority task executes.[1] For example, under priority-based scheduling, a low-priority task that holds a computational resource, such as a shared lock, blocks a higher priority task from this resource until the low-priority task completes. If several tasks of intermediate priority lie between the lower and higher priority tasks, the blocked high-priority task must wait for a period bounded only by the number of medium-priority tasks. This problem makes it very difficult to put an accurate bound on task execution times.

Specifying rigid timing constraints. Conventional real-time programs do not explicitly describe timing constraints in the program text. Instead, they describe

0018-9162/92/1000-0066$03.00 © 1992 IEEE

them in a separate timing chart or document. This makes it difficult to enforce timing constraints or detect timing errors during compile time or runtime.

Moreover, current systems pose difficulties in specifying the timing characteristics of a periodic task. Languages or operating systems often use the duration of a delay statement to implement a periodic task. However, this can lead to an inaccurate value for the waiting time. For example, consider the following program written in Ada:

```
1 loop
2    — . . . body of cyclic activity . . .
3    dtime := nexttime – currenttime;
4    delay dtime;
5 end loop
```

The execution of the statement at line 3 is not an atomic action, so the dtime variable may have a wrong value. For example, if the program's execution is suspended after currenttime is evaluated and resumed later, dtime is calculated with the incorrect value of currenttime. So the program might be delayed too long in the delay statement.

This delay problem and other issues related to real-time programming in Ada are addressed in a proposal for the coming Ada standard, Ada 9X.[2]

Scheduling

Two major approaches to developing schedulable real-time systems dominate the current state of the art.

The cyclic executive. This approach performs a sequence of actions during fixed periods of time. The execution is divided into two parts. The major cycle schedules computations to be repeated indefinitely. The major cycle is composed of minor cycles. A programmer divides each task into subcomponents so that the execution of each subcomponent fits into the minor cycles in a way that satisfies the timing constraints. In other words, this programming style forces a programmer to schedule programs using static analysis tools with some manual scheduling or reprogramming to ensure predictable execution timing.

Rate monotonic scheduling. This approach uses a preemptive fixed-priority scheduling algorithm that assigns higher priority to tasks with shorter periods. The CPU utilization of a task i, $U(i)$, is calculated by $U(i) = C(i) / T(i)$, where $C(i)$ and $T(i)$ are the execution time and period of task i, respectively.

Assume a task's deadline is the same as its period. Its CPU utilization is schedulable up to 100 percent in the case of a harmonic task set where all periodic tasks start at the same time and all periods are harmonic. In the general case, n independent periodic processes can meet their deadlines if the following formula holds:

$$\sum_{i=1}^{n} \frac{C(i)}{T(i)} \le n\left(2^{1/n} - 1\right)$$

This formula is very simple but pessimistic: A task set that does not satisfy this condition may or may not be schedulable. There is a more precise schedulability analysis of the rate monotonic algorithm (see references 1 and 5 in the main article). However, in this article, we use this pessimistic formula for simplicity.

Scheduling approach

Many developers use the cyclic executive to predict timing correctness for real-time systems with periodic tasks[3] (see the sidebar on scheduling). This approach offers a framework for scheduling periodic tasks, but it has some problems. First, a programmer must use tools for deterministic scheduling. These tools require much insight into timing requirements and program structure. Sometimes, a task's structure must be changed to satisfy the timing constraints: for example, a single logical task might be split into two parts that fit better into the timing structure.

Second, programs built with the cyclic executive are very difficult to extend or modify. Changes tend to violate timing structure and constraints that were tuned to specific characteristics of the original problem.

Instead of the cyclic executive, our approach employs the rate monotonic scheduling analysis.[4] Rate monotonic scheduling uses a preemptive fixed-priority scheduling algorithm that assigns higher priority to the tasks with shorter periods. With this algorithm, the schedulability of a given task set is analyzed by applying a closed formula (see the sidebar on scheduling).

Rate monotonic scheduling does not require programmers to split tasks by hand as the cyclic executive does, but the tasks must be preemptive and there is some penalty for context-switch overhead. Critical regions that require mutual exclusion interfere with the preemptability constraint of rate monotonic analysis, and the resulting potential for priority inversion must be accounted for.

