PROSE

A VERY HIGH LEVEL GENERAL PURPOSE LANGUAGE

by

Joe M. Thames, Jr. and Michael N. Robinson

PROSE, Inc.
8616 La Tijera Blvd.
Los Angeles, CA 90045

23 May 1974

Keywords: Mathematical, Calculus, Simulation, Optimization, Language

Presented at Mathematical Software II Conference at Purdue Univ, May 1974 by invitation from John R. Rice in session chaired by William Gear. Subsequently excluded from conference proceedings as "too philosophical". Reviewers claimed that such a language could not be implemented. They were unaware that the product had already been introduced by Control Data in January of 1974.

ABSTRACT

PROSE is a general-purpose scientific programming language with syntax and semantics for addressing problems at the calculus level as well as the algebra level. PROSE was designed to be the logical successor to FORTRAN and BASIC. Its design integrates under a consistent syntax the broad capabilities of general-purpose programming, higher-level calculus programming and simulation modeling.

In the domain of general purpose programming, PROSE provides the full capabilities of contemporary procedural languages, including:

- o Simple arithmetic and algebra
- o Vector/matrix algebra
- o Convenient input and output
- o Program logic path analysis
- o Variable auditing and trapping

The calculus operations are concerned both with differential and integral calculus and with algebraic problems whose solution techniques require the values of derivatives. A partial list of the types of calculations and related problem areas is as follows:

- o Automatic evaluation of first and second order analytic derivatives, with no limit on the number of independent or dependent variables or on the complexity of their relationships.

- o Evaluation of maxima and minima of functions with no limit on function complexity or dimensionality.

- o Solution of systems of equations, including both algebraic and ordinary differential equations, whether they be implicit or explicit, linear or nonlinear.

- o Solution of process-identification problems, boundary-value problems, and optimal control problems.

- o Linear and nonlinear programming subject to arbitrary combinations of linear or nonlinear equality and inequality constraints.

PROSE simulation capabilities unify the techniques of discrete-event modeling and continuous-system modeling into a single, coherent simulation language. The simulated systems are represented in terms of parallel, independent processes, sharing and competing for a variety of resources. Their simulation models may employ any PROSE capabilities for general-purpose and calculus programming.

INTRODUCTION AND OVERVIEW

Today's major cost in computer problem solving is the cost of manpower and manpower delay in programming. Since the mid-sixties there has been a need for a powerful "do it yourself" programming language for engineers and scientists. The need was partially fulfilled by the introduction of time-sharing and BASIC, which specifically appealed to problem solvers, but did not provide much problem-solving power. For the most part, problems have been limited in both size and complexity to approximately the range of small scale machines like the IBM 1620. Yet, this limitation is not imposed by the lack of computing power, but rather by the low *problem level* of the programming languages available.

Implicit Mathematics

At the problem level of a language, one need only program *what* is to be solved, without having to specify *how* it is computed. The problem level of FORTRAN, BASIC, PL/I, etc. is *explicit algebra*. Higher level problems must be broken down into explicit algebra before they may be programmed in these languages. However, the great bulk of scientific problems are higher level. In the realm of mathematics, five problem levels naturally occur as shown in Table 1.

TABLE 1

MATHEMATICAL PROBLEM LEVELS

Problem Level	Primary Languages	Secondary Languages
Arithmetic	Assembly Languages	FORTRAN, etc.
Explicit Algebra	FORTRAN, etc.	PROSE
Explicit Calculus	CSSL, DYNAMO, etc.	PROSE
Implicit Algebra	PROSE	
Implicit Calculus	PROSE	

As illustrated, of the five problem levels, contemporary languages provide capabilities only in levels involving explicit problems. To clarify: an explicit problem has the structure shown in Figure 1, namely that the known quantities are the

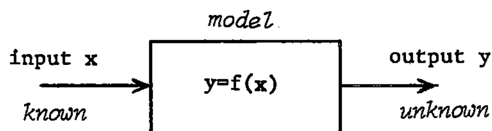


Figure 1. Explicit Structure

independent (input) variables and the unknown quantities are the dependent (output) variables. Problems having this structure include, at the explicit calculus level, initial value problems of ordinary differential equations. Contemporary languages have been able to easily handle problems of this structure because the methods of solving such problems require no more than basic arithmetic.

In the realms of higher mathematics, implicit problems abound. Implicit problems have the fundamental structure illustrated in Figure 2, namely that the known (or characterizable)

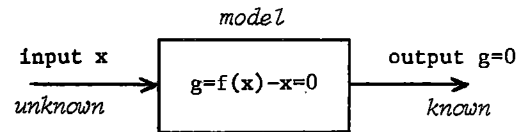


Figure 2. Implicit Structure

quantities are the output (dependent) variables, and the unknown quantities are the input (independent) variables. Algebraic problems having this structure include high order polynomials, transcendental equations and systems of simultaneous equations. Calculus problems having this structure include boundary-value ODE problems, implicit ODE problems, optimization problems and model fitting problems.

General methods for solving implicit problems require more than basic arithmetic. The most general methods are successive approximation methods based upon repeated application of the Taylor series formula (e.g., Newton's method.)

$$f(x + \Delta x) = f(x) + \frac{\partial f}{\partial x} \Delta x + \frac{1}{2} \frac{\partial^2 f}{\partial x^2} (\Delta x)^2 + \ldots$$

Figure 3 is a modification of Figure 2 to illustrate the application of a first order Newton method.

