Construction and Use of a Simulation Package in C++*

M. C. Little† and D. L. McCue‡

†*Department of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK*

‡*Xerox Corporation, Webster, New York 14580*

Abstract

We have designed and built a simulation package in C++ which provides discrete process based simulation similar to SIMULA's simulation class and libraries. Inheritence was used throughout the design to an even greater extent than is already provided by SIMULA. This has allowed us to add new functionality without affecting the overall system structure, and hence provides for a more flexible and expandable simulation package. This paper describes the class hierarchy which we have created, and indicates how it can be used to further refine the simulation package. An example of the use of the simulation package is presented along with the results obtained.

Keywords: simulation, object-orientation, C++, SIMULA

---

* The software to be described in this paper is available via anonymous ftp from arjuna.ncl.ac.uk

1. Introduction

This paper describes the steps which we have taken when designing and building a simulation package in C++[Stroustrup 86]. Our package (C++SIM) provides discrete process based simulation similar to that provided by SIMULA [Birtwhistle 73][Dahl 70] and has been used in the work presented in [McCue 92]. Based on the facilities provided in SIMULA, our simulation environment provides active objects (instances of C++ classes) as the units of simulation using the type-inheritence facilities of C++ to convey the notion of "activity". Inheritance was used throughout the design of the simulation package to even a greater extent than is already provided in SIMULA. For example, our I/O facilities, random number generators and probability distribution functions are entirely object-oriented, relying on inheritance to specialize their behaviour. Hence, the addition of new functionality (e.g., new random number generators) can be provided with little effect on the overall system structure.

Using this framework, existing classes can also be replaced as long as they conform [Black 86] to the original class definition: for example, to enable each object within the simulation to possess an independent thread from that which created it, we have made use of Sun Microsystem's lightweight process (thread) package; however, this package has been added to our simulation class hierarchy through an abstract class definition so that other lightweight process packages can be used instead with very little modification. The paper describes the class hierarchy which we have created, and indicates how it can be used to further refine the simulation package.

2. The Simulation Library

2.1. The C++ Abstract Threads Interface

In keeping with the C++ programming model classes obtain the *thread* characteristic, necessary to convey the notion of "activity" within the simulation environment, by inheriting an appropriate base class (in simulation terms they become *processes*). There is a minimum functionality which we require from any threads library that may be used for the simulation package, and to enforce this all classes which provide the abstraction of threads must be derived from the `Thread` base class. This base class provides the definitions of the operations which must *at least* be provided by the deriving class: we use *pure virtual functions* to enforce this rule (C++ ensures that such functions must be defined by a deriving class before an instance of the class can be declared):

```
class Thread
{
public:
    virtual void Suspend() = 0;    // pure virtual function
    virtual void Resume() = 0;
    virtual void Body() = 0;
    virtual long Current_Thread() = 0;

    virtual long Identity();
    static Thread* Self();
};
```

When defined, the Suspend and Resume methods will give thread package specific ways of suspending and resuming execution of a thread respectively.

Body represents the controlling code for each object, i.e., the scope within which the controlling thread will execute.

Current_Thread must be defined by the derived class as it returns the identity of the currently executing thread, which is specific to the thread package used.

The implementations of the operations Identity and Self are provided by the base class because some threads packages do not provide similar functionality: Identity returns the unique identity of the thread associated with the given objects, and Self returns the currently executing thread. Because Self is a *static* member function it can be invoked without creating an instance of the Thread class, i.e., using Thread::Self().

2.1.1. Specific Thread Class Implementations

At the time of writing, we have two threads packages available to us: Sun's own lightweight process package, and the GNU threads library. We have derived two classes from the Thread base class and these respective threads packages. For example, user classes which require separate threads of control using the Sun thread package can be derived from the LWP_Thread class shown below:

```
class LWP_Thread : public Thread
{
public:
    virtual void Suspend();
    virtual void Resume();
    virtual void Body() = 0;

    virtual long Current_Thread();

    thread_t Thread_ID();

    static void Initialize();

protected:
    static const int MaxPriority;
    LWP_Thread(int priority = MaxPriority);
};
```

The MaxPriority constant represents the maximum priority at which a thread may execute (by default all threads derived from this class execute at this priority). All of the pure virtual functions declared in Thread are defined by this class, except Body, which must be defined by the deriving class.

