Object-Oriented
Discrete-Event
Simulation in C++

UNIVERSITY OF
NEWCASTLE UPON TYNE

C++SIM User's Guide

Public Release 1.5

Draft Version 1.0

Department of Computing Science,
Computing Laboratory,
The University, Newcastle upon Tyne,
NE1 7RU, UK.

Copyright Notice:

Research Funding:

C++SIM was developed as part of the research conducted for the *Arjuna* project.

The *Arjuna* project began in 1985 and is now funded by the UK Science and Engineering Research Council (SERC) grant No. GR/F 06494: The Design of High Performance Distributed Fault-Tolerant Systems, SERC grant No. GR/F 38402: Fault-Tolerant Multi-processor Systems and ESPRIT Project 2267: Integrated Systems Architecture (ISA) through a subcontract monitored by APM Ltd., Cambridge.

Table of Contents

1. Preface

C++SIM was developed as a direct consequence of research conducted for the *Arjuna* project [10]. *Arjuna* provides a set of tools for the development of fault-tolerant distributed applications in C++, using atomic actions and replication of objects. As part of the development of *Arjuna*, the *Replica Management System* [5][7] was designed which allows dynamic reconfiguration of object replica groups based upon changes in the characteristics of the underlying distributed system and the objects being replicated. To determine the best policies to use for this reconfiguration (e.g., the number and placement of replicas, based upon the desired quality of service), it was necessary to simulate them. Since *Arjuna* is written in C++ it was decided to write this simulation in C++ as well, as this would facilitate transferring the finished system to *Arjuna*.

1.1. Design Decisions

When designing *C++SIM* the following requirements were identified:

- *easy to learn and use*: the interface to the simulation library should be easy to understand.
- *correct abstraction*: existing C++ programmers should not find the simulation paradigm in conflict with the programming paradigm presented by C++. Simulation programmers used to other environments should find the transition to *C++SIM* straightforward.
- *flexible and extensible*: it should be relatively easy for anyone to add new functionality to the system, such as new distribution functions.
- *efficiency*: the system should be efficient and produce efficient simulation runs. Simulation packages which we have experience of tended to be extremely slow and consume large amounts of system resources.
- *portable*: it should be possible to build the system on a range of different architectures and operating systems. It should be possible to write simulations in *C++SIM* which are architecturally neutral. In addition it should be possible to interface other packages/libraries to the simulation system.

These requirements were realised in the following design decisions:

- the discrete-event process based simulation facilities provided by SIMULA [1][2] and its simulation classes and libraries have a considerable experience and user community which have found them to be successful for a wide variety of simulations. In later versions of the system additional simulation classes were added which provide extra functionality.[1]
- inheritance was to be used throughout the design to even a greater extent than is already provided in SIMULA. This certainly enable *C++SIM* to be more flexible and extensible, allowing new functionality to be added without affecting the overall system structure. For example, our I/O facilities, random number generators and probability distribution functions are entirely object-oriented, relying on inheritance to specialise their behaviour.
- No changes to the language or the operating system should be made, as this would affect portability. Because *C++SIM* is written in standard C++, this facilitates the ability to interface applications written in *C++SIM* with other C++ libraries. For example, we can use the *InterViews*[2] or *Tcl*[3] GUIs to easily build graphical simulations.

---

[1]It is not necessary for the reader to know anything about the SIMULA programming language or its simulation classes, but such knowledge would aid in the understanding of the concepts and classes presented within.

[2]InterViews is a trademark of ...

- The only architectural specific component of *C++SIM* is the threads package it uses to achieve the abstraction of active objects. However, by interacting with the threads package through a suitable abstract interface, portability has been achieved. Applications can be written in *C++SIM* without the programmer having to consider such issues as the architecture it will run on, the compiler, or the threads package.
- After having used both C++ and SIMULA 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. We have attempted to make those core components of *C++SIM* as efficient as possible, in terms of speed and resource utilisation. The fact that these components are written in C++ means that new implementations can be derived from them, perhaps tailored for particular applications.

1.2. Future Developments

Further modifications to *C++SIM* are under development. One of the possibilities we are investigating is making use of various facilities provided by *Arjuna* to enable us to build distributed simulations. Distribution transparency would be provided by *Arjuna*, and programmers could develop applications without having to consider whether or not they will eventually be distributed.

Further ports of the system to other architectures are being planned. For example, there is a requirement for *C++SIM* to be available on non-UNIX[4] platforms, most notably the PC and its compatibles. As such, *C++SIM* has been ported to the *Windows-NT*[5] and *OS/2*[6] operating systems.

Graphical tools for building and viewing simulations have been written in various GUIs and for a variety of systems. These will be included in future releases.

[3]Tcl is a trademark of ...

[4]UNIX is a trademark of AT&T.

[5]Windows-NT is a trademark of Microsoft Corporation.

[6]OS/2 is a trademark of IBM and Microsoft Corporation.

2. Introduction

This manual is not intended as a tutorial on the concepts of simulation in general, but rather how to write simulations in the *C++SIM* system. However, in order to be able to do this certain key simulation concepts will be briefly described. The interested reader is referred to [6] for detailed descriptions of these concepts and for further discussions on simulation modelling.

2.1. Simulation Models

To model a system is to replace it by something which is:

- simpler and/or easier to study.
- equivalent to the original in all *important* respects.

Therefore, before constructing the actual simulation, it is first necessary to abstract from the real system those components and their interactions that are considered important for the actual model. Building a simulation system model involves making certain simplifying assumptions to aid in the actual implementation and study of the simulation (without such simplifications the model would be as complex as the system it is meant to be simulating). However, the accuracy of the results obtained from the simulation depend upon how valid the initial assumptions are. For example, when considering the trajectory of a projectile through the atmosphere, the friction due to the air molecules is usually ignored. This assumption is valid only within certain boundaries: if the size of the projectile is on the same scale as the air molecules and its speed is sufficiently small then friction plays a significant role in its movement.

Thus, the first step towards building a simulation model of a system is to determine exactly what are the important features which are to be measured, and what characteristics of the system have an affect on them. Any boundary conditions for the simulation (e.g., size of projectile) should be considered at the same time in order to simplify this procedure. Building a final model can often take several phases, where results from the initial model are compared with those obtained from the real system to determine their accuracy. Any discrepancies are taken into account by possibly adding new components to the simulation until, within certain error boundaries, the simulation results match those from the real system.

2.2. Terminology

The system components chosen for the simulation are termed simulation *entities*. Associated with each entity in the simulation are zero or more *attributes* that describe the state of the entity and which may vary during the course of the simulation. The interaction of entities and the changes they cause in the system state are termed *events*.

The collection of these component attributes at any given time *t* defines the system state at *t*. In general, the system state can take any of a variety of values, and a given simulation run results in one realisation of a set of these values (the *operation path*) over the *observation period*.

