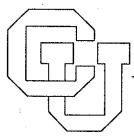
Continuous System Simulation Languages: Design Principles and Implementation Techniques

Richard E. Fairley*

CU-CS-034-73 December 1973



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

*This work supported in part by NSF grants GJ383 and GJ660.

ABSTRACT

Continuous system simulation languages are very high level programming languages which facilitate modelling and simulation of systems characterized by ordinary and partial differential equations. This paper discusses design principles and implementation techniques for continuous system simulation languages.

Following a brief introduction to very high level languages, design principles for continuous system simulation languages are presented. These principles are illustrated by examples from the Continuous System Modelling Program (CSMP) and the Partial Differential Equation Language (PDEL). A typical program in each language is included.

Batch and interactive implementation techniques for continuous system simulation languages are discussed. The classical batch implementation technique is to provide a preprocessor which translates the simulation language into an algorithmic language such as FORTRAN or PL/1. The PL/1 preprocessor is described as a useful language for the implementation of very high level language translators.

The final section of the paper presents an interactive implementation technique which interfaces a batch program processor to interactive graphics display and updating routines. In this manner, efficient simulation code is interfaced to flexible interaction routines. In addition, the batch processor is preserved intact, thus requiring only one implementation of the language for both batch and interactive applications.

INTRODUCTION

Programming languages facilitate the mapping of abstractions from various domains of application into the computer representation of those abstractions. A programming language is thus a vehicle of communication between man and machine. The well-known high level languages, such as FORTRAN, ALGOL 60, and PL/1 permit the implementation of algorithmic computer programs in notations that are both human oriented and machine translatable.

Very high level languages narrow the conceptual gap between man and machine by permitting a description of the problem to be solved; statement of the algorithmic solution is not explicitly required. The very high level language translator automatically synthesizes an algorithmic program to solve the relationships among selected attributes of the system, and to present the solutions to the user in human oriented formats. By restricting the universe of denotation to a specific area of application, a high degree of structure and context can be incorporated into a very high level language, thus making the language easy to use and relatively easy to implement.

Modelling and simulation languages form an important subset of the very high level languages. Modelling and simulation languages have been developed for a wide range of applications including computer simulation (hardware and software), discrete systems simulation, ordinary differential equation modelling, and partial differential equation modelling. Languages for modelling of ordinary and partial differential equations are termed continuous system simulation languages. The activities accommodated by these languages are analysis, synthesis, identification, and optimization of continuous systems. Analysis involves the determination of system output, given the structural description

and the external inputs to the system. Synthesis involves finding a structural description which transforms given inputs into desired outputs. Identification is concerned with finding inputs to produce desired outputs for a given structural description. Optimization involves finding a system which is optimal with respect to some criteria of optimality. Due to the lack of algorithmic techniques for synthesis, identification, and optimization, these activities are best accomplished in a language which permits trial and error analysis in an interactive mode. The following sections of this paper describe design principles, and batch and interactive implementation techniques for continuous system simulation languages.

LANGUAGE DESIGN PRINCIPLES

The high level languages, such as FORTRAN and PL/1, have certain technical characteristics which facilitate the specification of algorithmic processes. These characteristics include the syntactic structure, data types, data structures, operators, control mechanisms, I/O facilities, and storage management capabilities of the language. In a similar manner, very high level languages incorporate certain characteristics which facilitate the description of models. These characteristics include:

- structural description
- parameterization
- algorithm control
- default conditions
- multiple reruns
- initialization control
- termination control
- flexible output control
- user extensions
- free format syntax

The structure of a model specifies the interrelationships among the independent and dependent variables of the model. The structural description includes specification of equations, number of dimensions, geometry, the solution algorithm to be used, and the types of initial conditions and boundary conditions. For example, the Partial Differential Equation Language (PDEL) (1)

permits the transcription of partial differential equations into alphanumeric PDEL statements. A two dimensional diffusion equation of the form:

$$\frac{\partial}{\partial X}\left(\sigma_X \frac{\partial \Phi}{\partial X}\right) + \frac{\partial}{\partial Y}\left(\sigma_Y \frac{\partial \Phi}{\partial Y}\right) = K_t \frac{\partial \Phi}{\partial T}$$

is transcribed into the following EQUATION statement:

```
% EQUATION = 'PX, SIGX*PX, PHI + PY, SIGY*PY, PHI = KT*PT, PHI';
```

Expressions for SIGX, SIGY, and KT are provided in Parameter statements. Symbols PX, PY, PT, and PHI are reserved words in the language.

