A LAYERED APPROACH TO SIMULATING COMMUNICATION SYSTEMS USING SIMULA

James P. Hauser and Dennis J. Baker
Naval Research Laboratory
Washington, D.C. 20375

ABSTRACT

In this paper we describe a Simula package (or 'context') for simulating radio communication networks. The context is developed in layers, the higher layers in the context being patterned after the levels of the ISO/OSI Reference Model. Each of the layers in our context is developed as a separate subclass of the standard Simula class Simulation, and layers that model OSI layers include interfaces to adjacent layers. The context provides an easy-to-use interface to a model of the Physical Layer (Layer 1) of the OSI model. Within Layer 1 we provide models for several types of radio communication systems including spread spectrum transmitters and receivers. Users can redefine some of the attributes of the transmitters and receivers defined in Layer 1 without the need to modify Layer 1 itself. We illustrate our approach with the example of a radio network that uses a carrier-sense multiple-access protocol for channel access.

INTRODUCTION

During the first half of this decade the International Standards Organization reference model for open systems interconnection (ISO/OSI) [1] strongly influenced how communication systems were conceptualized and designed. The OSI model provides a framework for decomposing complex communication systems into layers, each of which performs a well-defined and limited set of functions. Moreover, the implementation details of each layer are hidden from other layers behind precisely defined interfaces. Thus, one has the flexibility of changing the implementation of a particular layer without disturbing the rest of the system (i.e., other layer implementations) as long as the interface to the modified layer remains unchanged. The motivation for the layered approach is to reduce the amount of complexity inherent in communication system designs. Through layering, the degree of system complexity is no greater than that which is found in the individual layers.

The motivation for a layered approach to the simulation of communication systems is much the same as for the design of the systems themselves. First of all, we would like to reduce the complexity of the codes which implement our simulation models. In addition, we want the simulation codes to reflect the designs of our systems as nearly as possible. Since many of the communication systems we wish to simulate are layered, simulation codes that are layered will not only be less complex, but will more accurately reflect the designs they are modeling. The desirability of a close resemblance between the simulation code, which models a system, and the system itself may not be obvious, since a correct simulation may be written in a fashion that obscures the simulation model. The experience of the authors has convinced them that simulation codes which mask the underlying simulation models are difficult to develop and maintain. The techniques for writing simulation programs, which we describe in this paper, has resulted in code that is flexible as well as easy to develop and maintain.

Previous work in this area consists mainly of Britwistle's paper on the advanced use of Simula [2]. In that paper, Britwistle layered approach to modeling OSI systems in Simula and gave an example based on the simulation of a fixed topology local area network.

Our paper focuses on the development of a Simula context for modeling variable topology communication networks. In addition, we include a description of techniques for modeling OSI layer interfaces, which is not covered in [2].

In this paper we begin with a discussion of the Simula class construct, which is used to develop layered contexts as well as object templates. We describe our own layered context and give an example of its use in building a simulation model for a communication system using a carrier-sense multiple-access (CSMA) protocol. Finally, we present our conclusions.

SIMULA CLASSES

Simula [3,4] is a block-structured, object-oriented language with class inheritance capability. Simula obtains its block structure from ALGOL [5], which is contained within Simula as a subset. It adds to ALGOL a special kind of block called a class. A block is designated as a class by prefixing it with the Simula keyword class:

```
CLASS class_name;
BEGIN
  class body
END;
```

Simula classes can be used as object templates or as Simula contexts. A context is a Simula class containing data structures, procedures and object templates which serve as a set of tools to extend the capability of Simula to handle problems in a specific area of interest. A context may be extended (using Simula's class inheritance features) by developing a subclass of the context class. Alternatively, one may use the context directly by prefixing the main program block by the name of the context. In either case, the context is normally compiled separately from the main program.

