REPORTS FROM ASSOCIATE EDITORS

Editor's note: One of the most valuable functions that our associate editors perform, both for the headquarters editorial staff and for our readers, is to serve as our "eyes and ears" in their particular technical areas. As a part of that service, we ask them to write brief annual reports on some aspect of those areas. The reports are designed to inform those who are not specialists in that particular area but would like to know what's happening in it. BRG

COMPUTER-ASSISTED MODELING

New problems in software complexity

François E. Cellier
Institute for Automatic Control
Swiss Federal Institute of Technology, Zürich
ETH – Zentrum
CH-8092 Zürich, Switzerland

INTRODUCTION

Complexity is a major problem in modern simulation software. Users always want more features, thus making both the implementation and the mastering of software constantly more difficult. Language designers have estimated that a good computer language should have less than 100 keywords. If it has more, the compiler becomes large and clumsy; the limited acceptance of PL/I illustrates the fate of languages with large, clumsy compilers. The user's manual for a language should be less than 100 pages long to allow the average user to master all its features. Clearly, these are serious limitations for simulation software, since the number of required keywords is dictated by the complexity of the underlying tasks rather than by the wishes of the language designer.

How can we overcome this problem? I believe one of the keys lies in separating data management from the language itself. A simulation system should have a data base management system (DBMS) adapted to the specific needs of simulation. Standridge[1,2,3] has described such a DBMS. Programs that perform different parts of a system analysis may then be implemented independently; they can communicate through the data base. This approach has many advantages; let me begin with some that are unique to simulation:

(1) *Ability to combine outputs from several runs (sometimes called an overplot).* Many simulation languages, such as CSMP-III, offer this feature. However, it becomes natural when all results are saved in a data base. A separate postprocessor can then readily retrieve and display data, regardless of which specific run created it. The postprocessor communicates with the simulation program only through the data base.

(2) *Ability to combine simulated and experimental results.* All that is necessary is a real-time data acquisition program that stores its results in the data base. Although this combination would greatly simplify model validation, it does not exist in any current simulation language of which I am aware.

(3) *Dynamic loading of tables.* The simulationist need not enter tabular data directly into the simulation program (e.g., using a CSMP FUNCTION statement). Instead, the tables are simply stored in the data base. This data may be user-generated, generated by other simulation runs, or even generated by real measurements. For example, this feature would greatly simplify the solution of the finite-time Riccati differential equation. This equation must be computed backward in time, while the system equations must then be computed forward using the backward solution.

(4) *Statistical analysis of noisy data.* One often would like to analyze stochastic results statistically. It is natural to store them in the data base and let an independent statistical analysis program (e.g., SAS or SPSS) perform the analysis. The simulation analyst then need not duplicate the functions of well-established packages.

(5) *Range analysis.* With stochastic models, one often wishes to display the range of the results. Managers generally prefer this representation, since it allows them to see trends and confidence limits that are difficult to express. Again, a postprocessor can perform this kind of analysis through the data base without involving the simulation language at all.

(6) *Actual storage of models, parameter values, experimental frames, and other information related to the simulation project.* Modeling projects, like other projects, have their

own internal data base; this may include partial models, models with different levels of detail or different orientations, validation tests, and base cases. Usually the analyst keeps this information in a file cabinet or in a stack of old runs. Logically, nowadays, like personnel files or accounting records, it belongs in a data base system. Ören and Zeigler have discussed the automation of model management.[4]

Another advantage of this approach is that the independent modules can have separate manuals. A manager may then, for example, study only the postprocessor's manual, since he or she will use the computer only to display data produced by other people's programs. The approach also allows a natural division of tasks, rather than requiring people to write programs that must communicate with each other directly.

SIMULATION ENVIRONMENT

Although one can hardly lament the decline of batch processing, it did have some advantages. The user only had to learn the simulation language and the mechanics of entering input into the batch and obtaining output from it. Some installations even generated control cards automatically.

