Object-Oriented Real-Time Language Design: Constructs for Timing Constraints

Yutaka Ishikawa†, Hideyuki Tokuda‡, Clifford W. Mercer‡

†Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba,
Ibaraki, 305, JAPAN

‡Carnegie Mellon University
Pittsburgh, PA 15213 USA

yisikawa@etl.go.jp, hxt@cs.cmu.edu, cwm@cs.cmu.edu

Abstract

We propose a new object-oriented programming language called RTC++ for programming real-time applications. RTC++ is an extension of C++ and its features are to specify i) a real-time object which is an active entity, ii) timing constraints in an operation as well as in statements, and iii) a periodic task with rigid timing constraints.

In this paper, we first discuss real-time programming issues and what language support should be provided for building real-time applications. Then, the key features of RTC++ are described. Some programming examples are shown to demonstrate RTC++'s expressive power. A comparison to other programming languages are also discussed.

1 Introduction

Real-time computer systems play a very important role in our society. They are used in multimedia systems, robotics, factory automation, telecommunication systems, and in air traffic control systems. The object-oriented concept and programming languages make it easier to design and develop such complex real-time application programs. However, unlike non real-time programs, real-time programs must satisfy the *timing* correctness as well as *logical* correctness. Satisfying the timing correctness of a real-time program is difficult because of the lack of explicit specification of the timing constraints in a program and the lack of schedulability analysis techniques.

For example, a real-time task must start at a specified time and complete its activity by its deadline. However, in a conventional real-time program, such timing constraints are not explicitly described in its program, rather described in a separate timing chart or document. Thus, it is very difficult to enforce timing constraints or detect timing errors during compile time or/and runtime. Although the data encapsulation in object-oriented programming languages will help us to confine *logical errors* in a program, *timing errors* often penetrate the module boundary.

The schedulability analysis of a real-time program is also a difficult problem. By the schedulability analysis, we mean that a program designer should be able to analyze or predict whether the given real-time tasks having various types of system and task interactions (e.g., memory allocation/deallocation, message communications, I/O interactions, etc) can meet their timing constraints. For instance, if real-time tasks are interacting via shared resources, prediction of the worst case blocking time is difficult due to a possibility of having unbounded blocking delay.

In order to eliminate such unbounded blocking delay, a real-time system must avoid *priority inversion* problems. A priority inversion problem occurs when a higher task must wait indefinitely for a lower priority task to execute. For example, priority inversion may occur in task scheduling: when a task attempts to get a shared resource, it is blocked if another task is keeping the resource. If the task keeping the resource has a lower priority and a middle priority task can run under priority-based preemptive scheduling, then the blocked task has to wait for unbounded time. Thus, the system cannot guarantee to satisfy rigid timing constraints when priority inversion occurs.

We have designed and implemented a real-time distributed operating system kernel called the ARTS kernel

and programming languages ARTS/C and ARTS/C++ both of which are running on the kernel[11]. The ARTS kernel provides primitives of remote object invocation with timing constraints, methods for specifying periodic execution, and mechanisms for avoiding priority inversion. ARTS/C and ARTS/C++ are an extension of C and C++ respectively. Those have a capability of defining a special object called a *real-time object*. A real-time object has a set of operation with timing constraints and threads each of which is an execution unit. Languages do not have linguistic support for avoiding priority inversion but they can call kernel primitives for that purpose.

Our experiment with those languages leads to design more suitable object-oriented programming language which copes with real-time issues easily and efficiently. What capabilities we need in a real-time language are: i) expressions for timing constraints for statement level in addition to operation timing constraints, ii) linguistic support for avoiding priority inversion problem, and iii) an inheritance mechanism in a real-time object.

In this paper, first we discuss real-time programming issues and describe what language supports are required. Then, a new object-oriented programming language called RTC++ is proposed in section 3. In section 4, some programming examples are shown in order to demonstrate the RTC++ programming power. In section 5, we compare RTC++ with other object-oriented programming languages and real-time programming languages.

2 Real-Time Programming Issues

2.1 Timing Specification

Both ARTS/C and ARTS/C++ allow us to specify timing constraints of each operation in a real-time object. The objective of this support is as follows[11]: In traditional real-time systems which use a cyclic executive, a timing error often penetrates the task or module boundary, so that it is very difficult to capture the error at run-time. By using the timing specification, we can bound the timing error at run-time as well as compile-time.

