SimPlus: An Experimental Simulation Tool

Hassan Rajaei, Scott Harper, Tom Mancine, Ryan Scott

Department of Computer Science
Bowling Green State University
Bowling Green, Ohio 43403
email: rajaei@cs.bgsu.edu

Abstract

SimPlus is a simulation toolkit in C++. It has evolved from a final class project for a computer science course on Simulation Techniques to its current release as a UNIX static library. A key objective of the final class project was to provide students with good understanding of the underlying mechanism of a simulation engine. Each group of 2-3 students was assigned to one project. SimPlus is the result if one group. Over the course of the project, SimPlus gained numerous advanced features such as a global static kernel implementing the Singleton design pattern, and callback-method automated event processing.

1. Introduction

Simulation tools aims at facilitating the tasks of analysts or engineers. Without such tools, writing a simulation program often becomes difficult and requires reinvention of the wheel at the beginning of each project. As a result, simulation tools have become very popular for industrial, defense, and educational applications.

For educational purposes, the final class project of our course on Simulation Techniques aimed to provide the students with practical experiences and good understanding of the underlying mechanism of simulation tools. Each group of students was assigned to one project for a period of 4 weeks. The class project was supervised through a project meeting concept, which is often used in engineering environment to safeguard the outcome and on time delivery. SimPlus is the resulting work of one group. SimPlus evolved from a final class project to its current release as a UNIX static library with advanced features.

We experimented first with a simple simulation library called Simlib, which was devised by Law & Kelton[1] for rudimentary simulation tasks. We completed several assignments with Simlib over the course of the semester and became intimately acquainted with its strengths and weaknesses. We also tried NS, Network Simulator[2], and found it lacking certain flexibilities. As a final project, we aimed to improve Simlib, concentrating on three requirements: to re-implement the code in C++, to improve event-list management, and to improve random number

generation. Exceeding these requirements, we managed to completely revise Simlib and transform it into SimPlus, a fully object-oriented simulation tool with numerous capabilities. SimPlus has a class hierarchy designed for flexibility and extensibility. In creating SimPlus, our prime goal was enhanced ease-of-use through abstraction. The class hierarchy with provided library models fulfilled these objectives. The kernel's Singleton design[3] allows application programs to obtain references to it and call any kernel functions. The callback method of the executives automates event processing, separating the actions of the user classes from the kernel. Other capabilities include: significant improvement of random number generator using the Mersenne Twister pseudo-RNG[4,5], an event list implemented as a heap with O(log n) access, advanced memory management, extensible statistics-gathering and reporting, and an extensive API.

The development of SimPlus was divided into several phases. In the first phase, which lasted most of the semester, we learned what we wanted in a simulation tool by using several limited engines and noting their shortcomings. This phase corresponded to the research phase of the project. In the next phase we discussed our goals for the project and formulated a set of objectives. This phase corresponded to the requirements gathering and specifications. The third phase focused on the design, laying out the class hierarchy and interaction between various objects. This was followed by implementation and slight modifications to the design as needed. Apart from the research phase, the project was completed in four weeks.

