Interactors: A Real-Time Executive with Multiparty Interactions in C++

Pierre Labrèche, member
Louis Lamarche
Objectif: Systèmes — Objective: Systems
P.O. Box 265, Ville Mont-Royal
Québec, Canada, H3P 3C5

Abstract

Interactors is a run-time environment for embedded real-time software, which adds concurrency to the C++ object-oriented language. *Interactors* allows sequential processes to interact synchronously or asynchronously, and provides user-definable multiparty interactions. Several forms of selective wait, inspired by Ada, are provided. Scheduling algorithms follow Carnegie-Mellon University's recommendations for implementing hard deadline scheduling. Concepts are illustrated by simple application examples: Producer/Consumer and Dining Philosophers. This paper concludes by a description of the current implementation.

1 Background

Object-oriented languages, especially C++, are generating a lot of interest in the embedded software development community [Eckel]. The Ada language has been a major influence in the diffusion of concurrency concepts in the software engineering community. Although Ada integrates tasking as a language feature, it is not object-oriented, as it is lacking inheritance, an essential feature of object-oriented languages.

In the simulation area, the concepts of object and process have been successfully merged. The Simula language, initially developed for discrete event simulation, is recognized today as being the first object-oriented language [Meyer]. Simula also offers coroutines, independent threads of control which are also sometimes referred to as light-weight processes.

It is argued in [Magnusson] that concurrent processing could be achieved by adding interrupt-based preemptive scheduling to languages providing coroutines. Indeed, several coroutine-based executives have been proposed for many languages.

Language designers are debating whether concurrency mechanism should be part of programming languages, as in the case of Ada, or if these mechanims should be external, to be accessed using basic language features only. There is a similar debate about I/O handling in programming languages such as C and Pascal.

Implementing concurrency outside of a programming language keeps the language small. The resulting application software may be portable, but the software will not integrate easily with software developed for another process abstraction, thus affecting reusability.

On the other hand, the inclusion of a specific concurrency model in a language imposes this model to all applications, with the risk that this model may not be suitable to all. The limitations of the Ada tasking model have been discussed in many papers, for example [Cornhill]. As a legacy to the limitations of the "one size fits all" approach, Ada's tasking model is now generally avoided for critical software applications. As a consequence, Ada tasking is currently undergoing a re-definition, in the scope of the Ada 9X Project.

Revision proposals of Ada in the domain of tasking are summarized in several Ada Language Issues of the Ada Language Issues Working Group [ALIWG1] [ALIWG2] [1].

As processors are evolving rapidly, application code must be portable. Because the *C language* is available on virtually any kind of processor, from 8-bit microcontrollers to supercomputers, it has been used as the implementation language of many commercially available run-time executives. As the C++ language [Stroustrup] is a superset of C, and is typically compiled using a portable front-end translator to C (called `cfront`), the C++ language has also achieved a high degree of availability and portability.

The C++ language has been used to implement complete operating systems, including for multiprocessors. An excellent discrete-event simulation package, Simulation And Modelling In C++ (SAMOC) [Lomow], integrates a simulated-time process abstraction into the C++ language. SAMOC is implemented as C++ classes, without introducing any changes to the base language. An advantage of this add-on approach over defining special-purpose simulation languages is that it gives access to all features and support tools of the base programming language, in this case C++.

Other sequential languages were also extended for concurrency. A process abstraction was successfully implemented in the Modula-2 language [Burns]. This abstraction provides: preemptive scheduling, the one-to-one channels synchronization model of Occam, and a comprehensive suite of selective wait constructs.

Concurrent object-oriented languages use various models for inter-process synchronization and communication. A good discussion of concepts is available in [OOPSLA'87].

- Messaging models cause an independent thread activation in the object receiving the message. For some models, messaging is the only possible inter-process communications method. *Actors*, discussed in [Agha], are typical of messaging models.
- Messaging models support either blocking (synchronous) or non-blocking send (asynchronous). The Transputer Occam model provides synchronous message passing only.
- Inter-process communication and synchronization may be generalized to more than 2 parties. According to the authors of the design language Raddle87 [Forman], multiparty interactions are becoming an important abstraction in the design of distributed systems. Multiparty Interactions, or meetings, have also been included in a recent book on visual design techniques [Buhr].
- Shared memory models may be applicable, depending upon the physical connectivity of multiprocessors.

