Efficient Simulation of Network Models in C++

Bernd T. Kluth *  
Mannesmann Mobilfunk GmbH  
Düsseldorf, Germany  

Carmelita Görg  
Institute for Teleprocessing  
Aachen University of Technology  
Aachen, Germany

This paper shows how object oriented programming concepts may be used to establish a framework for supporting simulation model programming. It describes how this has been implemented in a system written in C++ called SIC. Its support of stochastic simulation concerns three fields: First, a library of simulation tools for random number generation and statistical analysis is offered along with a mechanism for implementing process oriented simulation. Second, a modeling toolkit has been set up and may easily be extended by the user, facilitating model construction and reuse of components in different programs. Third, based on such a toolkit and the SIC program structure, a system for graphical generation of simulation programs has been established with an X window based graphic editor, which is also used for graphic output of intermediate and final results. The reusability of model components offered by the object oriented design techniques of C++ is illustrated by examples of queueing network models comparing different scheduling strategies. A simulation model of the DQDB MAN is presented showing how such structured models may be implemented appropriately in SIC.

1 Simulation Issues

Discrete Event Simulation is still the most widely used means for analyzing the performance of randomly driven, complex digital systems such as communication systems. The reliability of the results obtained by simulation strongly depends on the qualities of the random number generators applied and on the statistical methods used for analyzing the data observed during a simulation run. Their accuracy of course rises with the number of observations made. This leads to a demand for high event counts, which in turn calls for a good efficiency of the program execution. Finally, the methods used for mapping a model into a simulation program influence the modeling safety.

The significance of the latter aspect increases as the investigated models grow more complex. In the context of simple models the main issue was to retain the model's functionality inside the simulation program. To maintain models of higher complexity more safely, this aim needs to be supported by retaining the model's structure as clearly as possible inside a simulation program. Smaller programs certainly profit by a good program design as well.

2 Object Oriented Simulation

Recently, one topic seems to dominate many new software systems: object orientation. While the basic idea was originally established in the context of simulation programming by the language SIMULA, its recent upshot originated from other projects such as SMALLTALK. The use of these systems in simulation programming, however, is impeded by their relatively poor performance, as compared to "traditional" programming languages. Even though the object oriented programming language SIMULA has been available for a long time, it didn’t prevail since. This is largely due to the fact that SIMULA has developed as a mainframe language and implementations were not very efficient. Moreover, the UNIX operating system meanwhile emerged in the scientific and technical society, giving rise to a broader use of the C programming language.

On the other hand, it has been noted before [15] that object oriented concepts can help to maintain the model's structures inside a simulation program in a natural way. The rationale for this is that simulation models usually describe the components of a system and the interactions between them. This may naturally be reflected by object oriented structures which increases the reliability of the resulting model, for the correctness of the program can be assured more easily.

This perception was the motivation for STRUSTRUP to develop the programming language C++ [13] as a superset of the language C, adding concepts of object oriented programming to it. When introducing these new language features, care was taken to maintain as much efficiency as possible. Thus, C++ obviously combines the expressive power of object orientation with the efficiency of the C programming language, making it ideally suited for simulation programming.

*This work was done while the author was at the Institute for Teleprocessing, Aachen University of Technology, Aachen, Germany, E-mail: bernd@dfv.e-technik.rwth-aachen.de
3 SIC -- a System for Stochastic Simulation

Written entirely in C++, a simulation system called SIC\textsuperscript{1} [6] was developed. A portable kernel enabling process oriented simulation is at its core.

The key aspects of SIC are presented in this section. Based on the syntax of C++, SIC gives a framework for defining a process as well as for setting up an entire simulation program. On top of that, a set of building blocks for creating queueing network simulation models has been set up allowing simulation programs to be established in a well structured manner. This was extended to allow graphical specification of simulation models. The methods used for ensuring statistical reliability are outlined. Finally, some possibilities offered to further speed up simulation runs will be described.

3.1 Structure of a SIC Program

Processes are defined in a SIC program as 	extit{classes}. A class in C++ describes both a data object and the operations allowed for it. These operations are specified in C++ as 	extit{member functions} of a class. The properties of classes may be inherited by so-called derived classes. This allows a set of classes to share common properties easily.

