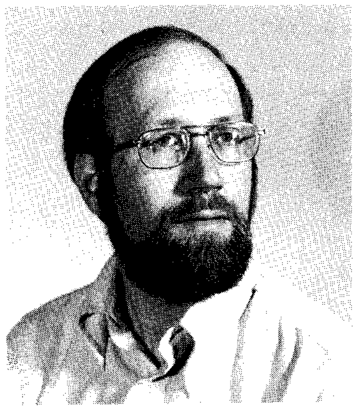
APPLYING SOFTWARE ENGINEERING TO PROTOCOL SIMULATION

Keywords: abstract types, communications, computer-network models, electrical engineering, information hiding, languages, military applications, philosophy and comment, protocols, queuing, SIMULA, simulation, software engineering



by

Carl E. Landwehr
Code 7593
Naval Research Laboratory
Washington, D.C. 20375

CARL E. LANDWEHR is a member of the Computer Science and Systems Branch of the Information Technology Division at the Naval Research Laboratory in Washington, D.C. He holds a BS from Yale University and a PhD in computer and communications sciences from the University of Michigan. Formerly, he was a research assistant at the University of Michigan Computing Center and a member of the computer science faculty at Purdue University. His present research interests include computer security, software specification and verification techniques, and the modeling, simulation, and performance analysis of computer systems. His outside interests include teaching computer science at Georgetown University.

ABSTRACT

Communications systems have widely varying equipment, traffic distributions, error characteristics, and measures of performance. To accommodate these characteristics, a simulator for the protocols controlling the transmission and reception of messages must be modular and flexible; it must also allow for the handling of concurrent processes and the efficient generation of reports and statistics. One way to construct such a simulator is to use techniques derived from software engineering: design prior to coding, design review, modular design based on information hiding, use of abstract types, code review by peers, co-operating sequential processes, and pseudo-code specifications. In the development of a simulator program for several protocols, outside design review, the information-hiding principle, the use of abstract types with a language that supports them (SIMULA), and the use of co-operating sequential processes were the most productive methods.

INTRODUCTION

Simulation is an important tool for studying the performance of complex systems. Recently, there has been significant progress[3,6,12,26] in the application of statistical techniques to the analysis of simulation results. Techniques for building simulation programs have received less attention, except in the sense that software engineering methods are intended to apply to the construction of programs in general. This paper reports on the use of certain software engineering techniques in building a simulator for satellite communication protocols.

Simulation provides a good test case for software engineering methods because simulations often require (1) concurrent processes, (2) modeling a complex system through abstraction, (3) validating a program against vague requirements, (4) modifying a program to model alternative systems or configurations, and (5) efficient use of computer time and storage to allow enough replications of experiments to provide statistically meaningful results. In this project, we applied the following software engineering techniques:

(1) Complete design prior to coding[20]

(2) Design review by knowledgeable outsiders[8]

(3) The information-hiding principle[21]

(4) Abstract types[18,23]

(5) Code reading by other than the coder prior to testing[1]

(6) Co-operating sequential processes[5]

(7) Use of pseudo-code.[11]

All these techniques were helpful, although some proved of greater benefit than others.

DESIGNING THE SIMULATOR

The first step in the design was to survey and document the requirements for the simulation and the environment. The general goal was to evaluate the performance of alternative channel management algorithms (protocols) for broadcast satellite UHF channels. Thus the simulator had to model accurately this type of channel and allow different communications protocols to be tested.

We first determined the primary factors that would affect the performance of protocols in the communications systems. We also developed secondary requirements that would allow modeling of more general systems (e.g., point-to-point networks or networks including voice links). The primary list of requirements was mandatory, the secondary desirable (extensions of the initial simulator were to allow their inclusion if possible). We provided brief descriptions of three protocols as examples of the operations that the simulator would have to model.