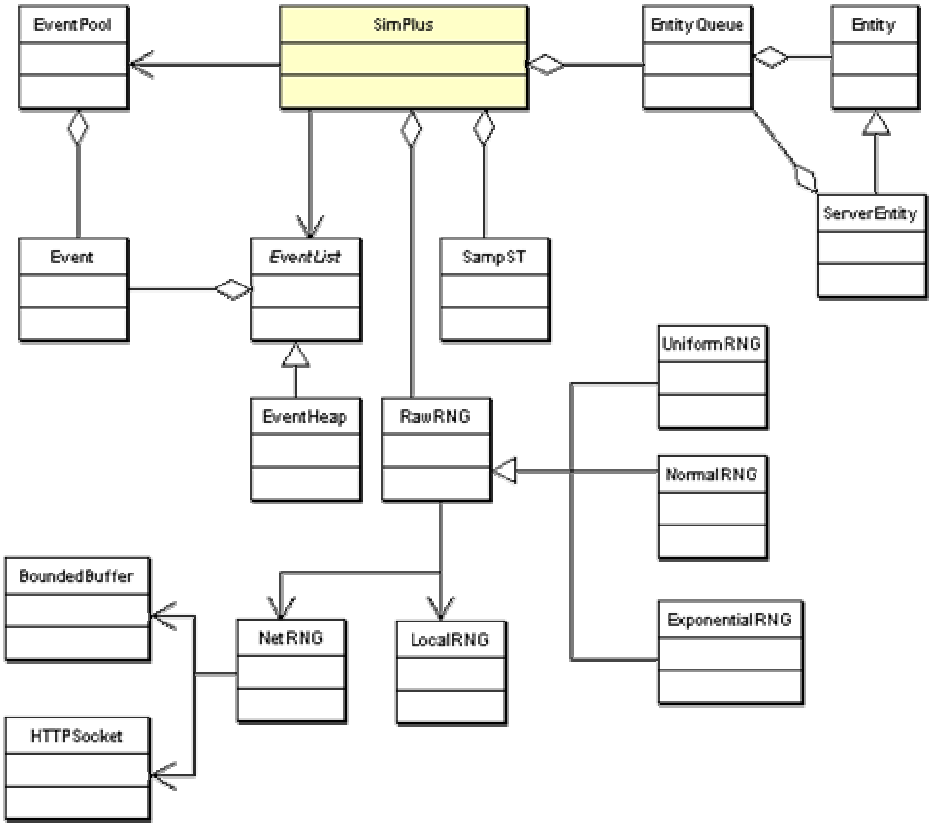

Figure 1: Overall Object-Oriented Architecture of SimPlus Kernel using UML diagram.

2. System Architecture

Key components of the system are detailed by the class diagram and the toolkit API. In this section we present an overview of the system architecture and the API. In the following section we describe the details of the kernel's features and related API followed by segments of simulation code from a sample application written using SimPlus.

The Class Diagram in Figure 1 is a UML class hierarchy diagram illustrating the overall system architecture. As shown in the figure, the kernel of SimPlus incorporates four major components: a Simulation Entity handler, an Event handler, Random Number Generation (RNG), and a Statistic Collection handler.

The Event handling subsystem takes care of event scheduling, event processing, and event-list management. The simulation objects are inherited from Entity and ServerEntity, and can be aggregated by EntityQueue. There are two types of Random Number Generation: Local RNG and Net RNG. The Local RNG uses a pseudorandom number generator based on the Mersenne Twister, whereas the Net RNG fetches the needed random numbers from the random.org CGI application through HTTP sockets. Statistics collection is handled by the SampST object class. Description of the major individual classes and their methods are provided in the next section. More details can be found in the project technical report[6].

3. SimPlus Kernel

The Kernel in SimPlus is implemented as a C++ class that encapsulates most of the functionality required for small to medium scale simulation projects. The SimPlus class (See Figure 1) manages the creation and distribution of most simulation-related objects, such as events, RNGs, and several kinds of queues and keeps track of these objects so that it can clean up after itself. SimPlus is loosely based on the simlib.c code[1] but has evolved as mentioned previously. The kernel class should be explicitly deleted; since no automatic instances of it may be created, the destructor is not implicitly invoked at program exit.

SimPlus is implemented using the "Singleton" design pattern. This means that within a given execution, only one instance of the Kernel object may be created. This is ensured by declaring the constructor, copy constructor, and overloaded assignment operator to be protected. References to the current kernel are obtained by SimPlus::getInstance which returns a pointer to the current instance of SimPlus. It creates a new instance and returns a pointer to it if this is the first call. The destructor explicitly deletes all the object references it has handed out through calls to the various getXXX methods. Below are the constructor, destructor, and error report methods of the SimPlus class. Details of four major components are provided in the following subsections.

```
static getInstance() : SimPlus*
~SimPlus()
static reportError( string )
```

3.1 Simulation Entity handler

Entity: Objects of type Entity represent any object that could possibly require control of the simulation. Objects that inherit from Entity are automatically assigned an unsigned int as a unique identifier. The methods generateEvent and processEvent are declared virtual to enable polymorphism. Any object that inherits from Entity should override these methods if that Object is capable of generating and/or processing events. This class provides following services:

```
Entity()
~Entity()
getID() : unsigned int
virtual operator==(Entity&) : bool
virtual operator!=(Entity&) : bool
virtual operator<(Entity&) : bool
virtual generateEvent(const unsigned short&, const double&) : Event*
virtual processEvent(Event*)
reportError( String )
```