2 Introduction to *Interactors*

Interactors is a concurrent C++ environment for real-time applications. *Interactors* is designed for embedded software applications, such as control and monitoring systems, test equipment, and telecommunications.

Interactors support single-processor applications. Multiprocessing is not directly supported, but is possible using user-defined communications. Advanced features such as memory management, exceptions, and security were not considered at this time.

Interactors concurrent objects are defined in a hierarchy of classes. The lowest-level is the thread. A thread possesses its own stack, and must explicitely be given control. The next level is the process. It possesses all the attributes of the thread, and has specific attributes used for scheduling purposes. The highest level is the interrupt service process (ISP), a process which can be connected to a vectored interrupt.

New processes can be created dynamically. There is no task configuration file or static description of objects.

[1] Some of the interesting proposals are: [L108] Preference Control for open select alternatives. [L140] Add mutual control to tasking. [L161] Use of task priorities in accept and select statements. [L162] Increased control over Ada task scheduling. [L163] Provide comprehensive race controls.

Inter-process synchronization and communication, or interactions, were designed for maximum application flexibility. Multiparty interactions are user-definable:

- Interactions may be synchronous, i.e. ready only when all interactors are ready.
- Interactions may be asynchronous, i.e. requiring some form of buffering to be used between the interactors. For example, transferring data using an asynchronous interaction will allow the sender to continue even if there are no receivers waiting.
- Broadcast communications may be designed, allowing a sender's message to be transferred to a multitude of receivers.
- Data transfers are not the only types of interactions available. Some forms of interaction involve synchronization only; other forms involve data transfers and compuitations.
- A shared memory model may be used.

Flexible selective wait constructs are provided, modeled after Ada's own constructs. With selective waiting, a process can wait until the completion of one of many requests. The application may define its own criteria for arbitrating between many ready requests.

3 *Interactors* Built-in Components

In this section we describe the objects which are provided by the *Interactors* environment.

3.1 Interactors: Threads, Processes, and ISPs

The foundation of all concurrent objects (interactors) is the `THREAD`. A thread possesses its own stack, in order to support context switching. The stack size may be defined independently for each thread object. A thread may be initialized with interrupts enabled or disabled. Thread objects will not execute until the member function `Resume()` is called. When a thread is resumed, the executing thread's state is saved, and the resuned thread's state is restored. Thread behaviors are defined by the virtual function `Script()`. Thread-derived classes may have additional data supplied by constructors.

Threads may be used by applications but, in general, the increased capabilities of the `PROCESS` class will be required. Basic threads cannot be scheduled. Processes are threads, complemented with additional attributes, used for scheduling:

- Priority – An integral value indicating scheduling priority.
- Time Scheduled – The calendar time at which this task should be made ready for execution.
- Time Slicing Quota – The maximum number of Time Slicing periods for which this task should execute. Beyond that limit, other tasks of the same priority will be given control in a round-robin manner. A zero value disables time slicing.

Process member functions used for defining the script, for setting the priority, and for scheduling processes. A process may be scheduled with the following options:

- Immediate – `Schedule()`
- In a specified delay `d` – `Delay(d)`
- At a specified calendar time `t` – `Schedule(t)`

Most user-defined active objects will be derived form the process class. The MAIN built-in class defines properties of the unique instance object MainThread. A main program is defined by the function MAIN::Script().

Interrupt Service Processes (ISPs) can be resumed by the processor interrupt mechanism. ISPs are derived from processes, therefore they can be manipulated by the scheduling algorithms. ISPs are associated to an interrupt vector by the call to the SetVector(n) member function. Setting a vector installs an ISP so that interrputs at this vector will save the running thread's status on its stack, and then will change context to the ISP. The interrupted process's address is saved a field of the ISP. It is up to the ISP to re-schedule or resume the interrupted thread. The InterceptVector(n) member function is different: before giving control to the ISP, the interrupt vector's old Interrupt Service Routine (ISR) will be executed. The old ISR address which was in the Interrupt Vector may be called or restored using other member functions.

3.2 Interrupt Locks, Semaphores and Regions

Scheduling can be disabled by INTERRUPT_LOCK objects. An interrupt lock is usually unnamed. The constructor of interrupt locks will disable interrupt-driven scheduling, and its destructor will reestablish the condition which existed prior to the lock. Interrupt locks are designed for defining higher-level objects. Interrupt locks allows a structured access to a processor's interrupt control.