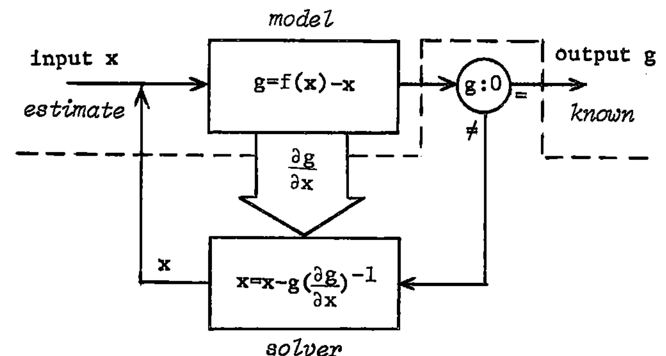


Figure 3. Implicit Problem Solution

The goal of a language designed for programming implicit problems as easily as explicit problems, is to have the user specify *only* what is shown in Figure 2, namely, the model, and identification of the input and output. Everything below the dashed line in Figure 3 must be provided automatically by the language, i.e., *submerged below the user's threshold of awareness and concern*.

To *effectively* submerge the mechanics of solution below the language, certain design requirements

must be met. First, the solution methods employed must be as reliable and independent of specific usage as possible, so that they may be used as "black boxes" by engineers who have no numerical expertise. They must not require "numerical tuning" to fit each problem. Second, the linearity or nonlinearity of problems must be largely immaterial, i.e., the fundamental solution mechanics must be applicable to both.

Procedural Calculus

The key semantic process of PROSE that satisfies both of the design requirements is an automatic arithmetic for *exact* evaluation of partial derivatives. Unlike symbolic manipulation (which we have found impractical because of excessive storage requirements), this process does not generate formulas for derivatives. Rather, it computes derivatives via an extended arithmetic which we call "digital calculus." This process is the foundation of implicit problem solving in PROSE.

Implicit problems are specified in PROSE by the FIND statement:

FIND $x_1, x_2, \ldots$ IN *model* {BY *solver*} TO *criterion*

where $x_1, x_2, \ldots$ are the names of the independent variables to be determined, *model* is the language block where the formulas of the model appear, *solver* is the name of the numerical solution method, and *criterion* is a phrase of either of the forms

MATCH $g_1, g_2, \ldots$

MINIMIZE f

MAXIMIZE f

where $g_1, g_2, \ldots$ are the names of dependent variables, computed in the model, which are constraints representing implicit algebraic equations, and f is the name of a dependent variable representing an objective function to be optimized. In the case of constrained optimization, the clauses

HOLDING $g_1, g_2, \ldots$ *(greater than or equal to zero)*

MATCHING $h, h, \ldots$ *(to zero)*

could appear in a FIND statement, representing inequality constraints and equality constraints, respectively.

In most cases, the FIND statement activates the digital calculus mechanism to compute gradient vectors

$$\left[\frac{\partial}{\partial x_1} \frac{\partial}{\partial x_2} \cdots\right]^t$$

for first order problems, and in addition, Hessian matrixes

$$\begin{bmatrix} \frac{\partial^2}{\partial x_1^2} & \cdots & \frac{\partial^2}{\partial x_n \partial x_1} \\ \vdots & & \\ \frac{\partial^2}{\partial x_1 \partial x_n} & & \frac{\partial^2}{\partial x_n^2} \end{bmatrix}$$

for second order problems, for each dependent variable computed in the *model* block (or blocks called by the *model* block). The values of these derivatives are stored temporarily, and are used by the *solver* in its solution process. In addition, the user may access the values of the derivatives from within the *model* block: (a) for printing purposes using statements such as

PRINT GRADIENTS OF $y_1, y_2, \ldots$

or (b) to assign the values of derivatives to variables using array functions such as

GRADY = .GRAD(Y)

which stores the gradient vector of the variable Y in the vector GRADY. (If the variable GRADY was previously a scalar, then this statement would make it a vector - a standard feature of array functions in PROSE.)

Actually, the FIND statement invokes the *solver*, and the *solver* controls the execution of the *model*. It may execute the *model* several times with different values of the independent variables $x_1, x_2, \ldots$, and it controls the activation of derivative evaluation. When the solution criterion has been met, the *solver* returns control to the user program at the statement following the FIND statement.

Other types of procedural statements are available for high level mathematics in PROSE, including solution of ordinary differential equations. All of these statements are backed up by a very advanced procedural algebra language which subsumes most of the scientific features of FORTRAN and PL/I, including array functions, interrupt, and simplified input/output. The high level statements may be easily combined in structured procedures to solve complex mathematical problems such as boundary value problems, implicit ODE's and optimal control problems.

Simulation

Digital simulation is a powerful method of analysis which is complementary to the problem solving methods of calculus and numerical mathematics. Traditionally, simulation modeling has been separated into two disciplines, discrete-event modeling and continuous-system modeling, each having its own set of simulation programming languages. PROSE provides a single modeling language for both disciplines which facilitates simulation of combined discrete-continuous models in addition to the combination of simulation and calculus problem solving in the same program.

In PROSE simulation, a modeled system is represented as a set of attributes called *facilities* and *registers*, and a set of programmed *processes*. Processes are PROSE blocks which define the generic behavior of independent *transactions*, where each transaction is a dynamic instance ("copy") of a process block. Each transaction competes with other concurrent transactions (of the same process or of different processes) for the possession of system facilities, and information is transmitted between transactions via system registers.

The programmed behavior of a transaction may result in a chain of discrete events, such as the behavior of an order in a job shop, or a continuous trace, such as a flight trajectory, or an intermittent discrete-continuous activity such as the movement of an automobile in a traffic network. Once created, all transactions are independent of one another, both with respect to the values of their variables and the manner in which they propagate through time. In the simulation mode, each transaction is the equivalent of a PROSE program in the nonsimulation mode; it has its own set of global variables whose values are communicated between the procedural blocks of the transaction, but are independent of the global variables of other transactions (even of the same process). Any transaction may contain a calculus procedure, such as an optimization problem.

