UNIVERSITÀ CA' FOSCARI DI VENEZIA
Dipartimento di Informatica
Technical Report Series in Computer Science

Rapporto di Ricerca CS-2004-1

Febbraio 2004

M. Marzolla

libcppsim: A Simula-like, Portable Process-Oriented Simulation Library in C++

Dipartimento di Informatica, Università Ca' Foscari di Venezia
Via Torino 155, 30172 Mestre–Venezia, Italy

`libcppsim`: A Simula-like, Portable Process-Oriented Simulation Library in C++

Moreno Marzolla

Dipartimento di Informatica

Università Ca' Foscari di Venezia

via Torino 155, 30172 Mestre (VE), Italy

e-mail: marzolla@dsi.unive.it

Technical Report CS-2004-1, Feb 2004

Abstract

In this paper we describe the design and implementation of `libcppsim`, a general-purpose, process-oriented simulation library written in C++. `libcppsim` provides a set of classes for implementing simulation processes, scheduling primitives, random variate generation and output data analysis functions. The main simulation entity provided by the library is the simulation process; the basic process scheduling primitives are modeled upon those provided by SIMULA's simulation class. The modular object-oriented design of `libcppsim` allows users to extend its functionalities with minimal effort. In order to improve efficiency, simulation processes are not implemented as operating system threads; instead, they are implemented on top of coroutine objects which implements a cooperative quasi-parallel process environment. Coroutines are implemented in a portable way in order to allow `libcppsim` to be used on different platforms.

Keywords Process-Oriented Simulation, Simulation Library, C++.

1 Introduction

Simulation is a general modeling technique which can be used for analyzing complex systems that can not be described and evaluated analytically. Many tools and languages have been developed over time in order to facilitate the implementation of simulation models. SIMULA [6] has been one of the first successful general-purpose programming languages providing special features targeted at simulation implementation purposes. SIMULA provided a set of basic facilities for implementing a simulation model as a set of cooperating simulation processes executing in a quasi-parallel system. These facilities were enough for describing many typical simulation models, and could also be used as building blocks for more sophisticated applications; DEMOS [4] was one of the alternative simulation systems developed in SIMULA.

Traditional general-purpose programming languages can be used for implementing simulation programs as well. Support for simulation facilities is usually provided as libraries which can be linked to user's applications. Example of simulation libraries based on general-purpose languages include CSIM19 [16] and C-Sim [10] (based on the C language), C++Sim [13], OMNET++ [18] and DESP-C++ [7] (based on the C++ language) and JavaSim [12] (written in Java). More sophisticated and user-friendly simulation tools have also been developed to make the implementation of simulation models easier for non-programmers. Such simulation environments (usually limited to particular models, such as those based on Queuing Networks or Petri Nets) provide users with visual editors for graphically building the model.

Simulation languages and libraries can be implemented according to an event-oriented or process-oriented paradigm. Event-oriented simulation models are described in term of the events which can change the system's state. A process-oriented simulation model is represented as a collection of concurrently executing simulation processes, each one with its own thread of execution. A simulation process is made of two parts: a set of local variables, and a sequence of actions to execute.

It should be noted that many existing languages and tools suffer either a long learning time, poor performances, or can only be used for special-purpose models as they lack generality. Simulation libraries based on the event-oriented paradigm, while usually easier to learn, are not suited for models with a high number of different event types, as the resulting simulation programs would be very difficult to understand and debug. On the other hand, implementing the process-oriented simulation paradigm on the top of a conventional, general-purpose programming language can be difficult due to limitations (the main being the lack of support for coroutines) of the host language. Nevertheless, using a standard programming language is desirable because modelers are more likely to be already acquainted with the language and do not require to learn some new idiom.

In this paper we describe `libcppsim`, a process-oriented, discrete simulation library written in C++. Our goal is to provide a simulation library based on a widely used and efficient object-oriented programming language. Object-orientation has long been recognized to be very useful for modeling complex systems in a compact and structured way; it is not a surprise that SIMULA was one of the first object-oriented programming languages. `libcppsim` implements a limited set of simulation primitives which can be learned quickly and can be immediately used for implementing simulation models. The provided simple and general framework can be extended if necessary with more complex, high-level functionalities.