3.1.1 Process Structure

A process class in SIC is derived from a base class 	extit{task}. The definition of a process consists of three distinct and distinguishable parts which match the modeler's view in a natural way:

- 	extbf{persistent data:} The persistent data comprise those variables of a process which describe its state and which thus have to retain their values during deactivation phases of a process. They are contained in the class definition of a process class.

- 	extbf{initialization:} The initialization of a process serves to maintain correct initial values of a process's persistent data. Additionally, it serves to parameterize a process.

- 	extbf{algorithmic behavior:} The algorithmic behavior of a process is specified in a special member function of a process class called \textit{run}. This function defines the operation of a process, its cooperation with other processes and its access to resources.

3.1.2 Process Oriented Simulation in SIC

Processes are the active entities inside a simulation model. In stochastic simulation, processes usually execute repetitively while interacting with their neighbors. Such interactions, however, may require the execution of a process to be suspended either unconditionally or until some conditions are met, e.g. until some resources are freed. Later, they have to be resumed at the point of their previous interruption. This leads to a behavior of processes which is usually referred to as co-routine execution.

For the sake of portability, process oriented simulation is implemented in SIC by mapping the co-routine behavior of a process to procedure calls. A preprocessor ensures that execution is resumed at the appropriate point in the sequence of a process's actions. This implementation does not depend on additional routines in assembly language.

Due to this, SIC could easily be installed on various UNIX systems. A portation of SIC to PC/AT-type personal computers with MS-DOS has been completed. As the implementation of process-oriented simulation in SIC does not rely on address-calculations, this could be done without major difficulties. Otherwise, the segmented address space of the PC architecture might have required additional efforts.

An aspect of this co-routine implementation is that statements which might interrupt processes may only be placed in their \textit{run} member function and not in an arbitrary function called from \textit{run} (although it is of course possible to call other functions from the \textit{run} function). This restriction has not been an impediment for the implementation of simulation models.

3.1.3 Program Structure

As mentioned, processes are defined inside SIC as classes. By setting up a process's class description, only the \textit{template} of a process is given; the \textit{incarnation}\textsuperscript{2} of a process is carried out separately. This concept allows the creation of several process incarnations from one process definition. As it is possible to parameterize a process in SIC, each process incarnation may be set up individually, e.g. concerning its stochastic characteristics and the model resources it utilizes. Thus sets of processes with the same structure are maintained efficiently in SIC.

Based upon the process structure described above, a typical SIC program is shown in Fig. 1. Its first part consists of \textit{definitions} of processes in the above-mentioned form and of passive components used to specify the processes's cooperation. Incarnations of these components are \textit{created} and parameterized in the subsequent main program. In the context of queueing models, the structure of the investigated system is thus defined here, specifying both the model's topology and parameterization. In the case of the network shown in Fig. 2, its structure is completely defined by naming the queues to read from resp. to write to for each server.

3.2 A Modeling Toolkit

Process specifications and the main program may of course be grouped in different modules which are translated separately. This offers a path to elegantly establish a library with components of simulation models. Their use considerably reduces the effort of writing a simulation program while simultaneously advancing its clearness.

A toolkit of components for queueing network models

\textsuperscript{1}SIC: Simulation in C++

\textsuperscript{2}The term \textit{incarnation} is used throughout this paper to denote both the action of creating an object from a C++ class and the resulting object itself.
Process 1
- <persistent data>
- <initialization>
- <algorithmic behavior>

Process 2
...
...

Main program
- <Creation of queues etc.>
- <Creation of random number generators>
- <Creation of process incarnations>

start of simulation
- <output of results...>

Fig. 1 Structure of a SIC program

was established for SIC. It contains several predefined process types for setting up the active parts of a simulation program. Additionally, building blocks for modeling the interaction of processes are included. These are various queue objects according to the structure mentioned below, and semaphore objects for controlling access to resources.