Binary SEMAPHOREs are provided for exclusion control. Semaphores consist of a priority list of waiting processes, and a pointer to the process owning the semaphore. The classical Wait() and Signal() operations are provided. If a wait operation cannot be satisfied because the semaphore is busy, the requesting process will be enqueued into the semaphore's list. Upon signalling to the semaphore, the highest priority waiting process will acquire the semaphore and will be scheduled. If configured so, semaphores implement priority inheritance (refer to 3.6.1).

REGION objects provide a structured access to semaphores. As for interrupt locks, regions are usually unnamed. A region consists of a pointer to a semaphore, which is initialized when the region is constructed. The semaphore is waited upon during construction of the region. The semaphore is released when the region is destructed. Regions can be used asis, or application-specific derived classes can also be defined.

3.3 Interactions and Requests

All inter-process synchronization and communication are based on two foundation classes: requests and interactions. The basic interactions and requests are virtual classes, i.e. which cannot be used directly by application software. Derived classes must provide the specialized features required for application use.

REQUESTs are data structures created by processes for communicating and synchronizing with other processes. A request usually carries information to the other processes which participate in some interaction. Requests can be submitted for unconditional execution by simply calling the Perform() member function. Requests are executed by processes in a mutually exclusive manner. Request types define a synchronizing condition allowing the request to proceed, and a script to be executed when allowed to proceed. Request instances are tied to an instance of an interaction. Those request types which can be pending because of synchronization will also indicate a queue within the interaction, where pending requests will be deposited.

The Perform() function contains all the logic to apply request's synchronization conditions, to enqueue requests, and to switch context when required.

INTERACTIONs are the meeting place of participating interactors. *Interactors* rendez-vous are different from the Ada tasking model. In Ada, tasking synchronization mechanisms are not symmetric: tasks communicate and synchronize on one end by entry calls and on the other end by call accepts. Moreover, a calling task must explicitly reference the desired acceptor task. An acceptor task needs not knowing the identification of the caller.

By contrast, *Interactors* allows symmetrical communications between interactors. This is made possible by restricting all exchanges between processes to use named, passive, INTERACTION objects. When a rendez-

vous takes place, the last ready interactor will effectively complete applicable pending requests prepared by all participating interactors, including itself.

Interactions typically include queues for requests which cannot immediately be serviced, if synchronous interactions are desired.

If asynchronous operations are required, the interactions will rather contain buffers allowing to immediately release the requesting process, providing a wait-free operation.

3.4 The Channel Built-In Interaction

The *Interactors* environment includes a `CHANNEL` built-in interaction class. Channels are synchronous, unbuffered, portals where information may be exchanged in one way only, from the point of view of a requestor. Two classes of requests are associated with channels: `SEND`s and `RECEIVE`s. Although each request is unidirectional, a process may alternately send and receive on a given channel.

A send request identifies the channel and source of data, and the number of receivers which need the information. Usually, only one receiver will be required. A receive request identifies the channel, and the destination of data.

Channels allow broadcast communications to be implemented. A transfer will be ready only when (1) one sender is ready, and (2) as many receivers as requested by the sender are ready.

The definition of channels contain inline members functions, which are more programmmer-friendly than `SEND(...).Perform()` and `RECEIVE(...).Perform()`. These functions are naturally named `Send(...)` and `Receive(...)`.

3.5 Selective Waiting

Selective waiting, for a process, is the ability to initiate a set of possible requests. Only one of these requests, at most, will be selected. For each select choice, the programmer specifies a statement to be executed, should that request be selected.

Selective waiting has been inspired by the rich set of options which are offered by the Ada language. The selection is not constrained to data transfer types of requests. Any request type may be specified in a selective wait construct. For instance, a process may selectively wait on sending on channel A, receiving on channel C, and obtaining a resource from a programmer-defined interaction.

Selective waiting is specified by the `select ... endselect` pair. Select alternatives are specified by `when(...)` clauses.

The following selective wait forms are mutually exclusive:

Unconditional will block the selecting process until a request is served.

Conditional will not block the selecting process except perhaps for mutual exclusion into the interactions. The interaction's own synchronization conditions will not block the selecting process. This option is specified by `when (nothing)` as the last alternative.

Delayed will block the selecting process for a maximum delay. This option is specified by `when (delay_is (...))` as the last alternative. The argument of the `delay_is` construct is a time duration.

