OOS '94, Tempe AZ

AN OBJECT-ORIENTED LIBRARY FOR SIMULATION OF COMPLEX HIERARCHICAL SYSTEMS

Hartmut Kocher
Rational[†]
Rosenstraße 7
D-82049 Pullach im Isartal
Germany

Martin Lang
University of Stuttgart
Institute of Communications Switching
and Data Technics
Seidenstraße 36
D-70174 Stuttgart
Germany

ABSTRACT

As today's systems are becoming more and more complex, simulation is often the only viable way to verify the functionality of a system, or to estimate its performance. Simulating a complex system is itself a complex task.

In this paper, we will present a flexible general purpose library for the simulation of complex hierarchical systems. The library is implemented in C++ and uses high-level abstractions that are closely related to the problem domain. This eases the mapping from a simulation model to an actual simulation program. The library supports hierarchical decomposition of simulation models into submodels and model components. Model components are strictly encapsulated and communicate with each other using a handshake protocol. This offers the ability to highly reuse standardized model components and quickly create or modify a simulation model using a 'plug-and-play' approach.

After discussing the main abstractions of the library, we will describe a sample application. Finally, we outline the modifications that are necessary in order to support distributed simulations.

1 INTRODUCTION

Today's systems are becoming more and more complex. The human mind is unable to comprehend complex systems in their entirety. Therefore, complex systems need to be structured in way that allows humans to cope with this complexity. This is usually achieved by breaking down the system into a hierarchy of subsystems and modules (Booch 1991). In such cases, simulations can help in evaluating different design choices. Simulation programs can be used to verify the functionality of the system as well as to estimate the performance of the target system.

Simulating a complex system is itself a complex task. Therefore, it is important to structure simulation software in a suitable way. The complexity of simulation programs can be reduced by decomposing the simulation model into a hierarchy of submodels that can be refined in further steps. Although many parts of a simulation model could potentially be reused across different applications, this is not supported very well in current simulation programs due to strong coupling between model components. Reuse of models and submodels would be an important step towards economic development of simulation programs that could be used to investigate several design choices in a cost efficient way. This is a general problem of software development and is one of the reasons for the so-called software crisis. The object-oriented paradigm promises to improve the situation dramatically. That's why more and more simulation libraries are written in an object-oriented programming language.

Currently, two approaches can be differentiated: on the one hand, simulation environments are mostly targeted at specific application areas. They provide high-level abstractions that are taken from the problem domain. Therefore, they are easy to learn and use. Sometimes, they even offer graphical user interfaces and their own simulation language, e.g., (Belanger 1990; Melamed and Morris 1985). Unfortunately, most simulation environments cannot be extended by the user, or do not adapt very well to different needs even within the same application domain.

On the other hand, general purpose simulation libraries promise to overcome these difficulties. They are developed using general purpose programming languages. Therefore, they can be extended and adapted easily by the user. Recently, many simulation libraries have been written in object-oriented languages, mostly C++. Unfortunately, most of them focus on implementation issues rather than using abstractions from the problem domain. There is a semantic gap between the low-level abstractions they offer, such as process classes, and the problem-oriented abstractions the user wants. Simulation libraries that are more problem oriented are just emerging, e.g., (Mak 1991; Zheng and Chow 1993; Vaughan et. al. 1991). However, most systems don't offer support for hierarchical systems. This makes it difficult to implement reusable submodels, and components that can be further

[†]This research was performed while the author was a member of the Institute of Communications Switching and Data Technics at the University of Stuttgart.

refined as the design evolves. As we have already pointed out, these are essential features for simulating complex systems.

This paper describes a general purpose simulation library that has already been used extensively for the simulation of complex communication systems. It tries to narrow the gap between the problem domain and implementation by using high-level abstractions. Therefore, users can easily map their simulation models to actual simulation programs. The library can be easily extended because it is written in standard C++. The implementation takes advantage of the latest additions to the C++ language, namely templates and the exception handling mechanism (Ellis and Stroustrup 1990). It uses few basic abstractions and emphasizes a clean software architecture. Hierarchical system decomposition is supported, and it is even possible to write distributed simulation programs. Strict encapsulation with clean interfaces between model components enables massive reuse of simulation models. A detailed description of the simulation library can be found in (Kocher 1993).