2.3. Categories of Simulation Models

There are three categories of simulation model, described by the way in which the system state changes as a function of time:

- C*ontinuous time:* is one whose state varies continuously with time; such systems are usually described by sets of differential equations.
- *Discrete time*: the system is considered only at selected moments in time (the *observation points*). These moments are typically evenly spaced. Some economics models are examples of this, where economics data becomes available at fixed intervals. Changes in state are noticed only at observation points. By choosing a suitably small interval between observation points, a continuous time simulation can be approximated by a discrete time simulation.
- *Continuous time-discrete event*: the time parameter is (conceptually) continuous and the observation period is a real interval, usually starting at zero for simplicity. The operation path is completely determined by the sequence of event times (which need not be evenly spaced and can be of arbitrary increments) and by the discrete changes in the system state which take place at these times (i.e., the interactions of the events). In between consecutive event times the system state may vary continuously. Although it is possible to model the passage of real time by suitable event timing, this is not necessary for a discrete event system: the simulation model can advance its own internal time directly from one discrete event to another, taking any appropriate action to advance the state accordingly.

It is this latter category of simulation modelling that *C++SIM* supports. Examples of discrete-event simulations are most queuing problems: entities (e.g., customers in a bank) arrive according to a given distribution and change the system state instantaneously (e.g., the number of customers in the queue). The operation paths for this system are step functions: they jump up (or down) by one when a customer joins (or leaves) the queue.

2.4. Event Scheduling

Given that a simulation consists of a series of interacting events (the operation path), a simulator can be defined as that program devoted to the generation of operation paths. The simulator allows the creation of events and controls their interactions according to a set of rules, using an internal "clock" to keep track of the passage of (simulation) time.

It maintains an *event list*, which indicates which events are to be scheduled for execution at specific simulation times. Events are executed according to their simulation times. There are two approaches to the way in which a simulator can schedule events to produce an operation path:

- *event-oriented*: there is a procedure associated with each type of event in the system: it performs the action required to handle that type of event and it is invoked every time such an event occurs. In an event-oriented approach, an operation path is obtained by taking a global view of everything that happens in the system; the manipulation of events is explicit.
- *process-oriented*: an operation path is obtained by the interacting of a number of processes running in parallel. The management of events is implicit in the management of the processes. The simulation system provides primitives for placing processes at particular points on the event list, removing and re-scheduling them.

The process-oriented approach best fits with the object-oriented paradigm which we want to present to the programmer of *C++SIM*. As in SIMULA, simulation processes then become active objects which interact with each other through message passing and the simulation primitives. Refinements of these objects can then be obtained by inheriting from them and redefining the appropriate methods.

3. Active C++ Objects

In a process-oriented simulator it is necessary to be able to convey the notion of activity to the processes involved in the simulation. *C++SIM* does this by using *active objects*. An active object is one which has an independent thread of control associated with it at creation time. This chapter will describe the classes necessary to create and use active objects in C++.

Note: these active objects can be used without the simulation component of *C++SIM*.

3.1. Threads

C++SIM uses *threads* (*lightweight processes*) to create active objects. The class `Thread` is the base class from which all thread specific classes must be derived in order to ensure they provide at least the minimum functionality required by *C++SIM*. The following section will describe this class in detail. Section 3.2. describes additional functionality which must be provided by these derived classes, and which users of *C++SIM* must be aware of before writing applications involving active objects.

The `Thread` class maintains a list of every threaded object in the application; whenever a new threaded object is created it is automatically added to this list, and is removed when it is goes out of scope. In *C++SIM* these objects will typically be simulation processes, but for every simulation there will be at least two threaded objects which do not fall into this category: the simulation scheduler, and the main system thread. These will be described in detail in later sections.

3.1.1. Thread Class Interface

The `Thread` class interface is shown below. As mentioned above, this represents the minimum functionality that is required by *C++SIM* from any threads package. To enforce this, a thread class is written for each specific thread library, and must be derived from `Thread`. This base class provides the definitions of those operations which must *at least* be provided by the deriving class: *pure virtual functions* are used to enforce this rule.

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

    virtual long Current_Thread () const = 0;

    virtual long Identity () const;

    static Thread *Self ();

    virtual ostream& print (ostream&) const;

protected:
    Thread ();
    virtual ~Thread ();

    long thread_key;
};

extern ostream& operator<< (ostream& strm, const Thread& t);
```

Because *C++SIM* was designed to support a variety of thread implementations, it was a requirement that applications could be written in a thread-independent manner. This enables such applications to be moved from one configuration of *C++SIM* to another without changes to the code. Therefore, although *C++SIM* must know which thread implementation class (derived from `Thread`) is being used for a specific configuration, this information is hidden from the user by the `Thread_Type` macro. This macro is defined to point to the actual thread implementation when *C++SIM* is built. Portable applications which use threads should manipulate them only through this macro, with the exceptions described below.

The methods `Suspend()` and `Resume()` must be defined by the deriving class, and suspend and resume threads respectively.

Active objects are derived from the thread specific class (through the `Thread_Type` macro), and the thread is created when they are instantiated. `Body()` represents the code within which the thread executes, and *must* be defined by one of the deriving classes. The thread begins execution at the start of `Body()`, and is deleted if it ever returns from this method. However, because of restrictions imposed by certain threads packages, this should be prevented if code is intended to operate on different configurations of *C++SIM*.

All threads in the application are uniquely identified by a `long` key. `Current_Thread()` returns this key for the currently active thread. Because this is specific to a given thread package this must be defined by the thread class derived from `Thread`.

`Identity()` returns the key of the object it is invoked on. If this object is the currently active object then `Identity()` is equivalent to `Current_Thread()`.

In a general active object environment it may be necessary to obtain a reference to the currently active object/thread. However, some thread packages do not provide a means to obtain the currently active thread. Therefore, `Thread` provides this basic functionality through `Self()`. Because `Self()` is *static* it can be invoked without obtaining an instance of the class.

The `print` method and the overloaded `operator<<` make it possible to print information on each threaded object.

3.2. Important Thread Specific Methods

Although Chapter 5 describes how to create new thread specific classes, it is necessary for users of *C++SIM* to have some knowledge of basic functionality which must be provided by them. This functionality (accessed through `Thread_Type` for portability), concerns initialising the threads packages, suspending and resuming the main thread, and exiting a threaded application.

To illustrate this, we shall consider the thread class `XThread`, shown below:

```
class XThread : public Thread
{
public:
    virtual void Suspend ();
    virtual void Resume ();

    virtual void Body () = 0;

    virtual long Current_Thread () const;