Deadlined will block the selecting process until a specified calendar time. This option is specified by `when (time_is (...))` as the last alternative. The argument of the `time_is` construct is a calendar time.

The above select keywords are implemented as simple C pre-processor macros.

3.6 Scheduling

3.6.1 Algorithms

In real-time systems, the scheduling algorithms must be carefully selected, so that critical tasks be executed within their deadline. Carnegie-Mellon University research [Cornhill] has identified important requirements for real-time scheduling, which have all been incorporated and validated by at least one Ada vendor. These recommendations are being implemented into *Interactors*:

- The scheduler should be preemptive.
- Integration of the same scheduling algorithms for both interactions and for task activation and suspension.
- When multiple requests are ready for a selecting process, the request which is tied to the highest-priority interaction will be given priority.
- Priority queues are used as the standard mechanism for both the ready list and for the interaction waiting queues.
- Priority inheritance raises the priority of processes while they utilize a shared resource which blocks a higher-priority process.

3.6.2 Priority of Requests

Each request type has a virtual member function returning its priority. In general, that priority should be specified as the highest priority of any waiting process which will be scheduled when executing that request. This will be used by selecting processes to arbitrate between open alternatives, in favor of the waiting process with highest priority.

3.6.3 Scheduling lists

Two scheduling lists are used:

- The `ReadyList` is used to keep processed which are ready for execution, but which are not currently executing. This list is sorted by priority. Whenever the executing process is suspended, it is normally the first process on the ready list which is given control by the scheduler.
- The `DelayedList` is used to keep processes which are scheduled for later activation. This list is sorted by scheduled time. Periodic interrupts will trigger the time-based scheduler, which will determine if any delayed process can be made ready or if the running process has elapsed the duration of its time slice, if enabled.

3.6.4 Time

Calendar time is internally represented as a two-fold data structure: one part is the `time_t` ANSI C type, which is the current time in seconds since 00:00:00 1 January 1970; the other part is the 16-bit binary fraction of a second. This format (1) is independent of the actual duration of the hardware clock tick, (2) lends itself to easy manipulation with integer arithmetic, and (3) is compatibile with ANSI C standard time.

Durations are kept in the same format, seconds and fraction, with the semantic of being relative. Durations are easily be added to calendar time. In order to allow implementations without floating point arithmetic, a maco was defined (`TIME_F(...)`) which accepts a duration in seconds and converts it into a class constructor with integer arguments only.

```
#include <channel.hpp>
CHANNEL  CHAN;
class PROC : public PROCESS
{
public:
  PROC(){Schedule();}
  void Script()
    {
      int data;
      while(1)
      {
        CHAN.Receive(data);
        CHAN.Send(++data);
      }
    }
};
```

```
void MAIN::Script()
{
  PROC();
  for (int data=1; data != 10;)
  {
    CHAN.Send(data);
    CHAN.Receive(data);
  }
}
```

Figure 1: Consumer/Producer processes on a single channel.

4 Examples

The following are examples of classical problems coded in the *Interactors* environment.

4.1 Two Producer/Consumers, One Channel

In this example, two processes, exchange data back and forth. The first process will initially send, while the other will initially receive. A single channel is necessary for this type of interaction, because the roles of sender and receiver are synchronized. The complete example's source code is shown in figure 1

A static channel object, `CHAN`, is declared. A `PROC` process class is defined. The constructor of the `PROC` class schedules the process for immediate execution. The process's script is a never-ending loop: an integer value is read from the channel, is incremented, and sent back to the channel.

The main process creates an un-named `PROC` process. The main process sends to the channel, initially the value one, receives a new value from the channel, and will loop until the value 10 is received.

4.2 One Producer, Two Consumers, Two Channels

In this example, the main process sends data to either of two channels, each channel being used by a consumer. The consumer processes are not always receiving. Although two channel are used in this example, similar results could be achieved using a single channel. The complete example's source code is shown in figure 2

Two static channel objects, `CHAN1` and `CHAN2`, are declared. A `PROC` process class is defined, containing a private pointer to the channel to be used by the consumer process. The constructor of the `PROC` class will receive as an argument a pointer to a channel. The channel address is saved by the constructor, and the process is also scheduled for immediate execution. The consumer process's script is a never-ending loop: an integer value is read from the channel, and the consumer waits for 0.5 seconds.