Whether a transaction is discrete or continuous depends upon how it propagates through time from one active state to the next. An individual transaction may propagate discretely at one time and continuously at another, according to the logic of the program. Discrete propagation is performed by executing the statement:

WAIT | FOR *deltime*
UNTIL *abstime*
WHILE *register relation expression* |

The active transaction becomes *passive* (inactive for a specified time) for either WAIT FOR or WAIT UNTIL. It becomes *suspended* (inactive for an indefinite period) for WAIT WHILE. If the WAIT criterion is currently unsatisfied, the WAIT statement is disregarded and the transaction remains active. This may arise from a nonpositive *deltime*, an *abstime* earlier than the current time, or a register value which fails to satisfy the relational expression.

Continuous propagation is performed by executing the statement:

ADVANCE *model* EQUATIONS $\dot{y}_1/y_1, \dot{y}_2/y_2, \ldots$

STEP *delta* | FOR *deltime*
UNTIL *abstime*
WHILE *register relation expression* |

where *model* is a procedural calculus block containing the differential equations for $\dot{y}_1, \dot{y}_2, \ldots$ to compute the dependent variables $y_1, y_2, \ldots$ as functions of simulated time. The propagation technique is to integrate the model through time until the ADVANCE criterion is no longer satisfied. The maximum integration step size is either *delta* or some function of *delta* computed by the integration solver to hold discretization errors within specified bounds.

The interaction of all of the transactions of the system is reflected in the *system state*, the content of all of the *system attributes:* time, facilities, and registers. A snapshot of the system state gives an instantaneous picture of the progress of a simulation. A time history of the system state therefore yields the complete behavior profile of the system for a given transaction loading. This behavior profile is measured by the accumulation of statistics for each facility and register, and the results are obtained from the display of these statistics in a time profile at the end of a simulation run.

Programming and Program Execution

PROSE programs may be developed and executed in both batch or conversational timesharing modes. In either mode, the user must communicate through an executive which manages the facilities of the PROSE programming system. These facilities include:

- The PROSE *compiler*, which translates PROSE language blocks into relocatable object decks.
- A general-purpose macro-processor capable of translating macro languages into PROSE, FORTRAN or other languages supported by the PROSE system (e.g., assembly language).
- The PROSE *library*, which contains processing functions required by the PROSE language and a library of utility procedures available to service user requests.
- The PROSE *assembler*, which combines the output of the PROSE compiler with selected library routines to compose an executable PROSE program.
- Program file management facilities which provide automated configuration management of developing programs and production programs.

Conversational programming and execution is controlled by a timesharing executive called PRIMP, which provides an operational environment for interactive conversation at a "minimal terminal" such as a teletype. The PRIMP user may perform two types of tasks, *program synthesis and job processing*. While they may be distinct and, in fact, could be distributed over several terminal sessions, PRIMP presumes that the tasks are integrally related, and therefore automatically performs maintenance operations to augment user actions.

Program Synthesis - The functions comprising program synthesis include:

- Input, organization, and editing of source text.
- Source language compilation: PROSE, FORTRAN, assembly language and macro languages
- Management of object decks resulting from compilation

The maintenance operations performed by PRIMP to support program synthesis consist of the interactive execution of user commands and "bookkeeping." Whenever a user is communicating with PRIMP, the permanent context of his conversation is recorded in a file structure called a *book*, as though a stenographer were carefully noting every substantive change in a running log. This book, called the *active* book is a conversational workspace whose contents are altered by certain PRIMP commands called bookkeeping commands, which include editing, copying, and miscellaneous recording commands.

Each user may have any number of books and he may share their contents with other users in a controlled fashion. The physical entries in a book are of two types, text *items* and object *decks*. For convenience in manipulating book entries, a book may also contain groups of items and decks, called *sets* and *packs* respectively. Each set may contain any number of items and, possibly, other sets. Similarly, each pack may contain any number of decks and other packs.

Although all entries in a book are structured, the only type of entry which may be edited is an item, which physically consists of one or more lines of text in a prescribed sequence. Thus, the PRIMP text editing functions are designed solely to manipulate text items and their constituent lines. All other functions, e.g., copying entries from one book to another, may affect any named entity, i.e., items, decks, sets, packs, and books.

PRIMP treats textual data generically, i.e., it is not classified with respect to content within a book unless it is processed in a user-prescribed manner. If it is compiled, however, the resulting object deck is correlated to the source item as an aid to program configuration management. Furthermore, for selective retrieval, object decks are classified according to their source language.

Job Processing - Job processing under PRIMP is a sequence of correlated functions leading to, though not necessarily culminating in a program execution. Once job processing operations begin, a *jobstream* is initiated. Thereafter, all subsequent program submission operations modify or use this jobstream.

A jobstream may be viewed as a job in development. It may contain execution data in text form, relocatable object decks or an absolute object program. It has no specific relationship to any books except insofar as they contain common information. Whereas, a book is viewed as a place to synthesize a program, the jobstream is viewed as a "launching platform." A jobstream may be saved at any state of completeness, thus a user may have any number of jobs in preparation concurrently.

The final step of job processing is job submission for execution. If the job is to be interactive, then it is given immediate control of the user's conversation. If it is executed in batch mode, then the state of its progress is monitored for query by the user.

Time-Sharing Input/Output - The executing PROSE program may request immediate input from the terminal by the statement

READ TTY DATA

or may request data input after a sequence of iterative calculations, by virtue of the *solver control variable* INPUT, which may be used to schedule user queries:

INPUT = 0, *no terminal input (nominal)*

= n, *terminal input every nth iteration*

In the latter case, input occurs after the sequence of iterations is complete, and a summary of the iterative results have been printed. This feature is particularly useful in solving difficult optimization problems by adjusting independent variables to recover from poor initial values or to compensate for locally ineffective bounding.