Efficiency was one of our main concerns. For that reason the pseudo-parallel simulation process system is not based on the threading model of the underlying Operating System (OS). Threads are not handled efficiently in every OS, as thread switching may incur in the same overhead as process switching. Moreover, complex simulation models made of tens or thousands of simulation processes would probably overflow the resources (process or thread table size) of many operating systems. We address this problem by implementing the *coroutine* primitive as a C++ class. Coroutines implements a simple, user-level cooperative multitasking environment; while this lacks the features of multitasking, it is exactly what is needed to implement process-oriented simulations.

This paper is organized as follows. Section 2 describes how coroutines and simulation processes are implemented in `libcppsim`, and the process scheduling facilities are illustrated in Section 3. Random variate generation is described in Section 4 and output data analysis in Section 5. In Section 6 we show a usage example of the library, and conclusions are reported in Section 7.

2 Simulation Processes

Coroutines [14] are a programming construct which can be very helpful for implementing process-oriented simulation languages and libraries; SIMULA had coroutines as a built-in facility. Unfortunately, the C and C++ programming languages, which are widely used and for which efficient compilers are available, do not provide coroutines natively.

Coroutines are blocks of code with associated state. A coroutine can suspend itself by calling another coroutine; however, unlike traditional subroutine call in which the caller is suspended until the callee terminates, coroutines may be reactivated in any order.

In C/C++, each function is associated to a data structure called *activation record* which is stored in the run-time (LIFO) stack. The activation record contains informations about the status of the routine, such as the value of the local variables and the return address used to resume the execution after a function call. When a function call occurs, a new activation record is created and put on the top of the stack. All the local variables which are defined by the called routine are stored on the newly created activation record. When a routine terminates, its activation record is pushed from the stack. LIFO handling of the activation records does not work anymore with coroutines, because the currently active coroutine may not be the one associated with the topmost activation record. This implies that the order of activation of coroutines is not given by the LIFO stack handling. It also means that the stack of all the routines associated with all but the topmost activation record cannot grow.

This problem can be solved in different ways. The first approach is the "copy-stack implementation" described by Helsgaun [9]. The stack of the currently operating coroutine is kept in the C++ runtime stack. When a coroutine suspends, the runtime stack is copied in a buffer associated with the coroutine. The buffer is allocated in the dynamic heap, so it does not interfere with the normal stack operation. A coroutine is resumed by copying the content of that buffer to C++'s runtime stack and setting the program counter to the saved execution location.

A second approach is simpler and more efficient, and consists of making use of the context handling facilities provided by most Unix SysV-like environments. The OS provides functions allowing user-level context switching between different threads of control within a process using the getcontext(), setcontext() and makecontext() functions. In this approach, each coroutine has its own run-time stack in a pre-allocated block in the heap.

Both approaches have advantages and disadvantages. The "copy-stack" approach is the most portable, as it can be implemented almost in every modern OS which implements the setjmp() and longjmp() system calls (almost everyone does). However, it requires a copy of the C++ runtime stack to be saved and restored every time a coroutine passes control, which may cause a considerable overhead. On the other hand, the approach based on the context handling facilities is more efficient, as the stack content is not copied, but less portable as not every OS implements the required system calls. Also, the context-based approach does not allow the coroutine context to grow past the dimension defined when the context is allocated. The user is required to set the maximum dimension of the context in advance; choosing a too small buffer results in erroneous runtime behaviors caused by stack corruption which can be difficult to debug; see [9] for a complete discussion, and [8] for detailed portability considerations. The `libcppsim` library implements both stack-handling mechanisms, and the user can choose which one to use at compile time. As a general rule, the "copy-stack" variant is useful when developing the simulation program, in order to ensure that stack corruption errors do not happen and measure the maximum stack size required by the application; then, when the program is verified, production runs may be done with the more efficient context-based approach.

Once coroutines are available, it is very easy to define simulation processes on top of them. Fig. 1 shows the class hierarchy related to simulation process implementation.

A simulation process is represented by the process class, and is derived from the coroutine class; a simulation process is a coroutine whose state must be preserved across invocations.