Interactive processing is often more difficult initially for the novice, since that person must learn about file manipulation (creating a file of the right type, naming it, copying it, etc.) and data manipulation (using a line or screen editor). Unfortunately, the introduction of a DBMS can make things even more complicated. Instead of just stating which printed results are wanted, the beginner must now specify what should go in the data base and in what form. Simulation thus no longer consists of a single well-defined task. Instead, we have a number of different programs (operating system, editor, simulation language, data base management system, statistical analysis package, graphics package, etc.) that form a *simulation environment*.

We can reduce the complexity of what the beginner sees. Modern operating systems allow special operating environments, such as a UNIX environment within a machine running some other system. Of course, these special environments involve extra overhead and are inefficient for large tasks, but that limitation does not matter to the beginner. We could use this approach to implement a special-purpose SIMULATION OPERATING SYSTEM. I suggest this term for new systems rather than *simulation language* or *simulation package*.

Let me describe a simple example. I have implemented a command procedure that simplified the running of ACSL (Advanced Continuous Simulation Language) on a DEC VAX 11/780. The procedure, consisting of roughly 200 lines of code, compiles, links, and executes ACSL. A typical command statement is

@ACSL PILOT LIST FORT GIGI

This statement compiles the pilot ejection study (file PILOT.CSL), producing listings of both the ACSL program (option LIST) and its FORTRAN precompilation (option FORT). The program is linked to the graphics driver for the GIGI terminal from which it is operated. The program is then executed; afterward, the procedure asks the user whether hard copies of the plots and printed results are wanted. The command procedure also has help files and an interrogative mode in which it asks the user for parameters.

This command procedure is useful but does not solve all the beginner's problems in using ACSL. Students still must learn how to call up the editor, copy problems, etc. We have, however, coded another command procedure (roughly 400 lines of code) that simplifies these problems by providing menu-based interaction. This command procedure is acting as a LOGIN file on the students' account. After signing in to the VAX, the students get the following menu displayed:

Current ACSL Problem: NONE

Possible action:
(A) Run ACSL problem
(C) Clean up files
(D) Delete ACSL problem
(E) Edit ACSL problem file
(F) Edit ACSL data file
(G) Display general HELP information
(H) Start/stop HELP menu
(L) List of existing ACSL problems
(M) Display the message of the day
(N) Print Non-ACSL files (after error)
(O) Make old version current again
(P) Purge old versions
(Q) Show disk quota
(R) Read file from other problem
(S) Select ACSL problem
(T) Display status of queues
(V) Display VAX-specific information
(W) Write file to other problem
(Z) Exit from ACSL account

---------->

The user may, for example, press L to obtain a directory, S to select a program for execution, and E to edit a program. In this way, our students are able to master the simulation operating system without difficulties after a very short demonstration.

I am currently implementing a similar operating system for discrete-event simulation, using Pritsker and Associates' SLAM II™ simulation language,[5] together with the SDL™ data base management system, the AID™ statistical analysis package, and the SIMCHART™ graphical postprocessor. The advantages of this combination would be difficult to appreciate if the user had to approach the programs independently. The simulation operating system makes the combination useful and manageable.

In conclusion, I advocate a separation of simulation functions into independent modules that communicate through a data base management system. This separation increases the flexibility of the overall system, while keeping the modules simple enough for the user to manage and the system developer to implement efficiently.

REFERENCES

1 STANDRIDGE, C.R. and WORTMAN, D.B.
"The Simulation Data Language (SDL): A Database Management System for Modelers." *Simulation* 37:2 (August 1981), 55-58.

2 STANDRIDGE, C.R.
"Using the Simulation Data Language." *Simulation* 37:3 (September 1981), 73-81.

3 STANDRIDGE, C.R.
"The Simulation Data Language (SDL): Applications and Examples." *Simulation* 37:4 (October 1981), 119-130.

4 ÖREN, T.I. and ZEIGLER, B.P.
"Concepts for Advanced Simulation Methodologies." *Simulation* 32:3 (March 1979), 69-81.

5 PEGDEN, C.D. and PRITSKER, A.A.B.
"SLAM: Simulation Language for Alternative Modeling." *Simulation* 33:5 (November 1979), 145-157.