APPLICATIONS OF FORMAL SIMULATION LANGUAGES IN THE CONTROL AND MONITORING SUBSYSTEM OF SPACE STATION FREEDOM

Final Report

NASA/ASEE Summer Faculty Fellowship Program - 1990

Johnson Space Center

Prepared By:	**R. C. Lacovara, Ph.D.**
Academic Rank:	**Assistant Professor**
University & Department	**Stevens Institute of Technology Dept. of Electrical Engineering and Computer Science**
NASA/JSC	
Directorate:	**Engineering**
Division:	**Tracking and Communications**
Branch:	**Communications Performance and Integration**
JSC Colleague	**James C. Dallas**
Date Submitted	**August 17, 1990**
Contract Number	**NGT-44-005-803**

1. Abstract

The notions, benefits, and drawbacks of numeric simulation are introduced. Two formal simulation languages, Simscript and Modsim are introduced. The capabilities of each are discussed briefly, and then the two programs are compared.

The use of simulation in the process of design engineering for the Control and Monitoring System for Space Station Freedom is discussed. The application of the formal simulation languages to the CMS design is presented, and recommendations are made as to their use.

2. Simulation and the Scope of this Paper

This paper discusses certain formal simulation techniques and tools, and makes observations and recommendations on the use of these techniques in a specific environment: the Control and Monitoring Subsystem (CMS) for Space Station Freedom.

The concept of simulation and varieties of simulation are discussed briefly. After that discussion, the application of simulation to the CMS is presented. The formal languages are presented and compared. Lastly, some recommendations for the use of these tools are advanced.

2.1. Rationale for Simulation

Simulation is one of many methods of obtaining information about physical or conceptual systems. The chief feature of simulation is that the information is not obtained by methods of formal analysis. Instead, a model of the system which includes the pertinent behavior is constructed and the model is exercised. Possibly, the model is exercised using pseudo-random input conditions, and the model may be exercised many times to smooth out variations in results due to the pseudo-random inputs, or internal psuedo-random events.

(In this paper, the term "random" will be used to describe the behavior of pseudo-random sequence generators. The differences and implications of this choice, and the definition of pseudo-random are beyond the scope of this work.)

The chief advantage of simulation accrues from the chief feature. Simulation is an interesting methodology precisely when (a) no formal analysis is possible or (b) no physical system is available from which to make observations. Any number of reasons will occur to the reader which may cause case (b). Case (a), the lack of formal analysis, occurs whenever the system to be studied has no known system of equations which describe it, or when the complexity, inherent non-linearities, pathologies, or plain old intractability of the system make the search for analytical description impractical. A cold appraisal of the world leads workers to conclude that case (a) is the rule, not the exception.

In brief, simulation is applied to problems for which formal analysis is unavailable or for which no physical system is available for observation.

3. Categorization of Simulation

Simulations may be usefully categorized in several ways. The process domain of a simulations refers to the description of time in which the system operates. Discrete time and continuous time descriptions will be familiar to most electrical engineers.

The computational domain describes the facility used to model the system. Although most facilities are digital computers, some analog computers are used in simulations.

The simulation scheme refers to the method in which the simulation passes time. This may be synchronously, in which the program advances time in small quanta and determines what, if anything, should occur. Otherwise, the simulation might be asynchronous, in which the program maintains a list of scheduled events and proceeds directly from one scheduled event to another regardless of the intervening time.

3.1. Process Domain

Process domain refers to the basic view of time taken by the simulation. A natural view of the progression of time is that of continuous time. The time variable may take on any value, and this is certainly the most common view of processes such as ballistic bodies, electronic circuits and similar systems. The other common description of the passage of time is discrete time, in which the time as an independent variable takes on only specific values, usually of the form

$$t = nT$$

where T is the smallest identifiable duration, and n is the actual independent variable.

The difference between continuous and discrete time signals is more fundamental however. Continuous time systems are those which may be described by differential equations, whereas discrete time systems are described by difference equations. Analytic solutions to continuous time systems are therefore the solutions to ordinary and partial differential equations with boundary conditions, usually facilitated by the use of Laplace transforms. Analytic solutions for discrete time systems satisfy difference equations with forcing functions, and a common technique is that of z-transforms.

Often it is convenient to model a naturally continuous system with a discrete time model. This may entail a choice of T such that no significant events occur in a period of time less than 2T. Unfortunately, it is not always clear what constitutes a "significant event", and further, other subtleties (such as solution stability concerns) may intrude.

As a practical matter, most formal simulations are in fact discrete time. Philosophical arguments aside, digital computers (the most common vehicle of simulations) represent quantities discretely (in finite precision.)