2 AN OBJECT-ORIENTED SIMULATION LIBRARY

2.1 Software Architecture

The following section describes the overall software architecture of the simulation library. Figure 2.1 shows a typical system. Two main parts can be distinguished. The simulation support subsystem contains all components that are necessary to control the execution of a simulation program. It also contains classes that simplify the development of simulation programs, like a modular I/O concept. It is further described in Section 2.5.

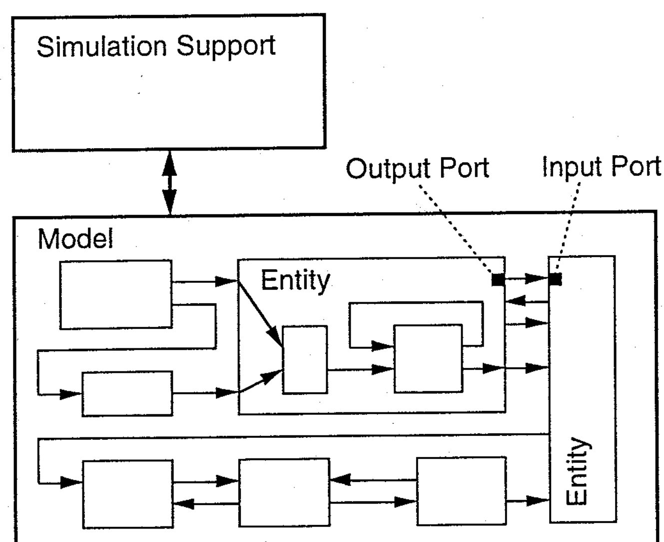

Figure 2.1: System Overview

The main part of the system is the simulation model. The simulation model can be hierarchically decomposed in submodels and in model components. The latter are called entities. Entities communicate with each other by exchanging messages. All messages are derived from an abstract base class that defines some common properties, e.g., the message type. Contents and meaning of a message are user defined. Each entity can evaluate only those aspects of a message which it is interested in.

A port mechanism is used for communication. Transferring messages between entities is as simple as connecting the input and output ports of these entities. A handshake protocol ensures that both entities are ready to exchange messages before they are actually sent. Entities can be seen as black boxes that communicate with the outside world using ports. This strict encapsulation allows separation of the behavior of an entity from the structural arrangement within the model. Therefore, it is easy to insert a new entity between existing entities without modifying the existing ones.

Our simulation library is based on an event-driven paradigm. We selected this approach over a process-oriented paradigm because it was clear how a hierarchical event concept should work, whereas we could not come up with a suitable definition of hierarchical processes. In event-driven simulations, events are used to plan future activities. Events are entered in a sorted event list and processed later. The meaning of an event depends on the entity that generated it. Events are passed to the entity for processing. The entity might process the event itself, or may pass it on to its parent entity. This supports hierarchical processing of events.

A model entity is a special entity that has a built-in event list. Usually, the model entity stays at the top level of the simulation model hierarchy. Since a model may be composed of more than one submodel, the library supports more than one event list. The submodels are responsible for synchronizing distributed event lists.

The following sections describe the above-mentioned mechanisms in more detail.

2.2 Model Components

A simulation model can be seen as a network of model components, which we call entities. All entities are derived from a base class *TEntity* that defines the common properties of all entities. An entity has a local name, which can be chosen arbitrarily. A global name can be formed by chaining the local names of all owning entities, just like path names in a file directory tree. Names are mainly used for identifying entities in error messages during program development, and for I/O-purposes. *TEntity* also defines methods for dealing with ports and events.

Decomposing a model into a hierarchy of entities is an important means to reduce overall complexity. Therefore, we spent a lot of effort to come up with a useful concept. Each entity can contain other entities internally. If this principle is applied recursively, it leads to a tree structure of entities with the model entity as the root. Each entity has a reference to its parent entity. Figure 2.2 depicts a simple queuing model that shows two network nodes in a communication system. Each node consists of an input queue and a server entity. Figure 2.3 shows the resulting object tree using the notation of (Booch 1991; Booch 1992). The mapping from the components of the simulation model to the implemented entities is straightforward.