After identifying the requirements, we proposed a design. Since the characteristics of the traffic on the channel (arrival rates, sources, destinations, priorities, and message lengths) could vary greatly, the design had to provide a flexible way to specify them. A goal for the design was that changes in traffic characteristics should require only parameter changes in the simulator, not coding changes. We expected coding changes to be necessary for testing different protocols, but we wanted the changes to be limited to the protocol module. We expected that changing the measurements to be recorded would require changes in code, but our goal was to minimize and localize these changes for each variable. The code that generates reports should not be affected by changes in the variables actually measured.

We specified the initial design informally as a set of functions grouped into modules based on the principle of information hiding.[21] Each module isolates particular design decisions and "hides" those decisions from the other modules. If a design decision is changed, the change affects only a single module. The "secrets" of a particular module are the design decisions it hides. The design also defined a structure for processes to model the activities of a real communications system: message generation, transmission, and reception. We defined our processes as sequentially predictable activities; actions that could occur simultaneously in the real world (e.g., creation of one message and transmission of another) were modeled by separate processes. The design included pseudo-code (an ad hoc programming-language shorthand) for example processes; this code consisted of function invocations embedded in ALGOL-like control structures.

The design document also included a table listing system characteristics likely to be changed in experiments (such as traffic distributions, communications equipment, statistics collected, and channel error characteristics), cross referenced with the proposed modules. At the intersection of each module (row) and experimental variable (column), we indicated whether a change in the variable would require no change, a parameter change, or a coding change in the corresponding module. This table provided a convenient way to check how well our modularization followed the information-hiding principle.

A technical memorandum[16] describing the system requirements and the proposed design was circulated to three interested observers for a thorough review. Keeping the documents together allowed reviewers to judge the design with respect to the requirements. We incorporated the results of these reviews into a revised design memorandum that served as the basic document guiding the implementation. The differences between the initial and final designs derived principally from a sharpening of the concepts of process, function, and module. The revised document included definitions of these terms and omitted one synchronization process that had been present in the initial design. This process was found not to represent a real-world activity.

The final design consisted of two major parts: (1) the user interface and simulator control, which controlled a given simulation run, providing checkpointing and restarting facilities, parameter input/output, report generation, and initialization, and (2) the communications system simulator, which actually simulated the system.

The communications system simulator included two types of processes: message generation processes (MGPs) and channel access processes (CAPs). Each node in a broadcast communications network would have one or more MGPs to generate traffic according to specified distributions for message lengths, arrival rates, etc., and one CAP to transmit messages across the channel. (Later versions split each CAP into two separate processes, one for transmitting messages and the other for receiving messages.) These processes would invoke the functions contained in the following five modules:

(1) *Message generation.* This module hides such details as the probability distributions in use for determining message interarrival time, message lengths, etc. It provides functions to generate a new message and to generate the time between messages.

(2) *Message storage.* This module hides the queues of messages awaiting transmission and the priority queuing algorithms. It has functions to return the waiting message with the highest priority, to remove a message from the store, to enter a message into the store, etc.

(3) *Channel.* This module hides the noise and delay characteristics of the communications channel. There are functions to transmit or receive data and to obtain the propagation delay between a pair of nodes.

(4) *Protocol.* This module hides the detailed behavior of the protocol for transmitting and receiving messages. The specific functions vary according to the protocol but generally include the handling of acknowledgments and retransmissions, storage of portions of messages, and so forth.

(5) *Statistics.* This module provides a common set of functions for the collection and reporting of statistics recorded during a simulation run. It hides the algorithms and data structures used to generate the statistics and reports.

The design of the user interface and the simulator control was specified in less detail, since the functions to be performed were typical of many simulation programs. These modules were identified as follows:

(1) *Control.* This module provides the user interface to the top-level simulation functions, such as reading parameters, initializing the simulator, initiating a simulator run, and generating reports. This module reads user requests, checks them for validity, and invokes lower-level functions to execute them. It requires no detailed knowledge of the lower-level functions.