Queues in SIC present themselves to the user as two different classes — qhead and qtail, which are interconnected internally. This concept improves modeling security by assigning specific operations to a queue’s head resp. tail: messages may be inserted with queue_put only at a qtail object, reading with queue_get is only allowed at a qhead object. This allows some programming errors to be discovered already at compile time.

Fig. 2 Queueing model with several stations

An advantage of distinguishing between head- and tail-objects of queues becomes apparent when considering the open queueing network model in Fig. 2 [8]. It consists of nine stations S1,...,S9. Each station consists of a server, drawn as a circle here, and a queue. All servers operate in the same manner: they pick a message from a queue’s head, process this message, and forward it to another destination after simulating their service time. Thus it seems obvious to create all server incarnations from the same process class. Taking a closer look, however, one notes that some servers put their messages immediately into queue-tails, some lead to exits from the network Sk1,...,Sk3, others forward them through branching points D1,...,D6 into one of several possible directions with a specific probability assigned to each branch. If implemented in a straightforward way, for each of these cases a specific server process type would have to be defined. This would make the resulting program more complicated. As the different servers would operate similarly with respect to their internal behavior while forwarding the messages to different targets, the same code would have to be replicated several times. This prevents code reuse and makes it necessary not only to ensure its correctness repeatedly, but also to maintain the equivalence among all copies after changes have been made.

Using the object oriented features of C++, however, it is possible to develop this model in a more structured way. A generalization of the concept of queues was introduced in SIC by setting up a class hierarchy for queues. This paves the way to treat all servers in this model equally.
The classes qhead and qtail are merely base classes in SIC. Derived from these there are several queue types, e.g. FIFO queues. Queues implementing other strategies, e.g. priorities, are included as well. This was for instance used to perform the examinations shown in section 4.1.

To cope with a model where messages are routed as in Fig. 2, two new object types have been derived from the base class qtail: a sink object for removing messages from the system and a decision object implementing a branch in the flow of messages.

The structure of a decision object is shown in Fig. 3. Like any other object derived from class qtail it has a member function queue.put to enter a message. Here, this function forwards the message to one of a choice of qtails, again using their queue.put member function. Internally it uses a random number generator RG to choose the target direction according to a branching probability \( p \). To take advantage of this scheme, processes in the toolkit are written to operate on objects of the base class qtail and thus may interface to any object derived from this type.

Using these components allows all process incarnations in the model in Fig. 2 to behave identically and thus to be created from the same process class. Only one additional process type has to be defined to implement the stations S01,...,S03, where messages are generated and entered into the network.

### 3.3 Graphical Program Generation

Program generators are often used to simplify the creation of programs in a special and limited context of applications. They usually compose a program by arranging and parameterizing predefined components according to user specifications [5]. For the sake of efficiency and simplicity, they often immediately generate machine code or some other intermediate language which the average user typically is not acquainted with.

The SIC program structure naturally leads to a separation between the definition of the model's components and their incarnation and parameterization. This not only enables one to establish a modeling toolkit in form of a library of simulation model components. It also forms a basis to further facilitate the generation of simulation models by program generators. In conjunction with a library of model components as described in section 3.2, a program generator may concentrate on creating the main program for SIC defining the model's structure and parameterization.

A system for graphical generation of queueing network simulation programs was developed for SIC. It is based on a graphic editor written for SIC in C++ called NETCAD. The integrated program generator is able to create a SIC program from a schematic drawing that the user has composed. An example for its operation is the network diagram in Fig. 2. It was created using the graphic editor NETCAD. After interactively adding parameters to the model, a SIC program can be generated from this drawing.

---

Fig. 4 Screen view of the NETCAD graphic editor

For drawing simulation models, a multi window mode has been added to the graphic editor. In this mode, diverse model components are offered in a menu which may be used to compose a queueing network model. A view of the editor screen is given in Fig. 4. After parameters have been added, a simulation program may be generated, translated and run without leaving the editor. The results of such a simulation run may in turn be visualized in the editor display, e.g. as distribution functions.

The program generator creates programs which look like those written manually. This increases transparency and enables the user to quickly create a program source by means of the graphic editor which can be extended and modified later on if desired.