Therefore, our approach employs the priority inheritance protocol[1] to bound the duration of priority inversion. In the priority inheritance protocol, if a task has to wait for the completion of a lower priority task's execution, the low-priority task's priority is temporarily changed to the priority of the higher task. Thus, tasks of intermediate priority cannot disturb the execution of the lower priority task. This lets us bound a task's blocking time (that is, the time a task spends waiting for a resource, such as a mutual exclusion lock, to become available). Note that the term inheritance as used in priority inheritance protocol has no relation to the inheritance of objects in the object-oriented methodology.

Using the priority inheritance protocol under rate monotonic scheduling, all periodic tasks meet their deadline if the following formula holds[1,5]:

$$\frac{C(1)}{T(1)} + \cdots + \frac{C(n)}{T(n)} + \max\left\{\frac{B(1)}{T(1)}, \cdots, \frac{B(n-1)}{T(n-1)}\right\} \le n\left(2^{1/n} - 1\right) \quad (1)$$

where $C(i)$, $T(i)$, and $B(i)$ are the execu-

tion time, period, and blocking time, respectively, of the task i and n is the number of tasks. In this formula, a task whose subscript is smaller has a shorter period and a higher priority. To use these methods effectively for scheduling analysis, we need a good methodology to specify the execution and blocking times (due to both synchronization and communication) in the program text.

Real-time object model

Timing encapsulation. The real-time object model extends the object-oriented model to describe real-time properties in programs. In the real-time object model, active objects with timing constraints describe a system, together with their interaction through message passing. Such an active object is called a *real-time object.*

An active object, as described here, has one or more threads that can be executing when a message arrives. Various message-passing schemes have been introduced to describe concurrency among objects in object-oriented concurrent programming.[6] Figure 1 illustrates the typical execution flow between active objects. The sender object at (1) sends a message to the receiver at (2) and waits for the reply message. After the execution of (3) in the receiver, the receiver sends a reply message at (4). Then both the sender and receiver objects execute concurrently at (5) and (6).

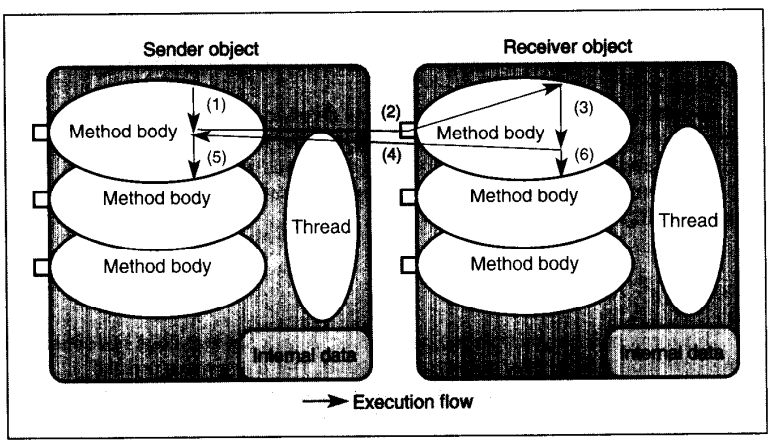


Figure 1. Execution flow between active objects.