The main process creates two un-named `PROC` processes, one for each channel. The main process will selectively send data to channel `CHAN1`, to `CHAN2`, selecting for at most 0.2 seconds. The main process will loop 10 times.

```
#include <channel.hpp>
#include <select.hpp>
#include <stream.hpp>

CHANNEL  CHAN1, CHAN2;

class PROC : public PROCESS
{
  CHANNEL *chan;
public:
  PROC(CHANNEL *c)
    {chan = c; Schedule();}
  void Script()
    {
      int data;
      while(1)
      {
        chan->Receive(data);
        delay(0.5);
      }
    }
};
```

```
void MAIN::Script()
{
  PROC (&CHAN1);
  PROC (&CHAN2);
  for (int data=0; data <10; data++)
  {
    select
      when(CHAN1.Send(data))
        {cout << "CHAN1 "; cout.flush();}
      when(CHAN2.Send(data))
        {cout << "CHAN2 "; cout.flush();}
      when(delay_is(0.2))
        {cout << "Timed out "; cout.flush();}
    endselect
  }
}
```

Figure 2: Single Producer, two channels, a Consumer on each channel

4.3 Dining Philosophers

The problem of dining philosophers is a classical synchronization problem cited in numerous articles and textbooks, for instance [Ringwood] and [Milenkovic]. This introduction to the problem is extracted from [Ringwood]:

> "Though the problem of the dining philosophers [Dijkstra] appears to have greater entertainment value than practical importance, it has had huge theoretical influence. The problem allows the classic pitfalls of concurrent programming to be demonstrated in a graphical situation. It is a benchmark of the expressive power of new primitives of concurrent programming and stands as a challenge to proposers of these languages."

There are five philosophers living in a monastery. The philosophers endlessly think and eat rice which is continually replenished by servants (not part of the problem). The dining room has a round table with five places, five bowls, and five chopsticks. A philosopher needs two chopsticks to eat. The problem is to provide synchronization between the philosophers providing mutual exclusion, avoiding deadlock and lockout. The complete example's source code is shown in figures 3 and 4.

The approach starts with designing a `CHOPSTICKS` interaction class which will synchronize all the philosophers, and keep a record of allocated chopsticks. There will be a single instance of this interaction. The chopsticks interaction has two data components: (1) the current status of each of the five chopsticks, `inUse[5]`, and (2) five lists of pending philosopher's requests, `waitList[5]`. Lists are used rather than a single pointer, allowing to borrow the built-in mechanisms of interactions. The constructor of the interaction will mark all chopsticks as not in use. Two access functions, `Get()` and `Put()` will be defined later. An instance object, `CHOPSTICKS`, is created statically. By convention, chopsticks are assigned to philosophers as follows: the left chopstick has the same number as the philosopher's seat; the right chopstick is the philosopher's number plus one, modulo five.

Request classes are needed for getting and returning chopsticks. Each of those requests will contain a single data item, the requesting philosopher's identification, a number between zero and four.

The get request class, GET_CHOPSTICKS, when constructed, will save the philosopher's identification. Virtual member functions are defined:

- `Interaction()` returns a pointer to the CHOPSTICKS interaction;
- `InteractionList()` will return a pointer to the request waiting list associated with the requesting philosopher's identification.
- `Ready()` returns a boolean indicating if the request may be performed. It is computed by verifying that both the left and right chopsticks of the requesting philosopher are not in use.
- `Script()` is associated with executing the request. It simply assigns the chopsticks as in use, therefore preventing other philosophers from using the left and right chopsticks.

The return request class, RETURN_CHOPSTICKS, when constructed, will save the philosopher's identification. Virtual member functions are defined:

- `Interaction()` returns a pointer to the CHOPSTICKS interaction;
- `Ready()` is always true: chopsticks may be returned any time.
- `Script()` is associated with executing the request. It marks the chopsticks as not in use, therefore allowing other philosophers to use the left and right chopsticks. Then, the returning philosopher, being very courteous, will see if any of its neighbors need help. Any pending requests to its left or to its right will be processed, in this order.

The chopsticks interaction, as written, could be used without modification to allocate chopsticks to more than five philosophers, as long as only five seats are assigned.

Two programmer-friendly functions are defined on chopsticks: `Get()` and `Put()`. These functions will only call the generic interaction's `Perform()` member function.