EntityQueue: A class composed of a doubly linked list, representing queues capable of holding Entity objects. Implemented as a doubly-linked list. Care should be taken to avoid inserting a single Entity into multiple queues. Doing so will likely cause unexpected behavior. This class provides the following functions:

```
EntityQueue()
~EntityQueue()
addFirst( Entity* )
addLast( Entity* )
removeFirst()
removeLast()
removeEntity( unsigned int )
find( unsigned int )
```

ServerEntity: This class is a subclass of Entity designed to represent Entities that are capable of performing service on/for other Entities moving through the system. When constructed, the ServerEntity is assigned a numeric prefix and registered with the kernel for callbacks. A binding function binds a specified EntityQueue to the current ServerEntity. Pointers to EntityQueues are stored in associative arrays (STL maps) keyed by the second argument. A ServerEntity may be bound to more than one EntityQueue and an EntityQueue may be bound by more than one ServerEntity. Binding only implies that a ServerEntity may insert into/extract from an EntityQueue. The unbind function will unbind the queue with key equal to sole argument. `generateEvent(...)` wraps a call to Entity::generateEvent. This adds ServerEntity's prefix to the Event's eventType. Care should be taken to account for prefixes in subclasses' Event processing. `getCheckDelay()` returns a normally distributed random number with mean 1.0 and standard deviation 0.1. The preferred method for performing service is to generate

and process BEGIN_SERVICE events at offset equal to checkDelay->getRandom() until an Entity is available for service.

```
ServerEntity()
virtual ~ServerEntity()
Virtual destructor.
bind( EntityQueue*, string ) : bool
unbind( string ) : bool
getBoundQueue( string ) : EntityQueue*
virtual generateEvent( const unsigned short&, const double& ):
Event*
getCheckDelay() : double
getPrefix() : unsigned short
```

3.2 Event handler

The kernel uses the function timing() to process the next event contained in the event list. If the Entity where the event is scheduled to take place has registered itself with the system (which is the default behavior for objects of type ServerEntity), timing calls the processEvent method for that Entity and returns NULL. If the target Entity is not registered with the kernel, a pointer to the Event object is returned so that the Event may be processed manually. scheduleEvent(Event*) inserts the sole argument into the system's event list and calls reportError if the event list cannot or will not accept more Events. cancelEventID(...) cancels a specific event with eventID equals to its sole argument. The event canceller operates in O(n) time and returns true if the event is cancelled, false otherwise.

To save time once a simulation has begun executing, an initial pool of Event objects is allocated. The getEvent method returns a handle to the top Event in the pool if the pool has any events in it, and returns a new Event if the pool is empty. expandEventPool(...) causes the EventPool to allocate a number of new events equal to its sole argument (usually in anticipation of a large number of new events.) `getEntityQueue()` allocates a new EntityQueue for the user, saves a reference to it for cleanup, and returns a pointer to it. Methods provided by Event handler are:

```
timing() : Event*
registerServer( ServerEntity* )
scheduleEvent( Event* )
cancelEventType( const unsigned short& ) : bool
cancelEventID( const unsigned int& ) : bool
getEvent() : Event*
releaseEvent( Event* )
expandEventPool( const unsigned short& )
availableEvents() : unsigned int
getEntityQueue() : EntityQueue*
getSampST() : SampST*
getSimTime() : double
```

Event: Event objects are used to control the flow of a system. Functions associated with Event object are:

```
Event()
Event( double, unsigned int, unsigned int, unsigned short )
~Event()
operator==( const Event& theEvent ) : bool
operator!=( const Event& theEvent ) : bool
operator<( const Event& theEvent ) : bool
operator>( const Event& theEvent ) : bool
reset()
```

EventHeap: A heap data structure for storing SimPlus Events. Implemented as an in-place heap; permits insertion, retrieval, cancellation, and resizing. Acts like a priority queue ordered by Event::timeStamp. Implements the EventList interface.

```
EventHeap(unsigned int)
~EventHeap
get() : Event *
put(Event *) : bool
resize(unsigned int) : bool
cancelNext(unsigned short) : bool
```

EventList: An abstract base class defining the interface for the various event queues provided by SimPlus. Note that there is currently only one event queuing implementation; however, all other implementations are guaranteed to use this interface. Functions related to EventList includes:

```
EventList()
~EventList()
get() : Event *
put(Event *) : bool
resize(unsigned int) : bool
cancelNext(unsigned short) : bool
```