Initialize is used to initialize the threads package prior to use.

Thread_ID returns more detailed (package specific) information on the associated thread.

2.2. The Simulation Scheduler

As in SIMULA, simulation processes (entities) execute according to their simulation time, which is typically drawn from an appropriate distribution function. Only one process executes in any instance of real time, but many processes may execute at any instance of simulation time. Those processes which are currently inactive are placed on to a simulation queue, which is arranged in increasing order of simulation time.

To co-ordinate the execution of these processes, there is a simulation scheduler which manages the simulation queue: when no process is currently active, the scheduler selects a process to run from the head of the queue and (re-) activates it. When no processes are left to execute, i.e., the queue is empty, the simulation ends.

To improve the efficiency of our scheduling algorithm, the simulation queue is organised as a tree, with nodes (processes) at the same level of the tree possessing the same simulation time, as shown in figure 1:

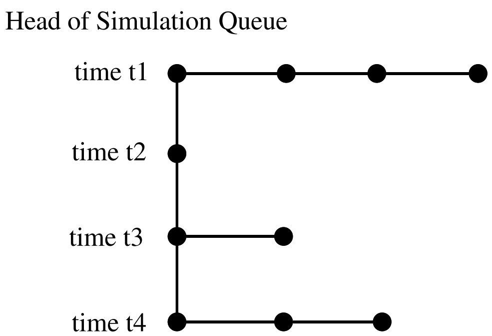

Figure 1: Simulation Queue

Because the scheduler manages processes in the simulation environment it cannot itself be a process. Like the `main` thread to be described later, it is a priority thread within the environment and as such must be controlled in a slightly different manner to other simulation entities. The structure of the scheduler is extremely simple and is shown below:

```
class Scheduler : public LWP_Thread
{
public:
    Scheduler ();
    ~Scheduler ();

    void Body ();
    double CurrentTime ();
};
```

Every simulation application *must* start one scheduler before the simulation can begin. The example to be described in Section 3 will illustrate this.

2.3. Simulation Processes

To become a *process* in the simulation environment, the class must be derived from the `Process` base class. The `Process` class provides all of the operations required by the simulator (the scheduler and other simulation processes) to control the execution of all of the processes in the simulation.

At any point in time, a process can be in one (and only one) of the following states:

- *active*: the process is at the head of the queue maintained by the scheduler and its actions are being executed.
- *suspended*: the process is on the queue maintained by the scheduler, scheduled to become active at a specified time in the future.
- *passive*: the process is not on the scheduler's queue. Unless another process brings it back on to the list, it will not execute any further.
- *terminated*: the process is not on the scheduler's queue and has no further actions to execute.

```
class Process : public LWP_Thread
{
public:
    virtual ~Process ();

    static double CurrentTime ();

    void ActivateBefore (Process&);
    void ActivateAfter (Process&);
    void ActivateAt (double AtTime = CurrentTime());
    void ActivateDelay (double AtTime = CurrentTime());
    void Activate();

    void ReActivateBefore (Process&);
    void ReActivateAfter (Process&);
    void ReActivateAt (double AtTime = CurrentTime());
    void ReActivateDelay (double AtTime = CurrentTime());
    void ReActivate ();

    void Cancel ();
    double evtime ();
    void set_evtime (double);

    boolean idle ();
    boolean terminated ();

    virtual void Body () = 0;

protected:
    Process ();

    void Hold (double t);
    void Passivate ();
};
```

`idle` returns either TRUE or FALSE depending upon whether the process is currently on the simulation queue.

`terminated` returns either TRUE or FALSE depending upon whether the process is terminated.

The simulation time at which a process is due to be reactivated can be obtained through the `evtime` method and it can be changed by `set_evtime`.

The `Hold` method removes the active process from the head of the event list and schedules it to become active a specified number of time units later.

`Pasivate` removes the currently active process from the event list altogether. If the proceess is to be scheduled again in future another process is required.

