THE INTERACTIVE EXTENSIBLE SIMULATION CAPABILITY OF CML

Ronald E. Mills, M. Phil.
President, Puter Associates, Inc.

ABSTRACT

The Conversational Modelling Language is a system for the implementation of hierarchies of computer languages to provide simulation and other modelling capabilities to decision makers. The basic language is interactive, flexible, extensible, structured and exhibits features of particular use to the simulation of health care facilities.

THE CONVERSATIONAL MODELLING LANGUAGE

The construction of decision support systems to serve as a basis for the rational planning of health care delivery organizations necessarily involves modelling of their demand and service processes. The models developed are based on different mathematical structures or modelling techniques for different aspects of the overall health care delivery planning problem: e.g. classificational models of the demand for health care, queueing models of the servicing of this demand, statistical models of its effectiveness, linear optimization models of the consequent resource allocations. Given the integrated implementation of these models on a computer and sufficient empirical demand and service time data, it is then possible to provide planners with the projected behavior of the delivery system for any proposed set of budgets and utilization policies.

Any practical implementation of the above scheme must take into account the fact that the individuals responsible for and knowledgeable about health services planning have a great deal of information about their decision problems that the computer must know if it is to be of significant help to them. However, these people are usually not expert in modelling and rarely expert in computer programming. It is not practical for them to communicate this structural knowledge directly to the machine. The problem can be resolved by the formation of a team: health planners and operations researchers interact and generate specifications for programs that transform empirical data into useful projections. Unless the hierarchical modelling structure is well understood (i.e. everyone perceives correctly what is variable, what is constant, what is structural and what is parametric) small changes in the planning environment or objective can lead to enormous changes in the programs. The decision support system consisting of this team and its programs will not be sufficiently responsive to be used.

What is needed, instead, is a mechanism that allows the health planner to communicate his knowledge and objectives to a computer that is ready to understand him: the computer previously has been given a model and framework in which to place the information being given to it. Further, the construction of this model should be through a direct communication based on a yet more fundamental model and framework. At the base of this hierarchy of models and specifications of models is the computer's understanding (i.e. executability) of its own internal structure.

For example, a systems programmer, working at the machine language (assembler) level constructs facilities for the generation of random variables and the storage of data in lists of entities. A higher level language programmer utilizes this facility to construct a general multiserver queueing model. An operations researcher casts this model in terms of patients and nursing personnel. The health planner provides data and receives expected staff utilization profiles. Each contributor provides both additional structure (knowledge) to the computer and a means of communications (language) for the next contributor.

To some extent, this hierarchical specification of knowledge is already in general practice. Computer languages bridge the gap between machine structure and "models" of data and procedures. Some go further to provide basic simulation world-views: events (SIMSCRIPT), processes (GPSS), activities (SIMULA). With these languages, models with a high degree of generality can be constructed. These may be parameterized by their users at run-time to produce particular cases of interest. However, this hierarchy is usually two-level and at most three, and the style and language of the interaction at each level is radically different. For greater flexibility and efficiency, a mechanism admitting of more levels of specification, consistent language and more sharing of capabilities between levels is needed.

The Conversational Modelling Language (CML) [3] represents an attempt to provide such a mechanism. CML is an extensible, interactive, general purpose programming system with special capabilities for discrete event simulation, linear optimization and statistical analysis. CML is constructed hierarchically from an initial set of fundamental programs (called the CML Nucleus) written in Assembler for IBM 360 or 370 computers. The Nucleus provides

- -all interfaces to the operating system for I/O or supervisor services
- -linguistic facilities providing CML's language extensibility
- -program library management and dynamic object program loading
- -an extensible 360/370 Assembler

With the fundamental programming facilities provided by the Nucleus, a second level algorithmic, structured, programming language is defined. This language (generally referred to as CML) was modelled after the general purpose languages in popular use during the early 1970's when CML was built: PL/I, ALGOL 68, SIMSCRIPT II. CML provides flexible data representations: several datatypes, structures (entities), arrays, lists, queues, trees, and dynamic memory management. Program control statements are generally block oriented, which encourages structured programming, and numerous linkage conventions are supported, which encourage modularization and allow recursion.

CML, the CML Assembler, and the CML linguistic facilities together form a basis on which hierarchies of special purpose, problem oriented, languages can be constructed. This system has, in addition to its extensibility, several attributes not generally available in simulation languages.

Consistent Environments

CML cannot be classified strictly as a compiler or as an interpreter. Like an interpreter, CML can accept and directly execute CML source statements. Like a compiler, CML can translate CML source statements into machine code and save the machine code for later execution. Unlike most compilers, however, CML does not generate object decks that must be link-edited into load modules before being executed. Instead, whenever a CML program is being executed or compiled, a total CML environment is maintained for that program: a small set of CML service routines are always in core when anything related to CML is running and the entire CML compiler is available to any system written in CML. Thus, in a sense, like an interpreter, CML provides support for the execution of its statements while they are being executed. Although it is often not obvious to their users, many strictly compiler languages (particularly PL/I and SIMSCRIPT) provide an execution time environment by forcing the link-editing of modules from their run-time libraries. CML makes this environment explicit, removing the separation between compile time and evecution time.