The management of program output from the executing program is handled in three ways. Some or all of the output may be disposed to a line printer (PRINTER output), either at the computer site or at a remote-batch terminal. Portions of the output may also be disposed for interactive access at the user terminal. This output may either be saved for post-execution examination (SCROLL output) or it may be displayed immediately (TTY output). The last option, of course, is not permitted for deferred batch jobs for which any TTY output request is automatically converted to SCROLL output.

If SCROLL output is specified, it is organized as a single file consisting of one or more pages, each containing one or more lines. Each page of SCROLL output is identified by a title specified in the PROSE output statement

EJECT *'title'* PAGE

The first page contains a table of contents prepared by PRIMP. The user may interactively select output from the SCROLL file using a control program called PROMP.

PROCEDURAL CALCULUS PROGRAM

This section describes an example of a procedural calculus program in PROSE with explanation of the problem and its program. Instead of treating individual operations, this program illustrates the combination of calculus operations under language control to address a problem which could not be solved by single operations alone. Thus it stresses the essential nature of a procedural language to provide flexibility in the configuration of problem statements to fit built-in methods.

Problem-Structured Programming

The form of the calculus operations in PROSE impose a "top-down" hierarchial structuring of programs in which each level in the hierarchy is a classical mathematical problem stated in procedural terms. To illustrate the structuring, the following shorthand is introduced:

=> means "is structured as"

() denotes problem subordination (nesting)

, denotes problem coordination

An example of a problem structured program is the optimal design and control problem[2] to find the design parameter a by minimizing the functional

$$J(y,a) = \frac{1}{2}\int_0^1 (x^2 + y^2)dt + \frac{a^2}{2} \qquad (1)$$

where the state variable x and the control variable y are governed by the equations

$$\dot{x} = -ax + y \ , \qquad x(0) = c \qquad (2)$$

$$\dot{y} = x + ay \ , \qquad y(1) = 0 \ . \qquad (3)$$

By introducing the variable z we may convert the integral to an ODE,

$$\dot{z} = x^2 + y^2 \ , \qquad (4)$$

so that the objective function becomes

$$J(y,a) = \frac{1}{2}[\, z(0) + z(1)\,] + \frac{a^2}{2} \qquad (5)$$

At the first (top) level of this problem, we want to minimize the variable J. This can be done by the PROSE *problem* block:

```
PROBLEM .OPTDES
  READ DATA
  FIND A IN .GETJ  TO MINIMIZE J
END
```

which reads the input data for the problem, including an initial guess for the variable A.

At the second level of the problem we must compute J according to equation (5), but this requires the integration of equations (2), (3), and (4), satisfying boundary conditions on x and z at t=0 and on y at t=1. To solve this *two-point boundary value* problem we must find the unknown initial condition y(0) which satisfies the constraint y(1)=0. This is accomplished by treating the constraint as an implicit equation for y(1) as a function of y(0). Thus the model block .GETJ has the following structure:

```
MODEL .GETJ
  FIND YO IN .GETYO TO MATCH Y1
  J=Z/2+A**2/2
END
```

where the initial guess YO is also read in as data in the PROBLEM block.

At the third level of this problem, we want to solve the *initial value problem* for equations (2), (3), and (4) given the initial conditions x(0)=1, z(0)=c (input) and y(0) (solved for in the block .GETJ). The model block .GETYO has the following structure:

```
MODEL .GETYO
  Y=YO X=1 Z=ZO T=0
  INITIATE ISIS FOR .DIFF EQUATIONS
    XDOT/X,YDOT/Y,ZDOT/Z OF T STEP DT TO 1
  INTEGRATE .DIFF
  Y1=Y
END
```

The INITIATE statement defines the initial value problem to the solver ISIS by identifying the model block .DIFF (below) where the ODE's reside, and specifying the derivative variables XDOT,YDOT,ZDOT, their dependent variables X,Y,Z, to be computed, the independent variable T, the integration step size DT and the upper limit of integration (1). It also executes the model .DIFF at the initial point.

The INTEGRATE statement performs the integration from the initial point to the upper limit of integration. Then the constraint Y1 is set to the integrated value of Y. The model block .DIFF is the fourth level of the problem structure. It contains the ODEs to be integrated:

```
MODEL .DIFF
  XDOT=-A*X+Y
  YDOT=X+A*Y
  ZDOT=X**2+Y**2
END
```