Class inheritance is effected by prefixing one class with another. We rely heavily on the inheritance capability of Simula classes to provide layering in our context. For example, in figure 1 we depict the context in which our physical layer model, Layer 1, is built. The foundation of the context is Simula itself. Two standard Simula classes, Simset and Simulation, are added to Simula and together provide a context available to every Simula user. Next, we provide a class called Stats. Class Stats has the following form:

```
simulation CLASS stats(max_num_of_nodes);
  INTEGER max_num_of_nodes;
  BEGIN
    (code which implements class stats) ...
  END;
```

36.2.1

U.S. GOVERNMENT WORK NOT PROTECTED BY U.S. COPYRIGHT
In the first line of Stats code we see that the class declaration for Stats is prefixed by simulation. This has the effect of passing on to class Stats the entire context of class Simulation, which in turn has the entire context of class Simset. By the time we define queues class Layer1, Layer1 has inherited the entire context which has come before and in turn may pass it on along with its own object templates, procedures and data structures to any class or block that uses layer1 as a prefix. In this manner we can build a context incorporating all the OSI layers one layer at a time. Figure 1 gives a pictorial representation of a context developed along the lines of the preceding discussion and is followed by a short description of each layer.

| LAYER7 (Application) |
| LAYER6 (Presentation) |
| LAYER5 (Session) |
| LAYER4 (Transport) |
| LAYER3 (Network) |
| LAYER2 (Data link) |
| LAYER1 (Physical) |
| QUEUES |
| MATRICES |
| EVSIM |
| STATS |
| SIMULATION |
| SIMSET |
| SIMULA |

Figure 1. A layered context for modeling communication systems and the corresponding OSI layers

- Simset, a standard Simula class (context), implements queues as two-way linked lists by defining two new classes (templates), Link and Head. Any object prefixed by link can be inserted into or removed from any queue, which is an instance of class Head.
- Simulation, another standard Simula class, supports discrete event simulation. Class Simulation defines three new classes - link class event notice, link class process and process class main program. Also, an instance of class Head is created that serves as an event-notice queue. Class Simulation provides a set of procedures for scheduling event-notice. When an event-notice becomes current the process that it references becomes active. The main program is also a process with its own event-notice and thus becomes another member of the set of quasi-parallel processes that constitute a Simula system.
- Stats provides abstract data types for statistics collection [6]. In our implementation of class Stats we also introduce the concept of a node. The introduction of object-template class Node at this relatively low layer in the context facilitates the inclusion of Node reference variables and class prefixing in tailoring the statistics collection and higher layer facilities of the context to a node-oriented model.
- Evsim provides an event-process facility [7], which permits a process (i.e., an event-process) to have multiple event notices pending in the event queue. This is an important extension to the Simula process as defined in class Simulation, which permits each process to have only one event notice in the event queue at any one time. Also, Evsim provides for event tracing. Event-processes play a significant role in the design of Layer1 code and code for higher layers built on top of Layer1.
- Matrices supplies a template and procedures for manipulating matrices.
- Queues extends the basic queue handling facilities of class Simset to provide procedures Put and Get [8] along with an interface to queue servers.
- Layer1 (OSI Physical layer) [9] uses many of the features just described and adds to them its own set of classes and procedures oriented toward the modeling of communication systems. This is the first layer in our context that implements an OSI interface. Layer1 provides object templates to model communication hardware and communication channels. In addition, Layer1 implements part of the Layer1/Layer2 (1/2) interface.
- Layer2 – Layer7 (OSI Data Link - Application layers) use the entire context described above with all its features and capabilities, each adding its own unique protocol(s) and interface to the context which will become the final simulation model.

Another way to use a class is as an object template. An object template is used as a pattern to create one or more objects of the same type. This is accomplished by using the Simula keyword new and an object thus created is called a class instance. Object creation (i.e., class instantiation) is illustrated by the following Simula statement:

```
xmtr :- NEW transmitter;
```

The attribute xmtr is a special type of Simula variable called a reference variable. Reference variables serve as pointers to class instances and must be properly typed somewhere in the Simula context. In this case we need a type designation as follows:

```
REF(transmitter) xmtr;
```

which designates xmtr as a pointer to transmitter objects. The symbol " :- " is read as denote and serves as a replacement operator for reference variables. NEW causes the instantiation to occur.