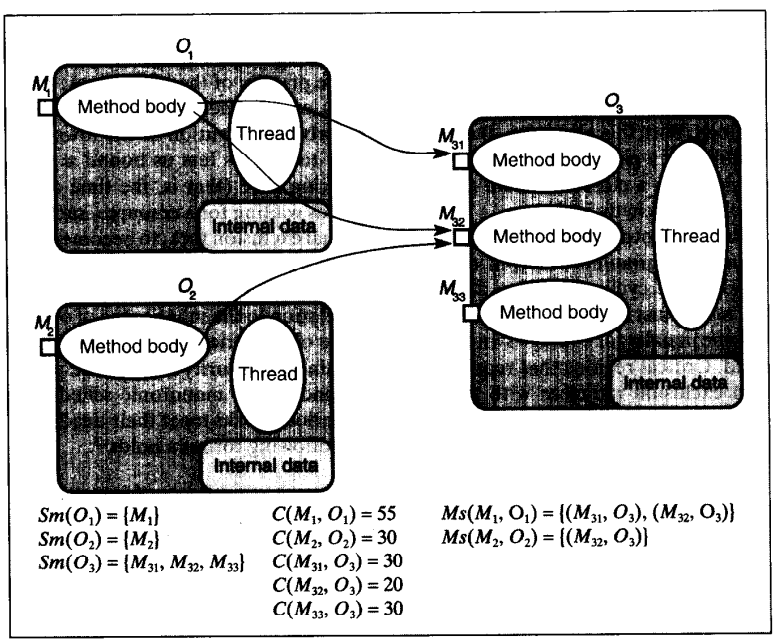


Figure 2. An example of real-time objects.

Nonpreemptive object. A nonpreemptive real-time object consists of internal data, operations called *methods* with timing properties, and a thread. We call the object nonpreemptive because the object performs the senders' requests sequentially and cannot interleave the execution of various requests. The following notation describes the timing properties of objects in the real-time object model:

- $Sm(o)$ is the set of methods in an object o.
- $C(m, o)$ is the worst case execution time (not including blocking time) of method m of object o.
- $Ms(m, o)$ is the multiset of other objects' methods called by method m of object o.

Figure 2 shows an example of real-time objects. Object O_1 has method M_1, whose worst-case execution time is 55 milliseconds. Object O_2 has method M_2, whose worst-case execution time is 30 milliseconds. Object O_3 has three methods, M_{31}, M_{32}, M_{33}, whose worst-case execution times are 30, 20, and 30 milliseconds, respectively. An arrow indicates an object's invocation sequence. Method M_1 in object O_1 invokes methods M_{31} and M_{32} in object O_3, while method M_2 in object O_2 invokes method M_{32} in object O_3.

By using the information about timing and execution dependency, we can analyze the timing constraints of the program as follows: Because M_1 of O_1 calls two methods (M_{31} and M_{32}) in O_3, the worst case execution time of M_1 must be greater than the summation of the worst-case execution times of M_{31} and M_{32}. Moreover, the worst-case execution of M_2 must be greater than the worst case execution of M_{32}. That is,

$$C(M_1, O_1) > C(M_{31}, O_3) + C(M_{32}, O_3) \rightarrow 55 > 30 + 20$$
$$C(M_2, O_2) > C(M_{32}, O_3) \rightarrow 30 > 20$$

One advantage of this model is that the schedulability of a task set is easily analyzed under the rate monotonic scheduling as described in the sidebar

on scheduling. Another advantage is that a reusable object is easily built for real-time applications. For example, we can provide a real-time object library such that several objects have the same functionality with the same interface but with different timing constraints, arising from their internal algorithms. Programmers can choose an object from the real-time library that fits their timing constraints.

Preemptive object. Nonpreemptive real-time objects can suffer from priority inversion due to blocking at an object invocation (see the sidebar on priority inversion in an active object). Two ways to reduce the blocking time are *concurrent execution* in the object or the abort-and-restart methodology.[7]

An object can execute requests concurrently if it has multiple threads, each of which is responsible for some methods. However, this doesn't eliminate blocking time due to the synchronization of internal data in an object. In the abort-and-restart methodology, if a process is going to be blocked at the request of an object, the current execution of the object is aborted. When the execution is aborted, the object is responsible for maintaining the consistency of the data. This methodology should be applied if the abort, recovery, and requeueing cost is less than the blocking cost. For simplicity, we do not consider the abort-and-restart methodology here.

The real-time object model can describe objects with multiple threads.[8] Each thread is responsible for performing one or more methods. A collection of threads may be responsible for the same set of methods, in which case the threads constitute a thread group.[9] Real-time objects with multiple threads are called *preemptive objects*. A preemptive object is described using the following notation in addition to the notation of the nonpreemptive object:

- $G(i)$ is thread group i (that is, the set of thread numbers), where $\forall i,j,i \neq j, G(i) \cap G(j) = \phi$.
- $Gm(m, o)$ is a thread group that executes the method.
- $Mr(m, o)$ is the multiset of pairs of critical region and its worst-case execution time in the method.