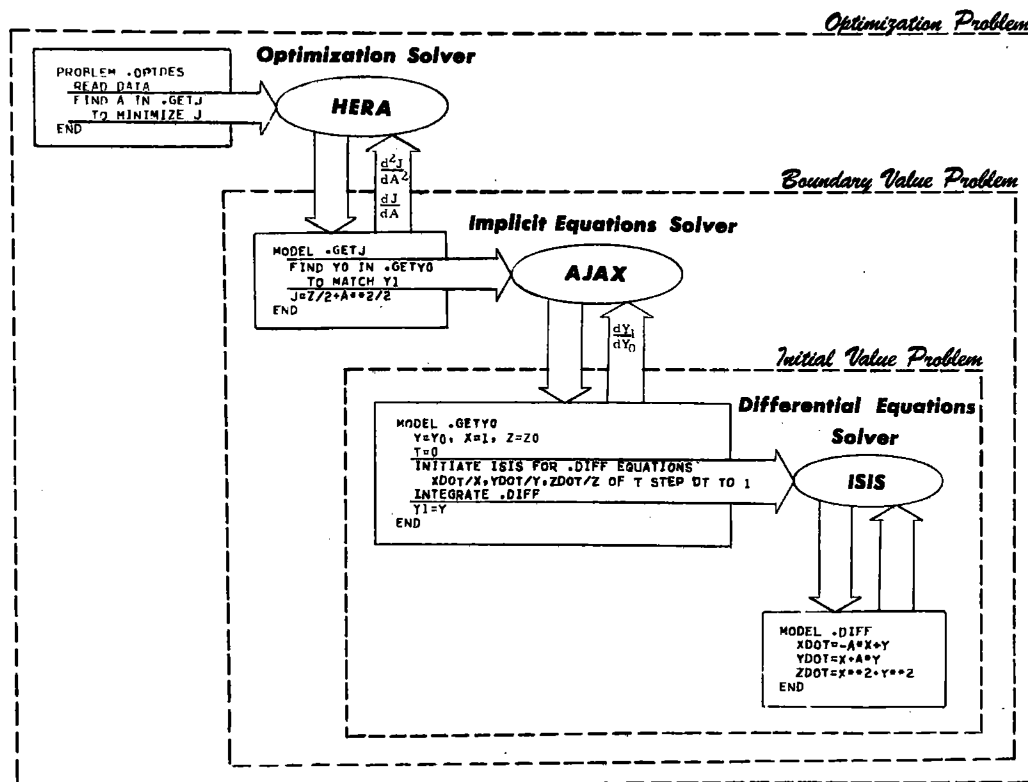
Figure 4 demonstrates the hierarchial structure of the PROSE program. The highest level problem in the hierarchy is a parameter minimization problem employing the solver HERA operating on the block .GETJ, which acts as the driver of the optimization model. As a by-product of the optimization model computations, the first and second derivatives of every dependent variable are evaluated with respect to the independent variable A. The derivatives of the objective variable J are used by HERA to search for the minimum of J.

The next level in the hierarchy is the solution of an implicit equation, represented by the constraint variable Y1, to determine the initial condition YO. As a by-product of the implicit model computations, the first derivatives of every dependent variable are evaluated with respect to YO. The derivative of Y1 is used by the solver AJAX, a first order Newton method to solve the implicit equation.

The third level of the hierarchy is the integration of the ODEs for a given set of initial conditions. This is performed by the solver ISIS, a Runge-Kutta-Gill procedure.

Finally the lowest level of the hierarchy consists of the integration model, the model block .DIFF, containing the algebraic representation of the ODEs.

Using the shorthand notation described above, the structure is as follows



.OPTDES => *Parameter minimization*(.GETJ, $\frac{d^2}{dA^2}$)

.GETJ => *Implicit Equation*(.GETYO, $\frac{d}{dYO}$, $\frac{d2}{dA^2}$)

.GETYO => *Integration*(.DIFF, $\frac{d}{dYO}$, $\frac{d2}{dA^2}$)

.DIFF => *Algebra*, $\frac{d}{dy0}$, $\frac{d^2}{dA^2}$

In words, the structure is described as follows:

"The problem .OPTDES is structured as parameter minimization of the model .GETJ with second derivatives evaluated with respect to the design parameter A. The model .GETJ is structured as the solution of an implicit equation in the model .GETYO with first derivatives evaluated with respect to the unknown initial condition YO and second derivatives evaluated with respect to A. The model .GETYO is structured as numerical integration of the model .DIFF with first derivatives evaluated with respect to YO and second derivatives evaluated with respect to A. The model .DIFF is structured as explicit algebra, with first derivatives evaluated with respect to YO and second derivatives evaluated with respect to A."

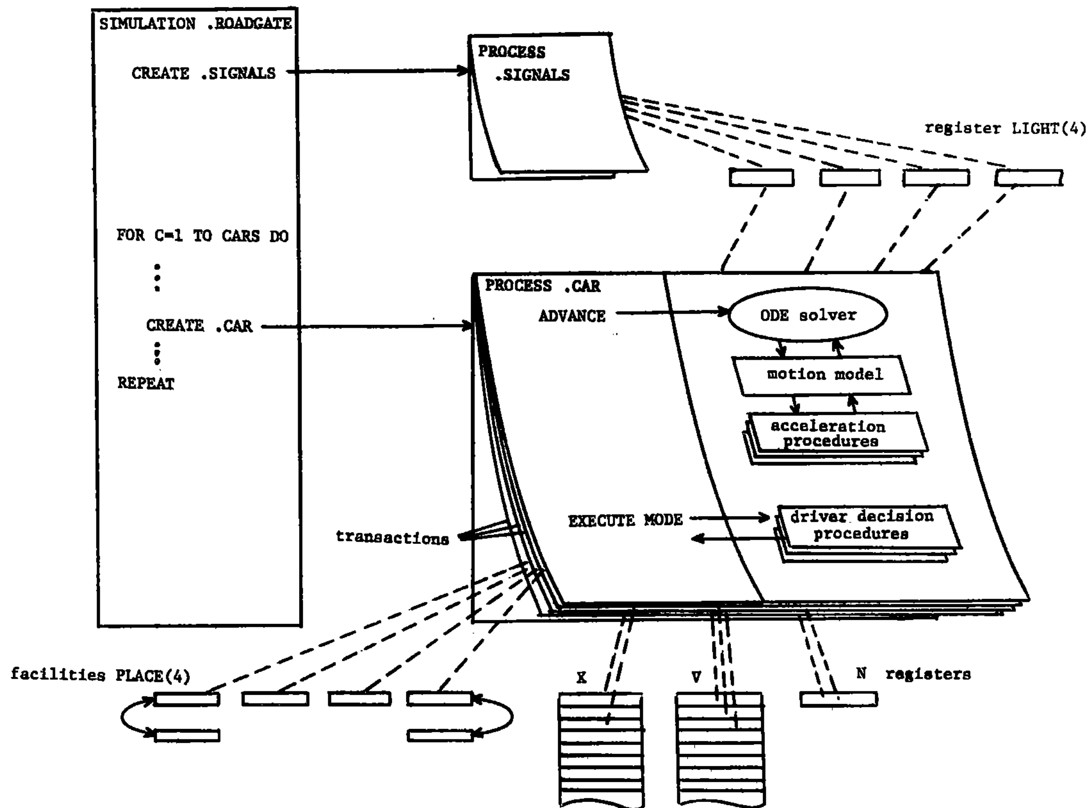
SIMULATION PROGRAM

This section describes an example of a PROSE program to simulate a discrete-continuous process. This is a representation of a model, developed by Fahrland[3] of a traffic intersection in which cars are approaching and passing a traffic signal. The model contains four traffic lanes, each approaching a crossing at right angles. For simplicity, no turns are allowed, so the model is merely a 4-way "roadgate."