(2) *Parameter input/output.* This module hides the user interface and functions for the acquisition, display, and saving of parameter values. In order to prevent this module from requiring detailed knowledge of the parameters and input/output formats for all modules in the system, we had each module in the communications system simulator implement functions to perform these operations for itself. The parameter input/output module needs only to know the names of these functions.

(3) *Report generator.* This module controls the printing of statistics. To prevent this module from knowing all the statistics in the system, a function was provided for each type of statistics variable to print a report. All instances of statistics variables are linked, and the report generator only has to know the name of the list head in order to sequence through the set of variables and print the required output.

IMPLEMENTING AND DEBUGGING THE SIMULATOR

We chose SIMULA[2,3,7] as the implementation language because of its support for abstract types and its availability on a local PDP-10. Although SIMULA is relatively little-used in the United States, it has a large base of European users. We found the DEC PDP-10 compiler to be relatively free of errors. The utility for interactive debugging provided excellent facilities for setting breakpoints, stepping through programs, and providing symbolic references to variables. The choice of SIMULA had no influence on the design, since the author had only a nodding acquaintance with the language prior to this project. No SIMULA code was written before the design was compiled. Consequently, we began by developing several small SIMULA programs to resolve questions on how certain features of the language worked.

The pseudo-code describing processes in the design had used an approach based on events: a process would wait for an event to occur, service an event passed to it, and wait for the next event. The test programs revealed that SIMULA did not support events directly; we either had to introduce events and a scheduling mechanism for them or we had to modify the design of the processes. Since this program affected the design, we wrote a memorandum posing the alternatives and circulated it to the reviewers of the earlier design document. We decided to avoid the use of events and to split the CAP into two separate processes, one for reception and one for transmission.

This decision made the mapping of the design into SIMULA straightforward. The CLASS structure in SIMULA provides a good mechanism for the implementation of type abstractions.[27] (An abstract type is a family of types in which family members implement closely related operations and data structures.) Of the abstract types used in the implementation, the one that proved most useful later is the abstract type for statistics collection. We have described the development and use of this abstract type elsewhere[15] but we will review it briefly here as an example of how we used type mechanisms to hide information.

The purpose of this family of types is to encapsulate the functions and storage required for recording such measurements as message delays and backlogs, calculating statistics (such as the mean and variance) from these measurements, and generating printed reports or histograms. A family of types, rather than a single type, is necessary because some measures (such as delay) are meaningful only for continuous quantities while others (such as the number of items queued) are meaningful only for discrete quantities.

Furthermore, creating histograms requires extra storage for collecting the statistics and more parameters (e.g., number of bins, bin size). Each statistics variable is specified as either real or integer and histogram or no histogram. The same operations are provided for all types: initialization (to generate a new instance of the type), update (to record an observation), and report (to print a summary of the observations). An additional operation prints the histogram.

Defining the types as SIMULA CLASSes satisfied the need for encapsulation of the storage and operations, but an additional goal was to hide from the report generator the identity and number of statistics variables (instances of these types) that were to be printed. If this information were concealed successfully, we could change the number and type of statistics collected without changing the report generator. To accomplish this goal, we joined all statistics variables in a single linked list. (SIMULA provides a built-in type for such lists.) Each time a new instance of one of the statistics types was created, it was linked to the end of this list. The report generator needed only the name of the list head and the SIMULA list operators. After printing a general heading, it merely invoked the report operation provided with each list element. By checking the type of a list element, the report generator could determine whether a histogram was required.

We performed coding and debugging in parallel. We first coded the lowest levels of the system (in the sense of the "uses" hierarchy[22,25]) and then compiled them immediately in order to detect errors as early as possible and to make the code more readily available for reading by an interested observer. Thus we first coded the abstractions for the definition of probability distributions and the generation of random numbers, followed by the abstractions defining messages, message stores, and message generator processes. Next, we implemented a skeleton control module, together with procedures to save and restore parameters, to allow execution of the completed portions of the code. The control functions and utilities were then expanded gradually as the coding of the abstractions for slots, blocks, channels, and—finally—a test protocol module was completed. The last functions to be added were the mechanisms for defining the statistics to be collected.