Other structural statements in PDEL provide information such as the number of spatial dimensions (the % DIMENSION statement), the geometry of the field (the % GEOMETRY statement), the solution algorithm to be used (the % ALGORITHM statement), the boundary condition type (the BCOND and BCGRAD statements), and the initial condition type (the SCOND and SCGRAD statements). Parameterization of a model permits specification of equation coefficients, boundary conditions, and initial conditions in the form of general arithmetic expressions, which can be arbitrary functions of the independent and dependent program variables. Thus, particular solutions to non-linear and time and space varying problems can be determined. For example a parameter statement in PDEL might be of the form:

```
% SIGX = 'X**2 + Y**2';
```

In addition to coefficients, and initial and boundary condition expressions, other parameterization statements in PDEL permit specification of:

- the number of grid points
- the interval between grid points
- maximum number of iterations/step
- the overrelaxation factor
- the output print interval
- the output plot interval
- maximum relative error

A complete PDEL program is contained in Appendix 1.

The Continuous System Modelling Program (CSMP) (2), which is similar to the Continuous System Simulation Language (CSSL) (3), permits the transcription of ordinary differential equations into alphanumeric CSMP statements. For example, a second order differential equation of the form:

$$\ddot{X} + a_2\dot{X} + a_1X = f(X,t); \qquad \dot{X}(0) = A, \quad X(0) = B$$

can be converted into a pair of simultaneous first order differential equations of the form:

$$\dot{X}_1 = X_2 \qquad X_1(0) = A$$

$$\dot{X}_2 = f(X_1,t) - a_2X_2 - a_1X_1 \qquad X_2(0) = B$$

This set of equations can then be programmed in CSMP as:

```
X1 = INTGRL(A,X2)
X2 = INTGRL  (B,Y)
 Y = F(X1,T) - A2*X2 - A1*X1
```

As in the case of PDEL, numerous statement forms permit parameterization of the model. A complete CSMP program is contained in Appendix 2.

The solution method for differential equations in CSMP and PDEL is by finite difference algorithms. A library of algorithms is provided in each language, and the user can select an appropriate one. Parameterized subroutines are retrieved from the translator library and linked together upon recognition of the appropriate source statements. The nonprocedural source program is thus translated into coordinated sequences of calls to subroutines which constitute a procedural program. In CSMP, the user selects an integration algorithm using the METHOD statement. Thus, a statement of the form:

```
METHOD        MILNE
```

will cause retrieval of the MILNE integration routine from the translator library.

In the absence of a METHOD statement, the translator retrieves a default algorithm. Default conditions, free format source text, and non-procedural ordering of statements allow the novice user to concentrate on the essential features of the language. As the user becomes more sophisticated, he can override the default conditions and gain greater control of the simulation program by specifying a particular algorithm, or by supplying his own algorithm in the form of a procedural subroutine. In CSMP, a user supplied integration routine is retrieved upon recognition of the statement:

METHOD CENTRL

The routine itself must be a FORTRAN coded algorithm, which is supplied between the STOP and END statements of the CSMP program.

The multiple rerun capability permits incremental updating of the model, thus permitting parametric solution runs in one batch processing cycle. In PDEL multiple reruns are specified by a % RUNNUMBER statement which specifies the total number of runs. The parameters for each rerun are supplied as illustrated in the sample program of Appendix 1.

Initialization control permits algorithmic manipulation of input data prior to the start of the simulation run. Termination control can assume various forms. In the simplest case, termination can be unconditional; for example, after 100 iterations of the integration algorithm. Conditional termination control can be phrased in terms of acceptable relative error between two solution iterations. Termination control can be combined with the user extension capability to permit, for example, numerical solution of a two point boundary value problem by iterative refinement of an initial value until two successive final values agree to within an acceptable error value.

Output control specifies the solution output interval, and the format of the solution presentation. For example, numerical listings, alphanumeric print plots, and hard copy analog plots may be requested. In the case of analog plots, a data set is prepared for off-line plotting. In CSMP the PRINT, and PRTPLT statements permit specification of output variables. The output interval is specified in the TIMER statement.

The user extension facility in CSMP permits incorporation of user defined macros and subroutines into the simulation program. Macro definitions permit the definition of new statement forms in terms of existing statement forms. Thus, complex functions can be constructed from the elementary ones. User supplied algorithmic subroutines can be provided as the semantic definition of a new statement form. User supplied routines can also replace existing subroutines in the translator library.

In a nonprocedural language, such as PDEL or CSMP, the ordering of source statements is not related to the sequential order of the algorithmic code produced by the translator. Thus, source statements can appear in any order. In some cases, sorting routines are used to automatically order the source statements prior to translation. For example, the sorting method used in CSMP assumes that all integrator outputs are available. The sorting routine then traces each input of each integrator backwards through all the preceding functional operations until each path is terminated by the appropriate integrator output. The computational sequence for a system of ordinary differential equations is established by reversing the order of operations encountered in each computational path. Within a source statement the syntax is free format. Delimiters such as blanks and column alignments are avoided in order to make the language easy to use.