Let us say O_3 is a preemptive object instead of a nonpreemptive object. As shown in Figure 3, threads Th_1 and Th_2 are responsible for executing methods

Priority inversion in an active object

Figure A shows an example of priority inversion in an active object. Suppose we have a server object *S* and client objects *L* and *H* where *L*'s priority is lower than *H* 's. If the server is executing for *L* as a result of a request received from client *L* and client *H* sends a message to *S*, client *H* 's request is postponed until the server's execution for client *L* finishes. Because *H* 's priority is higher than *L*'s but processing for *L* precedes processing for *H*, we have a case of priority inversion in the server.

Moreover, if we assume that another object *M* is running independently with a medium priority, effectively bounding the execution time of *H* requires *S* to run with no interference from *M* whenever *H* is waiting for *S* 's reply. Thus, the priority of *S* has to change based on the highest priority of the requests waiting for service. This scheme for dynamically adjusting the priorities is called the priority inheritance protocol.

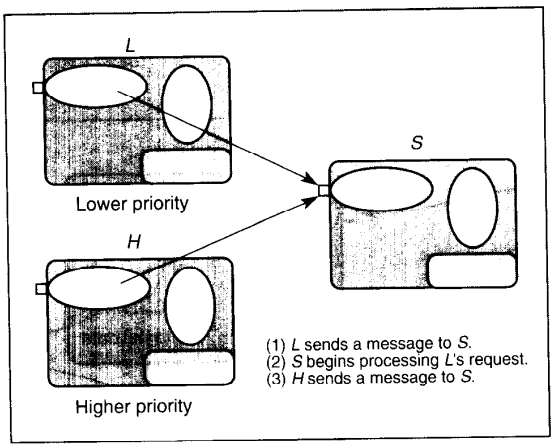


Figure A. Priority inversion in an object.

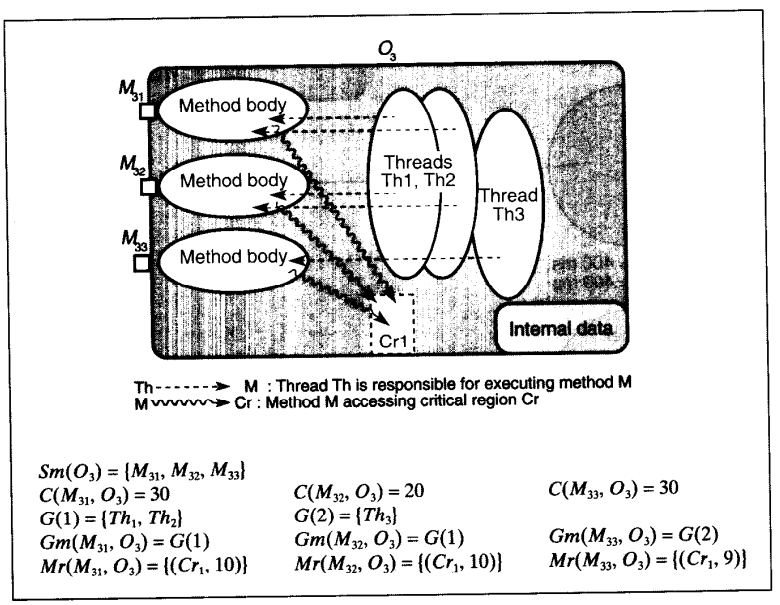


Figure 3. Preemptive object O_3.

Active object scheduling analysis

Suppose we have a real-time system composed of periodic tasks, active objects called by those tasks, and other independent active objects — all executing on a single CPU machine. We also assume that all method-calling sequences to other objects can be determined statically and that there are no recursive calls or unbounded iterations.

A periodic task has its period and deadline specified as timing properties. The task set is described by several objects and the interaction among those objects. Thus, a periodic task is defined as follows:

- $T(n)$ is the period of task n.
- $D(n)$ is the deadline of task n.
- $Ms(n)$ is the multiset of other objects' methods called by periodic task n.