Each simulation process has a unique identifier and a user-supplied name. The process class defines methods providing the classic SIMULA-like scheduling primitives. It is possible to suspend the current process for a given amount of time, schedule the activation of another process in the future or cancel a pending process from the sequencing set. The actions performed by a simulation process are specified by defining the pure virtual method inner_body(). Fig. 2 shows the C++ interface of the process class.

The handle$\langle$T$\rangle$ class implements a "smart pointer" to an object of type T, where the class T must inherit from class shared. A smart pointer [17] is a pointer to an object which

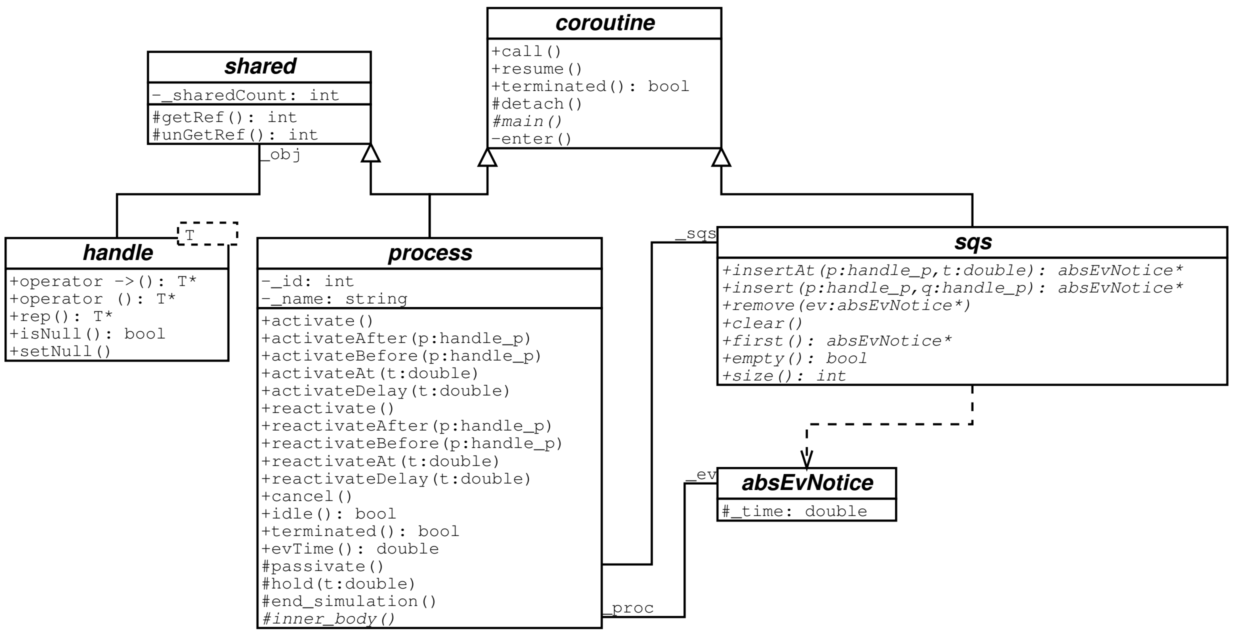

Figure 1: `libcppsim` process class diagram

keeps count of the number of times it is referenced. When the last reference goes away, the object is automatically destroyed. Smart pointers are used for implementing an automatic garbage collection mechanism. Our implementation of the handle class differs from the one proposed in [17] as we store the reference count in the object rather than in the handle. This allows to preserve the counter even if the handle is converted to a C-style pointer and then back to an handle.