A BATCH IMPLEMENTATION TECHNIQUE

The classical implementation technique for a very high level language is to provide a preprocessor which translates very high level language statements into a procedural language such as FORTRAN or PL/1. While this approach is less efficient than direct translation into assembly language or machine code, it is more flexible and easier to implement. Desirable features of a preprocessor include a pattern matching macro facility, and the ability to conditional retrieve source text from a program library. These capabilities permit retrieval and compilation of a procedural source program, whose final form is dependent upon the very high level language source statements.

The PL/1 preprocessor, which was used as the implementation language for PDEL (4), provides desirable features for the implementation of a very high level language translator. Preprocessor statements are distinguished by a leading percent sign (%). The preprocessor statement types and their usage are summarized in Table 1.

The %INCLUDE statement retrieves source text from a program library and incorporates it into a source program. Retrieved library text may contain both preprocessor and nonpreprocessor statements. The retrieved source text is subsequently scanned for preprocessor statements. In particular, the retrieved text may contain other %INCLUDE statements. Conditional retrieval of source text can be accomplished by using the %INCLUDE statement in conjunction with the %IF statement.

All preprocessor variables must be declared in a %DECLARE statement. Preprocessor variables can be of integer type or character string type. Integer variables can be used as counters to control the scan of the source text.

TABLE I

PL/1 Preprocessor Statements

Statement Type	Use
% INCLUDE	Retrieval of source text from a library.
% DECLARE % Assignment % ACTIVATE % DEACTIVATE	Declaration and manipulation of preprocessor variables
% PROCEDURE % RETURN	Declaration of preprocessor functions
% DO % END % IF % GOTO % Null	Control of the preprocessor scanner

Character string variables permit alteration of procedural program text. When a preprocessor variable is encountered in the text, it is replaced by the character string which is the current value of the variable. In this manner, procedural source text can be parameterized by the use of preprocessor variable names. Statements in the very high level language can be used to assign values to the preprocessor variables, thus particularizing the algorithmic program to the problem description.

The %ACTIVATE and %DEACTIVATE statements permit selective scanning of the source text. Thus, selected portions of the source program can be included or omitted from preprocessing.

Preprocessor function procedures are similar to preprocessor variables, in that the value of the function (an integer or character string) is copied into the source text in place of the function name. However, preprocessor functions are more powerful; they can generate new source statements whose formats are determined by the arguments passed to the function procedure.

The %DO and %END statements delimit preprocessor DO groups, and preprocessor DO loops. Preprocessor DO loops allow iterative preprocessing. The %IF statement resembles the normal IF statement; the %ELSE clause is optional, and nesting is permitted. The %GOTO statement is used to transfer control of the preprocessing scanner. The %GOTO statement must specify the label of a preprocessor statement. The %NULL statement can be used to provide a preprocessor transfer of control point, and to allow proper pairing of the %THEN and %ELSE clauses in nested %IF statements.

The primary disadvantage of the PL/1 preprocessor as an implementation language is its inefficiency. However, it is easy to use and provides a great deal of flexibility. This facilitates modifications and refinements to the language being implemented.

AN INTERACTIVE IMPLEMENTATION TECHNIQUE

The user of a continuous system simulation language is typically concerned with analysis, synthesis, identification, and optimization of continuous systems models characterized by ordinary and partial differential equations. Due to the lack of algorithmic techniques for synthesis, identification, and optimization, these activities are often realized by iterative analysis, and by other trial and error solution methods which require feedback from the user. These activities are well suited to an interactive computing environment. In addition, the use of an interactive graphics terminal permits human oriented description of models and input data, and the rapid assimilation of large quantities of output data.

The basic design criteria for an interactive graphics modelling system are flexible interaction, and efficient machine code. Flexible interaction is a necessity in dynamic modelling studies; the on-line user must be able to alter his model in an arbitrary manner by entering any valid set of statements (including an entire source program) via the interactive console. However, efficient machine code for analysis and display computation is also necessary; the finite difference algorithms utilized in the numerical solutions of ordinary and partial differential equations, and the high data rates inherent in the generation and manipulation of graphics console display orders place heavy computational demands on the largest and most sophisticated digital computers.

Flexibility and efficiency have traditionally been regarded as opposite extremes of a common spectrum; the position that a particular implementation technique occupies in that spectrum is a function of the trade-off decisions

made in the software design process. For instance, batch mode compilers produce efficient, optimized, but inflexible object code; incremental compilers and interpreters provide on-line flexibility, but are typically orders of magnitude slower in execution speed than batch implementations of the same language.