Nonpreemptive object. Figure B shows an example where periodic tasks send messages to the objects defined in Figure 2 of the main text. The system task Timer is defined to handle task scheduling. The context-switch overhead is accumulated in the execution of the Timer. To analyze the schedulability of this example under rate monotonic scheduling, we prioritize the tasks Timer, P_1, P_2, and P_3 as highest, high, middle, and low, respectively. This priority corresponds to the shortest to longest task periods.

We analyze the worst-case execution time of each task first. This is easy to do because each of an object's methods has timing constraints. The worst-case execution of P_1 is 85 milliseconds because it calls two methods, M_1 of O_1 and M_2 of O_2, whose worst-case execution times are 55 and 30 milliseconds, respectively. In the same way, we determine that the worst-case execution times for P_2 and P_3 are 30 milliseconds each.

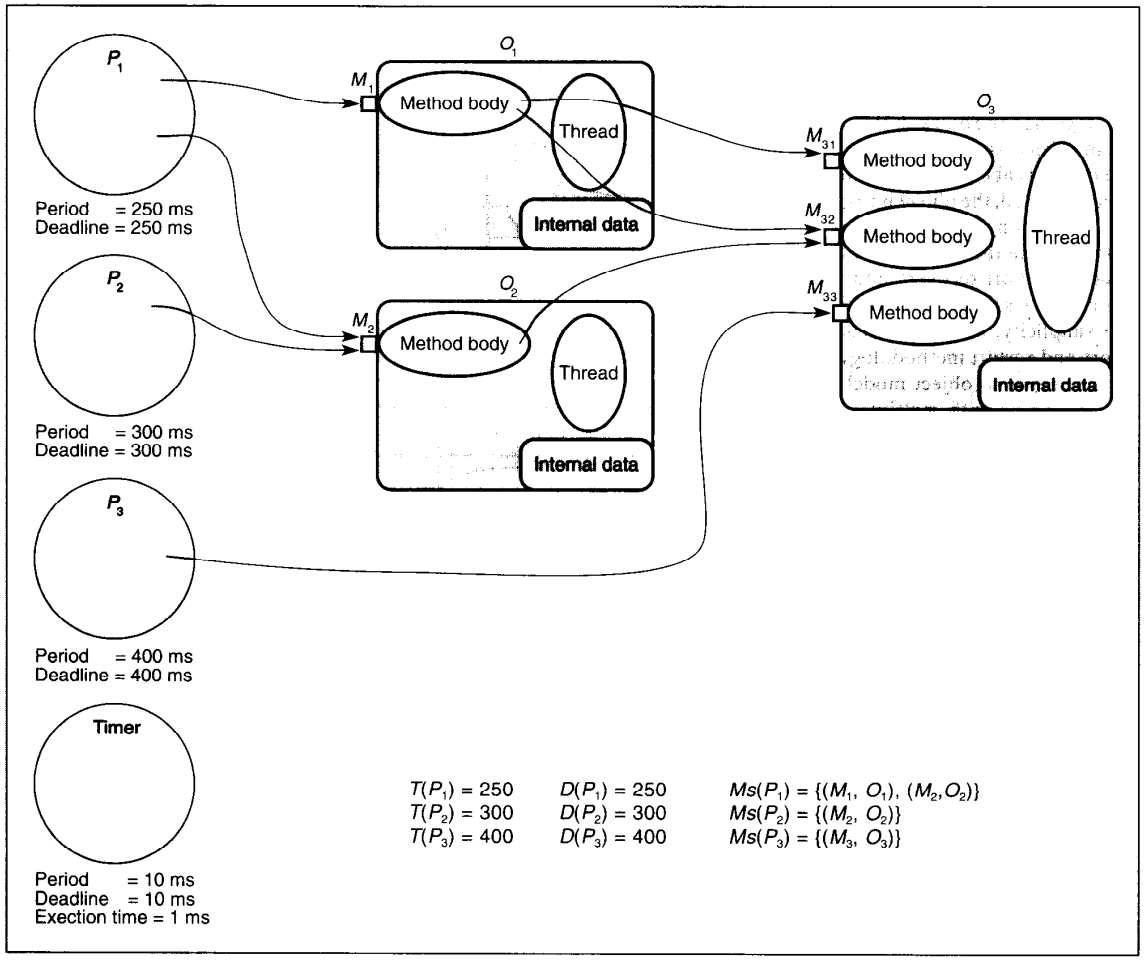


Figure B. A task set.