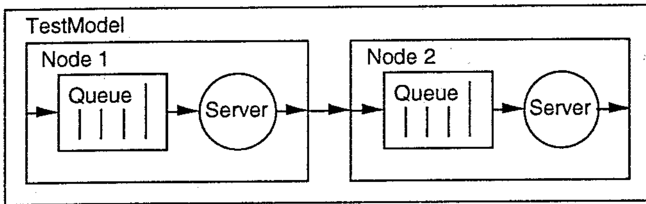

Figure 2.2: Simulation model of a simple queuing system

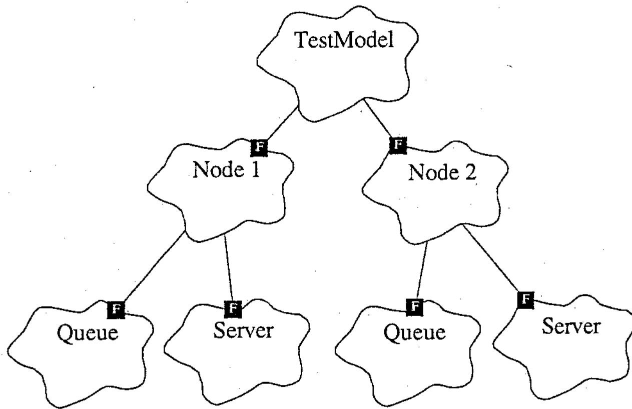

Figure 2.3: Object diagram of the simulation model

On one hand, an entity **is** a model component; therefore, it has to be derived from *TEntity*. On the other hand, it may **contain** other entities. These may be defined as class members and initialized in the constructor, or they may be created dynamically depending on some input parameters. Figure 2.3 shows such a containment hierarchy.

Obviously, it is not enough to just support nesting of entities. The more interesting part is how they interact. An important aspect of this is the coupling between entities in different layers of the hierarchy. In order to ease the reuse of entities, the coupling should only be as close as absolutely necessary. This can be achieved by following a few simple rules.

During initialization, each entity gets a reference to the parent entity. Because it does not know the exact type of the parent, it can only use services that are already defined in the base class *TEntity*. Because of polymorphism, the behavior of those services still depends on the actual type of the parent entity. That way, many services can be delegated to parent entities without knowing the structure of the containment hierarchy. Whereas child entities may only use anonymous services of the parent entity, parent entities do know all child entities. Therefore, they are allowed to call all methods of their children directly without sacrificing encapsulation. This is the most important principle of the simulation library. It guarantees loosely coupling between entities, and therefore supports independent reuse of entities and whole submodels in different applications. As will be shown in later sections, delegation is used in many places. Polymorphism together with delegation is the key to a flexible library, because it allows to change the behavior of the system dynamically without breaking encapsulation. To add new functionality, a new class with a different implementation is derived. Because the interface doesn't change, it is simply a matter of replacing some objects to get a completely new behavior.

2.3 Port Concept

The port concept is used to pass messages between entities. To improve type safety, input and output ports are distinguished. All connections are unidirectional point-to-point connections between two ports. Ports are registered with their owning entity during construction. Ports may be defined as member objects of the owning entity, or they may be created dynamically. The latter is useful for general purpose components like multiplexers, where the number of input ports depends on the simulation model. Two ports can be connected by calling the *Connect* method of the *TEntity* class. This method also checks if a connection is legal.

All messages must be derived from a common base class *TMessage*. The content of a message depends on the simulated problem and can be defined in derived classes.

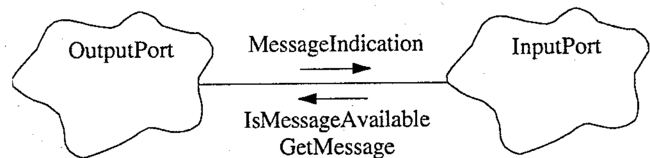

Figure 2.4: Handshake protocol between ports

Messages are passed between ports using a handshake protocol. This is shown in an object diagram in Figure 2.4. After an entity notifies an output port that a new message is available the port calls the *MessageIndication* method of the corresponding input port. The receiving entity can then decide if it is willing to accept the message. It can do so by calling the *GetMessage* method of the port. If it is unable to receive a message in the current state, it may simply ignore the call. In this case, the sender is blocked. Later, the receiving entity may call *IsMessageAvailable* to find

out if there are messages to receive, and call *GetMessage* to actually receive them.