Furthermore, we need to specify timing constraints for statement by statement in order to control execution depending on an external behavior. That is, when a part of statements (or transaction) in an operation cannot finish within the specified time, the current execution (or transaction) is aborted and then alternative statements are executed in order to satisfy timing constraint of the operation.

Another construct we need is the capability of describing a periodic task. Such capability is often realized as the duration of the waiting time by a language or an operating system. However, this leads to the unbounded waiting time. For example, the following program written in Ada is considered:

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

In this example, because the execution of the statement at line 3 is not an atomic action the `dtime` variable may have wrong value. That is, if the execution of the program is suspended after the `currenttime` is evaluated and then the execution is resumed later, the `dtime` is calculated with the wrong value of `currenttime`. So the program might sleep for wrong time at the `delay` statement.

2.2 Priority Inversion Problem

Using object-oriented paradigm, the client-server model is suitable in a distributed application. This model introduces a priority inversion problem in a real-time application. For example, suppose that there are server S and clients A and B where A's priority is higher than B's one. Suppose that when the client A sends a message to the server S, the server is performing for a request of the client B. In that case the request of client A is postponed until the server's execution for client B is finished. Because A's priority is higher than B's one but processing for B is prior to A, we call this phenomena *priority inversion* in the server. In order to avoid the priority inversion in a server, three methods can be considered: i) *preemption*, ii) *abort*, and iii) *priority inheritance*.

In the preemption method, the server's execution is preempted at the request of client A. Then, the server turns to perform for client A. After finishing the service for client A the service for client B is resumed.

In the abort method, the server's execution is aborted at the request of client A and then the server turns to perform for client A. At the abort of the execution the server must be responsible for maintaining the consistency. The principle of the recovery scheme in ARTS appeared in [9].

If the server's execution cannot be preempted and the cost of the abort procedure is too high compared with waiting for finishing the current server's execution, waiting is the best method.

However, this is not true when the server is running with other tasks. Suppose that the server's priority depends on the client's priority [1] and there is another task C whose priority is lower than the client A's priority but higher than the client B's priority. In this assumption, when C is ready to run, C begins to run while the server S performing for B is suspended. Thus, priority inversion occurs.

[1] This assumption is reasonable since the server has to serve for many clients whose priorities are different.

To avoid such priority inversion, we use the notion of priority inheritance [8] to propagate priority. That is, if a task provides a service on behalf of a client, the server should inherit the priority of the client. Furthermore, the server should inherit the priority of the highest priority task which is waiting for the service. Also, it is important to use priority queue instead of FIFO queue for a message queue.

If we apply the priority inheritance mechanism to the above example, the server S's priority is changed to client A's priority at the request from A. Thus, C could not run even when it becomes ready to run.

2.3 Single vs Multiple Threaded Object Model

Most object-oriented concurrent programming languages such as Actor[1], ABCL/1[14], ConcurrentSmalltalk[13], and Orient84/K[4] provide an object with a single thread model [2]. The idea of an object-oriented concurrent language is that an object is a sequential execution entity and concurrency is expressed by means of various message passing forms such as synchronous and asynchronous communication. The concept of sequential objects and message passing allows us to reduce programming complexity in a parallel application. Let us call this model the single threaded object model.

As described in the previous subsection, a highly preemptable server is required in a real-time application. The mechanism we need is that if a request message is coming at a server from a client during the server's execution for a lower priority's client, the server's execution is preempted or aborted and then performs for the higher priority's client. However, if the lower priority's client requests a service to the server, the request should be enqueued.

In object-oriented concurrent languages such a server can be implemented as follows. A server object consists of a root object and a set of objects each of which is responsible for one of the server's operation with a priority. Objects have to share the server's internal data so that each object can access the same data. For example, if two operations are defined in a server and the number of possible different priorities is three, then three objects each of which has a different priority are responsible for an operation and other three objects are responsible for the other operation. When a client sends a request message to the root object, the root object forwards the message to an object according to the client's request and priority.

Another approach to describing a preemptable server is that the server is implemented based on the multiple threaded object model. In the multiple threaded object model, there are some threads of control in an object so that a thread is invoked at the new client request whose priority is higher than the current client's priority. Because the object has multiple threads, the concurrency control has to be employed.

Conceptually the single threaded model and multiple threaded model have the same capability. So we have to consider the implementation of those models. The implementation of a highly preemptable server in the single threaded model needs more resources than one in the multiple threaded model because there are so many objects required. Moreover, if the internal data is primitive data such as integer or character, the compiler can generate an optimum code in the multiple threaded model while message passing forms are always needed in the single threaded model.