The procedures just described facilitated the integration of new sections of code with existing code, since the existing code had already been compiled, read by an observer, and executed in some fashion. We could focus our attention on the relatively small new section added rather than on the entire simulator. Initially, a reviewer read all the code, but we had to discontinue this because of competing demands for the reviewer's time.

The primary reviewing and debugging techniques included the code-reading already mentioned and the use of the compiler's features for compile-time syntax checking, run-time error detection, and interactive debugging. Although code-reading uncovered some errors, it was primarily helpful in ensuring that the code was well commented and consistent with the design documents. As we coded and compiled successive modules, the compiler detected as faulty block structures many errors that were in fact merely typographical. The errors reported by the compiler in such instances are, unfortunately, usually far removed from the true source of the problem. The structure provided by PDP-10 SIMULA for separately compiled procedures proved to be more hindrance than help, and separate compilation finally was abandoned as a development tool in favor of a single large compilation for the entire simulator. One problem centered on the requirement that PDP-10 file names (limited to a length of six characters) and the names for the separately compilable CLASSes they contain be consistent. This meant that all names referred to across the boundaries of separately compiled modules had to be distinguishable on the basis of their first six characters. Further, references were required to be strictly ordered: if two CLASSes contained references to each other, neither could be compiled first; they would have to be compiled as a unit.

The run-time debugging package (DDT) was very helpful (in fact, the control module uses its functions to allow display and alteration of parameters). The ability to display values of variables after an execution error was particularly valuable during debugging; its benefits would be of even greater if values returned by procedure calls could be made available.

An informal error log maintained during the development shows that the most numerous errors were clerical and, of these, the most bothersome were errors resulting in mismatched BEGIN-END pairs. Other errors included improper I/O function usage (caused partly by a lack of details in the SIMULA documentation), problems in attempting to use separate compilation, improper parameter passing, default initialization (SIMULA sets variable values to zero initially, masking some references that should be errors), and improper assumptions about the order of evaluation of Boolean expressions. The most difficult errors to find were problems that revealed themselves only in unusual statistical measurements from simulation runs. In one case, a few messages seemed to have inordinately long processing delays; the bug found was the improper resetting of an index in an infrequently exercised processing path. This error caused parts of a message to remain queued when they could have been transmitted. Debugging the simulator in this respect appeared quite similar to debugging an actual implementation of the protocol. It took six man-months to design, implement, and validate the simulator. The program consisted of roughly 2450 lines of SIMULA (this and subsequent measures of code length include blank lines and comments). About 200 of these lines were code specific to the test protocol.

VALIDATION

We validated the simulation by implementing the required processes and models for the slotted ALOHA protocol. In this protocol, time is divided into fixed length slots, with each slot representing the transmission time for a single packet of data. When a station receives a packet of data to send, it waits until the start of the next slot, transmits the packet, and then listens to the channel to hear whether the transmission was successful. If, by chance, two stations transmit packets simultaneously, the packets collide and both must be retransmitted. To avoid repeated collisions, each station trying to retransmit a packet waits for a random number of slots before retransmitting. Slotted ALOHA is called a contention protocol because the nodes in a network contend for time on the channel instead of adhering to a fixed schedule of allocations. We chose this protocol as a test case because it is widely understood, because it exercised the principal parts of the simulator without being too complex, and because analytic results were available for it.

We compared the simulated results with both the analytic and simulation results produced by Lam.[13] This comparison uncovered some errors in the implementation and led to a deeper understanding of both the details of the protocol and the assumptions on which Lam's analysis was based. Ultimately, satisfactory agreement between the simulation and Lam's work was obtained.[17]

MODIFYING THE SIMULATOR FOR OTHER PROTOCOLS

Following validation, we used the simulator to study three different protocols from two U.S. Navy systems. We have reported the results of these studies in References 14 and 19. Their relevance here lies primarily in the changes that we made to the simulator to accommodate the new protocols. Our design and development techniques were intended to reduce the need for changes and to limit them to particular areas of the program. Although we did not succeed in limiting changes solely to the protocol module and processes, our use of software engineering methods did simplify the few outside changes that were required. In this section we review the protocols that were modeled and the changes they required.