This simple protocol adds a lot of flexibility because entities don't have to know how messages are created or utilized, or if the receiving entity is in a state where it can accept new messages. Because of this loose coupling between ports, it is always possible to insert new entities between existing ones without influencing the way messages are transferred. For example, it would be possible to insert a multiplexer between a queue and a server. When the server is ready to accept a new message it calls the *IsMessageAvailable* method of the multiplexer. Depending on the polling strategy, e.g., priority, round-robin, etc., the multiplexer would query all connected input queues for new messages. Neither the queue nor the server entity would have to be modified. This greatly enhances opportunities for reuse because many models can be changed by simply inserting new entities. Other port schemes that don't implement flow control mechanisms are less flexible because messages cannot be blocked between entities, e.g., (Mak 1991).

Because entities know their ports, they can reference them easily, and call port methods directly. Ports also know their owning entity, but they don't know which method to call in case of a message indication. Especially, if an entity has more than one port, all ports would call the same method. Therefore, a class *TMessageHandler* was introduced to decouple entities and ports. A message handler may either handle a message directly, or delegate it to the owning entity. Template based message handlers allow to call arbitrary methods of the entity class in a type safe manner.

Although it would be possible to create special entities to count messages, or manipulate them, the overhead would be prohibitive. Message filters that can be installed in every port offer a more elegant solution for these kind of problems. If filters are installed in a port all handshake calls between ports are first dispatched to all filters before they are sent to the port or a message handler. Again, template based filters that are derived from the *TMessageFilter* class can be used to delegate these calls to other classes in a type safe manner. By installing two message filters that work together, message transfer times between any two ports can be evaluated.

2.4 Event Handling

Event processing is the core of any event driven simulation program. Normally, events are stored in an event list that is sorted by event time. When the current simulation time matches the event time, the event is processed. Additionally, the concept presented here supports hierarchical event handling.

All events must be derived from a common base class *TEvent*. Similar to the usage of message handlers and filters, users can derive their own event classes. Template based classes can be used to delegate event processing to arbitrary classes, e.g., the entity class that created the event.

The *PostEvent* method of class *TEntity* takes an event and the event time as parameters. Once an event has been handed over to an entity, the entity tries to find a suitable event handler that is willing to handle the event. Event handlers must be derived from the *TEventHandler* base class. Events have different types. Event handlers can either handle events of one specific type, or of all types. Event handlers may be installed in any entity. First, entities search their own handlers to find one that is willing to handle the event. If none is found, the *PostEvent* method of the parent entity is called. This technique is applied recursively until either a suitable handler is found, or the root of the model hierarchy is reached, which would cause an exception. To improve performance, entities cache the most recently used handler. Hierarchy levels with no registered handlers are skipped automatically.

Event handlers may either intercept events, or pass them on to the next higher level of the entity containment hierarchy. An event list is just a special case of an event handler that stores events and processes them later. In order to intercept events not only when they are posted, but also before they are processed, handlers may add an embedded event to the current event. When the *ProcessEvent* method of the original event is executed, all embedded events are processed before the original event.

With this concept, the parent entity is able to manipulate all events of their child entities without modifying the children. Therefore, this scheme can be used for conventional event processing, and for anonymous communication between entities in different hierarchy layers.

Model entities are special entities that include event lists. The *PostEvent* method of a model simply inserts the event in the event list. The simulation control class and the model entities work together to retrieve the next event in the event list, which is processed by calling its *ProcessEvent* method. The scenario in Figure 2.5 is based on the model that was shown in Figure 2.2. It shows a scenario where an event is posted by the server. Because no event handlers are installed in the server and network node entities, each entity delegates the *PostEvent* method call to its parent entity. That way, the event follows the hierarchy up to the model entity, which inserts the event in the event list. Later, the event is retrieved and executed. Because submodels may be installed at any part of the hierarchy, more than one event list may be used. If event lists are distributed, the models are responsible for synchronizing them.

Again, delegation and polymorphism are used to add flexibility to the simulation library. Because existing classes do not have to be modified to add new functionality, they can be more easily adapted to new uses, thereby greatly enhancing reuse opportunities. Also, hierarchical event processing is one of the key

concepts to support distributed simulation almost transparently (see Section 3.2). Although the concept is easy to understand and use, we found no other simulation library that supports it.