Based on the above observation, we choose the multiple threaded model to describe a real-time application. In the multiple threaded model, no restrictions on dynamically creating threads in an object may lead to increasing the complexity of concurrency control. Thus, the restriction we choose is that each operation may be executed concurrently but an operation has to be executed by one thread at a time. Moreover, the execution may be preempted to realize a highly preemptable server.

3 RTC++

In this section, we propose an object-oriented real-time language called RTC++ which is an extension of C++. RTC++ is designed based on the previous discussion. The syntax and semantics are described with examples.

3.1 Real-Time Object

In addition to C++ objects, RTC++ introduces an active object. If an active object is defined with timing constraints, it is called a real-time object. In the following example, the active class **Example1** is defined.

```
active class Example1 {
private:
  char   buf[BUF_SIZE];
  int    count;
  int    background();
public:
  int    read(char* data, int size) when(count > size);
  int    write(char* data, int size);
  int    open();
  int    close();
activity:
  slave[5]    read(char*, int);
  slave[5]    write(char*, int);
  slave       open(), close();
  master      background() cycle(;;0t30m;);
}
```

An active object definition is almost the same as an original C++ object definition except for adding the keyword **active** before the **class** keyword in RTC++. An

[2] Actor and ABCL/1 support a reentrant object if the object has no internal data.

active object has a single thread of control in default. A user can specify multiple threads of control which we call *member threads* in an active object. Member threads are defined in an *activity part* of a class definition. There are two types of member threads, *slave* and *master*.

A *slave thread* is an execution unit related to a member function or a group of member functions [3] . In the following definition, one slave thread is only responsible for the open and close requests.

```
slave          open(), close();
```

A slave thread inherits the priority from a sender. If there are some waiting messages, the priority of the slave thread is set to the highest priority of those messages. When a new message for those functions is comming and the sender's priority is higher than the current thread's priority, the thread's priority is changed to the higher priority. This mechanism is called the *priority inheritance mechanism in an object.*

In the following definition, the processing of the member function read can be preempted by up to five clients whose priorities are higher than the current execution of the read function. We call ''slave[5]'' a slave thread group.

```
slave[5]    read(char*, int);
```

A slave thread group does not realize just an interrupt mechanism. In order to illustrate the concept of the slave thread group, the following example is considered:

```
read(char* b, int s)
{
    <non-critical region A>
    <critical region B>
    <non-critical region C>
}
```

In this example, the read function consists of the sequence of the non-critical region, critical region, and non-critical region. Suppose that one of the thread group enters the critical region B and at that time a new read request where the sender's priority is higher than the previous sender's one is coming. Another new thread begins to execute the read function with the higher priority while the former thread is suspended. Since the critical region B is captured by the former thread, the higher priority's thread cannot enter the region B and is blocked. So the former thread executes again until the exiting critical region B. After that, the higher priority's thread is resumed. In this way, a slave thread group does not just realize an interrupt mechanism but supports a preemption mechanism.

The priority inheritance in a slave thread group is as follows: When all slave threads of a group are employed for clients' requests and at that time a higher priority's request than those threads' priorities is comming, then the highest priority's thread of the group changes its priority to the new highest priority.

A master thread is intended to use a background thread within an active object. For example, we want to specify the background thread which saves its internal data into a back up disk every 30 minute [4]:

```
master          background()  cycle(;;0t30m;);
```

In this example, the cycle expression specifies that the background thread is a cyclic task. The following is the syntax of the cycle expression:

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

In the example, <start-time>, <end-time>, and <deadline> are omitted so that those constraints are free.

In RTC++, a guard expression[2] may be defined in a member function definition in order to control concurrency. A guard expression may consist of primitive data types such as integer, primitive operations such as addition, internal variables (or instance variables in Smalltalk terminology), and message variables. For example, the following definition specifies that iff the expression ''count > size'' is true, the ''read(...)'' member function is invoked by a request, otherwise the invocation for the request is postponed until the expression becomes true:

```
int     read(char* data, int size) when(count > size);
```

In addition to a guard expression, we can specify a function which is invoked when a request is postponed. For example, the definition below specifies that if the guard expression is false then the *busy* function is invoked. If the function returns 0 the request is rejected, otherwise it is enqueued to the message queue.

```
int     read(char* data, int size) when(count > size)
                                   onwait(busy());
```

An expression for creating an active object is the same as an original C++ new expression except for adding priority. For example, the following expression means that an instance of class Example1 is created with priority 4 [5]:

```
Example1 *v = new Example1 priority 4;
```

When an instance object is created in the above expression, threads in the object have priority 4 at the first.