Navy tactical protocol

The first protocol (following slotted ALOHA) that we simulated was a design for a proposed U.S. Navy system for transmitting tactical data among ships and shore stations. This protocol is based on polling, with a central controller broadcasting the polling list. The polling list specifies both the quantity of data each station may transmit and the sequence in which stations can use the channel. As part of its transmission, the station can indicate whether it still has additional data to send, and it can acknowledge data it has received from other stations. At the end of a polling cycle, the central controller generates a new polling list based on this information, and the next cycle begins.

Unlike slotted ALOHA, this system requires explicit acknowledgments of received data. Consequently, we needed a functioning CAP-receive process to receive data and generate the acknowledgment. Because of the delay in the channel, parts of two transmissions for the same receiver might be outstanding at the same time. The process synchronization primitives in SIMULA, however, do not support multiple events pending for a single process. The solution was to introduce a channel process (in addition to the channel module). This process was responsible for receiving all transmissions, queuing them for the appropriate delay period, and then awakening the appropriate receiving processes. Besides adding this process, we had only to rewrite the transmission format blocks and protocol processes (as planned), and even these were able to use many of the utility functions implemented in the protocol module for slotted ALOHA. The message generation, message storage, statistics, control, parameter input/output, and report generator modulès did not change. Making the changes required for this protocol and debugging occupied about two man-months. The program grew to approximately 3100 lines, of which about 700 were protocol-dependent.

CPODA (contention-based, priority-oriented, demand access)

Next, we implemented a simulation of CPODA, a packet protocol with distributed control.[9,10] This protocol divides time into fixed length frames, each of which is divided into reservation and data-transmission subframes. Reservations are short requests for allocations of time in the data-transmission subframe. Each reservation also indicates a priority for the data to be transmitted. During the reservation subframe, nodes may broadcast reservation packets according to the slotted ALOHA protocol.

Reservations that are broadcast successfully (that don't collide) are queued in priority order by all nodes. When the data transmission subframe begins, the node whose reservation is first in the queue (the oldest reservation of the highest priority) may transmit its data. At the end of this transmission, the next queued reservation is served, and so on until the data-transmission subframe ends. There are provisions in the protocol to assure that all nodes maintain a consistent view of the reservation queue. As the queue of requests grows, the reservation subframe decreases and the data-transmission subframe increases. This protocol does not require a central controlling station; only a standard time reference is needed.

Nodes perform two types of timeouts. First, each time a node transmits a reservation or data packet it listens for the echo of that packet from the satellite. If the echo does not occur after the round-trip transmission delay to the satellite, the node requeues the reservation packet for transmission later. If the echo of a data packet occurs as expected, the node waits a specified length of time for the positive response from the intended recipient and, if none is received, queues a reservation packet to allow later retransmission of the data packet. The Navy tactical protocol described earlier had only one type of timeout. Furthermore, the timeout was tied to the polling cycle so that the channel process introduced to queue transmissions could handle the timeouts as well. The CPODA case was sufficiently complex to require a reexamination of this mechanism.

This reconsideration led to the introduction of an event mechanism in the simulation. We made each event an instance of a SIMULA PROCESS that would, on awakening, activate the process intended to receive the event and pass it a parameter defining the type of event. This approach avoids the requirement for a duplicate scheduler running on top of the SIMULA scheduler. (We discovered later that Franta proposes a similar implementation in his book.[7])

Again, the changes to the simulator were principally in the protocol module, as planned, but the event mechanism made possible the elimination of the channel process added in the previous protocol simulation so that there were some changes in the channel module as well. The definition of the event mechanism was made outside of the protocol module so that it would be available for use in future simulations. Making these took approximately three man-months. The simulator grew to approximately 3500 lines of SIMULA, of which almost 1000 lines were CPODA-dependent.