`Cancel` removes the process from the simulation queue or simply suspends it indefinitely if it is currently not on the queue (i.e., it is active at present).

There are five ways of activating a process, and similarly five ways to reactivate a waiting process (note that if a process is already scheduled, the reactivate will simply re-schedule the process):

- before another process (`ActivateBefore` and `ReActivateBefore`);
- after another process (`ActivateAfter` and `ReActivateAfter`);
- at a specified (simulated) time (`ActivateAt` and `ReActivateAt`);
- after a specified (simulated) delay (`ActivateDelay` and `ReActivateDelay`);
- activate now (at the current simulated time) (`Activate` and `ReActivate`).

The `CurrentTime` method returns the current simulation time, and is typically used to control action relative to a given time period.

2.3.1. Priority Threads

In the simulation there are two "priority" threads which cannot be derived from the `Process` base class and therefore must be activated and deactivated separately:

- the simulation scheduler: this must be activated using the `Resume` method of the thread base class from which it is derived (e.g., LWP_Thread);
- the thread associated with `main`. To allow other threads to run it is necessary to suspend this thread as it has the highest priority in the system. By making a call to the `Initialize` method of the `Thread` class within the `main` body of the simulation code this thread is added to the thread queue maintained by the `Thread` class. This then allows the `Suspend` method to be invoked on the thread later when it is required to become inactive (using the `Thread::Self()->Suspend()` operation).

2.4. Distribution Functions

In a simulation it is often necessary to specify the distribution functions of various events (e.g., the rate of arrivals of jobs at a processor, or the time between failures for a node). As such we have created a set of classes which provide access to various useful distribution functions. These include: RandomStream, UniformStream, Draw, ExponentialStream, ErlangStream, HyperExponentialStream, and NormalStream. By creating instances of these classes the simulation processes can gain access to the appropriate distribution function.

To illustrate how the distribution functions are derived and to show how further functions could be built, we shall examine the `RandomStream` class (from which all other distribution functions are derived) and the `NormalStream` class.

2.4.1. RandomStream

```
class RandomStream
{
public:
    RandomStream (long MGSeed = 772531L, long LCGSeed = 1878892440L);
    virtual double operator() () = 0;
    double Error ();

protected:
    double Uniform ();

private:
    double MGen ();
    double series[128];
    long MSeed, LSeed;
};
```

The `Error` method returns a chi-square error measure on the uniform distribution function. We experimented with several random number generators before settling on a shuffle of a multiplicative generator† (initialised by the `MGen` method) with a linear congruential generator, which seems to provide a reasonably uniform stream of pseudo-random numbers. The `Uniform` method uses the linear congruential generator based on the algorithm from [Knuth Vol2], and the results of this are shuffled with the multiplicative generator as suggested by [Knuth Vol2] ‡ to obtain a sufficiently uniform random distribution.

---

† The authors would like to thank Professor I. Mitrani for his help in developing the multiplicative generator used in the simulation. It is based on the following algorithm: Y[i+1] = Y[i] * 5^5 mod 2^26, where the period is 2^24, and the initial seed must be odd.

‡ As suggested by Maclaren and Marsaglia.

2.4.2. NormalStream

```
class NormalStream : public RandomStream
{
public:
    NormalStream (double Mean, double StandardDeviation);
    virtual double operator() ();

private:
    double Mean, StandardDeviation;
    double z;
};
```

The `operator()` uses the polar method in [Knuth Vol2][††] to implement the NormalStream by making use of the `Uniform` method of `RandomStream`.

2.5. SIMSET

The simulation package also provides entity and set manipulation facilities similar to those provided by the SIMSET classes of SIMULA. These facilities break down into two classes:

- `Link`: the `Link` class provides elements of a doubly linked list;
- `Head`: the `Head` class maintains doubly linked lists of `Link` elements.

The functionality provided by these classes is the same as that provided by SIMSET. However, for the sake of completeness we shall give a brief description of the methods provided.