Second, we analyze the blocking time of all tasks except for the lowest priority task. In other words, we determine the time each task must wait for synchronization or communication with other activities. There are two cases where the execution of P_1 is blocked due to P_2. One case is when P_2 has called method M_2 of O_2 and then P_1 tries to call the same method. In this case, the worst-case blocking time of P_1 is 30 milliseconds because the request can be postponed until the execution of M_2 is finished.

The second case is when M_{32} of O_3 has been called by M_2 under P_2's request and later O_1 calls M_{31} or M_{32} under P_1's request. The execution of M_{31} and M_{32} cannot both be blocked by P_2 during one period of P_1. However, under the priority inheritance protocol, one of them can be blocked because the execution of P_2 is temporarily given the highest priority until the completion of O_3's M_{32}. After P_2 executes, it cannot disturb P_1. Thus, the blocking time at O_3 is 20 milliseconds.

P_2 can block the execution of P_1 at O_2 for 30 milliseconds and at O_3 for 20 milliseconds. However, if P_2 blocks P_1's execution at M_{32}, then P_2 also blocks the execution of M_2 for P_1 during one period of P_1. Thus, we estimate that 30 milliseconds is the worst-case blocking time of P_1 due to P_2.

Let us consider the relation between P_1 and P_3 in terms of blocking time. P_1 can be blocked by P_3 when P_1 calls M_{31} or M_{32} of O_3 during the execution of M_{33} under P_3's request. Here, the worst-case blocking time of P_1 is 30 milliseconds because the execution time of O_3's M_{33} is 30 milliseconds.

To summarize this analysis of P_1, the blocking time of P_1 is 60 milliseconds — 30 milliseconds due to P_2 and 30 milliseconds due to P_3. In this way, we can estimate other blocking times. The execution of P_2 can be disturbed by P_3 at M_{33} of O_3. The worst-case blocking time of P_2 is 30 milliseconds.

Table A summarizes the timing analysis. Using the table we can analyze the schedulability of the task set under rate monotonic scheduling by applying formula (1) from the main text:

$$\frac{C(Timer)}{T(Timer)} + \frac{C(1)}{T(1)} + \frac{C(2)}{T(2)} + \frac{C(3)}{T(3)} + \max\left(\frac{B(1)}{T(1)}, \frac{B(2)}{T(2)}\right)$$

$$= 0.1 + 0.34 + 0.1 + 0.075 + \max(0.24, 0.1)$$
$$= 0.855 > 3(2^{1/3} - 1) = 0.780$$

Thus, using this simple (pessimistic) test, we cannot guarantee the schedulability of this task set under rate monotonic scheduling.

Preemptive object. Suppose we replace object O_3 described above with another implementation that is preemptive (as defined in Figure 3 of the main text). To analyze the schedulability of a task set with this object, we modify the implementation of object O_3 without changing the execution time. The execution times of all tasks are the same as in the previous example.

Now we estimate the blocking time of P_1 and P_2. P_1's blocking time due to P_2 does not change, because P_1 calls M_2 of O_2, which calls O_3. So the blocking time of P_1 by P_2 is still 30 milliseconds. The blocking time of P_1 due to P_3, however, changes to 9 milliseconds because the method M_{32} blocks only for the duration of the critical region shared in O_3. Thus, the blocking time of P_1 is 39 milliseconds — 30 milliseconds for P_2 and 9 milliseconds for P_3. P_2's blocking time is also reestimated as 9 milliseconds.

Table B shows the results of this analysis. Using the table we can analyze the schedulability of the task set under rate monotonic scheduling as follows:

$$\frac{C(Timer)}{T(Timer)} + \frac{C(1)}{T(1)} + \frac{C(2)}{T(2)} + \frac{C(3)}{T(3)} + \max\left(\frac{B(1)}{T(1)}, \frac{B(2)}{T(2)}\right)$$

$$= 0.1 + 0.34 + 0.1 + 0.075 + \max(0.156, 0.03)$$
$$= 0.771 < 3(2^{1/3} - 1) = 0.780$$

The result shows that the task set is guaranteed schedulable.