The structure of the simulation program is illustrated in Figure 5. It consists of the simulation block .ROADGATE (Figure 6), two process blocks .SIGNALS (Figure 7) and .CAR (Figure 8), a model block .MOTION and six procedures (Figure 9), describing acceleration under different control modes (.HOLD, .SLOW, .STOP, .PACE, .SLOWSTOP, .PACESTOP) and six procedures (Figure 10), describing driver decisions in each control mode (.CRUISE, .GAINING, .GAINED, .REDLITE, .FOLLOW, .FOLRED).

The simulation block creates a single transaction of the process block .SIGNALS, which cycles from green to yellow to red to green, etc., for the duration of the simulation. The simulation block also creates multiple transactions of the process block .CAR.

As each transaction is created, the global variables of the simulation block are copied to the transaction, and become *exclusive* attributes of the transaction. The procedural calculus blocks, referenced in .CAR are "copies" that are also exclusive to each transaction. They communicate within the transaction by virtue of the global variables of the transaction.



Communication *between* transactions is through the registers LIGHT, V, X, and N. These are *system* attributes which may be altered by any transaction in the system. The register vectors V and X are used to record the velocity and position of each .CAR transaction so that they may be referenced by other .CAR transactions in order to compute relative velocity and position.

```
NO.                                  PROSE STATEMENTS
---   --------+--------+--------+--------+--------+--

 1   SIMULATION .ROADGATE
 2     READ DATA, ALLOT PRED(4)
 4     DEFINE FACILITY PLACE(4)
 5     DEFINE REGISTER LIGHT, V(CARS), X(CARS), N
 6     CREATE SIGNALS
 7     FOR C=1 TO CARS DO
 8       XCAR=-L
 9       VCAR=.NORMAL(VM.VS)
10       VDES=.NORMAL(VDM.VDS)
11       TR=.NORMAL(TRM,TRS)
12       ALPHA=.NORMAL(ALM.ALS)
13       AMAX=.NORMAL(AMM,AMS)
14       LANE=.RANDI(1.4)
15       IF C EQ 1 GO TO $GEN
17       IF LANE EQ LAST
18         THEN WAIT FOR .MAX(.NORMAL(LAGM.LAGS).DMAL/VL)
19         ELSE WAIT FOR .NORMAL(LAGM.LAGS)
20       CLOSE
21   $GEN LEADC=PRED(LANE)
22         CREATE .CAR
23         PRED(LANE)=C
24         VL=VCAR
25         LAST=LANE
26         N=N+1
27     REPEAT
28     WAIT WHILE N GT 0
29   END

      --------+--------+--------+--------+--------+--
```

Figure 6. Simulation Driver

Statements 2-6 of the simulation block read input data for the run and define the system facilities and registers. Statement 6 creates a transaction of .SIGNALS. Statements 7-27 comprise a loop to create .CAR transactions. Statements 8-14 define statistical parameters of each car which initialize the global variables of the .CAR transaction when it is created by statement 22. The decision at statements 17-20 causes a statistical time lag to simulate to random spacing of cars in the roadway. Statement 21 determines the index of the car in the current lane which precedes the one being created. This index (LEADC) is used in the transaction to reference the velocity and position of the preceding car.

```
NO.                          PROSE STATEMENTS
---   --------+--------+--------+--------+------

 1   PROCESS .SIGNALS
 2     REGISTER LIGHT
 3   $ON LIGHT(1)=GREEN, LIGHT(3)=GREEN
 5       LIGHT(2)=RED, LIGHT(4)=RED
 7       WAIT FOR TGREEN
 8       LIGHT(1)=YELLOW, LIGHT(3)=YELLOW
10       WAIT FOR TYELLOW
11       LIGHT(1)=RED, LIGHT(3)=RED
13       LIGHT(2)=GREEN, LIGHT(4)=GREEN
15       WAIT FOR TRED-TYELLOW
16       LIGHT(2)=YELLOW, LIGHT(3)=YELLOW
18       WAIT FOR TYELLOW
19       GO TO $ON
20   END

      --------+--------+--------+--------+------
```

Figure 7. Traffic Light Process

```
NO.                           PROSE STATEMENTS
---  --------+--------+--------+--------+--------+--------

 1   PROCESS .CAR
 2     REGISTERS V,X,N,LIGHT
 3   $GO IDENTIFY MODE AS .CRUISE
 4       IDENTIFY ACCEL AS .HOLD
 5       UNTIL XCAR GT L DO
 6         ADVANCE .MOTION EQUATIONS
             ACAR/VCAR, VCAR/XCAR STEP DT FOR DT
 7         X(C)=XCAR, V(C)=VCAR
 9         IF LEADC EQ 0
10           THEN XGAP=L, VGAP=0 (FIRST CAR IN LANE)
12                XPRED=0, VPRED=0
14           ELSE XPRED=X(LEADC), XGAP=XPRED-XCAR  (NOT FIRST)
16                VPRED=V(LEADC), VGAP=VPRED-VCAR
18         CLOSE
19         IF XGAP LE 0 THEN DISPLAY XCAR IN
             #COLLISION AT X=****.* IN LANE *#
21           STOP
22         CLOSE
23         GAP=VCAR**2/KDMAX
24         ATHRESH=ALPHA*VGAP/XGAP**2
25         EXECUTE MODE
26         IF VCAR EQ 0
27           THEN TAKE A PLACE(LANE) (AT THE TRAFFIC LIGHT)
28                WAIT WHILE LIGHT(LANE) NE GREEN
29                GO TO $GO
30         CLOSE
31       REPEAT
31       N=N-1
33   END

     --------+--------+--------+--------+--------+--------
```

Figure 8. Automobile Dynamics Process

Each time a car is generated, the register N is incremented. Later, as each car leaves the region of the intersection, its transaction cancels itself and decrements the register N. Statement 28 of the simulation block causes the simulation transaction to be suspendent until N is zero, at which time the END statement is executed, terminating the simulation run.