When designing this program generator system, care was taken to ensure that the introduction of new model components is achieved by merely setting up configuration files. These are comprised of graphic files which contain the graphical appearance of model components and parameter files which describe their logical structure. For each model component, it specifies recursively the optional and required parameters, such as service time characteristics of a server or the maximum size of a queue. The model components mentioned in section 3.2 were implemented as a first example for this graphical program generation system. New model components may easily be added by editing the configuration files mentioned above.

### 3.4 Statistical Reliability

A central component of every simulation system is the statistical analysis of the values observed during a simulation run. In SIC, the new LRE algorithm\(^3\) [11] for obtaining the stationary distribution function of correlated random variables is implemented, which is able to determine this function with an a priori chosen maximum value of the relative error \( c_{\text{max}} \). Based on the accuracy desired and taking into account the variable's local correlation coefficient, the LRE algorithm determines the number of trials

---

\(^3\) LRE: Limited Relative Error
required and thus controls the length of the SIC simulation run objectively. Other simulation systems often leave it up to the user to determine the simulation run length; the accuracy of the observation is then given as an a posteriori result. In this paper, the results shown in Fig. 6 and Fig. 8 were obtained using the LRE algorithm. In addition to the LRE algorithm, conventional methods for statistical evaluation are offered which determine the moments of observed random variables such as their mean value and variance.

Together with the statistical evaluation, the quality of the random number generators is of significant importance for the reliability of simulation results. Especially high-dimensional correlation effects of these random numbers are among the less obvious factors which influence the reliability of simulation results [9]. Because of this, SIC offers a choice of random number generators: besides conventional pseudorandom number generators, a quasi-ideal table random number generator based on physical phenomena is provided [4].

3.5 Parallelization

Besides the ease and security of modeling, the efficiency of simulations is still an issue, especially in the context of communication systems modeling. Parallel processing offers a way to speed up simulations. In the context of a broader examination of parallel simulation systems [6], the SIC system was extended to enable simulations to be executed as parallel programs. Implemented on a UNIX shared local memory multiprocessor, it is used to explore the simulation speedup that may be gained by using a commercial parallel hardware for this purpose.

![Functional Parallelization in SIC](image)

**Fig. 5** Functional parallelization in SIC

The functional parallelization approach is used to parallelize SIC programs. In this approach, certain functions of simulation programs are distributed to several processors. This kind of task division is performed independent of the model examined. In the parallel version of SIC, the generation of random numbers and the statistical analysis are carried out by separate processors respectively (Fig. 5). This approach was also investigated on a special hardware [2], [7]. The tasks distributed here can work independently from each other, e.g. random numbers may be generated in advance and simulation results may be analyzed asynchronously from their production. A set of buffered communication channels $c_1, ..., c_n$ is required for their cooperation, providing for transfer of the different types of random numbers used in the simulation model, or of the results obtained at different observation points in the model respectively.

Using the object-oriented features of C++ and based upon the structure of SIC it is possible to translate the same simulation program source to either a sequential simulation program or to a parallel one. The internal structure of SIC forms a basis to achieve this in a way transparent to the user: both random number generators and the statistical analysis are called in SIC via member functions of their respective classes. In the single-program case, these classes are implemented to perform the desired operations themselves. In the parallelized program, they act as interfaces to the communication channels mentioned above. Thus the parallel version is integrated mainly by providing an alternative library for some internal classes inside SIC and by a preprocessor, which serves to create the additional processes needed.

4 Network Simulations in SIC

This section describes simulation studies which have been carried out using the system SIC. The first example gives new insight into the effect of different queuing strategies on the statistical properties of the traffic carried. The second example illustrates how the object oriented modeling approach employed in SIC was advantageously applied for modeling the DQDB MAN protocol.

4.1 Queueing Strategies in a Multi-Station Network

The specific advantages of using the modeling approach described show up clearly when examining various scheduling alternatives in more complex networks such as Fig. 2. The behavior of this model when running its stations with different scheduling strategies was observed. Besides the normal FIFO case, the effect of SRPT scheduling in the stations was examined.