2.5.1. Link

```
class Link
{
public:
    virtual ~Link ();

    Link* Suc () const;
    Link* Pred () const;

    Link* Out ();
    void InTo (Head*);

    void Precede (Link*);
    void Precede (Head*);
    void Follow (Link*);
    void Follow (Head*);

protected:
    Link ();
};
```

[††] Due to Box, Muller and Marsaglia

Because it makes no sense to be able to create instances of `Link` objects, the constructor for `Link` is protected: this class must be derived from.

`Suc` and `Pred` return the 'successor' and 'predecessor' of this list element respectively. They return 0 if no such element exists.

`Out` removes this object from the linked list it currently belongs to (if any). `InTo` places this object as the last element in the linked list if the list exists, otherwise it attempts to remove the object from any linked list it may belong to.

If `Precede` is passed another `Link` element (say, *L*) then, if *L* is a member of a linked list this object is placed into the same linked list as *L* immediately preceding it, otherwise the result is the same as for `Out`. If `Precede` is given a `Head` object then the result is the same as `InTo`.

`Follow` has similar action to `Precede`, except that `L.Follow(L1)` inserts *L* immediately after *L1*, and `L.Follow(H)`, where *H* is a `Head` object, places *L* as the first element in *H*.

2.5.2. Head

```
class Head
{
public:
    Head ();
    virtual ~Head ();

    Link* First () const;
    Link* Last () const;

    long Cardinal () const;
    boolean Empty () const;

    void Clear ();
};
```

`First` and `Last` return references to the first and last `Link` objects in the list respectively. If the list is empty then they return 0.

`Cardinal` returns the number of `Link` objects in the list, and `Empty` returns TRUE is the list is empty, FALSE otherwise. `Clear` removes all `Link` objects from the list.

3. Example

Having considered the simulation package, we shall now show how it can be used by looking at an example.

3.1. Job Service Simulation

This example is taken from [Mitrani 82] and simulates a process scheduler for a machine which attempts to execute as many process (jobs) as possible. The machine can only process one job at a time and job requests are queued until the machine can deal with them.

However, the machine is prone to failures, and so jobs started will be interrupted by such failures and delayed until the machine has been repaired (reactivated) at which point it is forced to restart execution from the beginning (i.e., it is placed at the head of the job queue). The main processes within this example are:

- `Arrivals`: this process controls the rate at which `Jobs` arrive at the service (`Machine`);
- `Breaks`: this process controls the availability of the `Machine` by "killing" it and restarting it at intervals drawn from a `Uniform` distribution;
- `Job`: this process represents the jobs which the `Machine` must process;
- `Machine`: this is the machine on which the service resides. It obtains `Jobs` from the job `Queue` for the service and then attempts to execute them. The machine can fail and so the response time for `Jobs` is not guaranteed to be the same;

3.1.1. Arrivals

The `Arrivals` class definition is relatively simple as no operations are invoked on it by any of the other processes involved in the simulation:

```
class Arrivals : public Process
{
public:
    Arrivals (double);
    ~Arrivals ();

    void Body ();

private:
    ExponentialStream* InterArrivalTime;
};
```

The constructor initialises the stream from which the rate of `Job` arrivals is drawn and the destructor simply cleans up before the object is destroyed:

```
Arrivals::Arrivals (double mean)
{
    InterArrivalTime = new ExponentialStream(mean);
}

Arrivals::~Arrivals () { delete InterArrivalTime; }
```

The main body of the `Arrivals` process simply waits for an amount of time dictated by the stream from which the rate of arrivals of `Jobs` is drawn, and then creates another `Job`. This procedure is repeated until the simulation ends.

```
void Arrivals::Body ()
{
    for (;;)                              // inifinite loop
    {
        Hold((*InterArrivalTime)());
        Job* work = new Job();
    }
}
```

3.1.2. Job

Unlike `Arrivals` which is an active entity within the simulation, the `Job` class does not need to be a separate process as it is simply enqueued when it is created and dequeued by the `Machine` when it can be executed. All a given `Job` must do is calculate how long it took to be "processed":

```
class Job
{
public:
    Job ();
    ~Job ();

private:
    double ArrivalTime;
    double ResponseTime;
};
```