    static void Initialize ();
    static void Exit (int = 0);
    static void mainResume ();
};
```

Apart from the `Thread` methods which must be defined by `XThread`, the additional methods are:

- `Initialize()`: this method, which is responsible for any initialisation code required by the thread package, *must* be invoked prior to the creation of the first application thread. This method is also responsible for adding the main application thread to the list maintained by `Thread`. This allows it to then be suspended and resumed as any active object (calling `Thread::Self()->Suspend()` while within the main thread).
- `Exit(int = 0)`: some threads packages do not allow the use of system `exit` to terminate applications. This method provides a thread specific means to achieve the same functionality. The parameter is the value returned by the application when it terminates.
- `mainResume()`: although the main thread can be suspended as shown above, unless the application code has retained a reference to its threaded object resuming it can only be performed through this method.

3.3. Using Active Objects

3.3.1. Example

4. Basic Simulation Classes

This chapter describes the core elements of the *C++SIM* simulation system. It is not intended as a tutorial on C++[8][9], and basic knowledge of the language is assumed.

4.1. The Simulation Scheduler

Chapter 2 described the event list and how simulation entities (processes) are executed according to their position on the event list (i.e., with increasing simulation time). In *C++SIM*, as in SIMULA, simulation processes are managed by a *scheduler* and are placed on a *scheduler queue* (the event list). Processes are executed in pseudo-parallel, i.e., only one process executes at any instance of real time, but many processes may execute concurrently at any instance of simulation time. The simulation clock is only advanced when all processes have been executed for the current instance of simulation time.

Inactive processes are placed on to the scheduler queue, and when the current active process yields control to the scheduler (either because it has finished or been placed back onto the scheduler queue), the scheduler removes the process at the head of the queue and re-activates it.[7] When the scheduler queue is empty, i.e., there are no further processes left to execute, the scheduler terminates the simulation.

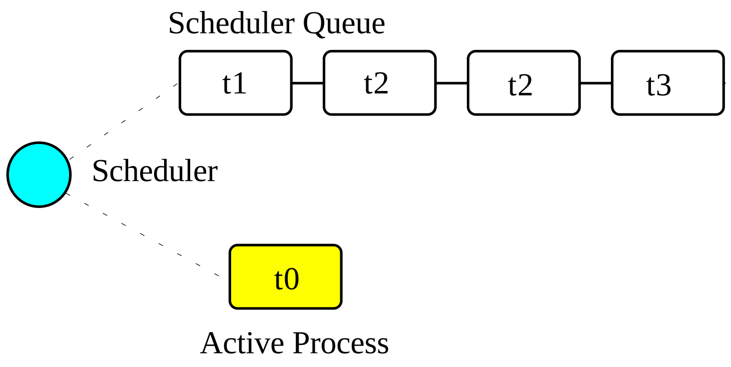

Figure 1: Scheduler-Process Interaction

As Figure 1 shows, the scheduler co-ordinates the entire simulation run, effectively monitoring the active and passive processes to enable it to determine when, and which, process to activate next. A simulation application cannot affect the scheduler directly, but can do so only indirectly through modifications of the scheduler queue.

Note: the scheduler queue can be structured in a variety of ways, including a linear list or a tree. The implementation of the queue can depend upon the type of simulation being conducted. For example, a simulation which involves many (concurrent) processes would suffer from using a linear ordered queue which would typically have insertion and removal routines with overheads proportional to the number of entries in the queue. However, a linear list may work best for a low number of simulation

[7]In SIMULA the currently active process is not removed from the head of the queue.

processes. *C++SIM* comes with a suite of scheduler queue implementations which can be chosen when the system is built.[8]

4.1.1. Scheduler Class Interface

The scheduler is an instance of the `Scheduler` class. It is the responsibility of the application programmer to ensure that only a single instance of this class is created.

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

    void Body ();
    double CurrentTime () const;

    void reset () const;
};
```

The scheduler maintains the simulation clock, and the current value of this clock is obtained by invoking the `CurrentTime()` method.

To enable multiple simulation runs to occur within a single application, it is possible to reset the scheduler and simulation clock by calling the `reset()` method. This causes the scheduler to remove all processes currently registered on the scheduler queue and to invoke a class specific method on each of them which also resets their states (detailed in the next section). Once this is finished the simulation is ready for an additional run.

4.2. Simulation Processes

As was described in the previous chapter, *C++SIM* supports the process-oriented approach to simulation, where each simulation entity can be considered a separate process. Therefore in *C++SIM* the entities within a simulation are represented by *process objects*. These are C++ objects which have an independent thread of control associated with them at creation time, allowing them to convey the notion of activity necessary for participating in the simulation.

In keeping with the object-oriented paradigm, and to make development of process objects simpler, classes inherit the process functionality from the appropriate base class (`Process`). This class defines all of the necessary operations for the simulation system to control the simulation entities within it, and for them to interact with it and each other.[9]

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

- *active*: the process has been removed from the head of the scheduler queue and its actions are being executed.
- *suspended*: the process is on the scheduler queue, scheduled to become active at a specified simulation time.

[8]The characteristics of each implementation are illustrated in the graph which accompanies the system. The ability to change the scheduler implementation without rebuilding *C++SIM* will be available in later versions of the system.

[9]At present only some of the methods from class `Process` are virtual, but in later versions many of these will be redefined to enable easy modification by derived classes.

- *passive*: the process is not on the scheduler queue. Unless another process brings it back on to the queue it will not execute any further actions.
- *terminated*: the process is not on the scheduler queue and has no further actions to execute. Once a process has been terminated it cannot be made to execute further in the same simulation run.

A process which is either active or suspended is said to be *scheduled*.

4.2.1. Process Class Interface

The `Process` class definition is shown below. Before considering how to build an example class derived from `Process` we shall discuss the methods which it provides.

Because the constructors are protected, it is not possible to create an instance of the `Process` class, i.e., classes must be derived from this. Processes are threaded objects, and typically each thread package schedules execution of threads according to a priority. By default, all processes in *C++SIM* are created with the same priority, but this can be altered by the `priority` parameter. In addition, the size of the stack allocated for each thread can be modified by the `stackSize` parameter.

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

    void Activate ();
    void ActivateBefore (Process &);
    void ActivateAfter  (Process &);
    void ActivateAt     (double AtTime = CurrentTime(), Boolean prior = FALSE);
    void ActivateDelay  (double AtTime = CurrentTime(), Boolean prior = FALSE);

    void ReActivate ();
    void ReActivateBefore (Process &);
    void ReActivateAfter  (Process &);
    void ReActivateAt     (double AtTime = CurrentTime(), Boolean prior = FALSE);
    void ReActivateDelay  (double AtTime = CurrentTime(), Boolean prior = FALSE);

    double evtime () const;
    const Process* next_ev () const;
    static const Process* current ();

    static double CurrentTime ();
    double Time () const;

    void Cancel ();
    virtual void terminate ();

    Boolean idle () const;
    Boolean passivated () const;
    Boolean terminated () const;

    virtual void Body () = 0;

    virtual void reset ();

protected:
    Process ();
    Process (int priority);
    Process (int priority, unsigned long stackSize);

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

There are five ways to activate a currently *passive* process, which results in it being brought to the correct position in the scheduler queue corresponding to its associated simulation time. If this is the head of the queue then it will become the active process.

- `Activate()`: this activates the process at the current simulation time.
- `ActivateBefore(Process& proc)`: this positions the process in the scheduler queue before `proc`, and gives it the same simulation time. If `proc` is not present then an error message is displayed.[10]
- `ActivateAfter(Process& proc)`: this positions the process in the scheduler queue after `proc`, and gives it the same simulation time. If `proc` is not present then an error message is displayed.
- `ActivateAt(double AtTime, Boolean prior)`: the process is inserted into the scheduler queue at the position corresponding to the simulation time specified by `AtTime`. The default for this time is the current simulation time. The `prior` parameter is used to determine whether this process should be inserted before or after any processes with the same simulation time which may already be present in the queue.
- `ActivateDelay(double AtTime, Boolean prior)`: the process is activated after a specified delay (`AtTime`). The process is inserted into the queue with the new simulation time, and the `prior` parameter is used to determine its ordering with respect to other processes in the queue with the same time.