Existing Navy polling protocol

A third protocol simulated was a Navy protocol currently in use for the exchange of messages over satellite channels. This protocol is again a polling scheme with a central controller; it has a simpler structure than the first protocol described, since its delay requirements are less stringent. It is used principally for the transmission of text messages from ships to a shore station over a UHF satellite link.

Some additional constraints placed on this study allowed the model to be simplified greatly: acknowledgments and channel noise did not have to be modeled, and specified distributions were to be used for message generation. These restrictions eliminated the need for the routines that defined an abstract type for probability distributions, the channel module, and processes to receive messages or queue and process acknowledgments. Consequently, instead of building trivial routines within our general framework, we decided to construct a separate, simple simulator for this protocol by borrowing only the appropriate modules from the original simulator.

We borrowed the statistics and report generator modules intact, but we simplified the message generation module to use specific probability distributions instead of more general descriptors. The abstraction implemented to model messages was simplified, since no specifications of destinations or priorities were required, and the message store module became a simple FIFO queue using the predefined SIMULA operators. We wrote the protocol module and a simple simulation control and parameter input routine from scratch.

This approach met the limited goals of the study. The simulator was constructed quickly and debugged easily. Other than the elimination of unneeded functions, virtually no changes were required in the borrowed modules. This simulator consists of about 700 lines, of which about half were borrowed. The simulator was constructed and debugged in less than a month.

The need for a minimal simulator lends further credence to the design concepts of program families and minimal subsets described by Parnas.[24,25] It is interesting to note that, although we did not employ those concepts explicitly in this project, the use of abstract types and the information-hiding principle led to a result consistent with them.

EXPERIENCE GAINED IN USING THE SIMULATOR

In addition to the lessons learned from the modifications of the simulator software, the effectiveness of the simulator as a tool for studying the performance of communications protocols is of interest. As mentioned above, we applied the simulator to two Navy problems. One involved the use of the distributed CPODA protocol to control communications among one to 20 ships and a shore station. The traffic in this case was primarily transactions with remote, land-based computer systems. The other problem involved evaluating the use of CPODA and the other protocols as vehicles for merging traffic from two presently independent Navy communication systems over a single channel. Both studies produced useful results; CPODA provided superior performance in nearly all the cases examined in the second study, and it provided adequate (though in some cases marginal) responsiveness in the first study.

In the course of performing these studies, we discovered several things about the simulator. The simulator was designed to be run interactively, and the commands provided by the user interface were generally helpful. The PDP-10 run-time support for SIMULA aided the implementation of some of these; the debugging package provides facilities to display and alter values of program variables, set breakpoints, and interrupt the simulator during execution. These facilities were helpful both during debugging and, occasionally, to alter parameter values during production runs. An unfortunate side effect of the run-time support was that it was impossible (without coding a special assembly-language routine) to save a complete core image in a file so that the run could be restarted later. We did not code the special routine because of our limited resources and our uncertainty as to how much effort it would involve.

The simulator itself provided facilities to save and restore parameter sets in files. This proved a crucial function, since the volume of information required to specify a given experiment was substantial. In fact, the interactive dialogue for parameter specification was burdensome enough that we preferred to edit a saved parameter file with a text editor rather than go through the entire dialogue if we only wanted to change a random number seed or one or two parameters.