Object templates may also be extended via class inheritance. For example, to extend the model of a transmitter to incorporate features specific to ultra-high frequency (UHF) transmitters we could define a new class transmitter class uhf transmitter which is then called a subclass of class Transmitter. Procedures that are declared virtual in class Transmitter can be redefined in class Uhf_transmitter, though they are still accessible from the Transmitter level of the object. When class Uhf_transmitter is instantiated it will be a concatenated object containing the attributes and actions of both class Transmitter and class Uhf_transmitter. Other varieties of transmitter object templates may also be defined using class Transmitter as a prefix. Thus, class inheritance provides a powerful and flexible capability for designing object templates as well as contexts.

Each class instance in a Simula system is an object with its own set of attributes and actions. Therefore, to implement the OSI Physical Layer in a context class named Layer1, we define a set of Simula object-template classes which provide templates for the various components of a communication hardware suite. The Simula code that a user of the Layer1 context writes instantiates the object-template classes as needed to implement a particular communication system model. This is analogous to building a communication system by assembling and interfacing hardware components. We explain this object-oriented approach in more detail when we discuss our example. In figure 2 we show a generic physical layer model using Layer1 objects.
LAYERED APPROACH USING INTERFACES

In the preceding section we described how layered contexts and object templates are built using Simula classes. We now carry the layered approach one step further by introducing our implementation of an interface. Interfaces play a significant role in the OSI architecture in that they are used to hide the implementation details of the protocols which support them.

In our code, we model each OSI layer interface as a separate object, as depicted in figure 3. Note that each of the layers implements the protocols specific to that layer as well as half of each interface to an adjoining layer. That is, each interface object consists of two parts, one of which is a subclass of the other. Layer(n-1) contains the super-class part of the (n-1)/n interface and layer n contains the subclass part. Each half of an interface object provides connections to other objects within its own layer.

Each interface services the protocol in the next higher layer. When layer n wishes to send a layer n message (n_msg) it calls send in the (n-1)/n interface, which in turn interrupts layer (n-1) by sending it an event. Likewise, layer n receives an n_msg when it is interrupted by a receive event. Note that layer (n-1) must implement the response to send while layer n must implement the response to receive. Through the use of Simula’s virtual procedure declaration, it is possible for any layer to make calls to interface procedures that are implemented in an adjacent layer. This facilitates the development of each layer as a separate software module.

The interfaces shown in figure 3 contain only a small subset of calls normally available at an interface. In our example we will expand these interfaces as necessary to meet the design requirements of our example communication system; however, every interface must at least contain send and receive procedure calls.

The interface objects and the protocol objects which support them are reproduced at every node in the communication system model. This is easily accomplished in Simula via the new construct once all the necessary object templates have been developed.

LAYER1

We present our discussion of Layer1 objects by using event graphs [10]. We take some liberty with event graphs as they are formally presented in reference 10 and, therefore, give the following explanation of the conventions we use in this paper.

Figure 4 shows our event graph conventions. Events are depicted as circles connected by directed edges. Each circle is tagged with the event name, and, when appropriate, the lower half of the circle is used to indicate a transition to a new state. All events create state transitions; however, we do not always rigorously spell out these transitions in the event diagrams. Where state transitions are given, the new state is entered after the occurrence of the event, not before.

Directed edges indicate causality between events. Edges may be tagged with conditions and/or delay times. If the edge is tagged with a condition it is marked with a “” to indicate its conditional nature. Conditions always represent some combination of state variables (i.e., system state) which must be satisfied for an event to occur. Moreover, the condition must be satisfied at the time of occurrence of the causing event, not after it. Thus, the new state indicated in the lower half of the causing event’s circle has not yet been entered at the time the condition must be tested. If a delay time is given, the event thus scheduled will occur at a time equal to the time of occurrence of the causing event plus the delay time.

36.2.3
Two more conventions complete our set of event graph tools. If a state or a condition has a line above it, the opposite is indicated and may be read as “not”. Also, a dashed edge indicates the canceling of an event as opposed to the scheduling of an event.

Once the actions of an object are described in terms of the events to which it can respond, which can be derived from event-graphs, it is a simple matter to generate Simula code using event-processes [7]. An event-process is a Simula process class whose actions are described by a list of events to which the object can respond with corresponding calls to procedures that handle the response to specific events. This approach is similar to Smalltalk’s Messages/Methods paradigm.