3 Sequencing Set implementations

The Sequencing Set (SQS) is a data structure containing the list of simulation processes to be executed; the list is sorted by nondecreasing activation time order. Each simulation process contained in the Sequencing Set (SQS) is associated with an event notice. The event notice basically contains the simulated timestamp at which the process is to be resumed, and references to that process. Additional informations may be present in order to make insertion or removal of event notices more efficient (see Fig. 3).

```
typedef handle<process> handle_p;

class process : public coroutine, public shared {
public:

    process( const string& name );
    virtual ~process( );

    //
    // Modifiers
    //
    void activate( void );                      // put in front of the sqs
    void activateAfter( handle_p& q );          // activate AFTER q
    void activateBefore( handle_p& q );         // activate BEFORE q
    void activateAt( double t );                // activate at time t
    void activateDelay( double dt = 0.0 );      // activate with delay dt

    void reactivate( void );
    void reactivateAfter( handle_p& q );
    void reactivateBefore( handle_p& q );
    void reactivateAt( double t );
    void reactivateDelay( double dt = 0.0 );

    void cancel( void );                        // remove from the sqs

    //
    // Accessors (const)
    //
    bool        idle( void )       const;       // Is this process idle?
    bool        terminated( void ) const;       // Is this process terminated?
    double      evTime( void )     const;       // The next reactivation time

protected:

    //
    // Modifiers
    //
    void hold( double dt );
    void passivate( void );
    void stop_simulation( void );

    //
    // Accessors
    //
    handle_p& current( void ) const;    // Currently executing process
    double time( void ) const;          // Current simulation time

    virtual void inner_body( void ) = 0; // Simulation process' body
};
```

Figure 2: Interface of the process class.

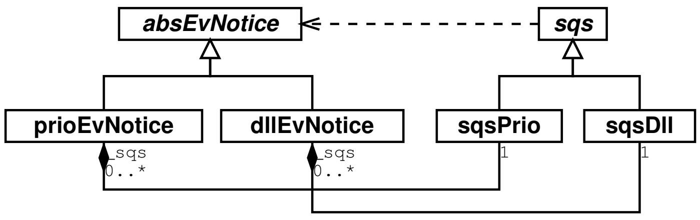

Figure 3: `libcppsim` SQS class diagram

`libcppsim` provides two different implementations of the SQS data structure. The first is defined in class sqsDll, and is based on a doubly linked list of event notices. This data structure is very simple, but insertions of event notices require linear time on average to be performed. A more efficient implementation is given by class sqsPrio, based on balanced search trees. The expected insertion time in this case is proportional to the logarithm of the SQS size. More efficient data structures (e.g. calendar queues [5]) can be implemented by

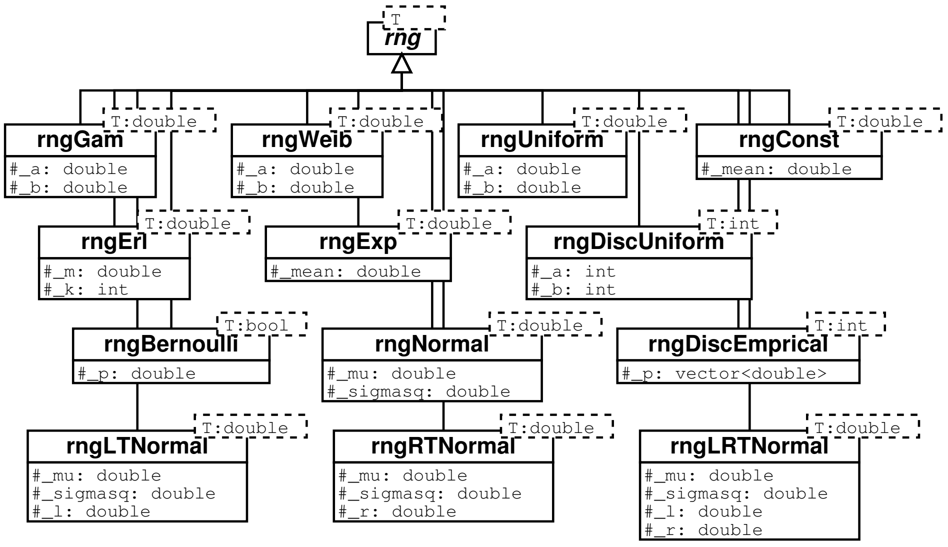

Figure 4: `libcppsim` Random Number Generators class diagram

supplying the corresponding implementation class.

4 Random variate generations

Simulation programs require an efficient and statistically robust mechanism for producing streams of pseudo-random numbers with given distribution. `libcppsim` defines an abstract templated class rng⟨T⟩ representing a pseudo-random number generator of numbers of type T. Thus, it is possible to generate streams of random integers, real or boolean values, by setting T to the appropriate datatype in a subclass. The rng⟨T⟩ class hierarchy is depicted in Fig. 4.