3.2. Computational Domain

Calculations may be performed in many different ways. Mechanical and hydraulic systems have been designed to perform logic and arithmetic. The only interest here is in electronic methods.

3.2.1. Analog Computers

Analog "computers" are good examples of analog simulators. These devices are actually arrays of operational amplifiers configured as summers, integrators and differentiators. As a result, analog computers are well suited to the solution of systems of linear differential equations. As an illustration, note that voltage or current is made the analogue of some physical quantity, and time is made the analogue of the independent variable, usually time. This is in contrast to something like a phonograph recording, in which vertical mechanical displacement of a grove is made the analogue of acoustic pressure, and linear displacement along the groove is made the analogue of time.

Analog computers are most often set up on plugboards to represent a particular differential equation. An applicable forcing function is applied, and the behavior of the system as a function of time is observed and scaled.

3.2.2. Digital Computers

Digital computers are the most common vehicle for calculation and simulation. Inherently, these machines represent discrete variables, and are naturally suited to represent systems which are characterized by difference equations. Various strategies allow the system to represent continuous time systems with acceptable results.

Digital "computers" could be constructed in the same manner as analog computers. Collections of adders, subtractors and delays could be assembled to exactly represent some particular difference equation. This is uncommon, although such an arrangement might be one of the fastest methods of obtaining a solution to a set of equations.

A slightly less hardware-intensive method is to use digital signal processing chips to implement the difference equations. These single-chip computers use specialized architectures and instruction sets to efficiently evaluate difference equations.

The majority of computational applications are performed on general-purpose computer architectures. This includes formal simulation languages.

3.3. Simulation Methodology

There are two primary simulation methods. These differ in the manner in which simulated time is advanced. An important distinction is that simulation time is unrelated to run time, or actual time. Simulation time is advanced either by identical small quanta (synchronous simulation) or advanced from event to event listed in a process list (asynchronous simulation.)

3.3.1. Synchronous Simulation

Synchronous simulations are used in systems which require interaction with exterior hardware. Examples are systems which are interfaced to hardware or certain recording systems. These systems have stringent timing requirements: the simulation must be able to advance simulation time at least as fast as real time.

Synchronous simulations are common even when there are no exterior requirements. This is not surprising, since synchronous simulations are often easier to code than asynchronous simulations.

Coding a synchronous simulation proceeds as follows. Prior to coding, the analyst determines the largest time slice which will suit the problem. An initial state of the problem is chosen. Then, for each element of the system, the analyst computes (guesses?) the probability (formally, a transition probability) that a particular element of the simulation changes state in during a time slice.

Code is written which can store the state of the system during any particular time slice, and update the state during the next slice. This is done by making a number of draws from a suitable probability distribution, and mapping the results into the new state of the system.

This technique has the advantage of simplicity. If the code is hosted on a sufficiently fast platform, the simulation is by default "real-time". (For example, if the time quanta chosen is 0.1 second, and the program completes all transition probability draws in less the 0.1 second, then the program is not a rate limiting part of a larger system, hence is real-time.)

There are drawbacks to this arrangement. If real-time performance is required, and the code does not run fast enough, then there is little recourse except a full re-write, or a port to a faster host. Further, a view of a process from the level of the time slice makes is difficult to implement a "process-wide" view of the system. Interactions between processes in the simulation are not obvious, and not necessarily easy to implement.

For certain classes of problems, the synchronous approach is sparing of a programmer's time, and may allow real-time operation in some cases. A discussion and exposition of a non-trivial synchronous simulation is found in [Lacovara 87].

3.3.2. Asynchronous Simulation

For more complex systems, or systems in which there is significant interaction between parts of the system, synchronous simulations present non-trivial coding problems. Further, the size of the time slice determines to a great extent the rate at which the simulation runs. A choice of small time slices will slow the system, and many slices may pass without any activity. Large time slices will allow simulated time to pass faster, but large slices may not allow sufficient fidelity in the simulated world.

Asynchronous simulation is an approach which resolves the time slice dilemma. In an asynchronous simulation, the time of occurrence of an event is predicted by a draw from an appropriate distribution. The predicted event is then placed as an event notice in a linked list of events, often called an event list. After all predictions which can be made from the present state and simulation time of the system have been make, simulation time is advanced to the

event notice closest in simulation time. Any changes needed to the state of the system implied by the current event are made, any new event notices are posted to the event list, and simulation time is advanced again.

The principal feature of asynchronous simulation is that simulation time advances from event to event. No compute time is spent evaluating time slices in which no events occur, and the "view" of the simulation is at the process level.