The concept of Batch Compatible Interaction (BCI) was developed to achieve the conflicting goals of efficiency and flexibility in interactive computing systems (5). In a BCI system, batch compiler generated analysis and display code is interfaced to interpretive routines which permit flexible output displays, and on-line updating of the source program.

Numerous advantages accrue by using existing batch program processors to generate the machine code for analysis and display:

1. Efficient, directly executable object code is generated.
2. One processor is utilized for both batch and interactive applications, thus insuring compatability between the two versions.
3. Existing higher level software for interactive display can be utilized.
4. Existing batch programs can be converted directly to interaction.
5. New language users can concentrate on learning the language in the batch mode, using existing documentation.
6. New on-line users already familiar with the batch language are easily acclimated to the interactive environment.
7. Programs can be debugged and checked out in either the batch or interactive mode, as desired.
8. Machine independent batch programs can be transferred from other machines to the interactive machine.

9. The interactive display routines can be easily modified.
10. The accumulation of sophisticated and elaborate program processors developed during the past decade is possible: a new implementation of the language is not required.

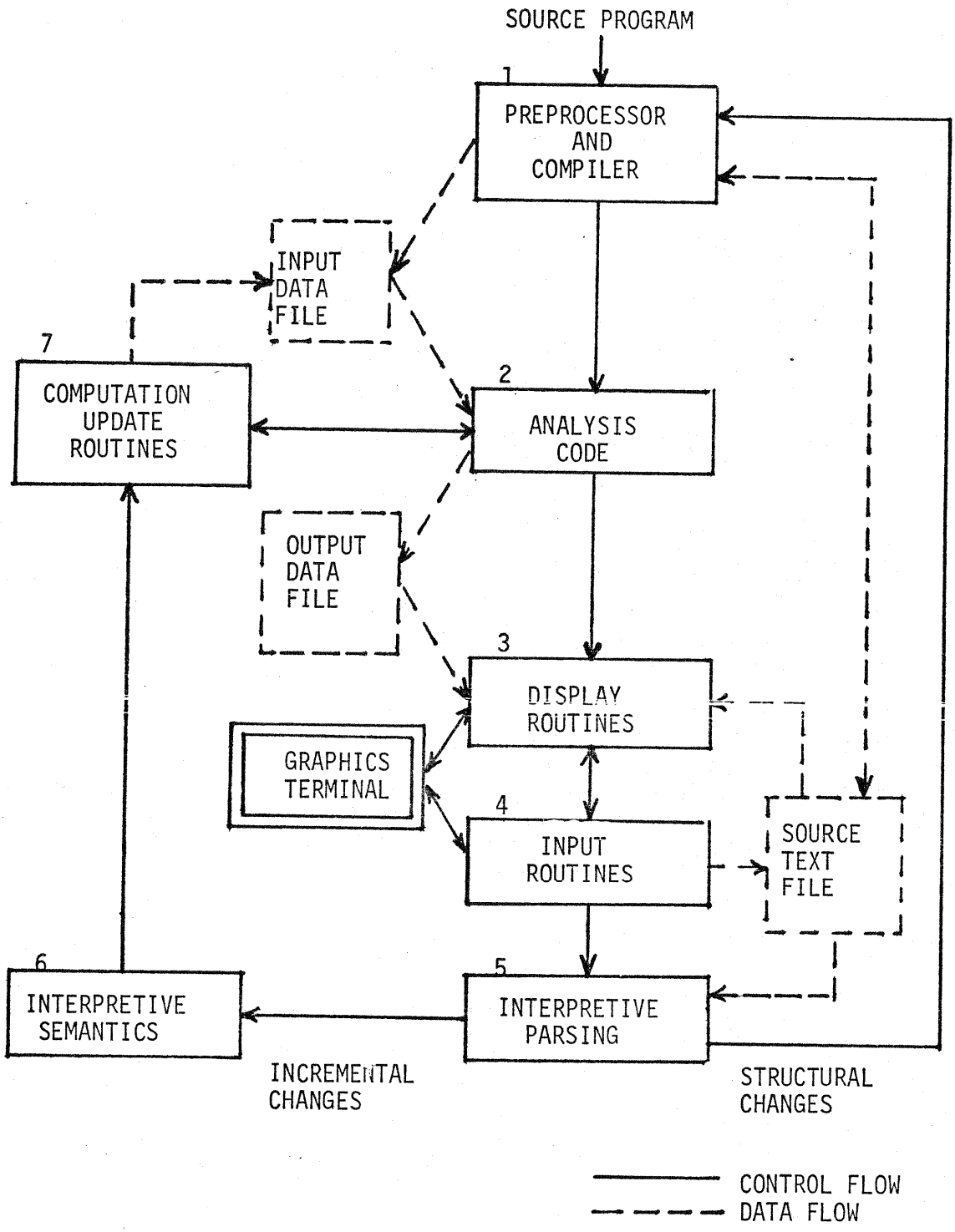
The flow of control in a BCI dynamic modelling system is illustrated in Figure 1, and explained in the following discussion:

STEP 1: The user enters a simulation program by the card reader or at the graphics terminal. During preprocessing, the source text is parsed to determine whether execution is to be accomplished in the batch or the interactive mode. If the program is to be processed interactively, interrupt routines, and interpretive display and input routines are retrieved from a program library and linked to the compiler produced computational analysis code.