We can now look at the philosopher class, PHILOSOPHER, defining the properties of philosopher processes. The class has a static data member, `n`, used for assigning unique identifiers to each new philosopher. Each philosopher has an idenfifier, and a counter of the number of times he ate. The philosopher's constructor simply assigns the next sequential number to the philosopher's identifier, clears the counter, and schedules the philosopher for immediate execution. The philosophers will endlessly think for three seconds, acquire chopsticks, eat for two seconds, and return the chopsticks.

The main program creates five philosophers, waits for some duration, and then displays how many times each philosopher ate. When runing this program, we see that resource allocation is fair, as philosophers eat approximately the same number of times.

5 Current Implementation

5.1 Development Environment

The *Interactors* development environment is currently IBM/PC based. Targets are the Intel 8086 processors or family, although other processors are possible in the future.

The run-time system is ROM-able, and targetable to bare microprocessor boards. The run-time system is also targeted to DOS Personal Computers. The *Interactors* environment is to be linked with the application code.

Interactors allows developing and testing embedded application software on the host PC. This ability to compile and test, in the PC environment, the actual code which will eventually be programmed into a target ROM, using source-level debugger, reduces the need for in-circuit emulation.

Zortech's DOS hosted C++ 2.0 compiler [Zortech] is currently used, along with Microsoft's Codeview debugger. Zortech's compiler has proven to be a good choice. Although a few compiler bugs have been observed, the code generator and optimizer produce excellent code.

5.2 PC Self-Targeted Applications

Simple PC-based applications have been implemented, including the Lift Problem of [Forman].

Reusable classes for handling the computer's mouse, sound, graphics, and asynchronous RS-232 serial communications have been created and used. Asynchronous serial communications have been tied to C++ streams.

5.3 Embedded Targets

A custom C++ locator was developed for translating DOS loadable `.EXE` files to the absolute code formats required for programming ROMs or transferring to in-circuit emulators. This locator performs all required ROM to RAM initializations which are required for Zortech's C++ compiler. The C++ locator currently translates to Intel Hex and Hewlett-Packard 64000 Formats.

Small demonstration software was uploaded to an Intel ETOX memory evaluation board [Intel], which is driven by an 80C186 processor. The 80186 processor's intialization code and a serial port driver were developed. The asynchronous serial communications driver was also tied to C++ streams.

6 Conclusion

The development of *Interactors* indicate that object-oriented languages such as C++ lend themselves to natural extensions even in core areas such as concurrency. The extensions did not require changing any part of the language, or to use a non-standard pre-processor or compiler.

Interactors's programming model should help to narrow the gap between system design and software implementation.

References

[Agha] Agha G., *Foundational Issues in Concurrent Computing,* Sigplan Notices, vol 24 no 4, April 89, Proceedings of the ACM SIGPLAN workshop on object-based concurrent programming, San Diego, Sept 26-27, 1988.

[ALIWG1] *Ada Language Issues Working Group (ALIWG), Minutes of 1 March 1989,* Ada Letters, vol IX no 4, May-June 1989, ACM Press.

[ALIWG2] *Ada Language Issues Working Group (ALIWG), Minutes of 9 August 1989,* Ada Letters, vol IX no 7, Nov-Dec 1989, ACM Press.

[Buhr] Buhr R.J.A., *Practical Visual Techniques in System Design, with Applications to Ada,* Prentice Hall, December 1989 Prepublication Manuscript.

[Burns] Burns A., Davies G.L., Wellings A.J., *A Modula-2 Implementation of Real-Time Process Abstraction,* SIGPLAN Notices, Vol. 23, No 10.

[Cornhill] Cornhill D., Lui Sha, et. al., *Limitations of Ada for Real-Time Scheduling,* Department of Computer Science, Carnegie-Mellon University, Proceedings of the first international workshop on real-time Ada issues, Moretonhampstead, Devon, U.K., 1987.

[Dijkstra] Dijkstra E.W., *Hierarchical ordering of sequential processes,* Acta Inf. 1 (1971), 115–138.

[Eckel] Eckel B., *C++ for Embedded Systems,* Embedded Systems Programming, Volume 3, Number 1, January 1990.

[Intel] Intel Corporation, *26F256 Flash Memory/80C186 Evaluation Pack, Monitor Software Manual Rel 1.0,* May 3, 1988.

[Forman] Forman I. R., *Design by Decomposition of Multiparty Interactions in Raddle87,* Proceedings of the Fifth International Workshop on Software Specification and Design, May 19–20, 1989, Pittsburgh, PA, USA.