Because no operations are invoked on instances of the `Job` class, all of the work is performed by the constructor and destructor:

```
Job::Job ()
{
    boolean empty;

    ResponseTime = 0;
    ArrivalTime = sc->CurrentTime();
    empty = JobQ.IsEmpty();
    JobQ.Enqueue(this);               // place this Job on to the queue
    TotalJobs++;

    if (empty && !M->Processing() && M->IsOperational())
        M->Activate();                // Machine idle as no Jobs in queue
                                      // and not broken
}

Job::~Job ()
{
    ResponseTime = sc->CurrentTime() - ArrivalTime;
    TotalResponseTime += ResponseTime;
}
```

3.1.3. Queue

The jobs which are not being serviced are placed on to a job queue. As with the `Job` class, the instance of the `Queue` class is not required to be active, and as such is not derived from the `Process` class. Because this is simply a standard queue code, we shall not give the implementation.

```
class Queue
{
public:
    Queue ();
    ~Queue ();

    boolean IsEmpty ();                   // returns TRUE if no Jobs in queue
    long QueueSize ();                    // returns number of Jobs in queue
    Job* DeQueue ();                      // returns head of queue
    void Enqueue (Job*);                  // places Job at tail of queue
};
```

3.1.4. Machine

The `Machine` process obtains `Jobs` from the queue and processes them. As it is prone to failures `Jobs` can take extended periods of time to process. Various operations are invoked on the machine by other simulation processes, for example to determine whether or not it has failed:

```
class Machine : public Process
{
public:
    Machine (double);
    ~Machine ();

    void Body ();

    void Broken ();
    void Fixed ();
    boolean IsOperational ();
    boolean Processing ();
    double ServiceTime ();

private:
    ExponentialStream* STime;
    boolean operational;
    boolean working;
};
```

As with the `Breaks` and `Arrivals` processes, the constructor and destructor initialise and delete the stream from which the time taken to process a `Job` is drawn.

```
boolean Machine::Processing () { return working; }

void Machine::Broken () { operational = false; }

void Machine::Fixed () { operational = true; }

boolean Machine::IsOperational () { return operational; }

double Machine::ServiceTime () { return (*STime)(); }
```

```
void Machine::Body ()
{
    for (;;)
    {
        working = true;

        while (!JobQ.IsEmpty())    // continue as long as Jobs are available
        {
            Hold((*STime)());
            Job* J = JobQ.Dequeue();

            ProcessedJobs++;       // keep track of number of completed Jobs
            delete J;              // remove finished Job
        }

        working = false;          // no Jobs in queue so become idle
        Cancel();
    }
}
```

3.1.5. Breaks

The `Breaks` class defines a process which simply waits for a specific period of time before "killing" the `Machine` process. It then waits again before re-activating the machine. Because none of the other simulation processes are required to invoke operations on the `Breaks` process, the class definition is relatively simple:

```
class Breaks : public Process
{
public:
    Breaks ();
    ~Breaks ();

    void Body ();

private:
    UniformStream* RepairTime;
    UniformStream* OperativeTime;
    boolean interrupted_service;
};
```

The constructor and destructor simply initialise and delete the streams used by the `Breaks` process respectively.

The main body of the process activates and deactivates the `Machine` process:

```
extern Machine* M;     // This is the machine used to service requests
extern Queue JobQ;     // This is the queue from which Jobs are drawn

void Breaks::Body ()
{
    for (;;)
    {
        Hold((*OperativeTime)());
        M->Broken();               // de-activate the Machine
        M->Cancel();               // remove Machine from Scheduler queue

        if (!JobQ.IsEmpty())
            interrupted_service = true;

        Hold((*RepairTime)());
        M->Fixed();                // re-activate the Machine
        if (interrupted_service)
        {
            interrupted_service = false;
            M->ActivateAt(M->ServiceTime() + CurrentTime());
        }
        else
            M->ActivateAt();
    }
}
```

3.1.6. MachineShop

The `MachineShop` class is the core of the simulation: it starts up all of the main processes involved, and when the simulation ends it prints out the results.

```
class MachineShop : public Process
{
public:
    MachineShop ();
    ~MachineShop ();

    void Body ();
    void Await ();
};
```