EventListException: An exception which can be thrown by objects implementing the EventList interface to indicate an unrecoverable error (such as during construction).

```
EventListException(char *)
~EventListException()
getMessage() : char *
```

EventPool: EventPool is used to speed up the SimPlus kernel by reducing context-switching overhead associated with use of the new operator. By pre-allocating a stack of Event objects, we save the user from using the new operator by allowing Events to be checked in and checked out

from the pool when needed. The EventPool will somtimes need to use the new operator itself, but the Pool structure's pre-allocation minimizes that need.

```
EventPool()
EventPool( const unsigned int )
~EventPool()
release( Event* )
get() : Event*
reserve( const unsigned int )
getSize() : unsigned int
```

3.3 Random Number Generation (RNG)

Two types of random number generation are implemented: LocalRNG and NetRNG. The RawRNG class aggregates these to present a uniform interface for Uniform, Normal, and Exponential distributions.

LocalRNG : A pseudorandom number generator based on the Mersenne Twister; basically a C++ port of the code by Nishimura and Matsumoto; the workhorse of the SimPlus RNG tree. The period of this RNG is 219937-1. Various random functions are provided, but the only one we use in SimPlus is genRandReal1() which returns a double between interval [0,1].

```
LocalRNG()
seedRand(unsigned long)
genRandReal1() : double
```

RawRNG: A class for random number generation that aggregates the NetRNG and LocalRNG behaviors. The type of behavior is a construction-time parameter, so it's possible to have instances possessing both behaviors. Two static constants are defined that let the user select behavior, such as the following:

```
RawRNG lRNG(RawRNG::Local);
RawRNG nRNG(RawRNG::Net);
```

Currently we default to the RawRNG::Local behavior.

```
RawRNG(unsigned short)
~RawRNG()
genRandReal1() : double
seedRand(unsigned long)
```

NormalRNG: NormalRNG extends RawRNG. It receives two arguments at construction, μ and σ (mean and standard deviation respectively). A polar method is used to generate two new random numbers (one is saved) that conform to a normal distribution. The getRandom() function will return a random double from an N(mu,sigma) distribution.

```
NormalRNG( double, double, unsigned short )
~NormalRNG()
getRandom() : double
```

UniformRNG: The UniformRNG class inherits from RawRNG. Two parameters are received at construction, the upper and lower bound of the new uniformly distributed random number. The getRandom() function will return one random double.

```
UniformRNG( double, double, unsigned short )
~UniformRNG()
getRandom() : double
```

ExponentialRNG: The ExponentialRNG class extends RawRNG. It will sample a random double from an exponential distribution with the specified mean. The constructor parameters are mean inter-arrival time for distribution and an unsigned short specifying the RNG source.

```
ExponentialRNG( double, unsigned short )
~ExponentialRNG()
getRandom() : double
```

NetRNG: A class for random-number "generation" which fetches numbers from the random.org CGI application. It requires a working PThread implementation. NetRNG uses a static member variable to track active instances. If a NetRNG being constructed is the first, a new thread of execution is created whose only job is to fill a bounded buffer with doubles on the interval [0,1]. These are acquired by TCP connection with a random.org CGI app that generates true random numbers based on digitized radio-band noise. When the last active NetRNG has its destructor invoked, the static stop() method is called to tear down the Bounded Buffer and collect the "fetcher" thread. An operator checks to see if NetRNG statics are in a state where we can get numbers from it, returning true if we are ready. genRandReal1()will return a double on the interval [0,1]; this call can block the calling thread if the buffer is empty. NetRNG has known issues related to a race condition in the initialization of the socket and is not recommended.

```
NetRNG()
~NetRNG()
operator!() : bool
genRandReal1() : double
```

BoundedBuffer: Used in NetRNG; a templatized, thread-safe circular buffer structure which implements the Mesa monitor semantic. Its size is a construction-time parameter. When it is full the thread(s) calling put() is(are) blocked, and when the buffer empties the thread(s) calling get() is(are) blocked. This construct provides following services:

```
BoundedBuffer(unsigned short)
~BoundedBuffer()
get(T& ) : bool
put(T )  : bool
tearDown()
empty() : bool
```