The process block .CAR (Figure 8) is the control program for simulation of automobile dynamics and driver decisions. For each transaction of .CAR, the driver control starts in the cruise mode in which a constant speed is maintained. This is specified by statements 3 and 4 which select .CRUISE as the mode procedure and .HOLD as the acceleration procedure for the next integration step of the second order differential equation

$$a_{car} = \frac{d^2x_{car}}{dt^2} = (a_{des} - k_{damp}\, v_{car})/m_{car}$$

where a_{car} is the actual acceleration, a_{des} is the desired acceleration, k_{damp} is a damping coefficient, v_{car} is the car velocity, and m_{car} is the mass of the car (assumed the same for all cases in this simulation). Whenever the car is in motion and its position lies in the interval $-L \le x_{car} \le L$, it is governed by the above equation.

Statement 6 of .CAR advances time by the amount DT by integration of the model block .MOTION (Figure 9). The desired acceleration ADES, in this equation is determined by executing one of the acceleration procedures, identified by the contents of the variable ACCEL.

Statements 7 and 8 of .CAR set the registers X and V to the integrated position and velocity so that these values may be referenced by other transactions of .CAR, as in the computation of the relative position and velocity of the current car and its predecessor (statements 14-17).

```
NO.                           PROSE STATEMENTS
---  --------+--------+--------+--------+--------+--------

 1   MODEL .MOTION
 2     EXECUTE ACCEL
 3     ACAR=(ADES-KDAMP*VCAR)/MASS

 4   PROCEDURE .HOLD (CONSTANT SPEED)
 5     ADES=-KACV*(VDES-VCAR)
 6   END

 7   PROCEDURE .SLOW (TO APPROACH CAR AHEAD)
 8     ADES=-KACX*(XPRED-XCAR)
 9   END

10   PROCEDURE .STOP (AT RED LIGHT AHEAD)
11     ADES=KACX*XCAR
12   END

13   PROCEDURE .PACE (FOLLOWING CAR AHEAD)
14     ADES=THRESH-ALPHA*(VPRED-VCAR)/(XPRED-XCAR)**2
15   END

16   PROCEDURE .SLOWSTOP (TO APPROACH CAR AND RED LIGHT)
17     EXECUTE .SLOW
18     ADSV=ADES
19     EXECUTE .STOP
20     ADES=.MIN(ADSV,ADES)
21   END

22   PROCEDURE .PACESTOP (TO FOLLOW CAR AND STOP AHEAD)
23     EXECUTE .PACE
24     ADSV=ADES
25     EXECUTE .STOP
26     ADES=.MIN(ADSV,ADES)
27   END

29   END (.MOTION)

     --------+--------+--------+--------+--------+--------
```

Figure 9. Motion Model and Acceleration Procedures

Statement 25 of .CAR simulates the driver decisions by executing one of the procedures (Figure 10) corresponding to the current mode of control. The action of these procedures is to change or not change the current control mode for the next integration step, according to the conditions of motion and conditions of the traffic light.

If the velocity of the car is zero statements 26-28 cause it to take a place in line at the traffic light, and wait until the light has turned green. Each time this happens, a statistic associated with the facility PLACE is accumulated by PROSE. At the end of the simulation run, a statistical summary of each facility is printed, describing the transaction loading for the run.