STEP 2: Processing is completed for the submitted program, utilizing the input data file. An output data file containing user requested output data is prepared for subsequent display and/or printing. When the initial analysis cycle (including multiple reruns) is completed, the program either terminates normally, in the batch mode, or else branches to the display routines to initiate interaction.

STEP 3: The display routines permit the user to observe the output data and the source text which produced that output. Numerical solution values are transformed into display orders which generate solution graphs and equipotential contour plots when executed by the terminal.

STEP 4: The user moves freely between display and input routines; examining solutions, viewing old source texts, and entering new source statements for on-line processing.



BATCH-COMPATIBLE INTERACTION
FIGURE 1

STEP 5: When the user has completed updating the model, control is passed to the interpretive routines for initiation of the next solution cycle. The interpretive routines have two primary functions: parsing of console entered source text, and generation of semantics corresponding to valid source statements. Parsing results in one of two alternatives, depending on the type of statements entered by the user: structural processing or incremental processing. Structural statements are those which change the form of the integration algorithm; for instance, changing the equation statements, the number of dimensions, the equation type, the boundary condition type, or user selection of a different algorithm. Structural changes necessitate recompilation of the analysis code; the flow of control returns to step 1. A longer response time delay will be required for recompilation of structural statements than for incremental changes to the current model.

STEP 6: Incremental changes invoke interpretative routines which update the computational analysis code.

STEP 7: In addition to supplying new input data, the computational updating routines can alter the execution sequence of the analysis routines, and selectively exclude or include compiled analysis code to be used in the next solution cycle.

STEP 8: During reexecution of the analysis routines in step 2, incremental updating is incorporated as follows:

Loop indices and tests relating to geometry, step size, errors, and numbers of iterations, etc. are set in step 6. In step 2, all of the computational parameters are evaluated as if the present solution cycle was the last of the initial reruns. At each step of the computation, just prior to

execution of the integration algorithm, a branch is made to the updating routines of step 7, where updated parameters are evaluated interpretively. Any new values generated in this evaluation overwrite the values used in the last initial rerun. The integration routine is executed to completion in this manner, and new output data is included in the output data file. The entire process between steps 2 and 7 (or 1 and 7) can be repeated iteratively until the desired solution is obtained, or until the user determines that the desired solution is not obtainable.

The three major software components in a BCI system are the analysis (simulation) code, the display code, and the interpreter. The interaction of these components is illustrated in Figure 2. Interfacing of the analysis and display routines in BCI-PDEL was accomplished by compile time replacement of calls to batch output routines with calls to interactive graphics routines. These routines have access to the output data file utilized by the analysis code. Thus, solution displays can be prepared and presented by the display routines. The implementation was slightly complicated by the fact that analysis code is generated by the PL/1 compiler and display code is generated by the FORTRAN compiler. For instance, PL/1 arrays are referenced by dope vectors and are stored in row major order, while FORTRAN accesses arrays in column major order with no dope vectors assumed. Despite minor problems, the two languages were successfully interfaced.

The basic functions of the interpreter are:

- parsing of interactively entered source statements
- formatting and writing input data into the analysis code input file
- preparation of a source code file for recompilation (if necessary)
- control of the execution sequence of batch generated routines

The interpreter was generated by a meta-translator based on the APAREL system designed by Balzer and Farber (6). APAREL consists of a set of PL/1 callable subroutines which permit definition of syntax in a BNF-like notation, and the association of PL/1 coded semantics routines with a successful parse.

Execution of the same batch compiled code for both batch and on-line entered source statements can be accomplished by specifying the label of a call statement to the batch compiled procedure as the entry point of the semantics generator for the on-line entered source statement. FORTRAN generated subroutines can be invoked by supplying a PL/1 call statement which calls the FORTRAN subroutine. Control is passed to the PL/1 call statement upon successful parsing of the appropriate on-line input statement. Thus, it is always possible to execute PL/1 or FORTRAN compiled object code in an on-line mode, independent of the particular function of the compiled program being converted to batch-compatible interaction. If necessary or desirable, the original batch program can be partitioned into sets of subroutines, each of which is a meaningful atomic program unit in the on-line mode. The segmentation of a program into sets of subprograms can always be accomplished; no new algorithmic code need be inserted into the program being converted to batch-compatible interaction. The primary advantage of this approach is that the on-line interpreter can coordinate the execution sequence of batch compiled routines. Thus, the compiler generated object code can be restructured at run time to reflect the new processing requirements imposed by interactive processing. The entire program must be recompiled if the interactive input requests a subroutine that has not been compiled and linked to the program. In this event, a warning message is issued, and a somewhat longer response delay is incurred.