[3] A member function is called a method in Smalltalk terminology.

[4] An example of the notation of time is that 8 hour 20 minute 30 second and 10 millisecond 10 microsecond is specified as "0t8h20m30s10.10".

[5] Priority is defined as number. Larger number represents higher priority.

3.2 Inheritance

Unlike the previous language ARTS/C++, RTC++ supports the inheritance mechanism in an active object. An example below defines the **Example2** active class derived from class **Example1** which we call a base class in C++ terminology [6]. In class **Example2** member functions **read** and **write** are redefined and the **control** member function is defined. The activity parts of among a class and its base classes are merged consistently. That is, an instance of class **Example2** has two slave thread groups defined in class **Example1** and one slave thread defined in class **Example2**.

```
active class Example2 : public Example1 {
public:
    int     read(char* data, int size);
    int     write(char* data, int size);
    int     control(...);
activity:
    slave   open(), close(), control(...);
}
```

It should be noted that if there are no activity parts of among an active class and its base classes, an instance of the class has only one thread which is responsible for all member functions.

3.3 Communication

RTC++ supports synchronous communication. The syntax of communication among active objects is the same as C++ syntax. An example is shown below:

```
Example1  *v;
...;
n = v->read(buf, size);
...;
```

RTC++ provides two means of sending a reply message, *return* and *reply* statements. The semantics of a *return* statement is that a reply message is sent to the sender and the execution of the function is finished. The semantics of a *reply* statement is that a reply message is sent and the subsequent statements are executed instead of finishing the execution of a member function.

It should be mentioned that a message has a priority which is the same as the thread's priority of a sender.

3.4 Exception Handling

A block started with the **except** keyword is called an *exception handling block*. An exception handling block is led by *do*, *within*, *cycle*, or *region* blocks. The *within*, *cycle*, and *region* blocks are described later. In an exception handling block, we can catch and handle an exception from an object, a thread, or a kernel. In the following example, *timeout* and *abort* exceptions can be handled during the execution of "<do region>":

[6] In Smalltalk terminology, class **Example2** is a subclass of **Example1** while class **Example1** is the superclass of **Example2**.

```
do {
    <do region>
} except {
case abort:
    ...;
    break;
case timeout:
    ...;
    break;
}
```

In order to protect a sequence of statements from exceptions such as *timeout* and *abort*, a protect region is supported in RTC++. In the following example, a block started with the **protect** keyword is a protect region. That is, even if *timeout* or *abort* occurs while the region is executed, the exception is postponed until exiting the region.

```
protect {
    ...;
}
```

The following example specifies that the **read** function is a protected region:

```
active class Example {
    protect int read(....);
};
```

3.5 Timing Facilities

As described in Section 2.1, a real-time programming language should support i) timing specification in each operation, ii) timing constraints for statement by statement, and iii) specifying a periodic task. RTC++ supports those requirements.

3.5.1 Time Encapsulation

The time encapsulation mechanism allows us to specify timing specification in an operation. The following definition specifies that the **read** function has to be finished within 20 milliseconds, otherwise the **read_abort()** function is called.

```
active class Example1 {
private:
    ...;
    int   read_abort();
    int   write_abort();
public:
    int   read(char* data, int size)
              when(count > size)
              within(0t20) timeout(read_abort());
    int   write(char* data, int size)
              within(0t20) timeout(write_abort());
     ...;
}
```

3.5.2 Within, At, Before Statements

Statements *within*, *at*, and *before* express statement-level timing constraints. A *within* statement expresses the duration of execution. An *at* statement expresses the constraint of starting time while a *before* statement expresses the constraint of ending time.

For example, an example of a *within* statement is as follows:

```
within(time) {
    ...;
} except {
case timeout:
    ...;
}
```

The *time* variable keeps an instance of the `Time` class which is supported by the ARTS kernel. We can specify the duration of a time [7]. When the execution of statements surrounded by ''`within(time) {`'' and ''`}`'' cannot be finished within the time specified by the `time` value, statements led by ''`case timeout:`'' are executed.

3.5.3 Cycle Statement

A cycle statement is to specify a periodic task. As shown below, a cycle statement can be followed by an exception handling block:

```
cycle(starttime; endtime; period; deadline) {
    ...;
} except {
case timeout:
    ...;
}
```

In this example, `starttime`, `endtime`, and `deadline` specify starting time, ending time, period, and the deadline time of the execution respectively. Those are instances of class `Time`.