Table A. Timing information for Figure B (in milliseconds)

Task	Period (T)	Deadline	Execution (C)	C/T	Blocking (B)	B/T
Timer	10	10	1	0.100	0	0
1	250	250	85	0.340	60	0.24
2	300	300	30	0.100	30	0.10
3	400	400	30	0.075	0	0

Table B. Timing information for Figure B with preemptive object (in milliseconds)

Process	Period (T)	Deadline	Execution (C)	C/T	Blocking (B)	B/T
Timer	10	10	1	0.100	0	0
1	250	250	85	0.340	39	0.156
2	300	300	30	0.100	9	0.030
3	400	400	30	0.075	0	0

```
1 active class O3 {
2   private:
3     // private data definition
4   public
5     int      m31(char* data, int size) bound(0t30m);
6     int      m32(char* data, int size) bound(0t20m) timeout(m32_abort);
7     int      m33(float f) bound(0t30m);
8   activity:
9     slave[2] m31(char*, int), m32(char*, int);
10    slave    m33(float f);
11  };
```

Figure 4. A real-time object in RTC++.

```
1 active class P1 {
2   private:
3     // private date definition
4     void     main();
5   activity:
6     master   main() cycle(0; 0; 0t200; 0t200);
7 };
```

Figure 5. A periodic task in RTC++.

M_{31} and M_{32}, while thread Th_3 is in charge of performing the method M_{33}. Suppose there is one critical region inside the object. During the execution of method M_{31}, it accesses the critical region for 10 milliseconds. The time of the critical region accessed by M_{32} is 10 milliseconds while the time of the region accessed by M_{33} is 9 milliseconds. All execution times of methods in O_3 are the same as they were in the nonpreemptive case.

The sidebar on the previous two-page spread analyzes the schedulability of a nonpreemptive active object and compares it to a preemptive active object. The results show that a system built using preemptive active objects provides better schedulability.

RTC++

RTC++[9] is an extension to C++. Its design is based on the real-time object model. In addition to C++ objects, RTC++ provides active objects. If an active object is defined with timing constraints, it is called a real-time object. Figure 4 shows the declaration of the active object O_3. An active object declaration is almost the same as the original C++ object declaration, except for the addition of the keyword Active before the keyword Class and the addition of a part for Activity.

Activity part. An active object has a single thread by default. A user can specify multiple threads, which we call *member threads* in the active object. Member threads are declared in the activity part of the class declaration. There are two types: slave and master.

A *slave thread* is an execution unit related to a method or a group of methods. Line 10 of Figure 4 declares that one slave thread is dedicated to handling the M_{33} requests. Line 9 specifies that two threads are responsible for executing methods M_{31} and M_{32}. That is, at most two requests of either M_{31} or M_{32} can be interleaved. These threads are called a *slave thread group*.

We employ the priority inheritance protocol in object invocation. That is, a slave thread inherits the priority from the sender. If there is a queue of waiting messages, the messages are ordered according to priority, and the priority of the slave thread is set to the highest priority of the invocations in the queue. When a new message for those methods arrives and the sender's priority is higher than the current thread's priority, the thread's priority is changed to the higher priority, and the message is enqueued at the head of the priority queue.

Figure 5 shows an example of a periodic task in RTC++. The *master thread* in line 6 is declared to specify the periodic task within an active object. The syntax of the cycle clause is as follows:

```
cycle(<start-time>; <end-time>;
  <period>; <deadline>);
```

In Figure 5, <start-time> and <end-time> are unspecified, so those constraints are free, and 0t200 indicates a time duration of 200 milliseconds. Therefore, the period is 200 milliseconds and the deadline coincides with the period.

Timing specification. Two types of timing information must be specified in RTC++: execution time and deadline time. RTC++ allows us to specify this timing information by using the Bound and Within constructs. The Bound construct asserts the worst-case execution time, while the Within construct asserts the deadline time.

As shown in Figure 4, all methods are declared with the worst-case execution time constraint. For example, the CPU usage in the execution of method M_{31} must be completed within 30 milliseconds. Line 6 shows that method M_{32} has a worst-case execution time of 20 milliseconds and that if this constraint is violated at runtime, the exception handler, m32_abort, is called.