SRPT is a preemptive scheduling procedure where processing of a message is suspended whenever a shorter one enters the queue. It has been shown that the SRPT strategy minimizes sojourn times in queueing systems and that it is optimal in this respect over any other scheduling strategy [10,12], even if overhead times are taken into account [3]. By including SRPT scheduling in the SIC toolkit, its effect on sojourn times in the complex network in Fig. 2 could easily be examined [6]. To accomplish this, all nine stations of this model had to be replaced by stations performing SRPT scheduling.

Here an advantage of the toolkit approach can be noted: it allowed the implementation of the SRPT strategy to be coded and verified first in a simple model whose results can be obtained analytically. Without touching their code, the stations performing SRPT scheduling could later be integrated into the complex network model.

Using the LRE-algorithm for statistical analysis, distribution functions of the message delay in each station were obtained. To investigate the influence of the SRPT

---

4SRPT: Shortest Remaining Processing Time first
strategy on the delay time, SRPT was introduced into all stations. The example given in Fig. 6 compares the results obtained for station 9 in Fig. 2. For these results no analytical methods are available. With parameters selected as in [8] and hyperexponential service times with a coefficient of variation of 2 in the stations, Fig. 6 compares the effect of SRPT with that of FIFO scheduling. It can be noted that the complementary distribution function $G(\tau)$ of delay time $\tau$ for the SRPT case lies below that for FIFO by a factor of up to 10. Roughly the same factor is gained for the mean value as well. Additionally, the influence on the local correlation coefficient $\rho(\tau)$ should be noted. While in the FIFO-case it quickly approaches the value 1, it stays below 0.1 in the SRPT case. This is due to the resequencing of messages done by the SRPT strategy.

4.2 Modeling the DQDB Man

To examine the applicability of the modeling approach of SIC the implementation of a real-world problem was examined. The DQDB MAN media access protocol as described in the IEEE 802.6 Draft Proposal [1] was chosen because it represents a modeling problem that contains easily distinguishable components which can be implemented as separate entities in an object oriented simulation model. However, the operation principles of this model may lead to an inefficient simulation implementation. The goal was to find an efficient implementation without sacrificing structure.

4.2.1 The DQDB Protocol

The DQDB\(^5\) network is based on a dual bus architecture (see Fig. 7) providing asynchronous (packet switched) and an equivalent to isochronous (circuit switched) services by reserving part of the bandwidth. Both modes of operation work independently sharing the total capacity of the network.

Access to the bus for asynchronous traffic is controlled by the distributed queue protocol. For each bus free slots are generated by a slot generator. The slot size is 53 octets with a segment payload of 48 octets. This definition is in agreement with the ATM\(^6\) cell size, which will facilitate the connection of DQDB networks and Broadband ISDN.

The distributed queue protocol is based on two control bits in each slot, the busy bit, which is set if the slot is used, and a request bit, which indicates that a station has a packet it wishes to send.

Each station maintains a request counter and a countdown counter for each bus. The request counter is incremented for each request bit received and decremented for each empty slot that passes the station on the other bus. The request counter approximates the number of packets waiting in the distributed queue for one bus. To send a packet on Bus A a station first sets a request bit on Bus B, it then copies the request counter into the countdown counter after which the request counter is set to zero. Now the countdown counter is decremented for each empty slot passing the station allowing access to an empty slot when it reaches zero.

The standard also defines 4 priority levels which are implemented by the corresponding number of request bits within each slot and request and countdown counters within each station, i.e. a distributed queue is implemented for each priority level. These priorities can be used to approximate the SRPT strategy [12] that was discussed in section 4.1. This strategy can be applied to the message level by using a special packet numbering mechanism. This leads to lower mean transfer times for messages because short messages have priority.

4.2.2 Model Structure

First a straightforward model of the DQDB network was implemented in SIC for validation purposes and speed comparison. In this model SIC tasks were defined for the packet generator, the bus and the station. Bus and packet generator are created only once, but each station has a separate task incarnation. In this straightforward model each station maintains its request and countdown counters as described above. This means that each arrival of a slot at a station is an event to be handled by the simulation.