The advantages of asynchronous simulation are often great. A more intuitive view of the system may be use to create, maintain, and interpret the simulation. The simulation will permit interaction between its component processes. The disadvantage, however, is that the creation and maintenance of event lists and process notices are non-trivial, and certainly not advised for casual, even if experienced, programmers.

The formal simulation tools discussed below have the great advantage that the mechanics of the simulation, linked lists, queues, priority chains and so on, are hidden from the programmer. With the use of these tools, powerful simulation constructs are available to users who do not plan to spend their entire career writing code.

4. Formal Languages: Simscript and Modsim

In this study, two formal asynchronous simulation tools produced by CACI Products Company of La Jolla, California were used. These languages are Simscript and Modsim. They are entirely different approaches to the problem of simulation, and have overlapping but not identical domains of applicability.

4.1. Simscript

Simscript (from simulation script) is an old (1962) hence mature product. It imposes on the programmer a "view" of a system divided into processes. This is not altogether artificial, as the Simscript model of a bank consists of processes which represent a) customers, b) tellers, and c) customer arrival generator.

Simscript maintains a complex internal system of queues, lists, and other data structures. The maintenance of these structures is transparent to the programmer for the most part. In the bank example, customer arrivals are

"generated" by introducing event notices of arrival in an event list. The bank teller is represented by a teller "resource", a process which incorporates a waiting queue and other flags to indicate teller status. When a customer arrives at a teller, the Simscript system determines whether the teller is "busy" with a previous customer. If busy, the customer is placed in a waiting queue until the teller is free to process the customer.

Simscript manages the teller's waiting queue transparently. It checks the teller's status flags and determines whether the customer may be served or must wait. In addition to manipulation of these internal structures, Simscript monitors selected parameters of the simulation automatically. Quantities such as queue size, maximum, minimum, average and other statistics are accumulated without explicit programmer intervention.

As a result of the internal features, a certain class of simulation may be implemented in Simscript in an elegant and straightforward fashion. From a descriptive point of view, Simscript is well suited to systems which involve queuing theory: processes in which requests for service contend for limited resources.

Simscript includes facilities for screen graphics. Pre-defined screen constructs include graphs, indicators and clocks. Moving icons may be designed and animated by the simulation. These are relatively straightforward to use, and seem to be quite useful. Some are quite impressive.

4.2. Modsim

Modsim is a relatively recent language product. It is syntactically based on Modula 2, which is similar to Pascal. Basically, Modsim adds formal object-oriented structures to Modula-2, and provides simulation capability by providing a library of objects which can pass simulated time. Due to the Modula 2 underpinning, Modsim has considerable general purpose characteristics. It would be perfectly feasible to use Modsim wherever Modula 2 or Pascal is employed.

A description of object-oriented programming is beyond the scope of this paper. However, it is necessary to note that the pertinent features of "objects" are these. Objects are data structures comprised of typed-fields, and a list of procedures (called methods) which are the only procedures which act

on the data structure. The implications are these: objects may interact only through a object's allowed methods. An object's external fields may be read by other objects, its internal fields remain invisible. An object's fields may not be modified directly, but only through the agency of its allowed methods. The purpose of this strict control of access is to impose a discipline on the construction of the program, and to provide a type of safety in the control of actions performed on data structures.

Modsim extends the notions of object-oriented programming in the following manner. Methods are available in two classes, "ASK" and "TELL". An ASK method corresponds to the original methods noted above. When invoked upon an object, some action ensues in instantaneous simulation time. TELL methods, however, are asynchronous: simulation time may elapse before the desired changes occur. This is accomplished by the mechanism of an event-list. As in Simscript, the control and maintenance of the event list for the asynchronous simulation aspects of Modsim are implicit.

Modsim provides a complex and general programming environment. (Translation: Modsim is powerful, difficult, and sometimes obscure.) It would appear to be suitable for a very wide range of simulations. Like Simscript, it contains provisions for complex screen graphics.

4.3. Comparison of Simscript and Modsim

Be advised that the author's programming experience is as follows: considerable expertise in assembly languages, Fortran, Pascal, and C, and operating systems. The author has written numeric simulations on several systems, but is new to Simscript, Modsim, and object-oriented programming.

It seems to be easier to write simulations in Simscript than Modsim. In many ways, Simscript seems to bridge the conceptual gap between the system under consideration and the programming model in a more direct fashion. Modsim's object constructs however correspond to identifiable processes in the real world. Modsim's power and complexity initially interfere with the process of writing code: Modsim requires a substantial amount of "overhead" code to just get started.