There are correspondingly five `ReActivate` methods, which work on either passive or scheduled processes. These will not be described in detail as they have similar signatures to their `Activate` counterparts and work in the same way.

`Hold(double period)` schedules the currently active process for re-activation after the simulated delay of `period` time. If this is invoked by the object (e.g., through a publicly available method) when it is not the current active process then it does nothing.

`evtime()` returns the time at which the process is scheduled for activation.

`next_ev()` returns a reference to the next process to be scheduled for execution. If the queue is empty then `NULL` is returned. Note that this is a constant pointer, and should not be modified by caller.

The *static* method `current()` returns a reference to the currently active process. Note that this is a constant pointer, and should not be modified by caller.

The current simulation time can be obtained by using either the `CurrentTime()` or `Time()` methods. The former method is *static* and as such can be invoked without an instance of the `Process` class.

`Cancel()` removes the process from the scheduler queue or suspends it if it is the currently active process. In either case, the process is set to the *passive* state. `Passivate()` functions similarly but only works on the currently active process, i.e., if it is invoked by the object (e.g., through a publicly available method) when it is not the current active process then it does nothing.

[10]These methods will eventually return `Boolean` values to indicate success or failure.

`terminate()` removes the process from the scheduler queue or it is suspended if it is currently active. The process is then set to the *terminated* state, and can take no further part in this simulation run.

`idle()` returns `FALSE` if the process is either active or scheduled to become active. Otherwise `TRUE` is returned.

`passivated()` and `terminated()` indicate whether the process is in the *passive* or *terminated* state, respectively.

Section 4.1.1. described how it is possible to reset a simulation to enable multiple simulation runs to occur within a single application. When the scheduler object's `reset()` method is invoked it removes all scheduled processes from the queue and calls their `reset()` methods, which can perform class specific actions to enable the object to be re-initialised for the subsequent simulation run. By default, this *virtual* method does nothing unless debugging is enabled in which case a suitable message is output to show that it has been called.

The independent thread of control associated with a `Process` object executes within the context of a procedure (method) in the same way that an operating system process uses `main`. This method is `Body()` which can only be defined by one of the classes which derives from `Process`. If this method ever returns then the thread is destroyed. However, because of restrictions imposed by certain thread packages, this should be prevented: the `terminate()` method *must* be used instead.

4.2.2. Example

To illustrate how a simulation process could be implemented from the `Process` class we shall consider the example of a queue of customers arriving at a bank. For this example, this involves three classes:

- `Customer`: instances of this class represent the customers in the queue.
- `Queue`: the instance of this class (`queue`) is the queue into which customers are places.
- `Arrivals`: this is the process which creates new customers for insertion in `queue`.

The implementations of the `Customer` and `Queue` classes are not important to this example. The implementation of the `Arrivals` class could be:

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

    void Body ();
};

void Arrivals::Body ()
{
    for (;;)
    {
        Customer* c = new Customer();
        queue.insert(c);
        Hold(20.0);
    }
}
```

4.3. Initialising the Thread System

Section 3.2. described how certain threads packages require initialisation code to be executed *prior* to the creation of the first application thread. This code must be provided by the `Initialize()` method of each thread class, but should be invoked using the `Thread_Type` macro.

```
int main (int, char**)
{
    // do anything we want before using threads

    Thread_Type::Initialize();

    // we can now use active objects
}
```

4.4. Starting, Ending and Controlling a Simulation

When a `Process` object is created in *C++SIM* it starts in the *passive* state, and must be activated before it can take part in the simulation. This is typically performed by the first process object to which control is transferred after the simulation is initially started. When writing *C++SIM* applications it is typical for the main thread to create a single *controller process* which is responsible for co-ordinating the entire simulation run. This creates and activates all of the simulation entities and the scheduler, and provides methods for suspending the main thread, thus allowing the controller object to execute, and exiting the application. An example controller interface is shown below, and the implementations for its methods will be described in the following sections:

```
class Controller : public Process
{
public:
    Controller () {};
    ~Controller () {};

    void Body ();

    void Await ();
    void Exit ();
};
```

Because `Controller` is a simulation process itself, it derives from `Process` and defines a `Body()` method, which will do the actual controlling of the simulation. It also provides the following methods:

- `Await()`: this method is called within the main application thread and suspends it, effectively transferring control the `Controller` process.
- `Exit()`: this method is called to exit the simulation.

4.4.1. Suspending the Main Thread

When a threaded application is started it is important to realise that before any application threads are created, the operating system has already created one to run the application. Since a process object's thread executes within `Body()`, then the main system thread can be considered to execute within `main`. This thread, which, in a single-threaded application is the operating system process, must be suspended before any simulation threads can run. (Strictly speaking this is not necessary where threads are prioritised, but we are considering the generalisation).

The `Await()` method of `Controller` is responsible for suspending this thread:

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

It must first resume the thread associated with the `Controller` instance (since `Controller` is a `Process` it starts in the passive state). This thread does not execute until the main thread is suspended; since `Await()` is called from within the main thread, `Thread::Self()` returns a reference to this thread, allowing `Suspend()` to be called on it.

The code for `main` would then become:

```
int main (int, char**)
{
    // do anything we want before using threads

    Thread_Type::Initialize();

    // we can now use active objects

    Controller c;
    c.Await();
}
```

4.4.2. Exiting the Application

Section 3.2. described how it is not possible to use the `exit` system call in some threads packages, and how each thread implementation class must therefore provide an `Exit(int = 0)` method. Although this can be called directly by any object in the simulation, the controller's method can hide any tidy-up code specific to the simulation.

```
void Controller::Exit ()
{
    Suspend();
    Thread_Type::Exit();
}
```

4.4.3. Controlling the Simulation

The controller's body creates and activates the other simulation entities and the scheduler, and controls the overall simulation (e.g., resetting the system between consecutive runs).

```
void Controller::Body ()
{
    sc = new Scheduler();

    // create and activate any other simulation entities

    sc->Resume();  // we must create a scheduler for the simulation to run

    // execute the simulation

    // print results

    sc->Suspend();  // suspend scheduler

    // suspend simulation entities

    Thread_Type::mainResume();
}
```

The final call to `mainResume` prevents `Body()` from exiting, which we must do to ensure the application is portable between thread implementations.

4.5. Resetting a Simulation

Resetting a simulation involves resetting all of the objects involved in it which will be required for subsequent runs. The scheduler calls the `reset` method for each of the process objects on the queue, and this method should restore the object to an initialised state, possibly resetting all non-process objects.

There is a difference between resetting the state of a process object and resetting the thread associated with the object. The former is accomplished by the scheduler and the reset method, but the latter can only be accomplished by code executed within the object when the thread regains control, i.e., when the object is reactivated.[11] Because a reset can potentially occur at any time, it is therefore necessary for a process to be able to determine whether it has been reset so it can take appropriate action, e.g., returning the thread to the start of `Body()`. There are two macros available for this, both based upon the CSP notion of guarded commands: operations are performed and a reset condition can be checked before the object acts upon results returned.

When writing a method body which needs to be concerned with resets, a *reset point* should be chosen: this is the place to which the thread of control will be transferred should a reset be detected. This is marked with the `RESET` label.

If a reset occurs, the `reset()` method should set a flag to an appropriate state. Each of the macros can then check this flag and possibly transfer the thread of control back `RESET`.

- `MONITOR_RESET(x,y,z)`: statement `x` is executed and upon completion statements `y` and `z` and compared. If they are equal then control is returned to `RESET`.
- `CHECK_RESET(y,z)`: statements `y` and `z` are compared, and if they are equal control is returned to `RESET`.

[11]This is a result of *C++SIM* being able to support multiple thread packages and not using its own, where direct control of thread program counters etc. would be possible.

4.5.1. Example

If we take the `Arrivals` example above and add a reset method then the code could be:

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

    void Body ();
    void reset ();

private:
    Boolean _reset;
};

void Arrivals::Body ()
{
RESET:

    _reset = FALSE;

    for (;;)
    {
        Customer* c = new Customer();
        queue.insert(c);
        MONITOR_RESET(Hold(20.0),_reset,TRUE);
    }
}

void Arrivals::reset () { _reset = TRUE; }
```