HTTPSocket: Used by NetRNG; a C++ class that sends and receives STL strings via TCP/IP--designed as an HTTP client but will probably work for other TCP-based protocols. Uses many C include files to provide networking functionality, and may require linking with -lxnet on the compile line. Constructor argument is size of char[] buffer used to read from the socket--probably not important, so it defaults to 1024 and tinkering is not recommended. The destructor de-allocates the buffer & closes socket if it's open. An open and close operation handles the connection and closing the socket.

```
HTTPSocket(unsigned short)
~HTTPSocket()
open(string, unsigned short) : bool
close() : bool
operator!() : bool
eof() : bool
operator<<(string) : HTTPSocket&
operator>>(string) : HTTPSocket&
```

Users should obtain RNGs through the following kernel calls:

`getExponentialRNG( const double&, unsigned short ) : ExponentialRNG*`
Allocates a new ExponentialRNG for the user, saves a reference to it for cleanup, and returns a pointer to it. The second argument indicates the type of RawRNG to be used in the ExponentialRNG being created. The default is RawRNG::Local, which is seeded. Another possibility is RawRNG::Net, which retrieves truly random numbers via an HTTPSocket from random.org. The first argument specifies the mean of the numbers generated.

`getNormalRNG( const double&, const double&, unsigned short ) : NormalRNG*`
Allocates a new NormalRNG for the user, saves a reference to it for cleanup, and returns a pointer to it. The last argument indicates the type of RawRNG to be used in the NormalRNG being created. The default is RawRNG::Local, which is seeded. Another possibility is RawRNG::Net, which retrieves truly random numbers via an HTTPSocket from random.org. The first argument specifies the mean of the numbers generated. The second specifies the standard deviation.

`getUniformRNG( const double&, unsigned short ) : UniformRNG*`
Allocates a new UniformRNG for the user, saves a reference to it for cleanup, and returns a pointer to it. The second argument indicates the type of RawRNG to be used in the UniformRNG being created. The default is RawRNG::Local, which is seeded. Another possibility is RawRNG::Net, which retrieves truly random numbers via an HTTPSocket from

random.org. The first argument specifies the lower bound of the numbers generated. The second specifies the upper bound.

3.4 Statistic Collection handler

SampST class allows the user to specify handles to simple statistics-gathering mechanisms. The constructor initializes data members. An "observe" function keeps track of the smallest and largest observations, as well as the mean.

```
SampST()
~SampST()
observe(double)
getSum() : double
getMinimum() : double
getMaximum() : double
getSampleSize() : unsigned int
getMean() : double
```

4. A simulation Example in SimPlus

In this section we demonstrate a simulation application modeled for the healthcare clinic shown in Figure 2.