Communication. RTC++ supports synchronous communication. The syntax of communication among active objects is the same as C++ syntax. For example:

```
1 O3 *v;
2 // . . .
3 n = v->m31(buf, size);
4 // . . .
```

RTC++ provides two means of sending a reply message: return and reply statements. In a return statement, a reply message is sent to the sender and the execution of the method is finished. In a reply statement, a reply message is sent and the subsequent statements are executed instead of finishing the execution of a method.

In addition to the features described in this article, RTC++ provides sophisticated facilities for programming applications: statement-level timing constraints, guard expressions, critical regions with timing constraints, and exception handling. Moreover, RTC++ provides facilities for programming distributed applications.

We think the constructs we proposed can be adapted to many other object-oriented languages besides C++. We have compared RTC++ with other real-time programming languages in a previous paper,[9] and the Ada 9X proposal[2] describes the impact of these issues on Ada.

RTC++ is currently running under the ARTS Kernel[10] on Motorola MC68030-based machines such as Sun3, Force Board, and Sony News. The RTC++ compiler generates C++ source programs and uses additional runtime support routines. ■

References

1. L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers*, Vol. 39, No. 9, Sept. 1990, pp. 1,175-1,185.

2. T. Baker and O. Pazy, "Real-Time Features for Ada 9X," *Proc. 12th IEEE Real-Time Systems Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2450, 1991, pp. 172-180.

3. T.P. Baker and A. Shaw, "The Cyclic Executive Model and Ada," *Proc. Ninth IEEE Real-Time Systems Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 894 (microfiche only), 1988, pp. 120-129.

4. 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.

5. L. Sha and J.B. Goodenough, "Real-Time Scheduling Theory and Ada," *Computer*, Vol. 23, No. 4, Apr. 1990, pp. 53-62.

6. *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, eds., MIT Press, Cambridge, Mass., 1987.

7. H. Tokuda and T. Nakajima, "Evaluation of Real-Time Synchronization in Real-Time Mach," *Proc. Second Mach Symp.*, Usenix, Berkeley, Calif., 1991, pp. 213-221.

8. C.W. Mercer and H. Tokuda, "The ARTS Real-Time Object Model," *Proc. 11th IEEE Real-Time Systems Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2112, 1990, pp. 2-10.

9. Y. Ishikawa, H. Tokuda, and C.W. Mercer, "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints," *Proc. Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, New York, 1990, pp. 289-298.

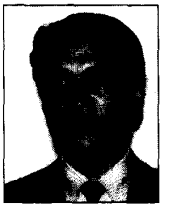
10. H. Tokuda and C.W. Mercer, "ARTS: A Distributed Real-Time Kernel," *Operating Systems Rev.*, Vol. 23, No. 3, July 1989, pp. 29-53.

Clifford W. Mercer is a PhD candidate in the School of Computer Science at Carnegie Mellon University. His research interests are in operating systems support for audio and video applications and distributed real-time systems.

Mercer graduated with university honors from Carnegie Mellon University with a BS in applied mathematics and computer science in 1988. He is a member of Sigma Xi and a student member of IEEE and ACM.

Yutaka Ishikawa is a senior researcher at Electrotechnical Laboratory, MITI, Japan. His research interests include real-time systems, distributed/parallel systems, and object-oriented programming languages.

Ishikawa received the BS, MS, and PhD degrees in electrical engineering from Keio University. He is a member of the IEEE Computer Society, ACM, and Japan Society for Software Science and Technology.



Hideyuki Tokuda is a senior research computer scientist in the School of Computer Science at Carnegie Mellon University and an associate professor of environmental information at Keio University. His research interests include distributed real-time systems, multimedia systems, communciation protocols, and massively parallel distributed systems.

Tokuda received BS and MS degrees in engineering from Keio University and a PhD degree in computer science from the University of Waterloo. He is a member of the IEEE, ACM, Information Processing Society of Japan, and Japan Society for Software Science and Technology.

Readers can contact Hideyuki Tokuda at Carnegie Mellow University, School of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15213; e-mail hxt@cs.cmu.edu.