5. Distribution Functions

Many of the aspects of the real world which a simulation attempts to model have properties which correspond to various distribution functions, e.g., inter-arrival rates of customers at a bank queue. Therefore, simulation studies require sources of random numbers. Ideally these sources should produce an endless stream of such numbers, but to do so either requires specialised hardware or the ability to store an infinite (large) table of such numbers generated in advance.

Without such aids, which are either impractical or not generally available, the alternative is to use numerical algorithms. No deterministic algorithm can produce a sequence of numbers that would have all of the properties of a truly random sequence [3]. However, for all practical purposes it is only necessary that the numbers produced appear random, i.e., pass certain statistical tests for randomness. Although these generators produce pseudo-random numbers, we continue to call the random number generators.

The starting point for generating arbitrary distribution functions is to produce a standard uniform distribution. As we shall see, all other distributions can be produced based upon this. (Interested readers are referred to [6] for a more complete treatment of this topic). All of the distribution functions in *C++SIM* rely upon inheritance to specialise the behaviour obtained from the uniform distribution class.

5.1. RandomStream

For historical reasons, the actual uniform distribution class is called `RandomStream`. This returns a series of random numbers uniformly distributed between 0 and 1. We experimented with several random number generators before settling on a shuffle of a multiplicative generator with a linear congruential generator, which provides a reasonably uniform stream of pseudo-random numbers.

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

    double Error ();

protected:
    double Uniform ();
};
```

The multiplicative generator uses the following algorithm[12]:

Y[i+1] = Y[i] * 5^5 mod 2^26

, where the period is 2^24, and the initial seed must be odd.

The `Uniform()` method uses the linear congruential generator (seed is `LCGSeed`) based on the algorithm in [4], and the results of this are shuffled with the multiplicative generator (see is `MGSeed`) as suggested by Maclaren and Marsaglia [3], to obtain a sufficiently uniform random distribution, which is then returned.

[12]Thanks to Professor I. Mitrani for his help in developing this.

The `Error()` method returns a chi-square error measure on the uniform distribution function.

By overloading the `operator()`, and ensuring that it must be defined by derived classes, we obtain a uniform means of accessing random numbers.

The `RandomStream` class returns a large sequence of random numbers, whose period is 2^24. However, unless the seeds are modified when each random distribution class is created, the starting position in this sequence will always be the same, i.e., the same sequence of numbers will be obtained. To prevent this, each class derived from `RandomStream` has an additional (default) parameter for its constructor which indicates the offset in this sequence from which to begin sampling.

5.2. UniformStream

The `UniformStream` class inherits from `RandomStream` and returns random numbers uniformly distributed over a range specified when the instance is created.

```
class UniformStream : public RandomStream
{
public:
    UniformStream (double lo, double hi, int StreamSelect=0);
    virtual double operator() ();
};
```

The range covers the interval specified by `lo` and `hi`, and `StreamSelect` indicates the offset in the random number sequence.

5.3. ExponentialStream

The `ExponentialStream` class returns an exponentially distributed stream of random numbers with mean value specified by `Mean`.

```
class ExponentialStream : public RandomStream
{
public:
    ExponentialStream (double Mean, int StreamSelect=0);
    virtual double operator() ();
};
```

5.4. ErlangStream

`ErlangStream` returns an erlang distribution with mean `Mean` and standard deviation `StandardDeviation`.

```
class ErlangStream : public RandomStream
{
public:
    ErlangStream (double Mean, double StandardDeviation, int StreamSelect=0);
    virtual double operator() ();
};
```

5.5. HyperExponentialStream

The `HyperExponential` class returns a hyper-exponential distribution of random numbers, with mean `Mean` and standard deviation `StandardDeviation`.

```
class HyperExponentialStream : public RandomStream
{
public:
    HyperExponentialStream (double Mean, double StandardDeviation,
                             int StreamSelect=0);
    virtual double operator() ();
};
```

5.6. NormalStream

`NormalStream` returns a normal distribution of random numbers, with mean `Mean` and standard deviation `StandardDeviation`. `operator()` uses the polar method due to Box, Muller, and Marsaglia [3].

```
class NormalStream : public RandomStream
{
public:
    NormalStream (double Mean, double StandardDeviation, int StreamSelect=0);
    virtual double operator() ();
};
```

5.7. Draw

The `Draw` class is the exception to the inheritance rule, instead using `RandomStream` through delegation (for historical reasons). This returns `TRUE` with the probability `prob`, and `FALSE` otherwise.

```
class Draw
{
public:
    Draw (double prob, int StreamSelect=0);
    virtual Boolean operator() ();
};
```

5.8. Example

6. Advanced Simulation Classes

Simulations formed by the interaction of objects derived from `Process` can be considered *causal* (synchronous) in nature: events occur at specific times and form a well defined order. However, it is sometimes necessary to simulate asynchronous real world events, e.g., processor interrupts. To do this requires finer-grained control of the scheduling of simulation processes than it provided by the scheduler; the scheduler simply activates according to simulation time, whereas asynchronous events may have different activation rules, e.g., activate when another process is terminated.

The `Entity` class and others to be described in the following sections gives this required level of control to the user, extending the types of simulation which are possible with *C++SIM*. Asynchronous simulation processes are derived from `Entity`, but the implementation enables these asynchronous process to execute in the same simulation as `Process` objects. However, because these processes are suspended and resumed outside of the control of the scheduler, it is possible for deadlock situations to occur. Therefore, some care must be taken when using these classes.

In addition to the active, suspended, passive and terminated states which a simulation process can be in, asynchronous objects can also be in the following states:

- *waiting*: the process is suspended waiting for a specific event to occur (e.g., a process to be terminated). The waiting process is *not* placed on the scheduler queue.
- *interrupted*: the process, which was in the waiting state, has been interrupted from this before the condition it was awaiting occurred.

The conditions on which a process can wait, and can thus be interrupted from, are:

- *time*: a process can attempt to wait for a specified period of simulation time.
- *process termination*: a process can wait for the termination of another `Entity` process before continuing execution.
- *semaphore*: critical regions of a simulation can be protected by semaphores, where only a single `Entity` process can acquire the semaphore; other processes are suspended until the semaphore is released.
- *user specific*: it is possible for other asynchronous conditions to occur which are not covered above.

6.1. Asynchronous Entities

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

    Boolean Interrupt (Entity& toInterrupt, Boolean immediate = TRUE);

    virtual void terminate ();

    virtual void Body () = 0;

protected:
    Entity ();

    Boolean Wait (double);
    Boolean WaitFor (Entity& controller, Boolean reAct = FALSE);
    Boolean WaitForTrigger (TriggerQueue& _queue);
    void WaitForSemaphore (Semaphore& sem);
};
```

Because `Entity` is derived from `Process`, all of the usual simulation methods are available, and can be used in conjunction with those provided by the derived class.