It turns out that the basic ingredient for generating stream of pseudo-random numbers is a good uniform generator RN$(0, 1)$ over the interval $[0..1]$. We chose to implement the algorithm called `MRG32k3a` in [11], which is known to have long period and good statistical properties. Different random variate generators, have been implemented, based on the algorithms described in [2, Ch. 5]. These include generators for the Uniform, Exponential, Gamma, Weibull, (Truncated) Normal, Bernoulli and Empirical distributions.

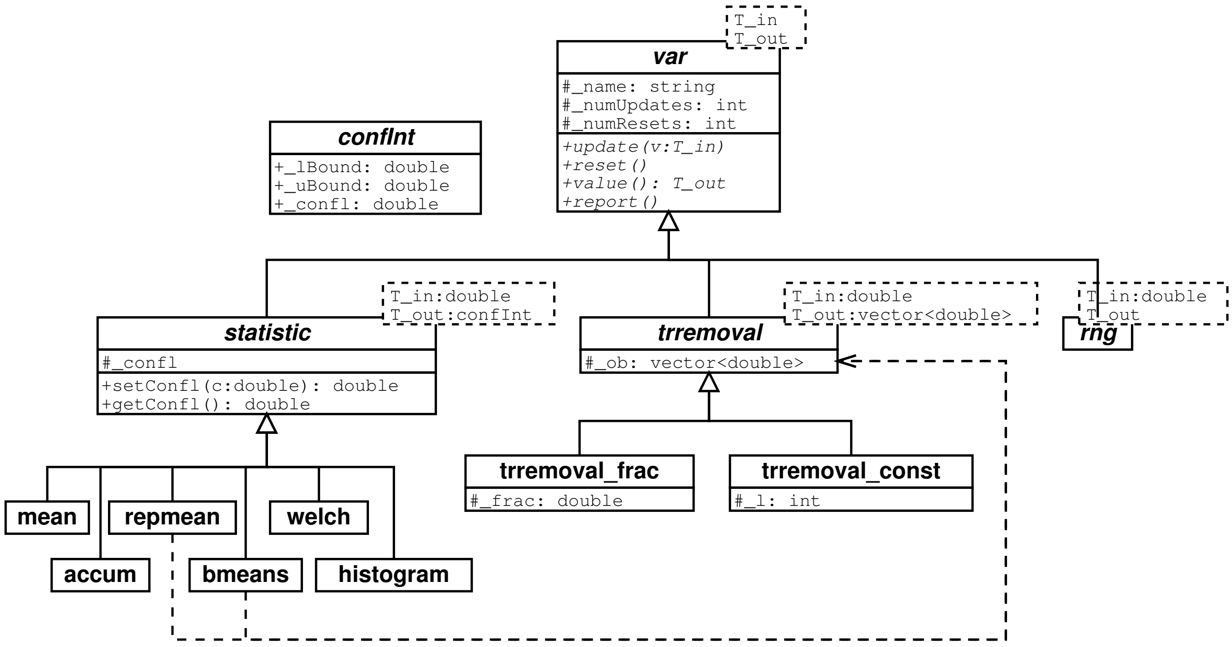

Figure 5: `libcppsim` statistics class diagram

5 Output Data Analysis

Simulation results are sequences of observations for quantities of interest. Appropriate statistical techniques should be applies to analyze these sequences, as the the observations are in general autocorrelated. `libcppsim` implements a set of classes dealing with collection of statistics. Figure 5 shows the relevant portion of the class diagram.

All types of statistical variables inherit from the $\mathsf{var} < \mathsf{T}_{\mathsf{in}}, \mathsf{T}_{\mathsf{out}} >$ abstract base class. This class represents "variables" which can be updated many times by feeding values of type T_{in}, and computes some function from the input data producing a result of type T_{out}. Each variable has two pure virtual methods: update() and reset(). These are used to insert a new value and reset the variable to a known state. The result can be computed by invoking the value() method. Variables are not stateless; thus, it is possible that successive repeated invocations of the value() method return different results. This is useful for representing a random stream as a variable which is initially updated with the seed of the pseudo-random number generator, and whose value() method returns the next pseudo-random number from the stream, at the same time updating the seeds. Class var defines also a pure abstract report() method which can be used to display a report about the content of the class.