3.6 Critical Region

A critical region is realized by implementing an object with a guard expression. In a critical region we need a mechanism that a thread entering a critical region can be aborted or inherit a priority from another thread trying to enter the region. RTC++ introduces a special class called `ActiveEntity` which supports the abort and priority inheritance mechanism.

The `ActiveEntity` keeps an active object and member thread information. All member functions in an active object can refer to special variables `myentity` and `sender` which are instances of `ActiveEntity`. The `myentity` variable keeps own object and thread information while the `sender` variable keeps the sender object and thread information. Class `ActiveEntity` has the `pinherit` member function to propagate priority, the `prelinquish` member function to cancel the propagated priority, and the `abort` member function to abort a member thread of an object.

The `ActiveEntity` class allows us to define a `Region` class which realizes a critical region with the abort and priority inheritance mechanisms. The following is a part of the class `Region` definition.

```
active class Region {
private:
    int             flag;
    ActiveEntity    *user;
public:
    protect int     use()    when(flag == USED)
                                onwait(check());
    protect int     release();
    protect int     who();
    protect int     abort();
activity:
    slave           use(), release(), who(), abort();
}
Region::check()
{
    if (sender->prio() > user->prio()) {
        user->pinherit(sender->prio);
    }
}
Region::use()
{
    flag = USED;
    user = sender;
}
Region::release()
{
    user->prelinquish();
    flag = FREE;
}
```

The `Region` class defines `use`, `release`, `who`, and `abort` member functions. The `use` and `release` functions are equivalent to the `P` and `V` operations in a semaphore, respectively. The `who` function returns who is currently occupying. The `abort` function is to abort the execution of a thread which is currently occupying the region.

Class `Region` supports the priority inheritance mechanism as follows. If a thread of an object has entered the region, another thread will wait for changing the `flag` variable at the `use` function. On waiting a message at the `use` function, the `check` function is called. If the waiting sender's priority is higher than the priority of the thread entering the region, the higher priority is propagated to the thread by the `check` function.

The thread leaving the region sends the `release` message to the region. The `release` function cancels the propagated priority to the thread so that the thread's priority becomes the previous priority. In a complex case, the `prelinquish` function in the `ActiveEntity` class must decide the new priority of the thread because the thread may have been inherited priorities from other threads.

[7] For example, 0t1s20.10 means that the duration of a time is second 20 milliseconds 10 microseconds.

An example of a critical region is shown below:

```
active class aClass {
    Region    *rr;
    ...;
}

aClass::aFunction() {
    do {
        t = rr->use();
        ...;
        t = rr->release();
    } except {
    case abort:
        ...;
        if (t == USED) {
           t = rr->release();
        }
        break;
    }
}
```

The reader may wonder if this example is too complex and a programmer forgets to call the `release` function. We also think many critical regions are used in order to realize a preemptable object. Thus, RTC++ provides a critical region statement which is a kind of macro:

```
region (rr) {
    ...;
} except {
case abort:
    ...;
}
```

The following critical region statement expresses that if the region `rr` cannot be entered within `tt`, the `timeout` exception occurs:

```
within(tt) region (rr) {
    ...;
} except {
case timeout:
    ...;
case abort:
    ...;
}
```

It should be noted that RTC++ predefines class `Region` which is the same semantics described above but the implementation is different. That is, functions are almost implemented by the ARTS kernel because of the efficient execution. However, a programmer can program an object which has the same semantics of class `Region` described above by using instances of class `ActiveEntity`.

4 Examples

In this section, we demonstrate capabilities of RTC++ by three examples. One is called *Dining Touchy Philosophers* problem and another one is called *Gluttons-Chef in Restaurant*. The other one is called *Dining Faithful Philosophers problem*. The full programs of examples appear in [6].

4.1 Dining Touchy Philosophers

Dining Touchy Philosophers problem is an extension of the dining philosophers problem stated by Dijkstra. The philosophers share a common dining room where there are a side table and a circular table surrounded by five chairs. There are four set of chopsticks in the side table. In the center of the circular table there is a large bowl of spaghetti, and the table is laid with five forks. On feeling hungry, a philosopher enters the dining room, sits in his own chair, and tries to eat spaghetti. He prefers to use forks but he can use chopsticks. He needs to pick up forks on the both side. He can wait for getting both forks for a while depending on his feeling. If he could not get them, he gives up using forks and goes to the side table to pick up chopsticks. When he has finished to eat, he puts down forks or chopsticks, and leaves the room.