`Interrupt(Entity& toInterrupt, Boolean immediate = TRUE)` interrupts the asynchronous process `toInterrupt`, which *must* not be **terminated** and *must* be in the **waiting** state. `toInterrupt` becomes the next active process (i.e., it is moved to the head of the scheduler queue). If immediate is `TRUE` (the default) then the current process is suspended immediately; it is scheduled for reactivation at the current simulation time. Otherwise, the current process can be suspended in an application specific way.

Because it is now possible for one process to wait for another to terminate the `terminate()` method must differ from that provided by `Process`. Before the terminating process ends it moves the waiting process to the head of the scheduler queue, and then calls `Process::terminate()`. Currently only a single process can wait on this termination condition, but this may change in future versions.

`Wait(double t)` is similar to `Hold(double t)`, with the exception that the process is moved into the **waiting** state as well as being placed on the scheduler queue. It is therefore possible to interrupt this process before the wait period has elapsed. `TRUE` is returned if the process was interrupted, otherwise `FALSE` is returned.

`WaitFor(Entity& controller, Boolean reAct = FALSE)` suspends the current process until `controller` has terminated. The process is placed in the **waiting** state. If `reAct` is `TRUE` then `controller` is moved to the head of the scheduler queue to become the next activate process, otherwise the application will have to activate `controller`. If the waiting process is interrupted then the method returns `TRUE`, otherwise `FALSE`. The `controller` and the current process must be different, i.e., it is not possible for a process to wait for itself.

Trigger queues are lists maintained by the simulation system of process waiting for specific events to occur, which are outside the scope of those described above. These will be described in the next section. `WaitForTrigger(TriggerQueue& queue)` places the current process on the trigger queue `_queue`, and passivates it. As with the previous methods, the return value indicates whether the process was interrupted, or triggered.

In addition to trigger queues, process can wait on semaphores, allowing the creation of monitor regions, for example. `WaitForSemaphore(Semaphore& sem)` causes the current process to attempt to exclusively acquire the semaphore. If this is not possible then the process is suspended. Currently, a process which is waiting on a semaphore cannot be interrupted, and is not placed into the **waiting** state. As such, when this method returns the semaphore has been acquired.

6.2. Trigger Queues

Processes waiting for the same application controlled event can be grouped together into a `TriggerQueue`, as described in the previous section. When this event occurs the application can use one of the two trigger methods to activate the queue members. This involves placing the process(es) onto the head of the scheduler queue.

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

    Boolean triggerAll ();
    Boolean triggerFirst (Boolean = TRUE);
};
```

- `triggerAll()`: triggers all of the members on the queue.
- `triggerFirst(Boolean = TRUE)`: triggers only the head of the queue. If the parameter is `TRUE` then the `trigger()` method of the `Entity` object is also invoked.

If the queue is not empty when it goes out of scope then all remaining queue members will be triggered, and placed back onto the scheduler queue.

6.3. Semaphores

Application code can be protected from simulation processes through semaphores, which are instances of the `Semaphore` class.

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

    virtual void Get (Entity* attempting);
    virtual void Release ();

    long NumberWaiting () const;
};
```

A semaphores is exclusively acquired by a simulation process, and can exist in one of two states:

- *available*: the semaphore is available to be acquired.
- *unavailable*: a process currently has the semaphore. If another process attempts to acquire the semaphore then it is automatically suspended until the semaphore is **available**.

To be able to manipulate semaphores, a process must be derived from the `Entity` class. To obtain the semaphore, the `Get(Entity* attempting)` method should be used, where `attempting` is the calling process. If the semaphore is **unavailable** then `attempting` is suspended.

When the semaphore is no longer required `Release()` should be called by the process which currently has it.

`NumberWaiting()` returns the number of processes currently suspended waiting for the semaphore.

If the semaphore goes out of scope with processes waiting for it then an error message is displayed. No further action is attempted on behalf of these waiting processes.[13]

[13]Hopefully this will be remedied later.

6.4. Example

7. List Manipulation Classes

It is frequently necessary to group objects in to linked-list structures. *C++SIM* also provides entity and set manipulation facilities similar to those provided by the SIMSET classes of SIMULA.[14] These classes present an abstract and uniform way to store and manipulate lists of objects. These objects do can be of arbitrary types: the list class can store instances of different types simultaneously in the same list.

These facilities are presented by three classes:

- `Link`: this class provides the abstract type of storage object for a doubly linked list.
- `Head`: this class maintains the double linked list of `Link` elements.
- `Linkage`: both `Link` and `Head` derive from this class, which indicates the core functionality required from each class.

7.1. Link Class

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

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

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

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

protected:
    Link ();
};
```

Classes which are to be manipulated using these linked-list mechanisms must be derived from `Link`. The linked list class `Head` treats all elements on the list as type `Link`, rather than their real type. Through this it is possible to store classes of arbitrary type in the same list mechanism.[15]

When applied to an object currently on a list, `Suc()` and `Pred()` return the next element (successor) and previous element (predecessor) respectively. They return `NULL` if the no such element exists.

If the object is currently on a linked list then `Out()` removes it.

`InTo(Head* list)` places this object as the last element in the linked list pointed to by `list` if the list is not `NULL` (i.e., exists). Otherwise it attempts to remove the object from any list it may

[14]These classes do no hold as a significance in *C++SIM* as they do in SIMULA: process objects do no inherit from them and they are not used by the core system.

[15]C++ templates can achieve the same goal.

belong to. In keeping with the list manipulation facilities of SIMULA an object can only be present on one list at a time.

The `Precede` method is overloaded to take either a `Link` or `Head` object:

- `Link`: if *L* is a member of a linked list then the current object is placed into the same list as *L* immediately preceding it, otherwise the result is the same as `Out()`.
- `Head`: the result is the same as `InTo(Head* H)`.

`Follow` is similarly overloaded:

- `Link`: if *L* is a member of a linked list then the current object is placed in the same list as *L* immediately after it, otherwise the result is the same as `Out()`.
- `Head`: this places the object as the first element in *H*.

7.2. Head Class

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

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

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

    void AddFirst (Link* L);
    void AddLast (Link* L);

    long Cardinal () const;
    Boolean Empty () const;

    void Clear ();
};
```

Instances of this class represent the linked lists used to store objects of abstract type `Link`. Any objects still on the linked list when it goes out of scope are automatically deleted; therefore it is important to guarantee that such objects are created on the heap.[16]

`First()` and `Suc()` return the first element on the list, `NULL` otherwise.

`Last()` and `Pred()` return the last element on the list, `NULL` otherwise.

`AddFirst(Link* L)` and `AddLast(Link* L)` add *L* as the first and last element to the list, respectively.