The primary problems encountered in actual simulator runs were storage limitations. Although the initial design did not limit the number of nodes in a simulated communications network, the number expected in the first application was less than 15. Consequently, storage requirements were not a major consideration in the design or implementation and test parameter sets were generated with networks of more than 50 nodes with the slotted ALOHA protocol. The actual studies, however, often specified multiple priorities of traffic, a variety of message generation processes, extensive statistics collection, and other factors that multiplied the storage requirements per node simulated. Further, if a system were tested near saturation, message queues might generate sizable, although transient, backlogs that would exhaust available storage. All these factors made simulations involving more than 30 nodes impractical. The second study described above was made with a 30-node network and three priorities of traffic, after removing all statistics collection variables that were not of central interest. Project sponsors felt the 30-node network was sufficiently realistic for the case at hand, but would have liked the ability to investigate larger networks as well. The restricted version of the simulator needed less storage but had significantly less capability.

CONCLUSIONS

While all seven software engineering techniques proved beneficial to some degree and some of them overlap, we can summarize the impact of each as follows:

Complete design prior to coding. The requirement prevented premature freezing of the design but was less important than the design reviews.

Design review by knowledgeable outsiders. This technique required that the design be sufficiently documented that others could understand it and suggest revisions. Although the reviews caused design changes, the exercise of creating the documentation for the design was in itself beneficial. The requirements description generated to go along with design was also useful. Perhaps we should have devoted more time to the statement of requirements; in retrospect, establishing a better definition of the environment to be modeled (e.g., with respect to channel noise characteristics) and delineating performance requirements might have revealed problems that were not apparent until later in the project.

Use of the information-hiding principle in design. The modularization of the simulator based on the information-hiding principle was beneficial in several respects: it made the design easier to describe, it forced consideration of future changes early in the design process, it simplified the making of changes, and it led to a structure that provided useful parts to other simulators.

Use of abstract types. This concept, together with the information-hiding principle and use of a language supporting abstract types (SIMULA), aided in organization and led to the creation of at least one abstract type (for statistics collection) that is being used in new simulators.

Code reading by other than the programmer prior to testing. This procedure was carried out only for the first few weeks of the coding. A few errors were found, but the primary benefit was to force the inclusion of good comments and clean coding practices.

Cooperating sequential processes. Although the use of processes is hardly novel, it was central to this design. If processes had not been available in the programming language chosen for the implementation, it would have been worth the effort to implement them.

Use of pseudo-code. This widely used technique was used to provide initial sketches of code throughout the project. It helped significantly in communicating the design to readers not familiar with SIMULA.

We often hear that software engineering techniques are desirable but time-consuming. Although this project did not have firm deadlines, the initial simulator was completed and validated with the slotted ALOHA protocol in less than six man-months. The documents and programs have continued to be useful over a period of more than two years. Another staff member who is developing a simulator for HF radio networks has been able to incorporate significant parts of our simulator, given only the listing and the relevant documents.

There is, of course, room for improvement. In particular, we should have devoted more time to the performance implications (CPU time and, especially, storage requirements) of the design, to the level of detail actually required in the channel model, and to the user interface for parameter specification. Nevertheless, the project succeeded in applying software engineering techniques and in producing a tool for performance analysis applicable to a variety of protocols and able to be used and modified by other persons.

ACKNOWLEDGMENTS

We would like to thank several members of the Computer Science and Systems Branch at NRL for their assistance in this work. Constance Heitmeyer and John Shore participated in many helpful discussions throughout the project. They and David Parnas reviewed the initial design; John Shore reviewed some of the code as well. He, David Weiss, and Honey Elovitz provided helpful reviews of drafts of this paper.

REFERENCES

1 BAKER, F.T.
Chief Programmer Team Management of Production Programming
IBM Systems Journal vol. 11 no. 1 1972

2 BIRTWISTLE, G. DAHL, O.-J. MYRHAUG, B. NYGAARD, K.
SIMULA Begin
Auerbach Philadelphia 1973

3 BIRTWISTLE, G. PALME, J.
DECSYSTEM 10 SIMULA Language Handbook Part I
Available as NTIS PB-243 064 September 1974

4 CRANE, M.A. INGLEHART, D.L.
Simulating Stable Stochastic Systems. III: Regenerative Processes and Discrete-Event Simulations
Operations Research vol. 23 no. 1 1975
pp. 33-45