The flexibility of a BCI system is illustrated in Figure 1:

1. Blocks 1 and 2 provide a batch processing capability.
2. Blocks 1 - 2 - 3 - 4 - 1 establish a remote batch processing loop.
3. Blocks 3 - 4 - 5 - 6 - 7 - 3 form an interactive processing loop.

The BCI system thus provides the advantages of efficient batch processing and flexible interactive processing.

The BCI methodology has been used to convert CSMP and PDEL from the batch mode to Batch Compatible Interaction (7). The implementations are truly batch compatible: Batch mode CSMP and PDEL programs can be processed in the interactive mode without modification, and with complete agreement of obtained solutions.

Execution time for the interactive solution cycle is dependent on the types of statements processed, but in general, response time is short enough that the user experiences no significant time delays, except in the case of structural changes, which necessitate recompilation of the analysis routines. Execution time for the analysis routines is not noticeably different between batch and interactive modes, because the same object code is executed in both cases.

Summary

This paper has presented design principles and implementation techniques for continuous system simulation languages. The design features of a simulation language were summarized and illustrated by sample programs in the CSMP and PDEL languages.

The PL/1 preprocessor language was described as a batch implementation language for very high level language translators. The utility of preprocessor PL/1 has been demonstrated in the implementation of PDEL.

Batch Compatible Interaction was discussed as an interactive implementation technique which utilizes the batch processor. The advantages of batch efficiency and interactive flexibility are accrued in this manner.

REFERENCES

1. Cardenas, A. F. and W. J. Karplus, "PDEL - A Language for Partial Differential Equations", CACM, Vol. 13, pp. 184-191, March, 1970.

2. System/360 Continuous System Modeling Program (360A-CS-16X), IBM Application Program H20-0240-3.

3. SCI Software Committee, "The Continuous System Simulation Language (CSSL)", Simulation, pp. 281-303, December, 1967.

4. Cardenas, A. F. and W. J. Karplus, "Design and Organization of a Translator for a Partial Differential Equation Language", AFIPS Conference Proceedings, Vol. 36, pp. 513-523, Spring, 1970.

5. Fairley, R. E., "A Batch-Compatible Interactive Computing System for Dynamic Modelling", Ph.D. Dissertation, UCLA, July, 1971.

6. Balzer, R. M. and D. J. Farber "APAREL - A Parse and Request Language", CACM, Vol. 12, pp. 624-631, November, 1969.

7. Fairley, R. E. and A. F. Cardenas, "Batch and Interactive Simulation of Partial Differential Equation Models", 1971 Summer Computer Simulation Conference Proceedings, July, 1971.

APPENDIX 1

Sample PDEL Program

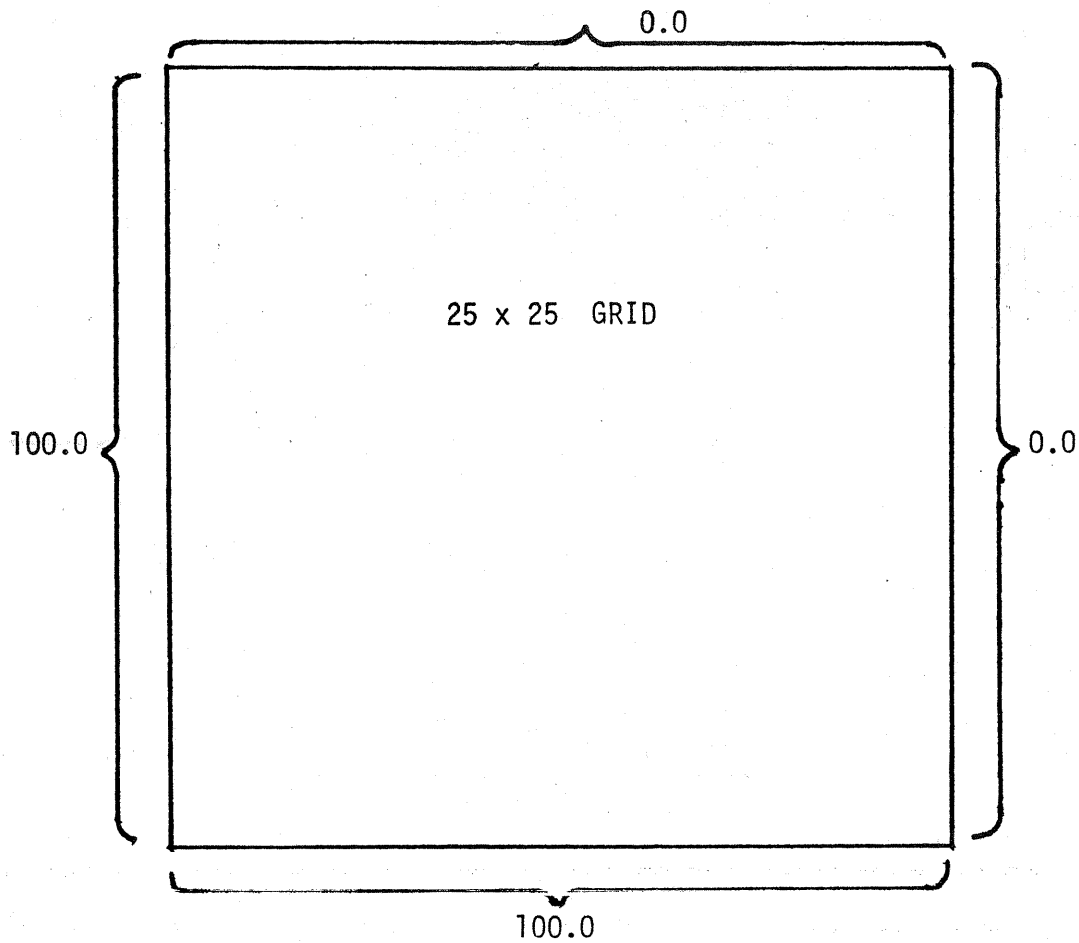
The following two-dimensional elliptical equation is to be modelled:

$$\frac{\partial}{\partial X}\left(\sigma_X \frac{\partial \Phi}{\partial X}\right) + \frac{\partial}{\partial Y}\left(\sigma_Y \frac{\partial \Phi}{\partial Y}\right) = K$$

where:

$$\sigma_X = \sigma_Y = 10 + Y \; ; \qquad K = 0$$

The initial solution run is for a discretized rectangular 25 x 25 grid, with a spacing of 2 units between grid points in both the X and Y directions. Dirichlet boundary conditions of 0.0 and 100.0 are specified as illustrated in Figure 3.



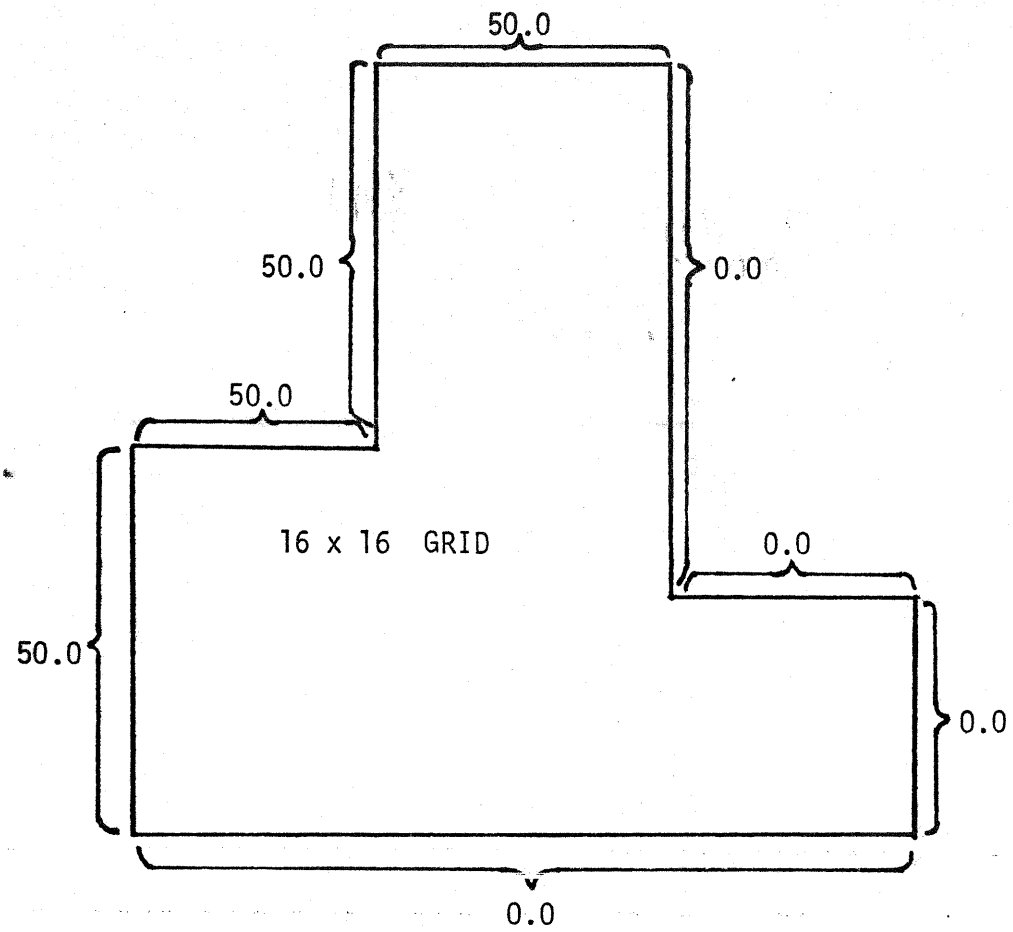
GEOMETRY AND BOUNDARY CONDITIONS FOR PDEL EXAMPLE
FIGURE 3

In addition, the following items are specified:

1. a overrelaxation factor of 1.7
2. a relative error tolerance of 0.05
3. a maximum of 100 iterations
4. solution printout at every grid point

The model is updated for the second run by changing the geometry and boundary conditions to those illustrated in Figure 4. In addition, the following changes are made:

1. a 16 x 16 grid
2. a spacing of 1.0 between grid points in the X and Y directions



GEOMETRY AND BOUNDARY CONDITIONS FOR PDEL EXAMPLE
FIGURE 4

3. an overrelaxation factor of 1.5

4. a relative error tolerance of 0.1

5. a maximum of 155 iterations.

The PDEL program has the following form:

```
 1   % INCLUDE $PDEL(INITIAL);
 2   % DIMENSION = '2';
 3   % DCL    (PARAM1,PARAM2) CHARACTER;
 4   % EQUATION='PARAM1*PX,PX,PHI + PY,PARAM1*PY,PHI=PARAM2';
 5   % PARAM1='10+Y';       %   PARAM2='0.0';
 6   % GRIDPOINTSX='24';    %   DELTAX='2.0';
 7   % GRIDPOINTSY='24';    %   DELTAY='2.0';
 8   % GEOMETRY='(1:23,1:23)';
 9   % BCOND='(*,0)=100;   (*,24)=0;   (0,*)=100;   (24,*)=0';
10   % MAXERROR='0.05';     %   ITERATE='100';
11   % ORF='1.70';
12   % PRINTINTX='1';       %   PRINTINTY='1';