Whenever the position XCAR exceeds L, the car has moved out of the region of interest of the model. In this case, the conditions of the loop beginning at statement 5 of .CAR are satisfied. The register N is then decremented and the END statement is executed, cancelling the current transaction of .CAR.

```
NO.                              PROSE STATEMENTS
---  --------------------+---------+-----------+---------+----------+-----
  1  PROCEDURE .CRUISE [MODE DRIVER DECISIONS]
  2    REGISTER LIGHT
  3    IF XGAP LT GAP
  4      THEN IDENTIFY MODE AS .GAINING [ON CAR AHEAD]
  5           IDENTIFY ACCEL AS .SLOW [TO PREVENT COLLISION]
  6    OR IF LIGHT(LANE) NE GREEN AND .ABS(XCAR) LT GAP
  7      THEN IDENTIFY MODE AS .REDLITE [AHEAD]
  8           IDENTIFY ACCEL AS .STOP [AHEAD]
  9    CLOSE

 10  PROCEDURE .GAINING [MODE DRIVER DECISIONS]
 11    REGISTER LIGHT
 12    IF XGAP GT GAP
 13      THEN IDENTIFY MODE AS .CRUISE
 14           IDENTIFY ACCEL AS .HOLD [CONSTANT SPEED]
 15    OR IF ATHRESH GT THRESH
 16      THEN IDENTIFY MODE AS .FOLLOW [CAR AHEAD]
 17           IDENTIFY ACCEL AS .PACE  [THE CAR SPEED]
 18    OR IF LIGHT(LANE) NE GREEN AND .ABS(XCAR) LT GAP
 19      THEN IDENTIFY MODE AS .GAINRED [GAINING AND RED LIGHT]
 20           IDENTIFY ACCEL AS .SLOWSTOP
 21    CLOSE
 22  END [.GAINING]

 23  PROCEDURE .GAINRED [MODE DRIVER DECISIONS]
 24    REGISTER LIGHT.V
 25    IF XGAP GT GAP
 26      THEN IDENTIFY MODE AS .REDLITE [AHEAD]
 27           IDENTIFY ACCEL AS .STOP [AHEAD]
 28    OR IF XCAR GT 0 OR LIGHT(LANE) EQ GREEN
 29      THEN IDENTIFY MODE AS .GAINING [ON CAR AHEAD]
 30           IDENTIFY ACCEL AS .SLOW [TO PREVENT COLLISION]
 31    OR IF ATHRESH GT THRESH
 32      THEN IDENTIFY MODE AS .FOLRED [FOLLOWING AND RED LIGHT]
 33           IDENTIFY ACCEL AS .PACESTOP [PACE AND STOP AHEAD]
 34    OR IF VCAR LT 1 THEN VCAR=0. V(C)=0
 36    CLOSE
 37  END [.GAINRED]

 38  PROCEDURE .REDLITE [MODE DRIVER DECISIONS]
 39    REGISTER LIGHT.V
 40    IF XGAP LE GAP
 41      THEN IDENTIFY MODE AS .GAINRED [GAINING AND RED LIGHT]
 42           IDENTIFY ACCEL AS .SLOWSTOP [SLOW AND STOP AHEAD]
 43    OR IF XCAR GT 0 OR LIGHT(LANE) EQ GREEN
 44      THEN IDENTIFY MODE AS .CRUISE [CLEAR AHEAD]
 45           IDENTIFY ACCEL AS .HOLD [CONSTANT SPEED]
 46    OR IF VCAR LT 1 THEN VCAR=0. V(C)=0
 48    CLOSE
 49  END [.REDLITE]

 50  PROCEDURE .FOLLOW [MODE DRIVER DECISIONS]
 51    REGISTER LIGHT.V
 52    IF LIGHT(LANE) EQ GREEN AND .ABS(XCAR) LT GAP
 53      THEN IDENTIFY MODE AS .FOLRED [FOLLOWING AND RED LIGHT AHEAD]
 54           IDENTIFY ACCEL AS .PACESTOP [PACE AND STOP AHEAD]
 55    OR IF VCAR GT VDES
 56      THEN IDENTIFY MODE AS .CRUISE [CLEAR AHEAD]
 57           IDENTIFY ACCEL AS .HOLD [CONSTANT SPEED]
 58    OR IF VCAR LT 1 THEN VCAR=0. V(C)=0
 60    CLOSE
 61  END [.FOLLOW]

 62  PROCEDURE .FOLRED [MODE DRIVER DECISIONS]
 63    REGISTER LIGHT.V
 64    IF XCAR GT 0 OR LIGHT(LANE) EQ GREEN
 65      THEN IDENTIFY MODE AS .FOLLOW [CAR AHEAD]
 66           IDENTIFY ACCEL AS .PACE [THE CAR SPEED]
 67    OR IF VCAR GT VDES
 68      THEN IDENTIFY MODE AS .REDLITE [AHEAD]
 69           IDENTIFY ACCEL AS .STOP [AHEAD]
 70    OR IF VCAR LT 1 THEN VCAR=0. V(C)=0
 72    CLOSE
 73  END [.FOLRED]

 74  END [.CRUISE]

     ---------+---------+---------+----------+----------+---------+-----
```

Figure 10. Driver Decision Procedures

CONCLUSION

Higher level languages are the critical factor in the future of computing because they are the *media* required for expanded computer usage. When FORTRAN was introduced in 1957 computers were far more expensive per computation than the manpower required to program them. Consequently FORTRAN could not capitalize on interpretive mechanisms that so greatly simplify computing and foster *true* machine independence. The net result was that FORTRAN placed the burden of programming on the man rather than the machine. Today, the situation is reversed. Manpower cost is two to ten times more expensive than the associated machine cost, yet FORTRAN-level languages are still the mainstay of computing.

PROSE is a general purpose programming language which subsumes the scientific capabilities of FORTRAN, BASIC, PL/I, etc., but offers a new, higher level of programming. The calculus and simulation operations of PROSE are designed to submerge the tedious aspects of numerical analysis and algorithmic logic below the threshold of the user's awareness, into the realm of machine mechanics. The same process has occurred many times in the history of modern technology. Within twenty years after the discovery of logarithms, the slide rule submerged the concept of logarithms below its user's threshold of awareness. He could multiply, divide, exponentiate and take roots without being aware of what logarithms were or that the scales of the slide rule were logarithmic.

PROSE enables scientists and engineers to program at the level of model concepts rather than at the level of solution mechanics. The net effect of PROSE will be to raise the level of problem solving power and sophistication beyond that which is now feasible with algebra languages, but without raising the level of programming difficulty.

REFERENCES

1. *PROSE - A General Purpose Higher Level Language, Calculus Operations Manual.* Control Data Corporation CYBERNET Service Publication No. 84003200.

2. R. E. Bellman and R. E. Kalaba Quasilinearization on Nonlinear Boundary Value Problems. American Elsevier Publishing Co., New York, 1965.

3. D. A. Fahrland, *Combined Discrete Event Continuous Systems Simulation,* February 1970.

4. M. N. Robinson, *PROSE Simulation External Specification,* PROSE, INC., 8616 La Tijera Blvd., Los Angeles CA 90045, March 5, 1974.

5. R. G. Kenedy, M. N. Robinson and J. M. Thames,Jr., *PRIMP* External Specification, PROSE, Inc., 8616 La Tijera Blvd., Los Angeles, CA 90045, February 26, 1974.