5 DIJKSTRA, E.W.
Co-operating Sequential Processes
In F. Genuys, editor, *Programming Languages* (Academic Press, New York, 1968), pp. 43-112

6 FISHMAN, G.S.
Achieving Specific Accuracy in Simulation Output Analysis
Communications of the ACM vol. 20 no. 5
May 1977 pp. 310-315

7 FRANTA, W.R.
A Process View of Simulation
Elsevier North Holland New York 1977

8 FRIEDMAN, D.P. WEINBERG, G.M.
Ethnotechnical Review Handbook 2nd edition
Ethnotech, Inc. Lincoln, Nebraska 1979

9 HSU, N.-T. LEE, L.-N.
Channel Scheduling Synchronization for the PODA Protocol
1978 International Conference on Communications Conference Record vol. 3 1978 pp. 42.3.1-42.3.5

10 JACOBS, I.M. BINDER, R. HOVERSTEN, E.V.
General Purpose Packet Satellite Networks
IEEE Proceedings vol. 66 no. 11 November 1978 p. 1448

11 KERNIGHAN, B.W. PLAUGER, P.J.
The Elements of Programming Style
McGraw-Hill New York 1974

12 KOBAYASHI, H.
An Introduction to System Performance Evaluation Methodology
Addison Wesley Reading, Massachusetts 1978

13 LAM, S.S.
Packet Switching in a Multi-Access Broadcast Channel with Application to Satellite Communications in a Computer Network
PhD dissertation Department of Computer Science
University of California Los Angeles 1974

14 LANDWEHR, C.E.
Performance Studies of the Distributed CPODA Protocol in the Mobile Access Terminal Network
NRL Memorandum Report 4804 September 1979

15 LANDWEHR, C.E.
An Abstract Type for Statistics Collection
ACM Transactions on Programming Languages and Systems vol. 2 no. 4 October 1980
pp. 544-563

16 LANDWEHR, C.E.
On the Design of a Simulator for Satellite Communications
NRL Technical Memo 5403-85:CL:la March 1977

17 LANDWEHR, C.E.
Construction and Validation of the Satellite Communication Simulator for the Slotted ALOHA Protocol
NRL Technical Memo 5403-259:CL:gls June 1977

18 LISKOV, B. ZILLES, S.
Programming with Abstract Types
SIGPLAN Notices no. 9 April 1974 pp. 50-59

19 MELICH, M. LANDWEHR, C.E. CREPEAU, P.
Alternative Satellite Channel Management Strategies
NRL Report 8404 Naval Research Laboratory
Washington, D.C. October 1980

20 MILLS, H.D.
Software Development
IEEE Transactions on Software Engineering
vol. SE-2 no. 4 December 1976 pp. 265-273

21 PARNAS, D.L.
On the Criteria to Be Used in Decomposing a System into Modules
Communications of the ACM vol. 15 no. 12
December 1972

22 PARNAS, D.L.
Some Hypotheses about the "Uses" Hierarchy for Operating Systems
Technical report of the Technische Hochschule Darmstadt 1976
See also Reference 25.

23 PARNAS, D.L. SHORE, J.E. WIESS, D.M.
Abstract Types Defined as Classes of Variables
NRL Report 7998 Naval Research Laboratory
Washington, D.C. April 1976

24 PARNAS, D.L.
On the Design and Development of Program Families
IEEE Transactions on Software Engineering
vol. SE-2 no. 1 March 1976 pp. 1-9

25 PARNAS, D.L.
Designing Software for Ease of Extension and Contraction
IEEE Transactions on Software Engineering
vol. SE-5 no. 2 March 1979 pp. 128-138

26 SAUER, C.H.
Confidence Intervals for Queuing Simulations
Performance Evaluation Review vol. 8 nos. 1-2
Spring-Summer 1979 pp. 36-44

27 UNGER, B.W.
Programming Languages for Computer System Simulation
Simulation vol. 30 no. 4 April 1978
pp. 101-109