13   % RUNNUMBER='2';
14   % INCLUDE $PDEL(HEART);
15   % /*   RUN2     */;
16   % MAXERROR='0.1';
17   % ITERATE='155';
18   % ORF='1.5';
19   % GRIDPOINTSX='15';    %   DELTAX='1.0';
20   % GRIDPOINTSY='15';    %   DELTAY='1.0';
21   % GEOMETRY='(1:15,1:4)'   (1:11,5:6);  (5:11,7:14)';
22   % BCOND='(*,15)=50.0;   (0,*)=50.0;   (*,0)=50.0';
23   % INCLUDE $PDEL(HEART);
```

APPENDIX 2

Sample CSMP Program

The following second order system is to be modelled by two simultaneous first order differential equations:

$$\ddot{X} + A\dot{X} + BX = Y \qquad X(0) = IC1$$

$$\dot{X}(0) = IC2$$

or:

```
X1 = INTGRL(IC1,X2)
X2 = INTGRL(IC2,F)
F = Y - A*X2 - B*X1
```

where

```
X1 = X.
```

The following conditions are specified for the model:

1. Milne integration
2. Solution interval: 0 - 7.5
3. Solution step size: 0.05
4. Print and plot X1, X2, and F
5. Parameters: A = 4.0, B = 6.0, IC1 = 2.0, IC2 = 0.0, Y = 10.0

The model is updated in the following ways:

1. Parameters: A = 2.0, B = 3.0, IC1 = 5.0, IC2 = 3.0
2. Solution interval: 0 - 15
3. Step size: 0.1
4. Integration method: Runge-Kutta

The batch CSMP program for this model has the following form;

```
*               FIRST RUN

                    F=Y-A*X1-B*X2
                    X2=INTGRL(IC2,X1)
                    X1=INTGRL(IC1,F)
METHOD              MILNE
TIMER               DELT=0.05,FINTIM=7.5,OUTDEL=0.05,PRDEL=0.1
PARAM               A=4.0, B=6.0, IC1=0.0, IC2=3.0, Y=10.0
PREPAR              X1,X2,F
PRINT               X1,X2,F
PRTPLT              X1,X2,F
END

*               SECOND RUN

PARAM               Z=2.0,B=3.0,IC1=5.0,IC2=3.0
TIMER               DELT=0.1,FINTIM=15.0,OUTDEL=0.1,PRDEL=0.1
METHOD              RKS
END
STOP
ENDJOB
```