Our physical layer code was originally designed to model HF spread-spectrum communication systems using multiple frequency-hopping codes and multiple frequency bands or channels. Though in the example given later we only require a single channel without code differentiation, we present the aspects of our physical layer model which handle multiple codes and channels along with the other portions of the model. We use two figures, 5 and 6, to illustrate our physical layer model in terms of event graphs. The process of transmitting a message is shown in figure 5 while the process of receiving a message is shown in figure 6.

The left-most box of figure 5 represents the interface to the physical layer, i.e., the 1/2 interface. Note that we show in figure 5 only that portion of the 1/2 interface which relates to sending messages. The part of the 1/2 interface which handles the receiving of messages is shown in figure 6. Three events - msg_sent, msg_ret, (figure 5) and msg_revd (figure 6) - are generated by Layer1 objects and behave like interrupts, which the Data Link layer protocol must handle in some fashion. The other events listed in the 1/2 interface are generated by the Data Link layer protocol to implement a channel access scheme.
The real work of transmitting a Physical layer message is done by the box in the middle, the transmitter object. If we ignore channels and codes for a moment, we see that a transmitter object has three fundamental states—sending, idle, or off. All edge conditions are related to one of these states. For example, to start a message the transmitter must be in the idle state. The idle state means that the transmitter has been turned on but is not sending any information. One could view it as a transmitter sending an unmodulated carrier or a synchronization preamble. If the transmitter has not been turned on or if it is in the process of sending a message, then calling start will have no effect. On the other hand, if the transmitter is idle, starting a message immediately queues a msg_arrived while_idle event which, in turn, schedules unconditionally a start_of_msg event in the channel and an end_of_msg event delayed by the message transmission time in the transmitter. The transmitter state is then set to sending. Upon completion of the transmission the transmitter end_of_msg event occurs which in turn schedules an end_of_msg event for the channel and schedules a msg_sent interrupt in the 1/2 interface where the Data Link layer protocol must decide what to do when a message has been sent.

Calls to turn_off and turn_on in the 1/2 interface will conditionally schedule events by the same name in the transmitter. The rationale for imposing these conditions should be fairly obvious. If the transmitter is already off, a call to turn_off should have no effect. Thus, the transmitter has to be in a "not" off state (i.e., either idle or sending) in order to schedule a turn_off event. Conversely, the transmitter must be off in order for a turn_on event to be scheduled. Both turn_off and turn_on events unconditionally schedule corresponding events in the channel. However, if a transmitter is turned off while it is sending a message a little extra work must be done to tidy things up. The end_of_msg event which had been scheduled must be canceled and the procedure msg_ret is called to enable the Data Link layer protocol to do something about the aborted transmission.

The set_channel and set_code events may only be scheduled if the transmitter is not sending, i.e., is off or idle. In the case of code or channel changes being made while the transmitter is in an idle state, additional xoff and xon events must be scheduled in the channel so that receiving nodes can sort out what is going on. Layerl performs these tasks so that the user does not have to consider these details.

The channel object has the task of forwarding all start_of_msg, end_of_msg, xon, and xoff events to the multicoopers at those nodes with which the transmitter using the channel has connectivity. Connectivity is determined by examining a Boolean connectivity matrix. The value of the connectivity matrix may be set externally by inputting fixed data or internally by the occurrence of events within the simulation in a dynamic fashion. If the channel connectivity matrix is dynamically altered, the Layerl channel generates some xon's and xoff's of its own to model the effects of changing connectivities at receiving nodes.

In figure 6, we diagram the effects of the start_of_msg, end_of_msg, xon, and xoff events as they are received. Reception, of course, is contingent upon having connectivity with the sending transmitter via a channel, as the conditions used to tag the left-most edges in figure 6 indicate. The multicooper intercepts xon and xoff events to maintain information concerning activity on the channel. Events are forwarded to any receiver at the node tuned to the channel which is sending the events. An xon event at a receiver schedules a sync_detected event delayed by sync_time, the time required to obtain synchronization, as long as these conditions are met: 1) the receiver is not already in sync, 2) a collision state does not exist, and 3) the receiver and transmitter codes are the same. Xon and xoff events can conditionally cancel the sync_detected event at a receiver. If the xon event results in excessive interference at the receiver, then the receiver will not be able to attain synchronization. Thus, the sync_detected event is canceled. Also, if the transmitter which caused the sync_detected event to be scheduled sends an xoff, the sync_detected event must be canceled. Changing the channel and code settings of a receiver will also cancel a sync_detected event which has been scheduled. Start_of_msg and end_of_msg events can be scheduled at the receiver only if the receiver is in sync and is set to the proper code and channel, as is indicated by the edge conditions. A start_of_msg event sets a pointer called current_msg to the message which initiated the start_of_msg event. When an end_of_msg event arrives it calls the 1/2 interface msg_revd procedure provided current_msg points to the message which initiated the end_of_msg event.