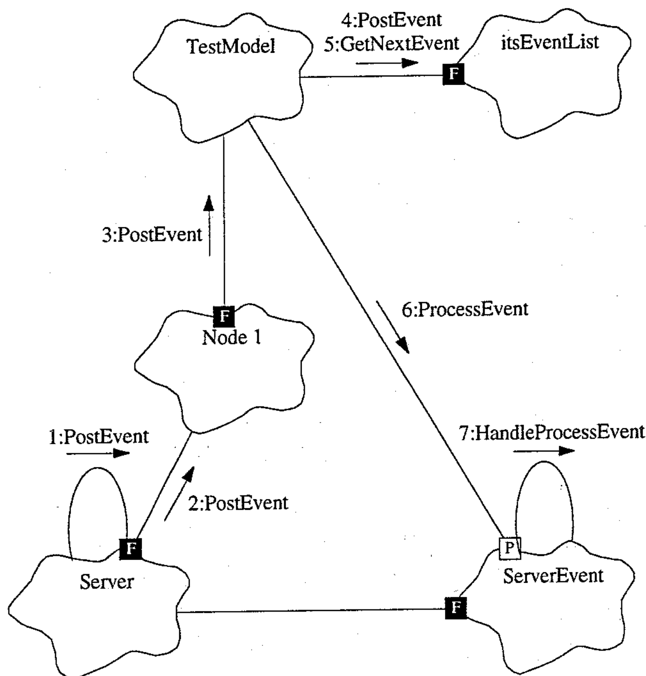

Figure 2.5: Hierarchical event handling

2.5 Other Concepts

2.5.1 Simulation Control

The execution of simulation programs must be controlled. The main tasks are initializing data structures, processing input parameters, running the simulation, collecting and printing the results the user is interested in, and finally stopping the simulation. Our library supplies a set of classes that provides a flexible environment for simulation control. It can easily be customized by overriding selected methods.

It must be emphasized that our environment is useful for both functional, and stochastic simulation. In the following, we will concentrate on stochastic simulation because of some additional requirements that have to be met to get statistically accurate results. Stochastic simulation is used to evaluate the performance of a system. In general, a stochastic simulation run consists of several phases. After launch, the system is in a nonstationary phase (warm-up period). Then, if the process is stable, it moves asymptotically towards a steady state (Pawlikowski 1990). To get statistically uncorrelated data, the observation process has to be replicated several times. This is usually done by dividing the simulation run into a series of batches. After all batches are finished, the means, variances, coefficients of variation, confidence intervals, etc., of all interesting measures can be estimated. Note, that simulation of nonstationary processes is also possible with this environment. The only change that needs to be made is the use of special data collection classes (see Section 2.5.2).

TSimulation is an abstract base class that provides virtual methods to start and stop the warm-up period, batches, and the whole simulation. The derived class *TStdSimulation* overrides these methods and implements the features needed for stochastic simulation. Additional functionality can be implemented by deriving new classes from *TSimulation.*

It is often necessary for objects to know in which phase (warm-up, or n-th batch) the simulation currently is, or when the individual phases start and stop, respectively. For example, at the end of each batch the characteristic values of all observations have to be calculated and all data collection variables must be reset for the next batch. The class *TSimulationControl* is provided for this purpose. All classes that need to be informed of phase changes must be derived from this base class. During creation of a simulation control object, it automatically registers itself with a global simulation control manager. Whenever the phase changes, the simulation object informs the simulation control manager which in turn notifies all registered simulation control objects.

How does the simulation object know when it should proceed with the next simulation phase? For this purpose, the simulation object has notification handler objects for each simulation phase. These notification handlers manage a collection of notifier objects, which are responsible for detecting the end of a simulation phase. Notifiers could be message counters, timers, or objects that watch the statistical measurements and stop when a predefined confidence interval is reached. Since each notification handler is able to manage more than one notifier, several conditions can be combined.

This distributed approach to simulation control adds significantly to the overall flexibility of the simulation library.

2.5.2 Support Classes

To further simplify the development of simulation programs, the library includes a number of useful support classes that will be briefly discussed in this section.

Our library provides a hierarchy of random number generator classes that implement some of commonly used algorithms (L'Ecuyer 1990). Based on the random number generators, we implemented a hierarchy of different distributions ranging from simple uniform, binomial, or poisson distributions, up to sophisticated state-dependent models for video sources that are needed for simulating broadband communication networks.