`Cardinal()` returns the number of `Link` objects in the list.

`Empty()` returns `TRUE` if the list is empty, and `FALSE` otherwise.

`Clear()` removes all of the `Link` objects from the list, deleting them as it does.

[16]We have a way around this limitation and will eventually make this available.

7.3. Linkage Class

This class represents the minimum functionality which must be provided by both `Link` and `Head` classes.

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

    virtual Link* Suc () const = 0;
    virtual Link* Pred () const = 0;
};
```

7.4. Example

8. Statistical Classes

The purpose of a simulation typically involves the gathering of relevant statistical information, e.g., the average length of time spent in a queue. *C++SIM* provides a number of different classes for gathering such information.

8.1. Mean

This is the basic class from which others are derived, gathering statistical information on the samples provided to it.

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

    virtual void setValue (double);
    virtual void operator+= (double);

    unsigned int numberOfSamples () const;

    double min () const;
    double max () const;
    double sum () const;
    double mean () const;

    virtual void reset ();
};
```

New values can be supplied to `Mean` using either the `setValue(double)` or `operator+=(double)` methods. The number of samples which have been give can be obtained from `numberOfSamples()`.

The maximum and minimum of the samples supplied can be obtained from the `max()` and `min()` methods, respectively.

`sum()` returns the summation of all of the samples:

$$\sum_{i=1}^{n} Si$$

`mean()` returns the mean value:

$$\frac{1}{n} \sum_{i=1}^{n} Si$$

An instance of `Mean` can be reset between samples using the `reset()` method.

8.2. Variance

This class is derived from Mean, and in addition to providing the above mentioned functionality also provides the following:

```
class Variance : public Mean
{
public:
    Variance ();
    virtual ~Variance ();

    virtual void setValue (double);
    virtual void operator+= (double);

    virtual void reset ();

    double variance () const;
    double stdDev () const;

    double confidence (double);
};
```

`variance()` returns the variance of the samples:

$$\frac{1}{n} \sum_{i=1}^{n} (\text{Si-Mean()})^2$$

`stdDev()` returns the standard deviation of the samples, which is the square root of the variance.

8.3. TimeVariance

The `TimeVariance` class makes it possible to determine how long, in terms of simulation time, specific values were maintained. In effect, values are weighted according to the length of time that they were held, whereas with the `Variance` class only the specific values are taken into account.

```
class TimeVariance : public Variance
{
public:
    TimeVariance ();
    ~TimeVariance ();

    virtual void reset ();

    virtual void setValue (double);
    virtual void operator+= (double);

    double timeAverage () const;
};
```

Whenever a value is supplied to an instance of the `TimeVariance` class the simulation time at which it occurred is also noted. If a value changes, or the `timeAverage()` method is invoked, then the time it has been maintained for is calculated and the statistical data is updated.

8.4. Histograms

`Mean`, `Variance`, and `TimeVariance` provide a snapshot of values in the simulation. However, histograms can yield better information about how a range of values change over the course of a simulation run. This information can be viewed in a number of ways, but typically it is plotted in graphical form.

A histogram typically maintains a slot for each value, or range of values, given to it. These slots are termed *buckets*, and the way in which these buckets are maintained and manipulated gives rise to a variety of different histogram implementations. The following sections detail this variety of different histogram classes.

8.4.1. PrecisionHistogram

The `PrecisionHistogram` class represents the core histogram class from which all others are derived. This class keeps an exact tally of all values given to it, i.e., a bucket is created for each value. Although buckets are only created when requires, over the course of a simulation this can still utilise a large amount of resources, and so other, less precise, histogram classes are provided.

```
class PrecisionHistogram : public Variance
{
public:
    PrecisionHistogram ();
    virtual ~PrecisionHistogram ();

    virtual void setValue (double);
    virtual void operator+= (double);

    virtual void reset ();

    long numberOfBuckets () const;

    virtual Boolean sizeByIndex (long index, double& size);
    virtual Boolean sizeByName  (double name, double& size);

    virtual ostream& print (ostream&) const;
};

extern ostream& operator<< (ostream& strm, const PrecisionHistogram& ph);
```

As with the `Variance` class from which it is derived, and whose methods are obviously available, values can be supplied to the histogram through either the `setValue(double)` or `operator+=(double)` methods.

The number of buckets maintained by the histogram can be obtained from the `numberOfBuckets()` method. Each bucket is uniquely named by the values it contains, and can also be accessed by its index in the entire list of buckets.

There are therefore two ways of getting the number of entries in a bucket:

- by the index number of the bucket: `sizeByIndex(long index, double& size)`.
- by the unique name of the bucket: `sizeByName(double name, double& size)`.

If the bucket does not exist then each of these methods returns `FALSE`, otherwise `TRUE`.

It is possible to output the contents of the histogram using either the `print(ostream&)` or overloaded `operator<<` methods.

8.4.2. Histogram

The problem with the `PrecisionHistogram` class is that it can use up a lot of system resources, especially over the course of a long simulation. `Histogram` attempts to alleviate this by presenting a histogram which is less accurate, but consumes less resources. Instead of maintaining a bucket for each individual value, it keeps a fixed number of buckets. Initially each bucket will store separate values as in the `PrecisionHistogram`, but when the number of required buckets would exceed the specified maximum number it merges pairs of buckets, thus reducing their total. The policy used when merging buckets it set on a per instance basis when created. Current policies are:

- `ACCUMULATE`: create a new bucket with the same name as the largest of the two buckets, and it has the sum of the two old bucket entries as its entry number.
- `MEAN`: create a new bucket with the name as the mean of the two old buckets, and it has the sum of the two old bucket entries as its entry number.
- `MAX`: create a new bucket with the name as the largest of the two buckets, and it has the same number of entries.
- `MIN`: create a new bucket with the name as the smallest of the two old buckets, and it has the same number of entries.

```
class Histogram : public PrecisionHistogram
{
public:
    enum MergeChoice { ACCUMULATE, MEAN, MAX, MIN };

    Histogram (long maxBuckets, MergeChoice = MEAN);
    virtual ~Histogram ();

    virtual void setValue (double);
    virtual void operator+= (double);

    virtual ostream& print (ostream&) const;
};

extern ostream& operator<< (ostream& strm, const Histogram& h);
```

When an instance of `Histogram` is created, the maximum number of allowed buckets must be specified. The merging algorithm can also be provided, with the default being the `MEAN` policy.

8.4.3. SimpleHistogram

As with the `Histogram` class above, `SimpleHistogram` keeps the number of assigned buckets to a minimum. However, it does this by pre-creating the buckets when it is created, i.e., the number of required buckets must be provided at the start. A width is the assigned for each bucket, and whenever a value if given to the histogram class it is placed into the bucket whose width it falls within.

```
class SimpleHistogram : public PrecisionHistogram
{
public:
    SimpleHistogram (double min, double max, long nbuckets);
    SimpleHistogram (double min, double max, double w);
    virtual ~SimpleHistogram ();

    virtual void setValue (double);
    virtual void operator+= (double);

    virtual void reset ();

    virtual Boolean sizeByName (double name, double& size);
    double  Width () const;

    virtual ostream& print (ostream&) const;
};

extern ostream& operator<< (ostream& strm, const SimpleHistogram& s);
```

When the class is instantiated, the range of values it will receive must be provided. Then, either the width of each bucket or the actual number of buckets can be given. If the width is provided, then the histogram automatically calculates the number of buckets, otherwise it calculates the width for each bucket by equally dividing the range between each bucket.