The statistics class represents a base class for a number of predefined statistic functions. In addition to the methods and attributes defined in its parent class var, statistics contains an attribute representing the confidence level of the computed result. Statistics are special kind of variables returning objects of type confInt (confidence intervals). The following statistics can be computed:

mean This class is used to compute a confidence interval for the mean of a sequence of *statistically uncorrelated* observations.

accum This class is used to compute a time-weighted sum of observations $Y_1, Y_2, \ldots Y_n$, $n > 1$ with timestamps respectively $t_1, t_2, \ldots t_n$. The result is computed as $A = \sum_{i=1}^{n-1} Y_i(t_{i+1} - t_i)$

repmean Computes the mean of a sequence of observations using the method of independent replications. The simulation must be repeated a total of R times, each run using different random streams and independently chosen initial conditions. Averages across replications can be used to obtain an accurate estimator of the sample mean [2].

bmeans The method of *batch means* divides the output data from one replication into a number of batches, and then treating the means of these batches as if they were independent to compute the sample mean [3].

welch This class implements the graphical procedure of Welch [19] for identifying the length of the initial warm-up period. R independent replications are collected; the values across the different replications are averaged and subsequently smoothed by computing moving averages. The plot of the moving average becomes approximately constant after the warm-up period is over.

histogram This class computes an histogram profile from a sequence of values. The user supplies an expected lower bound and upper bound for the observed values. Also, the user specifies the number of cells (bins) in which the histogram will be divided.

The trremoval class is used to model algorithms dealing with the removal of the initialization bias from a sequence of observations. If the simulation run produces the sequence of observations $Y_1, Y_2, \ldots Y_n$ for some parameter of interest. In general using the whole sequence to compute the statistics is dangerous as there is a bias associated to artificial or arbitrary initial conditions. One method to overcome this limitation is to divide the sequence of observations into two phases: first an initialization phase including observations $Y_1, Y_2, \ldots Y_d$, followed by a data-collection phase $Y_{d+1}, Y_{d+2} \ldots Y_n$. The parameter d, $0 < d \ll n$ is called *truncation point*. Classes inheriting from trremoval are used to identify the truncation point d according to some specific algorithm.

Currently, there is no automatic, general and correct method for detecting the length of the initialization bias. Different approaches have been proposed in the literature [20], although their effectiveness remains dubious. For that reason, simulation packages usually implement one of the following simple strategies:

- Removing a prefix consisting of a fixed fraction (eg, 20%) from the sequence of observations; this approach is implemented by class trremoval_frac. Class trremoval_const can be used is the exact length of the warm-up period is known by the modeler.
- Performing a very long simulation run such that the effect of the initial transient period can be neglected.

6 Example

We show in Fig. 6 a simple usage example of the `libcppsim` library. The main() function performs a set of initialization tasks. The first is the creation of a new simulation context with an associated SQS object which is used by all processes; as in the original SIMULA language, it is possible to nest simulation contexts, allowing independent simulations inside other simulations. At this point, one or more simulation processes can be created and scheduled for activation, after that the simulation can be started.

7 Conclusions

In this paper we described a SIMULA-like process-oriented simulation library implemented in C++. The library is based on the coroutine abstraction implemented as a C++ class, and

```
#include <iostream>
#include "cppsim.hh"

class job : public process {
public:
    job( const string& name ) : process( name ) {
        cout << "Created job '" << name << "' " << id() << endl;
    };
    virtual ~job( ) {
        cout << "Destroyed job '" << name() << "' " << id() << endl;
    };
protected:
    void inner_body( void ) {
        hold( 10 );
    };
};

class source : public process {
public:
    source( const string& name ) : process( name ) { };
    virtual ~source( ) { };
protected:
    void inner_body( void ) {
        handle<job> j;
        for ( int i=0; i<10; i++ ) {
            hold( 5 );
            j = new job( "job" );
            j->activateAfter( current() );
        }
        stop_simulation( ); // Returns to the main() function
    };
};
```

```
int main( void )
{
    // Instantiate a simulation context with an associated SQS
    simulation::instance()->begin_simulation( new sqsDll() );
    // Create a simulation process
    handle<source> src = new source( "source" );
    // Schedule the process for execution
    src->activate();
    // Run the simulation
    simulation::instance()->run();
    // Clean up the SQS
    simulation::instance()->end_simulation();
    return 0;
};
```

Figure 6: Example showing the usage of the `libcppsim` library

has been succesfully compiled on various versione of the Linux OS and other flavors of Unix including Digital/Compaq OSF and NetBSD. The library provides facilities such as simulation processes, random variate generators and basic statistical functions. The `libcppsim` library has been used to implement a process-oriented simulation tool for performance evaluation of UML specifications [1, 15].