The `Body` method starts up the other processes, e.g., the `Machine`, and then waits until the number of processed Jobs is at least 100000:

```
void MachineShop::Body ()
{
    sc = new Scheduler();      // create the simulation queue scheduler
    Arrivals* A = new Arrivals(10);
    M = new Machine(8);
    Job* J = new Job;
    Breaks* B = new Breaks;

    // activate the relevant simulation processes

    B->Activate();
    A->Activate();
    sc->Resume();          // start up the scheduler - it is not a process

    while (ProcessedJobs < 100000)
        Hold(10000);

    cout << "Total number of jobs processed " << TotalJobs << endl;
    cout << "Total response time " << TotalResponseTime << endl;
    cout << "Avge response " << (TotalResponseTime/ProcessedJobs) << endl;
    cout << "Avge number jobs present " << (JobsInQueue/CheckFreq) << endl;

    // end simulation by suspending processes

    sc->Suspend();
    A->Suspend();
    B->Suspend();
}
```

Note that we do not need to explicitly need to activate `Machine` as the `Breaks` or `Jobs` process will do this for us.

The `Await` method suspends the thread associated with `main`, thus allowing the other simulation threads to execute:

```
void MachineShop::Await()
{
    Resume();
    Thread::Self()->Suspend();
}
```

3.1.7. Main

The `main` part of the simulation code initializes the various thread specific variables used (e.g., the maximum priority of a thread), creates the main body of the simulation code (in this case `MachineShop`) and then suspends the thread associated with `main`:

```
void main ()
{
    LWP_Thread::Initialize();

    MachineShop m;
    m.Await();     // Suspend main's thread (NOTE: this MUST be done by all
                   // applications).
}
```

Conclusions

From the outset we endeavoured to provide a simulation package which provided similar functionality to that of SIMULA, as this has proved so successful in fulfilling the needs of users over many years. From our experiences of using SIMULA, both as a general programming language and as a simulation tool, we believe that we have been successful. As a result of using C++ we also believe that we have obtained several advantages over the use of SIMULA, for example:

- performance - it is our experience that C++ compilers typically generate code which is several times more efficient than similar SIMULA code, and as a result, simulations execute correspondingly faster;

- C++ provides more extensive object-oriented features than SIMULA, allowing, for example, class instance variables to be either publicly available or only privately available. In SIMULA, everything is public, affecting the way code is written and providing extra problems of debugging.

The results of the simulation package are encouraging and we intend to develop it further in light of our continued experience.

Acknowledgements

We would like to thank Professor Isi Mitrani for the help he has given us in the development of this simulation package and the time he has devoted to listening to our thoughts and problems. We would also like to than Ron Kerr for his help with the SIMULA language, and Dr Graham Parrington for his comments on drafts of this paper. The work reported here has been supported by SERC/MOD Grant GR/H81078 and Esprit Broadcast (Basic Research Project Number 6360).

References

[Birtwhistle 73]
G. M. Birtwhistle, O-J. Dahl, B. Myhrhaug, K. Nygaard, "Simula Begin", Academic Press, 1973.

[Black 86]
A. Black et al, "Object Structure in the Emerald System", Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, October 1986.

[Dahl 70]
O-J. Dahl, B. Myhrhaug, K. Nygaard, "SIMULA Common Base Language", Norwegian Computing Centre, Technical Report S-22, 1970.

[Knuth Vol2]
Knuth Vol2, Seminumerical Algorithms, Addison-Wesley, p. 117.

[McCue 92]
D. L. McCue and M. C. Little, "Computing Replica Placement in Distributed Systems", Proceedings of the 2nd Workshop on the Management of Replicated Data, November 1992, pp. 58-61.

[Mitrani 82]
I. Mitrani, "Simulation Techniques for Discrete Event Systems", Cambridge University Press, Cambridge, 1982, p. 22.

[Sedgewick 83]
R. Sedgeqick, "Algorithms", Addison-Wesley, Reading MA, 1983, pp. 36-38.

[Stroustrup 86]
B. Stroustrup, "The C++ Programming Language", Addison Wesley, 1986.