In the previous section, we mentioned the collection of samples and the calculation of characteristic statistical values. This functionality is provided by a separate class hierarchy where

all classes are derived from a common base class *TStatistic*. To be more flexible, the creation of statistic objects is done using a global manager object. For example, after the user has changed the properties of the statistic manager all classes that create statistic objects will automatically use the new properties.

To simplify the collection of sample data during a simulation run, a number of meter classes are provided. These meter classes can easily be connected to any port of an entity. Currently, two types of meters can be distinguished, meters that simply count messages, and meters that measure transfer times, i.e., the time a message needs to travel from one point of the model to another. When a meter is attached to a port, it installs a message filter. Because the filter concept is a basic part of the port mechanism, no modifications of entities are required. The message filter notifies the meter as soon as a message arrives at the port so it can count the message, or place a time stamp on it for measuring transfer times. The measurement regions of different meters may overlap.

During development of a simulation program, and for functional simulations, the user needs some debugging support. For this purpose, a sophisticated trace facility that is based on different user definable trace levels has been added to the library.

Finally, the I/O mechanisms of the library are presented. For input of simulation parameters, a parser environment is provided. The parser reads text from an input stream. If it detects a predefined keyword it reads a value that is assigned to the keyword, or it invokes other user-defined operations. The parser can be extended easily so that users can add their own keywords and operations. This file oriented input has advantages for time consuming simulations that can be started in the background and need no user interaction. If interactive control of the simulation is desired, an additional graphical layer can be added. This layer writes the commands in a parser readable form to a temporary stream which serves as input to the parser. This method has already been used successfully for functional simulations of communication protocols.

The output of simulation results can be controlled through usage of styles. A print manager class reads the definitions of styles from a file. Styles are used to define which values are printed, the output format for these values, and for printing headers and comments. Actual simulation results are marked by keywords, and are replaced with current data in the output stream, similar to mail merge applications. Each entity defines a number of keywords for the results it can offer. Styles can be defined hierarchically, so they fit into the hierarchical structure of the entities. This flexible concept simplifies the reuse of entities or models, because no code needs to be modified in order to print results in a different format. After editing the style file, the new format is in effect immediately without the need to recompile the simulation program.

3 APPLICATIONS OF THE LIBRARY

3.1 Simulation of a Satellite Communication System

In the following section, we will describe some insights that we gained while using the library for the simulation of a satellite communication system. In addition, we implemented the same simulation model using a simulation library written in the procedural programming language Pascal. The development process and the final versions have been compared.

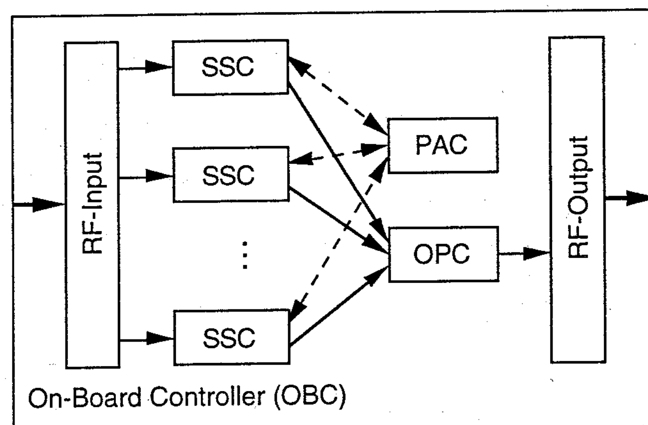

Figure 3.1: Block diagram of a satellite switching system

The system we wanted to simulate was a modular satellite switching system, where a satellite is used to interconnect several terrestrial terminals with different communication protocols (Piontek 1989). Switching inside the satellite is performed by a multiprocessor system, the so-called „On-Board Controller". Figure 3.1 depicts a block diagram of the system. The „Signalling and Switching Controllers (SSCs)" are responsible for connection management, the „Path Allocation Controller (PAC)" manages all resources of the satellite, and the „Output Processing Controller (OPC)" sends the packets down to the terrestrial terminals. A detailed queuing model of the simulation system is depicted in Figure 3.2.