Due to its object-oriented nature, the library can be easily extended with additional functionalities. We are currently extending the library in several directions. First, we are implementing more high-level synchronization mechanisms (such as semaphores and mailboxes) which can be useful in developing certain complex simulation models. Moreover, we are implementing more statistical functions in order to provide the modeler with a set of more advanced mechanisms for evaluation of the simulation results.

Availability

Full source code of the `libcppsim` simulation library is available on the author's web page http://www.dsi.unive.it/~ marzolla.

Acknowledgments

This work has been partially supported by MIUR research project FIRB "Performance Evaluation of Complex Systems: Techniques, Methodologies and Tools".

References

[1] S. Balsamo and M. Marzolla. Simulation modeling of UML software architectures. In D. Al-Dabass, editor, *Proc. of ESM'03, the 17th European Simulation Multiconference*, pages 562–567, Nottingham, UK, June 9–1 2003. SCS–European Publishing House.

[2] J. Banks, editor. *Handbook of Simulation*. Wiley–Interscience, 1998.

[3] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 2001.

[4] G. Birtwistle. *DEMOS–A system for discrete event modelling on Simula*. MacMillan Press, 1979.

[5] R. Brown. Calendar queues: A fast $0(1)$ priority queue implementation for the simulation event set problem. *CACM*, 31(10):1220–1227, Oct. 1988.

[6] O.-J. Dahl and K. Nygaard. SIMULA–an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, Sept. 1966.

[7] J. Darmont. DESP-C++: a discrete-event simulation package for C++. *Software–Practice and Experience*, 30:37–60, 2000.

[8] R. S. Engelschall. Portable multithreading–the signal stack trick for user-space thread creation. In *Proc. USENIX Ann. Tech. Conference*, San Diego, CA, June 18–23 2000.

[9] K. Helsgaun. A portable C++ library for coroutine sequencing. Datalogiske Skrifter (Writings on Computer Science) 87, Roskilde University, 1999.

[10] J. Hlavièka, S. Racek, and P. Herout. C-Sim v.4.1. Research Report DC-99-09, DCSE CTU Prague Publishing, Prague, Dec. 1999.

[11] P. L'Ecuyer. Good parameters and implementations for combined multiple resursive random number generators. *Operations Research*, 47:159–164, 1999.

[12] M. C. Little. JavaSim Home page. http://javasim.ncl.ac.uk/.

[13] M. C. Little and D. L. McCue. Construction and use of a simulation package in C++. Computing Science Tech. Report 437, University of Newcastle upon Tyne, July 1993.

[14] C. D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Impleme ntation*, volume 95 of *LNCS*. Springer-Verlag, 1980.

[15] M. Marzolla. *Simulation-Based Performance Modeling of UML Software Architectures*. PhD thesis, TD-2004-1, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Feb. 2004.

[16] H. Schwetman. CSIM19: a powerful tool for building system models. In *Proc. of the 33nd conference on Winter simulation*, pages 250–255. IEEE Computer Society, 2001.

[17] B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, 3rd edition, 1997.

[18] A. Varga. The OMNET++ discrete event simulation system. In *Proc. of ESM'01, the 15th European Simulation Multiconfer ence*, pages 319–324, Prague, Czech Republic, June 6–9 2001.

[19] P. D. Welch. The statistical analysis of simulation results. In S. Lavenberg, editor, *The Computer Performance Modeling Handbook*, pages 268–328. Academic Press, New York, 1983.

[20] K. P. White Jr., M. J. Cobb, and S. C. S. t. A comparison of five steady-state truncation heuristics for simu lation. In *Proc. of the 32nd Winter Simulation Conference*, pages 755–760, Orlando, Florida, 2000. Society for Computer Simulation International.