The following is a part of the program written in RTC++:

```
78      for(;;) {
79          room.enter();
80          left = right = STATUS_WAITING;
81          method = FORK;
82          within(wait_t) {
83              /* gettting right fork */
84              right = fork[n].use();
85              /* gettting left fork */
86              left = fork[(n + 1)%N].use();
87          } except {
88          case timeout:
89              if(right == STATUS_GETTING) {
90                  right = fork[n].release();
91              }
92              if(left == STATUS_GETTING) {
93                  left = fork[(n + 1)%N].release();
94              }
95              ch = chopstick.use();
96              method = CHOPSTICK;
97          }
98          /* eating */
99          if(method == FORK) {
100             left = fork[(n + 1)%N].release();
101             right = fork[n].release();
102         } else {
103             ch = chopstick.release();
104         }
105         room.exit();
106         /* thinking */
107     }
```

In order to wait for getting both forks for the specified time, the `within` statement is used in lines 82 to 87. If the timeout occurs, lines 89 to 96 are executed to give up using forks and try to use chopsticks.

4.2 Gluttons-Chef in Restaurant

An example called *Gluttons-Chef in Restaurant* is considered in order to show a highly preemptable object written in RTC++.

Two gluttons go to a restaurant to eat stir-fry vegetables or pork stir-fry every lunch. They always have seats.

Table 1: Recipe

menu	instructions	time(min)
Stir-Fry Vegetables	chopping tomato	1
	chopping onion	1
	chopping green pepper	1
	chopping cucumbers	1
	getting fry pan	
	making stir-fry	6
	releasing fry pan	
Pork Stir-Fry	chopping pork	1
	chopping mushrooms	1
	chopping onion	1
	chopping green pepper	1
	getting fry pan	
	making stir-fry	9
	releasing fry pan	

Unfortunately only one chef is cooking for them. Gluttons A and B's action patterns are as follows: Glutton A's lunch time is only 25 minutes and he decides his order within 2 minutes. Glutton B's lunch time is 50 minutes and he decides his order within 1 minute.

The recipe of stir-fry vegetables and pork stir-fry is shown in Table1. There are one poor food processor, one fry pan, and seasoning. The food processor is capable of chopping one stuff at a time.

If the chef is only cooking for one order at a time without preemption, the worst case cooking time is 23 minutes (i.e., 13 + 10). In this case, glutton A could not eat. Suppose that the chef can suspend cooking while he is using the food processor but he cannot suspend cooking when he is using the fry pan. If the chef can suspend to cook when glutton A orders, he starts to cook for glutton A. In this case, the worst case cooking time is 19 minutes.

The **Chef** class is defined below. The **request** function is to accept entree from gluttons and cook it. The activity part specifies that the **request** function can be executed by two threads.

```
71  active class Chef {
72   public:
73                        Chef();
74      void              request(menu req);
75   private:
76      Foodprocessor     *fproc;
77      Fpan              *fpan;
78      Region            *fp;
79   activity:
80      slave[2]          request(menu req);
81  };
```

The following list is a part of the definition of the **request** function. Each request to an instance of the food processor **fproc** is performed inside the critical region **fp** so that another thread cannot request the food processor even if the thread's priority is higher than the running thread's priority.

Table 2: Faithful Philosophers Action

	period(min)	eating(min)	thinking(min)
P	15	3	10
Q	30	5	20
R	60	10	40

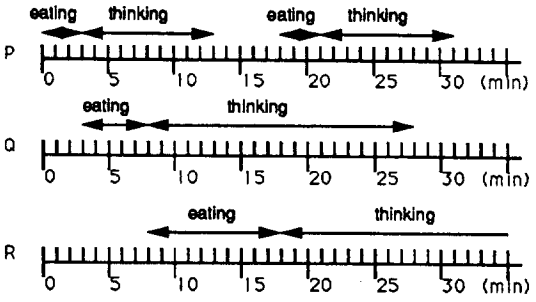

Figure 1: Timing Chart of Faithful Philosophers

```
93  switch(req) {
94  case V_STIR_FRY:
95     region(fp) { fproc->chopping(Otim, "tomato"); }
96     region(fp) { fproc->chopping(Otim, "onion"); }
....
103 case P_STIR_FRY:
104    region(fp) { fproc->chopping(Otim, "pork"); }
105    region(fp) { fproc->chopping(Otim, "mushrooms"); }
....
112 }
```

4.3 Dining Faithful Philosophers

In order to demonstrate how to use the abort method, another extension of the dining philosophers problem is considered. We call it the *Dining Faithful Philosophers* problem.