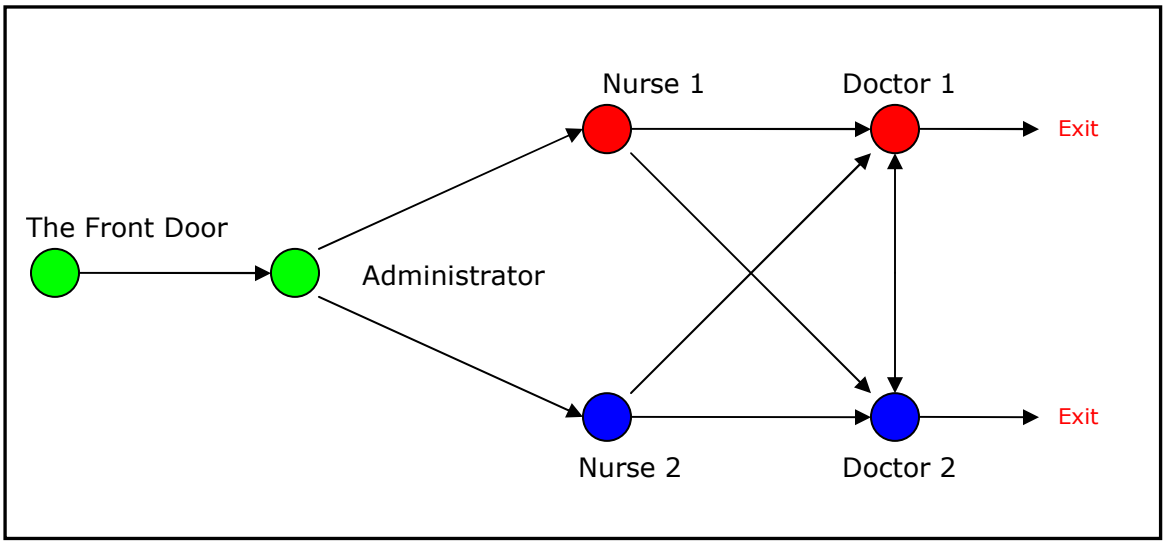

Figure 2: The Healthcare Clinic Model to demonstrate use of SimPlus

In order to demonstrate the robust features incorporated into SimPlus, we chose to implement the Healthcare Clinic model, which was also used as a lab assignment in the class. The following baseline parameters were used as data model as shown in Table 1. Segment of codes for implementation of the model in SimPlus is provided in Figures 4 & 5 in following pages.

Table 1: The input parameter if the Healthcare Clinic model

Administrator		Nurse		Doctor	
Redirection %	50%	**Redirection %**	15%	**Redirection %**	5%
Service Time μ	1.5 Minutes	**Service Time μ**	11.0 Minutes	**Service Time μ**	20.0 Minutes
Service Time σ	0.25 Minutes	**Service Time σ**	2.0 Minutes	**Service Time σ**	5.0 Minutes

The simulation was run through 31 iterations, with 1000 patients per iteration. The goal was to begin with a system that had a relatively light load (25 minute mean interarrival time) and progress to a heavily loaded system (10 minute mean interarrival time.) Therefore, each iteration saw mean interarrival time decremented by 0.5. The system behaved as expected, with wait times at a node increasing exponentially as the mean interarrival time decreased below that node's service time. The chart below demonstrates this effect, showing an exponential increase in wait times in doctor queues as the mean interarrival times decrease. The nurse wait times, however remain relatively constant because the mean interarrival time never drops below nurse service time. The simulation result is shown in Figure 3.