The author designed a simple multiplexer as a simulation example for both Modsim and Simscript. The multiplexer accepts "data packets" about every 75

milliseconds. The packets face two servers. One is a high speed server, the other is a low speed server. The multiplexer enforces the following service discipline. If the queue size for the low speed server is twenty-five or fewer packets, the incoming packet joins the low speed queue. Otherwise, the packet joins the high speed queue. The simulation accumulates statistics on the sizes of the queues, and the distribution of the time taken by every packet to traverse the system.

The Simscript version of this code is about 40 lines. These 40 lines compile into about 250K bytes for a Sun Sparcstation. The Modsim version is only about 50% larger, but compiles into about 1M bytes for the same machine. Modification of the server discipline is about the same difficulty in either language. As features are added to either version, the Simscript object file grows somewhat, but the Modsim does not seem to expand much at all. The simplest interpretation of this behavior is that the Simscript object file is growing proportionally to the additional lines of code. The Modsim compiler seems to be able to perform one of the selling points of object-oriented programming: the ability to reuse many parts of the code through inheritance and recursion.

Of course, the Modsim object starts out about four times the size of the Simscript model, but this is not really serious on large machines.

Both Modsim and Simscript seem to utilize whatever processor power is available, as judged by a Sun performance meter.

The screen graphics package for Modsim is very similar to that of Simscript, but available in a more advanced version, and is therefore preferable.

Overall, Modsim seems to be a more general and versatile simulation system than Simscript. It carries with it corresponding penalties is size and programmer learning curve. Simscript is almost as versatile, and seems to be a very good choice if only one package is to be available to the general analyst working with small to medium sized systems. For very large projects, Modsim may be a better choice, since in large systems the organizational advantages of object-oriented programming will begin to be felt.

5. Simulation of Control and Monitoring Subsystem

The Control and Monitoring Subsystem (CMS) for Space Station Freedom is a collection computers and busses which determine the configuration and operate

various hardware systems. Some tasks include health observation, fault diagnosis and recovery, and other system-wide tasks.

5.1. Functional Simulation

The process of design of this system includes several generations of simulation of the CMS. However, these simulations are not operational, but functional. As an example, other applications on the Space Station communicate with the CMS via a local area network. The present simulations of CMS accept these commands, and provide a simulated response. This simulation system does not depend heavily on stochastic processes. Instead, the simulation must "merely" accept communication from exterior agencies and provide a sensible response. As a result the current simulations are written primarily in ADA and C.

5.2. Use of Operational Simulations in the CMS

There are several areas in which operational simulations may be of use in the CMS scheme. These are primarily incidental, but of some interest.

The current fault generation in the CMS simulator is manual. If there is no automatic fault diagnosis and isolation, this is probably adequate. If automatic detection, diagnosis, isolation, and correction are to be implemented, good testing practice would dictate that the simulated flaws occur without warning to the simulation operators.

Some aspects of the simulation are time-varying. Some require knowledge of the state of other components of the simulation. Either simulation tool would be of some use here. If complex time-dependent behavior was required, the tools would have some advantages over C or ADA.

5.3. Disadvantages in the CMS Simulation Environment

Simscript and Modsim have common properties which do not fit well with current simulation and software philosophy on Space Station. Neither tool resembles ADA, is written in ADA, or is likely to be available in ADA in the foreseeable future. An ADA ideology permeates the Space Station software engineering efforts, possibly to the exclusion of other useful concepts and directions in computer science.

A secondary item of dogma is object-oriented programming. Modsim fills this

bill nicely, and brings with it all of the overhead and performance penalties built in to this paradigm. However, Simscript is not strictly object-oriented. Simscript is modular, but it retains some characteristics from its early origins. (A new line is read from a disk file by the keywords "START NEW CARD".) The disadvantages, quirks and structural awkwardness of either program are far outweighed by the advantages they bring to simulation.

6. Recommendations

In the context of the current software requirements of EE7 and in support of engineering efforts for Space Station Freedom, I offer the following recommendations.

a) One civil service employee of the branch should learn Simscript or Modsim well enough to write a small, but non-trivial simulation. This will bring some experience in formal simulation tools into the branch. I suspect that this will pay off rather sooner than later, as requirements and tasks evolve.

b) A portion of the CMS demo which could use graphics output might be identified and coded in Modsim. Modsim has convenient and useful graphics capability, and it is possible that it could be the most efficient means of generating many graphics displays.

c) Some demonstration of Simscript's or Modsim's capabilities should be shown to staff involved in analysis tasks in other branches. These tools are potentially most useful to people who write operational simulations or perform formal analysis. They should know that the tools exist on-site.

7. Reference

Lacovara 87

R. C. Lacovara, Dissertation: Adapted Packet Speech Interpolation, Stevens Institute of Technology, University Microfilm 88-17309 Issue 4907 DAI January 1989.