For simplicity, suppose only three philosophers. They are so faithful that their action is strictly decided shown in Table 2. For example, philosopher P eats spaghetti for 3 minutes and then thinks for 10 minutes every 15 minutes. He doesn't complaint even if someone disturbs his eating. However, because he is so faithful to his decision, when he could not perform his action on his schedule he might get angry and break the dining room.

As shown in Figure 1, philosopher P could not finish his second action within that period if a traditional program is used. This is because a philosopher is always keeping forks until finishing eat. In this example, we may preempt a fork which philosopher R is using because philosophers are so kind not to complain. This example should be implemented by using the abort method because the philosopher R has to give up eating by requesting from philosopher P.

The **Tool** class which is the base class of the **Fork** class is implemented so that if an object tries to use a tool

while another object having a lower priority is keeping the tool, the keeping object is aborted. The following list is a part of the **Tool** class definition. The function **abort** is invoked when an object has to wait for getting the tool. In the **abort** function, if the sender's priority is higher than the current user's priority, then the current user is aborted:

```
 8  active class Tool {
 9   public:
10      protect int use()  when(used == 0)
11                         onwait(abort);
12      protect int release();
13   private:
14      protect void abort();
15      ActiveEntity *user;
16      int          used = 0;
17  };


31  Tool::abort()
32  {
33      if(sender->prio() > user->prio()) {
34           user->abort();
35      }
36      return 1;
37  }
```

The following program is a part of the **Philosopher** class. In order to catch the **abort** exception, the exception handling block is defined:

```
67  cycle(start_t; end_t; period; worst_t) {
68      room.enter();
69      do {
70          left = right = STATUS_WAITING;
71          do {
72              /* gettting right fork */
73              right = fork[n].use();
74              /* gettting left fork */
75              left = fork[(n + 1)%N].use();
76              /* eating */
77              right = fork[n].release();
78              left = fork[(n + 1)%N].release();
79          } except {
80          case abort:
81              if(right == STATUS_GETTING) {
82                  right = fork[n].release();
83              }
84              if(left == STATUS_GETTING) {
85                  left = fork[(n + 1)%N].release();
86              }
87          }
88      } while (/* if he needs to eat more spaghetti
89           because someone disturbs him, continue */ );
90      room.exit();
91      /* thinking */
92  }
```

5 Comparison

RTC++ has two profiles, an object-oriented concurrent language and a real-time language. We compare RTC++ with those languages in this section.

In object-oriented concurrent languages such as ABCL/1[14], Actor[1], and Orient84/K[4], all objects are active entities each of which may have a thread of execution. That is, there are no distinction between active and passive objects. In contrast with those languages, RTC++ introduces the notion of the active object which is distinct from an original C++ object. The major reason is that we want to bound the memory resource and kernel resources statically when a program is written without dynamic properties such as the dynamic creation of active objects. If all objects are active entities, it is impossible to assign resources for a thread of execution to all objects. So a run-time routine must decide whether or not an object needs new resources for a thread of the execution[4]. In this case, it is difficult to bound the amount of the resource statically. In such an unbounded behavior, it is very complex to analyse the schedulability of a system because an object may wait for an unbounded time to get a resource from a resource pool when all resources are used by other objects.

In terms of communication primitives, many object-oriented concurrent languages support both asynchronous and synchronous communication facilities which imply the reliable facilities. In RTC++, asynchronous communication is not supported. This is because it is difficult for recovery from aborting in asynchronous communication. Moreover, in the underlying system only supporting reliable communication, asynchronous communication is equivalent to a function which is realized by a communication buffer object using synchronous communication. If unreliable asynchronous communication is required in a distributed environment, the language and its kernel may support such semantics.

Real-Time languages such as Real-Time Euclid[7] and real-time Mentat[3] provide facilities for rigid timing constraints such as specifying a periodic task. Real-Time Mentat is an object-oriented real-time language which is also an extension of C++. In Real-Time Mentat, a programmer may specify timing constraints in statement level. There are two ways of specifying timing constraints, *soft* and *hard* deadlines. In a block with *soft* deadline, the execution may be optionally skipped if the hard real-time tasks cannot meet their deadlines. The *soft* deadline constraint is currently not supported in RTC++. In contrast with RTC++, the facilities for a preemptable object are not supported in Real-Time Mentat.