The values of a bucket can be obtained from the `sizeByName(double name, double& size)` method.

The width of each bucket is provided by the `Width()` method.

8.4.4. Quantile

The `Quantile` class provides a means of obtaining the p-quantile of a distribution of values, i.e., the value below which p-percent of the distribution lies.

```
class Quantile : public PrecisionHistogram
{
public:
    Quantile (double = 0.95);
    virtual ~Quantile ();

    double operator() () const;
    double range () const;

    virtual ostream& print (ostream&) const;
};

extern ostream& operator<< (ostream& strm, const Quantile& q);
```

The p-quantile probability range must be specified when the object is instantiated, and can be obtained via the `range()` method.

The actual quantile value is provided by the overloaded `operator()`.

9. Development Classes

In addition to the classes described in the previous chapters, *C++SIM* also provides classes to help in the development and debugging of simulation applications. These classes enable debugging to be enabled at compile-time or run-time, and it is possible for the application programmer or the user to be selective about the type of debugging required at any time. There are also classes for the output of prioritised error messages.

9.1. Debug

The debug classes below provide a means of selective run-time control over debugging output statements. When executing an application, the user specifies the type of debugging required, in terms of the modules, the visibility of the methods (e.g., private or protected), and the types of methods (e.g., constructor or destructor). This information is provided in the form of a global debugging control mask.

Therefore, associated with each debugging statement is a control mask, which the run-time system compares with the main control mask provided at the start of execution; if the mask is a match or subset of the main control mask then the debugging statement is output. However, all debugging statements should occur within `#ifdef DEBUG` and `#endif` statements. This enables debugging to be completely disabled at compile time, thus removing the overhead incurred even when output is not required at run-time.

Debug statements are controlled by the following three types of variable:

- `FacilityCode`: this represents the context, or module, within which the debugging statements occur. If the required run-time output facility code does not match this then the statement is not output. As shown below, *C++SIM* pre-defines some facility codes for its own modules, but leaves room for user defined codes.
- `VisibilityLevel`: this represents the visibility of the methods within which the statement occurs, e.g., public or protected.
- `DebugLevel`: this represents the type of method within which the statement occurs, e.g., constructor or destructor.

```
enum FacilityCode
{
    FAC_SCHEDULER = 0x0001,
    FAC_PROCESS = 0x0002,
    FAC_THREAD = 0x0004,
    FAC_SIMSCRIPT = 0x0008,
    FAC_SEMAPHORE = 0x0010,
    FAC_ENTITY = 0x0020,
    FAC_PROCESSLISTS = 0x0040,
    FAC_GENERAL = 0x0080,
    FAC_USER1 = 0x1000,
    FAC_USER2 = 0x2000,
    FAC_USER3 = 0x4000,
    FAC_USER4 = 0x8000,
    FAC_ALL = 0xffff
};

enum VisibilityLevel
{
    VIS_PRIVATE = 0x0001,
    VIS_PROTECTED = 0x0002,
    VIS_PUBLIC = 0x0004,
    VIS_ALL = 0xffff
};

enum DebugLevel
{
    NO_DEBUGGING = 0,
    CONSTRUCTORS = 0x0001,
    DESTRUCTORS = 0x0002,
    CONSTRUCT_AND_DESTRUCT = CONSTRUCTORS | DESTRUCTORS,
    FUNCTIONS = 0x0010,
    OPERATORS = 0x0020,
    FUNCS_AND_OPS = FUNCTIONS | OPERATORS,
    ALL_NON_TRIVIAL = CONSTRUCT_AND_DESTRUCT | FUNCTIONS | OPERATORS,
    TRIVIAL_FUNCS = 0x0100,
    TRIVIAL_OPERATORS = 0x0200,
    ALL_TRIVIAL = TRIVIAL_FUNCS | TRIVIAL_OPERATORS,
    FULL_DEBUGGING = 0xffff
};

class DebugController : public StreamFilter
{
public:
    DebugController ();
    virtual ~DebugController ();

    void set_all (DebugLevel, FacilityCode, VisibilityLevel);
    void set_debuglevel (DebugLevel);
    void set_facility (FacilityCode);
    void set_visibility (VisibilityLevel);
};

extern DebugController *_cppsim_debug;

#define debug_stream ((_cppsim_debug != 0)?(_cppsim_debug->stream()):
             (_cppsim_debug = new DebugController(),_cppsim_debug->stream()))
```

The `debug_stream` macro is the main component in the debugging sub-system. It is used to control accessibility of debugging statements, and also to output them. To indicate the level etc. for each debugging statement the overloaded `operator<<` is used. That is, the facility code, level and visibility for the statement is inserted into the debug stream. For example:

```
debug_stream << CONSTRUCTOR << VIS_PUBLIC << FAC_PROCESS;
```

The actual debugging statement can be inserted using the same operator, e.g.,:

```
debug_stream << "Sample debugging statement." << endl;
```

It is preferable to flush the output also. Output will only be produced if the inserted code, level, etc. match the currently requested debugging level etc.

To set the desired debug tracing level, the `_cppsim_debug` variable must be set accordingly. This can be done with the individually provided methods `set_debuglevel(DebugLevel)`, `set_facility(FacilityCode)`, and `set_visibility(VisibilityLevel)`, or can be set all together using the `set_all(DebugLevel, FacilityCode, VisibilityLevel)` method. Since each level is represented by a bit in the variable, multiple levels are produced by OR-ing the fields together.

Using `_cppsim_debug` provides a static means of selecting the level of debugging required. Typically this would be initialised within `main`. However, a more dynamic mechanism is available through the shell environment variables: `DEBUG_LEVEL`, `DEBUG_VIS`, and `DEBUG_FAC`. By setting these accordingly prior to executing the application then different levels of debugging can be achieved without recompiling.

9.2. Error

Although error messages can be output using either the previous debugging classes or `cerr`, the `Error` class to be presented below provides additional functionality which may prove useful when building applications.

```
enum ErrorSeverity
{
    FATAL, WARNING
};

extern ostream& operator<< ( ostream& strm, enum ErrorSeverity es );
```

The `operator<<` has been overloaded to allow the addition of an error severity variable with each error message. This is the output with each message and can make identification of important error message easier.

10. Worked Example

11. Modifying C++SIM

11.1. Adding Thread Classes

12. References

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

[2] O-J. Dahl, B. Myhrhaug, K. Nygaard, "SIMULA Common Base Language", Norwegian Computing Centre

[3] Knuth Vol2, "Seminumerical Algorithms", Addison-Wesley, 1969, p. 117.

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

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

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

[7] M. C. Little and D. L. McCue, "The Replica Management System: a Scheme for Flexible and Dynamic Replication", Proceedings of the 2nd International Workshop on Configurable Distributed Systems, March 1994, pp. 46-57.

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

[9] S. B. Lippman, "C++ Primer", Addison Wesley, 1989.

[10] G. D. Parrington et al, "The Design and Implementation of Arjuna", Broadcast Project Technical Report, October 1994.