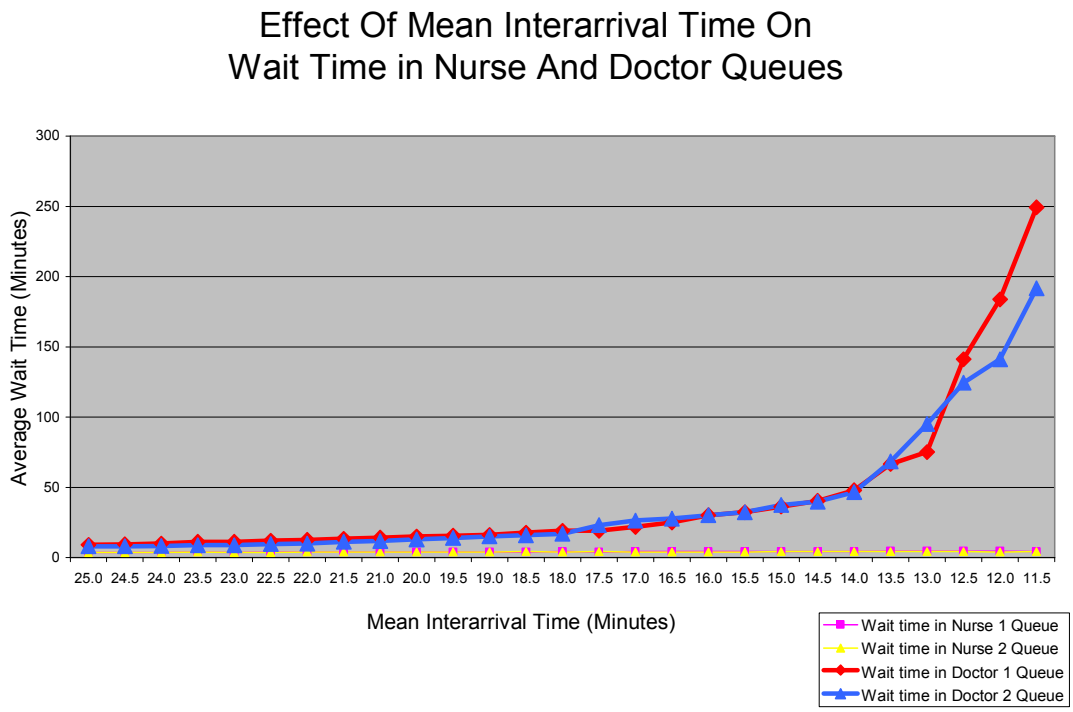

Figure 3: The result of simulation of Healthcare Clinic generate by SimPlus

```
/*******************************************************************************
 Doctor.h

The Doctor node processes incoming Patients and either removes them from the
system or sends them to another Doctor's queue (5% probability).

METHODS:

Doctor( double, double, EntityQueue*, EntityQueue*)
Sole constructor.  First argument is mean service time.  Second argument is standard
deviation of service time.  Third argument is the queue from which the Doctor will pull.
Fourth argument is the queue of the other Doc to which the Doctor can send Patients.
Generates a BEGIN_SERVICE event to force the node to begin checking for Patients.

~Doctor()
Destructor.

virtual generateEvent( const unsigned short&, const double& ) : Event*
Calls generateEvent in the ServerEntity class.

virtual processEvent( Event* )
Handles event callbacks from the kernel.  For a BEGIN_SERVICE event, the Doctor grabs
the first Patient in its queue.  If the Patient grabbed is null (zero) a BEGIN_SERVICE
event is generated at a random time offset (a call to getCheckDelay in ServerEntity.)
If the patient is not null, the Doctor goes to work (schedules an END_SERVICE event
at time offset equal to the next random available from the mean interarrival time.)
For an END_SERVICE event, the Patient is either removed from the system or is added to
the end of the other Doctor's queue (with 5% probability).

friend operator<<( ostream&, Doctor& ) : ostream&
Allows the Doctor to output its statistics to any ostream, such as
STDOUT or a file.  Returns a reference to the ostream.
*******************************************************************************/

#include <iostream>
using std::iostream;
using std::endl;

#include <string>
using std::string;

#include "SimPlus.h"
#include "Patient.h"

#ifndef DOCTOR_H
#define DOCTOR_H

class Doctor : public ServerEntity
{
    public:
            Doctor( double, double, EntityQueue*, EntityQueue* );
            ~Doctor();

            virtual Event* generateEvent( const unsigned short&, const double& );
            virtual void processEvent( Event* );
            static unsigned short patientsProcessed() { return numPatients; };

            friend ostream& operator<<( ostream&, Doctor& );

    protected:
            static unsigned short numPatients;
            Patient* currentPatient;
            SampST* idleTimeStat;
            SampST* waitTimeStat;
            SampST* serviceTimeStat;
            NormalRNG* myServiceTime;
            UniformRNG* myRedirectProb;
            double lastStart;
            double lastStop;

    private:
};
#endif
```

Figure 4: Header file for the Doctor node.

```
/*******************************************************************************
 Doctor.cpp

The documentation within this file is sparse, and is only intended to provide
an overview of coding practices.  For a more detailed description of Doctor,
see Doctor.h.

*******************************************************************************/

#include "Doctor.h"

unsigned short Doctor::numPatients = 0;

Doctor::Doctor( double mu, double sigma, EntityQueue* myQ,
        EntityQueue* otherQ ) : ServerEntity()
{
        SimPlus* handle = SimPlus::getInstance();
        idleTimeStat = handle->getSampST();
        waitTimeStat = handle->getSampST();
        serviceTimeStat = handle->getSampST();

        // this doctor's service time is a normal distribution with a mean of
        // fifteen minutes and a standard deviation of five minutes
        myServiceTime = handle->getNormalRNG( mu, sigma );

        // for redirection we will use a uniform between zero and one
        myRedirectProb = handle->getUniformRNG( 0.0, 1.0 );

        lastStart = 0;
        lastStop = 0;

        bind( myQ, "MY_QUEUE" );
        bind( otherQ, "OTHER_DOC_QUEUE" );

        generateEvent( BEGIN_SERVICE, getCheckDelay() );
}

Doctor::~Doctor()
{
}

Event* Doctor::generateEvent( const unsigned short& eventType,
        const double& eventTime)
{
        if( eventType != BEGIN_SERVICE && eventType != END_SERVICE )
                return 0;
        return ServerEntity::generateEvent( eventType, eventTime );
}

void Doctor::processEvent( Event* anEvent )
{
        double simTime = SimPlus::getInstance()->getSimTime();

        if( anEvent->getEventType() == (getPrefix() + BEGIN_SERVICE) )
        {
                // see if there are any patients waiting
                currentPatient = (Patient*)getBoundQueue("MY_QUEUE")->removeFirst();

                // if no patients are waiting, check again in 1 minute
                if( currentPatient == 0 )
                        generateEvent( BEGIN_SERVICE, simTime + getCheckDelay() );
                // otherwise go to work
                else
                {
                        generateEvent( END_SERVICE, simTime +
                                myServiceTime->getRandom() );
                        lastStart = simTime;
                        idleTimeStat->observe( simTime - lastStop );
                        waitTimeStat->observe( currentPatient->endWait( simTime ) );
                }

        }
        else if( anEvent->getEventType() == (getPrefix() + END_SERVICE) )
        {
                // we redirect 5% of the time
```

```
                if( myRedirectProb->getRandom() < 0.05 )
                {
                        getBoundQueue( "OTHER_DOC_QUEUE" )->addLast( currentPatient );
                        currentPatient->beginWait( simTime );
                }
                // otherwise we kick the patient out of the system
                else
                {
                        ++numPatients;
                        currentPatient->exitSystem( simTime );
                        //delete currentPatient;
                }

                // send my patient to the corresponding doc
                currentPatient = 0;

                lastStop = simTime;
                serviceTimeStat->observe( simTime - lastStart );

                //check for patients again in 1 minute
                generateEvent( BEGIN_SERVICE, simTime + getCheckDelay() );
        }
        else
        {
                reportError( "Doctor" );
        }
        SimPlus::getInstance()->releaseEvent( anEvent );
}

ostream& operator<<( ostream& out, Doctor& theDoctor )
{
        out << "Doctor: Average idle time between patients for this doctor: "
            << theDoctor.idleTimeStat->getMean() << endl;
        out << "Doctor: Average service time for patients for this doctor:  "
            << theDoctor.serviceTimeStat->getMean() << endl;
        out << "Doctor: Average patient wait time in queue for this doctor: "
            << theDoctor.waitTimeStat->getMean() << endl << endl;
        return out;
}
```