[Lomow] Lomow G., Ungar B., Birtwistle G., *User Reference Manual for SAMOC,* University of Calgary, Canada, 1987.

[Magnusson] Magnusson B., *Process Oriented Programming,* Proceedings of the ACM Sigplan Workshop on Object Based Concurrent Programming, in Sigplan Notices, vol 24 no 4, April 89.

[Meyer] Meyer B., *Object-Oriented Software Construction,* Prentice Hall,1988.

[Milenkovic] Milenkovic, *Operating Systems,* McGraw Hill, 1987

[OOPSLA'87] *OOPSLA '87 Panel Discussion: Object-Oriented Concurrency,* OOPSLA'87 Addendum to the Proceedings, October 1987, ACM Press.

[Ringwood] Ringwood G. A., *Parlog86 and the dining logicians,* Communications of the ACM, vol 31 no 1, January 1988.

[Stroustrup] Stroustrup B., *The C++ Programming Language,* Addison-Wesley Publishing Company, Reading, Massachussets, 1986.

[Zortech] Zortech Limited, - *Zortech C++ Compiler,* Second Printing, 1988, Zortech Limited.

```
#include <stream.hpp>
#include <interact.hpp>
class CHOPSTICKS : public INTERACTION
{
public:
    REQUEST_LIST       waitList[5]; // Philosophers' Requests have own list
    boolean            inUse[5];    // Remember Chopsticks in use
    CHOPSTICKS_INTERACTION() {for (int n=0; n<5; n++) inUse[n]=FALSE;}
    void Get   (int);
    void Return(int);
};
CHOPSTICKS Chopsticks; // Unique Instance

////////////////////////////////////////////////////////////////////////
class GET_CHOPSTICKS : public REQUEST
{
    int id;
public:
    GET_CHOPSTICKS(int ID)             {id = ID;}
    INTERACTION     *Interaction()     {return &Chopsticks;}
    REQUEST_LIST    *InteractionList() {return &Chopsticks.waitList[id];}
    boolean          Ready()           {return !Chopsticks.inUse[id] &&
                                         !Chopsticks.inUse[(id+1)%5]  ;}
    void             Script()
        {Chopsticks.inUse[id] = Chopsticks.inUse [(id+1)%5] = TRUE;}
};

////////////////////////////////////////////////////////////////////////
class RETURN_CHOPSTICKS : public REQUEST
{
    int id;
public:
    RETURN_CHOPSTICKS(int ID)      {id = ID;}
    INTERACTION  *Interaction()    {return &Chopsticks;}
    boolean       Ready()          {return TRUE;};
    void          Script();
};

void RETURN_CHOPSTICKS::Script()
{
    Chopsticks.inUse[id] = Chopsticks.inUse [(id+1)%5] = FALSE;
    Chopsticks.waitList[(id+1)%5].ProcessRequests();
    Chopsticks.waitList[(id+4)%5].ProcessRequests();
}
```

Figure 3: Dining Philosophers

```
void CHOPSTICKS::Get   (int n) {   GET_CHOPSTICKS(n).Perform();}
void CHOPSTICKS::Return(int n) {RETURN_CHOPSTICKS(n).Perform();}
//////////////////////////////////////////////////////////////////////////
class PHILOSOPHER : public PROCESS
{
    static int n;     // Philosopher Counter
public:
    int  id;          // Philosopher's Id
    int  c;           // Count How many times the philosopher ate.
    PHILOSOPHER()         {id=n++; c=0; Schedule();}
    void Think()          {     Delay(TIME_F(3));}
    void Eat()            {c++; Delay(TIME_F(2));}
    void Script();
};
int PHILOSOPHER::n = 0;

void PHILOSOPHER::Script()
{
    for (;;)
    {
        Think();
        Chopsticks.Get(id);
        Eat();
        Chopsticks.Return(id);
    }
}
//////////////////////////////////////////////////////////////////////////
void MAIN::Script()
{
    PHILOSOPHER p[5];            // Create 5 Philosophers
    Delay(TIME_F(30.0));         // Let run for some duration ... 30 seconds
    for (int n=0; n<5; n++)      // Report results
        cout << "Philosopher " << n << " ate " << p[n].c << " times.\n";
}
```

Figure 4: Dining Philosophers (cont.)