Real-Time Euclid is an extension of Euclid. In order to bound the resources statically, Real-Time Euclid restricts language constructs such as recursion and dynamic memory allocation. In addition to specifying starting time of a task, Real-time Euclid supports facilities **signal** and **wait** to control concurrency. The **wait** statement is extended to specify a time bound. In RTC++, the statement "**within() region()** {}" provides the same

functionality which is a kind of macro and realized by the timeout mechanism in an object.

Real-Time Euclid is a system rather than just a language. The system consists of the language compiler, schedulability analyzer, and run-time system. The schedulability analyzer allows a programmer to find if a system is schedulable or not before the execution. This is one of the key issues in the real-time system research field.

In this paper, we have described on the language features of RTC++, but we also have been developing a schedulability analysis tool and monitoring tool for distributed real-time systems[10]. In particular, a schedulability analyzer called *Scheduler123* which takes the timing specification of a real-time task set and analyzes its schedulability based on various scheduling algorithms, and *ARM*, advanced real-time monitor, have been in use for a few years. We are currently extending the toolset for coping with the end-to-end schedulability analysis and designing a new timing analysis tool which extracts timing information from a RTC++ source code and transfers the information to Scheduler123. In order to extract timing information statically, the language constructs will be used restrictively or more language features will be added.

6 Conclusion

In this paper, we have proposed an object-oriented real-time language called RTC++ which supports explicit timing constraints, highly preemptable object, periodic task creation, and priority inheritance. We think, however, that the constructs we proposed are not only for the extension of C++ language, but also those can be adapted in many other object-oriented languages. We also introduced interesting real-time concurrent programs and demonstrated the usefulness of RTC++ features.

The RTC++ compiler which translates a RTC++ source program into C++ and C programs are currently implemented. [6] described the implementation hint of a RTC++ compiler and runtime routine. A technical report which describes the language features and implementation will be published soon.

Acknowledgements

One of the authors would like to express his gratitude to Dr. Akio Tojo and Dr. Kokichi Futatsugi, members of ETL, who support his research environment.

References

[1] G.A. Agha, "ACTORS," MIT Press, 1986.

[2] Dijkstra, E.W., "Guarded commands, nondeterminacy, and formal derivation of programs," Communication of ACM 18, August 1975, pp.435-457.

[3] Grimshaw, A.S, Silberman, A., Liu, J.W.S, "Real-Time Mentat, A Data-Driven, Object-Oriented System," Proceedings of IEEE Globecom, Dallas, Texas, November 1989, pp. 141-147.

[4] Ishikawa, Y., Tokoro, M., "A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation," Proceedings of OOPSLA-86, Portland, Sept. 1986, pp. 232-241.

[5] Ishikawa, Y., "A Study on Object-Oriented Concurrent Programming Languages for Describing Knowledge-Based System," Ph.D. Dissertation, Keio Univ., 1986.

[6] "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints", CMU Technical Report, 1990.

[7] Kligerman, E., Stoyenko, A.D., "Real-Time Euclid: A Language for Reliable Real-Time Systems," IEEE Transactions on Software Engineering, 12(9), September 1986, pp.941-949.

[8] Sha, L., Rajkumar, R. and Lehoczky, J.P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," Technical Report CMU-CS-87-181, Computer Science Department, Carnegie Meollon University, November, 1987.

[9] Tokuda, H., "Compensatable Atomic Objects in Object-oriented Operating Systems," Proceedings of Pacific Computer Communication Symposium, 1985.

[10] Tokuda, H., Kotera, M., "A Real-Time Tool Set for the ARTS Kernel," Proceedings of the 9th IEEE Real-Time Systems Symposium, December, 1988.

[11] Tokuda, H., Mercer, C.W. "ARTS: A Distributed Real-Time Kernel," Operating Systems Review, Vol.23, No.3, July, 1989, pp.29-53.

[12] Tokuda, H., Mercer, C.W., Ishikawa, Y., Marchok, T.E., "Priority Inversions in Real-Time Communication," 10th IEEE Real-Time Systems Symposium, December 1989, pp.348-359.

[13] Yokote, Y., Tokoro, M., "Experience and Evolution of ConcurrentSmalltalk," Proceedings of OOPSLA-87, Orland, Oct. 1987, pp.406-415.

[14] Yonezawa, A., Briot, J-P., Shibayama, E., "Object-Oriented Concurrent Programming in ABCL/1," Proceedings of OOPSLA-86, Orland, Oct. 1986, pp.258-268.