---

\(^5\)DQDB = Distributed Queue Dual Bus

\(^6\)ATM = Asynchronous Transfer Mode
program leading to a very inefficient solution. Each bus operating at 50 Mbit/s generates a slot of 53 octets every 8.48 μsec, accordingly a network with 25 stations will generate about 354 million events in 1 minute of simulated time not counting events for packet arrivals. Especially for low network loads most of the events do not change the state of the station. In the next section an optimized model is described which utilizes the information known within the simulation program to minimize the number of events.

4.2.3 The Optimized Model
In the optimized model a look ahead scheme is used to minimize the number of events. In this model the station does not maintain any counters. This information is now shifted into the slot description. For each slot a station can reserve the request bit or busy bit. If a station wants to set a request bit the slots that have not yet arrived at this station are searched for a free request bit and the slot is marked as reserved. The station will be activated only for this slot. If there is no free request bit a special mechanism is used which will reserve the request bit when new slots are generated. If a station finds a slot marked as reserved for a request, it can decide whether its own request will be first and it can then move the reservation of the other station to a later slot. A similar scheme is used for reserving slots with empty busy bits. Slots are defined within the Bus Task, which offers member functions for the operations described above and which generates new slots.

The speedup of processor time obtained with the optimized model is load dependent as was explained above. For a normalized throughput of 0.1 to 0.9 a speedup factor of 16 down to 2.3 was obtained.

This improved simulation model was validated against results obtained by the straightforward model as well as by another simulation program in PASCAL and by an approximation method [14].

4.2.4 Results
In this section some results are presented that use an exponential arrival process for packets with the arrival rate being equal for all stations. Increasing this arrival rate raises the network load. The packet destinations are generated according to a uniform distribution. This does not affect the network load because each slot can be used only once, but it modifies the transfer time of a packet. The number of stations is 25, the station latency is 8 bits, the bus length is set to 100 km, and the transmission rate is 50 Mbit/s for each bus with a slot size of 53 octets. In the examples given below the total bandwidth was available to the asynchronous service.

**Fig. 8** DQDB Network: mean transfer time versus normalized throughput

**Fig. 9** DQDB Network: Complementary Distribution Function $G(\tau)$, local correlation coefficient $\rho(\tau)$ and relative error $c(\tau)$ of transfer time $\tau$ for station 1.

It takes longer to set a request bit and because the mean propagation delay from station 1 to all the other stations as destinations is longer.

The second type of result obtained is shown in **Fig. 9**. Here the complementary distribution function $G(\tau)$ is shown together with the local correlation coefficient $\rho(\tau)$ and the relative error $c(\tau)$ for the transfer time $\tau$ of one station. Note that the relative error is always below the prescribed value of 5%. For the given network load (normalized throughput 0.9) the local correlation coefficient shows the influence of the distributed queuing strategy which is closely related to the FIFO strategy.
5 Conclusion

In the context of SIC, C++ has proven well suited for simulation programming in several respects. Besides the obvious fact that its object oriented features enable the implementation of a model scenario in a natural way, it maintains an efficiency of program execution which allows the utilization of these features adequately in a simulation without paying a performance penalty. The separation between the definition of model components and their incarnation was well supported by C++. This was especially important for the graphic program generation system. C++ also helped to implement a version of SIC offering transparent parallelization on UNIX multiprocessor systems.

An extension that is planned for SIC concerns the event list. Depending on its size, different algorithms are optimal for its implementation. Choosing the appropriate algorithm respectively may avoid wasting resources. Therefore, a dedicated event set class will be introduced for the implementation of this feature. Further, it is planned to include provisions for animated simulation into the simulation system resp. the graphic editor. Such a feature has educational as well as practical aspects: while helping to understand how simulation works in general, it may also help to verify the correct operation of a simulation program interactively.

6 Acknowledgement

The authors would like to thank Leonhard Schmickler and Asbjørn Haugen for their support concerning the DQDB results. In addition, we wish to thank Prof. Dr.-Ing. F. Schreiber for the continuing support of the project, within which the SIC system was developed, and for discussions concerning this paper.

References