It is useful to characterize the operational difference between CML and more conventional programming systems. With these languages, the programmer typically creates source programs and uses a compiler to translate them into machine language expressed in an as-yet- nonexecutable form called an object deck. These object decks are then link-edited, typically along with many standard routines from language-associated object libraries, into complete load modules. These are loaded into core and executed. For PL/I, and to a lesser extent for the others, space for program variables is dynamically allocated by the language routines in the load modules, and various capabilities are also available for debugging.

This program construction system has several major disadvantages. The distinct compile, link-edit and execute steps provide three completely dif-

ferent environments, each with its own rules and point of view. There is a set of errors detectable at compile time, another set at link-edit time, and facilities for finding bugs at execution time. This last set of capabilities is not available for finding or correcting compile or link-edit time errors. Further, these facilities must often be placed in the execution phase by steps performed by the programmer before the compilation phase. This program development is characterized by expensive iterations of the compile-linkedit-test cycle until a satisfactory load module is created.

CML differs from this approach by providing the same environment for both the compilation and execution of programs, and by substituting a program library and dynamic loading for linkage editing. When the user enters CML he has all the facilities of the language for both the construction and execution of programs. The programs he creates are represented and treated the same as the programs constituting the CML compiler itself, and the latter set can be easily used as subroutines of the programs he writes.

Interaction

CML utilizes the teleprocessing and time-sharing capabilities of its host system to interact with users. Because of the virtual presence of all levels of language simultaneously, users can enter commands or respond to prompts from running programs not only to provide data to the programs but also to add to or modify the programs themselves. There is a mode of operation in which any statement presented to CML is compiled and immediately executed, instead of becoming part of a program. This allows the incremental interactive solution of one-shot processing or analysis problems.

Facility Sharing Between Language Levels

It is possible, with conventional languages, to provide a problem oriented, interactive, user interface that is sufficiently flexible to call it a language. However, it is generally difficult to provide that interface with any of the data processing tools already available in the implementing language. The structure of CML permits the easy incorporation, at higher language levels, of the facilities of the basic languages. Two examples of this are worthy of note: the CML simulation language includes the CML assembler language as a subset. The simulation programmer can switch into assembler for the coding of tricky or efficient inner loops. The CML expression compiler and executor can be used in higher level user languages, providing those languages with the full computational power of arithmetic, logical and functional syntactic forms.

Events and Waits

The simulation extension of CML provides several mechanisms for the control of the simulation timer. Events (delayed subprogram calls) can be scheduled and timer-invoked programs can contain WAIT statements that simulate the passage of time during the program's execution. This allows the simulation programmer to use a mixture of process specification styles in describing the dynamic relationships of his model. These timer controls are naturally available to higher levels of language as well.

Debugging

Since experimentation with a simulation model often involves program changes, debugging is a constant activity of the simulation user. CML provides interactive explicitly-for-debugging facilities; however, debugging is most aided by the interactive execution capabilities of the languages. When the observed behavior in one level of a model is questionable, statements can be entered and executed at lower levels to inspect and modify the internal representation of the higher level. This allows debugging at any language level to be performed by any user who understands the operation of that level and the use of the language implementing that level.

CML Extensions

The simulation user is often assisted by extensions of CML that have been constructed to serve other ends. GROUPER is a CML extension that implements the mapping of a multi-dimensional variable space into the integers. It was constructed to support the easy specification and efficient implementation of the classification of hospital inpatients into groups for utilization review. It is useful in health services simulation as a mechanism for associating service patterns and service time distributions with pre-defined sets of patient attributes. AUTOGRP [4, 5] is a CML extension that implements an interactive statistical analysis capability. An AUTOGRP user specifies the generation of tables, histograms, various descriptive statistics, hypothesis testing and variance reduction algorithms via a set of commands issued at

his terminal. For the stimulation user, AUTOGRP is valuable as a device both for the derivation of parameters from empirical data before a simulation experiment and the analysis of simulation results. Finally, the Linear Modelling Capability (LMC) provides an interface between CML and MPSX for the formulation, solution, and presentation of linear optimization models.

CML has been used as the basis of several large and small scale health service simulations [1, 2]. The system is available free from the author and is easy to install on any IBM 360/370 with OS.

BIBLIOGRAPHY

1. Bisbee, G. The Relationship between Case-Mix and Management of Allocation of Medical Case Resources, Ph.D. Dissertation, Department of Epidemiology and Public Health, Yale University, 1975.
2. Fetter, R., Mills, R., "HOSPSIM: A Simulation Modelling Language for Health Care Systems", Simulation, March 1975, pp. 73-79.
3. Mills, R., Fetter, R., Averill, R., The CML Reference Manual, Institution for Social and Policy Studies, Center for the Study of Health Services, Yale University, Working Paper W6-49, September 1976.
4. Mills, R., Fetter, R., Riedel, D., Averill, R., "AUTOGRP: An Interactive Computer System for the Analysis of Health Care Data", Medical Care, Vol. 14, No. 7, July 1976, pp. 603-615.
5. Theriault, K., Mills, R., Elia, E., The AUTOGRP Reference Manual, Institution for Social and Policy Studies, Center for the Study of Health Services, Yale University, Working Paper W6-47, July 1976.