The mapping of the queuing model to the simulation program is straightforward. A hierarchy of entities can be derived directly from the model. During the development of the program another great benefit of the object-oriented approach became obvious. The library supports an incremental development process. Due to the encapsulation of the entities and the framework which is provided by the library, it is always possible to build a reduced model which can be tested and simulated separately. Later, individual entities are combined to hierarchical entities, and their ports are connected. This has the advantage that an executable program is available during every stage of the development process. The need to integrate a large and complex system in one step does not exist. Since the individual entities are already tested, testing of the whole program can be reduced to validating interactions between entities.

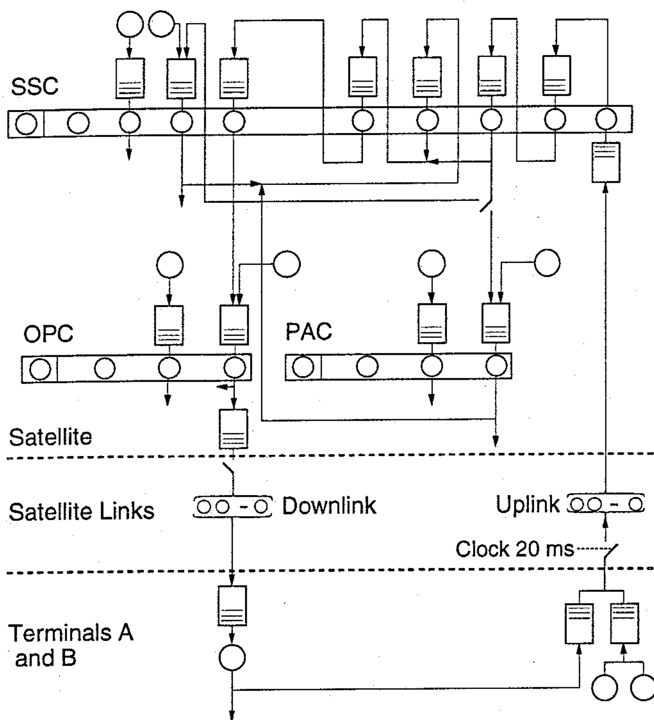

Figure 3.2: Simulation model of the communication system

During the development of the program, we could heavily reuse the framework provided by the library. The queues, service phases, packet generators, and transmission delay entities could either be used directly from the library, or had only to be slightly modified. These modifications could easily be accomplished by deriving new classes from the library entities, and overriding specific methods. Only very specialized entities had to be implemented from scratch, e.g., the entities that implement the communication protocol. Interestingly, we were able to find useful generalizations even for those specialized entities. The generalized entities were added to the library. Although this may lead to some additional work, it has two benefits. First, the library becomes more complete with every project, which reduces the work required to build future simulation programs. Second, it leads to clearer abstractions that makes it easier for developers to understand complex systems.

A comparison of the source code required to implement the object-oriented simulation program versus the Pascal program lead to 27% less code in the object-oriented program. If we subtract those parts of the program which could be added successfully to the library we end up with 42% less code. Also, the Pascal program was hard to understand because of many case-statements and global variables that were necessary due to functional decomposition.

To demonstrate improved maintainability, we added some functionality to the queuing model. Whereas a great deal of effort was necessary to incorporate the features into the Pascal program, the object-oriented program was changed within a couple of hours. Also, because the required modifications affected only a few entities, testing could be limited to these entities. In summary, the object-oriented solution required less effort, was better structured, easier to code, test, debug, and maintain.

3.2 Extensions for Distributed Simulation

This section describes briefly the extensions to the library that enable distributed simulation. Some of the concepts have already been successfully implemented. In general, we see two different strategies for implementing distributed simulations. The first one is to distribute simulation phases to different machines. This works well for stochastic simulations with a number of batches. The second strategy is to split the simulation model into several submodels, and to distribute these to different machines. Both strategies will be discussed in more detail in the following sections.

As mentioned above, one possible way to distribute a simulation is to run each batch on a different machine. The simulation is controlled by an object of a class that is derived from the *TSimulation* class. The easiest way would be to start identical copies of the simulation program on several machines. Each copy must be initialized with a different seed for random number generation. At the end of the warm-up period and one or more batches, the results are collected and evaluated. With its flexible output facility, the library simplifies the final evaluation. The distribution of the copies and the final evaluation can be incorporated in the simulation control class. No other changes to the simulation program are required.