Figure 6. Event graph of Layer1 multicooper and receiver

36.2.5
AN EXAMPLE

As an example, we show how to build a simple carrier-sense multiple-access (CSMA) protocol on top of our Physical layer. The CSMA protocol resides in the Data Link layer (Layer2). An event diagram of the protocol is shown in figure 7.

Our Layer1 model is accessed by using the 1/2 interface, which appears in the left side of figure 7. Other than knowing how to create Layer1 objects and link them together, we really don’t need to know anything more about Layer1 other than how to use its interface. In the 1/2 interface we show the procedures we shall use to implement the protocol. Toward the right side of the diagram we have another interface – the 2/3 interface. The 2/3 interface contains the procedures we want our Network layer to use. We will model the Network layer and above very simply by creating a traffic generator to generate the message traffic at each node and by creating a message handler which merely gets incoming messages and prints them. The work of designing the protocol then amounts to filling in the box between the two interfaces with a meaningful event diagram, which we have done.

Now, let’s see how the model works. To send a message via the 2/3 interface a call is made to the 2/3 interface send procedure. This puts the message in a queue – xq – and schedules an access_channel event if the protocol is not already in an accessing state. When an access_channel event occurs, the protocol checks to see if the channel is idle (not busy). If it is, the protocol turns on the transmitter and starts a message – the first message in xq, which is a FIFO queue. On the other hand, if the channel is busy, then the access_channel event reschedules itself to occur at some random delay time later. In either case, the protocol enters an accessing state which means that it is attempting – either successfully or unsuccessfully – to access the channel.

The rest of our CSMA protocol is implemented by specifying actions for the procedures msg_sent, msg_ret, and msg_rcvd. When the transmission of a message is completed, we must decide what to do next by specifying actions for the msg_sent procedure. Here we turn off the transmitter and, if xq has more messages, we schedule an access_channel event. If xq is now empty, however, we schedule a terminate_access event which changes the protocol’s state to not accessing. In other words, in our version of CSMA once the protocol gains access to the channel it keeps on sending messages until xq is empty.

The actions of the remaining procedures in the 1/2 interface are quite simple. Msg_ret merely prints out an error message. If our code properly implements the event diagram we should never see this message. If we do see it, it will aid in debugging the code. Msg_rcvd puts the incoming message into an FIFO queue – rq – and calls the receive procedure in the 2/3 interface, which, in turn, must handle the incoming message.

CONCLUSIONS

We draw the following conclusions. The first is that Simula’s concepts of class, inheritance, and virtual procedures can greatly simplify the task of building layered simulation models. Only a few languages have similar facilities. Among them are Smalltalk, Flavors, C++ [11], and Objective C. Each of these other languages has drawn upon Simula. Secondly, the OSI layered architecture is implemented directly in classes (contexts) which can be independently compiled and built one on top of another, thus, creating a one-to-one mapping between the simulation model and the code which implements it. Layer1 is a highly developed context that models the OSI Physical layer and uses an interface to

![Figure 7. Event graph of a CSMA protocol](image-url)
hide the implementation details. Developing a Data Link layer protocol, as illustrated by our CSMA example, becomes a relatively simple matter of properly using the Layer1 interface. The new context, extended by Layer2, provides an interface for developing Layer3, and so on. The methodology presented here is flexible enough to model a wide variety of communication systems and these models are useful in communication protocol development, verification, and performance assessment. Finally, event graphs provide a very useful technique for describing the simulation model apart from the code and it is very easy to write the simulation code once the event graphs have been determined.

REFERENCES