Figure 5: Code for Doctor class implementation.

5. Concluding Remarks

In this paper, we have reported the design, implementation, and result of an experimental simulation tool called SimPlus. Our base language was C++ and our tool takes full advantage of its object-oriented features.

Implementing SimPlus has been both challenging and rewarding. We feel that SimPlus has been taken as close to completion as was possible given our short time frame. The current version is well-suited for experimental use. SimPlus should be considered when deciding on a platform for a small to medium scale simulation project. The extensive API should result in shortened development times for experienced C++ programmers. The extensible architecture of the SimPlus object hierarchy additionally allows for highly customized behaviors to be used in specialized simulations.

We aim to further improve the tool and there are number of improvement candidates. The event list could be improved adding a Calendar Queue. The simulation engine can be modified to support distributed and parallel simulation. For the latter, we have recently implemented a parallel version of SimPlus whose description will be published soon. Finally, we plan to use the tool by other students to assess its possible weakness and strengths.

Bibliography

1. Law, Averill, and Kelton, David "Simulation Modeling and Analysis" Third Edition, McGraw Hill, 2000

2. The Network Simulator, NS2, http://www.isi.edu/nsnam/ns/

3. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns. Elemnts of Reusable Object-Oriented Software, Addisson-Wesley, 1995.

4. M. Matsumoto and Y. Kurita, "Twisted GFSR generatos II", ACM Trans. on Modeling and Computer Simulation, 4(1994),254--266

5. http://www.math.keio.ac.jp/~matumoto/emt.html

6. Harper, S., Mancine, T., and Scott, R. , "SimPlus, A Simulation ToolKit for C++", Dept. of Computer Science, Bowling Green State University, Ohio., Sprint 2003.

HASSAN RAJAEI
Hassan Rajaei is an Associate Professor in the Computer Science Department at Bowling Green State University. His research interests include computer simulation, distributed and parallel simulation, performance evaluation of communication networks, wireless communications, distributed and parallel processing. Dr. Rajaei received his Ph.D. from Royal Institute of Technologies, KTH, Stockholm, Sweden and he holds an MSE from U. of Utah.

SCOTT HARPER
Scott Harper was a graduate student of Computer Science at Bowling Green State University. He received a MS in December 2003.

TOM MANCINE
Tom Mancine is a graduate student in Computer Science at Bowling Green State University. He is currently completing his masters' project and plans to graduate in May 2004. His interests include generic programming and distributed systems.

RYAN SCOTT
Ryan Scot was a graduate student of Computer Science at Bowling Green State University. He received a MS in December 2003.