If the batches are distributed on different machines, all copies of the program have to perform the warm-up period. If this overhead is unacceptable an object-oriented database may be used to store the system state after the warm-up period. After that, each process could read the database, set a new seed, and run its batch. Concluding, we can state that our simulation library can easily be adapted to support the distribution of batches on different machines transparently.

The second strategy is to split the simulation model into several submodels. All entities of a submodel are connected using the standard port mechanism. Therefore, each submodel is complete and can be executed sequentially. The communication between distributed submodels is done by special entities. These entities must guarantee synchronization. For this purpose, a class *TSyncEntity* which provides a special synchronization method can be derived from *TEntity*. This approach has already been applied successfully to simulate a high speed communication network with ring topology.

Finally, special port classes may be derived from the standard ports. These new port classes are responsible for synchro-

nizing entities. Because of the handshake protocol used, the port mechanism can be distributed easily over a network. This enables the subsequent distribution of a sequential program. Due to strict encapsulation and the hierarchical concepts in the original library, important parts of the program can be modified with almost no impact on users of the library.

4 SUMMARY

In this paper, we presented a flexible object-oriented simulation library and some experiences gained from its use. One of the main concepts of the library is complete support for hierarchical decomposition of simulation models including hierarchical event processing. This enables direct mapping of complex simulation models to simulation programs, and also supports iterative refinement of models as the design evolves.

High-level abstractions close the gap between the problem domain and the actual implementation. Developers can focus on their simulation problems because the library provides all basic concepts. The resulting programs are well structured, easy to understand, and easy to implement. The chosen abstractions are very flexible and can even be used for distributed simulation. Strict encapsulation and a clear software architecture support reuse of model components and whole submodels.

Iterative development of complex simulation programs is encouraged because model components can be implemented and tested separately. The problem of integrating and testing large portions of code does not exist. Therefore, the time needed for system integration can heavily be reduced. Furthermore, modifications are easy to implement because they can be seen as another incremental step in the development cycle.

REFERENCES

Belanger, R.F. 1990. "MODSIM II: A Modular, Object-Oriented Language." In *Proceedings of the 1990 Winter Simulation Conference* (New Orleans, LA). 118-122.

Booch, G. 1991. *Object Oriented Design with Applications*. Benjamin Cummings, Redwood City, CA.

Booch, G. 1992. "The Booch Method: Notation." Rational, Santa Clara, CA.

L'Ecuyer, P. 1990. "Random Numbers for Simulation." *Communications of the ACM* 33, no. 10: 85-97.

Ellis, M.A. and B. Stroustrup 1990. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, Reading, MA.

Kocher, H. 1993. "Design and Implementation of a Simulation Library Using Object-Oriented Methods." Dissertation (Submitted). Institute of Communications Switching and Data Technics, University of Stuttgart, Germany. [In German].

Mak, V.W. 1991. "DOSE: A Modular and Reusable Object-Oriented Simulation Environment." In *Proceedings of the SCS Multiconference on Object-Oriented Simulation* (Anaheim, CA, Jan. 23-25). The Society for Computer Simulation, San Diego, CA. 3-11.

Melamed, B. and R.J.T. Morris 1985. "Visual Simulation: The Performance Analysis Workstation". *IEEE Computer* 18, no. 8: 87-94.

Pawlikowski, K. 1990. "Steady-State Simulation of Queuing Processes: A Survey of Problems and Solutions." *ACM Computing Surveys* 22, no. 2: 123-170.

Piontek, M.; W. Berner; H. Kocher; and M.N. Huber 1989. "Frame organization and signalling for an autonomous switching satellite." In *Proceedings of the First European Conference on Satellite Communication.* Munich, Germany.

Vaughan, P.W.; D.E. Newton; and R.P. Johns 1991. "PRISM: An Object-Oriented System Modeling Environment in C++." In *Proceedings of the SCS Multiconference on Object-Oriented Simulation* (Anaheim, CA, Jan. 23-25). The Society for Computer Simulation, San Diego, CA. 32-39.

Zheng, Q. and P. Chow 1993. "EXsim: A General Purpose Object-Oriented Environment for Descrete-Event Simulations." In *Proceedings of the 1993 Western Simulation Multiconference* (La Jolla, CA, Jan. 17-20). The Society for Computer Simulation, San Diego, CA. 15-21.