SIMULATION 52:4, 141-149

ISSN 0037-5497/89 $2.00 +.10
Printed in the United States of America

Key words: computer-aided analysis, software engineering, interactive programs, simulation environments

Matrix environments for continuous system modeling and simulation

François E. Cellier, Ph.D.
Associate Professor
Dept. of Electr. and Comp. Engr.
University of Arizona
Tucson, Arizona 85721

C. Magnus Rimvall, Ph.D.
Corporate Research & Development
General Electric
Schenectady, New York 12301

FRANÇOIS E. CELLIER received his B. S. degree in Electrical Engineering from the Swiss Federal Institute of Technology (ETH) Zürich in 1972, his M.S. degree in Automatic Control in 1973, and his Ph.D. degree in Technical Sciences in 1979, all from the same university. Following his Ph.D., Dr. Cellier worked as a Lecturer at ETH Zürich. He joined the University of Arizona in 1984 as an Associate Professor. Dr. Cellier's main scientific interests concern modeling and simulation, computer-aided modeling, and computer-aided design. He has designed and implemented the GASPV simulation package, and he was the designer of the COSY simulation language, a modified version of which under the name SYSMOD has meanwhile become a standard by the British Ministry of Defense. Dr. Cellier has authored or co-authored more than forty technical publications, and he has edited two books. He served as a chairman of the National Organizing Committee (NOC) of the Simulation '75 conference, and as a chairman of the International Program Committee (IPC) of the Simulation '77 and Simulation '80 conferences, and he has also participated in several other NOC's and IPC's. He is associate editor of several simulation related journals, and he served as vice-chairman of two committees on standardization of simulation and modeling software. Memberships include SCS and IMACS.

C. MAGNUS RIMVALL received an MS degree in Engineering Physics from the Lund Institute of Technology, Sweden in 1982, and a Ph.D. degree in Technical Sciences from the Swiss Federal Institute of Technology (ETH), Zürich, Switzerland in 1986. He became an Assistant Professor of Electrical Engineering at ETH Zürich in 1987, before receiving a US "green card" and subsequently joining the Control Systems Laboratory of General Electric Corporate Research and Development in 1988.

Dr. Rimvall has been active in the field of Simulation since 1980. His research interests also include Computer-Aided Control Systems Design, Discrete-Event Systems, and other areas of Automatic Control. He is a member of SCS, IEEE, and ACM.

ABSTRACT

In any future standard for continuous system simulation languages, it is of primary importance that the data and program structures are made sufficiently general and flexible. In this paper, the use of matrix structures in modeling and simulation, and the impact of matrix environments on the design of simulation languages and their experimental frames will be discussed.

1. INTRODUCTION

Presently several efforts are undertaken to replace the CSSL'67[2] standard for continuous simulation languages with a more modern template. Since the publication of the CSSL report in 1967, several successful commercial packages adhering to the standard have emerged, for example ACSL[14] and CSSL-IV[16]. Other languages (e.g. DSL/VS[8] and DARE-P[23]) differ syntactically from CSSL'67, yet retain many of its concepts. This has given the standard a quite remarkable life span of almost twenty years in times of strong technological advancements, something which must be attributed to the open endedness of the standard. The products derived from CSSL'67 are not forced to be static, but have been enriched by assimilation of algorithms and concepts from other fields.

Despite this evolution of CSSL languages, there is now an imminent need for a new standard. Most of the enhancements made within the CSSL framework were of a functional rather than structural nature, as any fundamental changes in the program/model structures would require a drastic deviation from the standard. Consequently, the work of present standardization committees (e.g. in IMACS and SCS) concentrates on structural matters like the partition between model and experiment sections, the inclusion of discrete elements and the design of submodel facilities. In this contribution, we want to elaborate on some concepts which should be carefully considered in the development of any new simulation language standard: matrix data structures and ideas from interactive matrix/control environments.

2. MATRIX ENVIRONMENTS

The term *matrix environment* designates a class of programs giving the user easy and interactive access to matrix manipulation algorithms. The earliest of these programs, MATLAB[15], was based upon the two state of the art numerical packages LINPACK[5] and EISPACK[6,21]. Newer programs (e.g. CTRL-C[12,22] and PC-MATLAB[13]) are derivatives of MATLAB, but also include additional algorithms for control theory and systems analysis (these programs are also called control environments). Yet another generation of packages (e.g. MATRIX$_X$[9,24] and IMPACT[4,18,19,20]), retains the simple functionality of the previous programs, yet complements the matrix structure with additional data structures to describe polynomial matrices, and linear and nonlinear systems.

Common to all these packages is their ease of use. To illustrate this, let us evaluate the stability of a linear system. For this purpose, we can enter the system matrix, and compute its eigenvalues:

```
[> a = [ -1, -2,  0
         1,  0, -2
        -1, -1,  2 ] ;
[> EIG(a)
ANSWER =
  -0.6348 + 0.6916i
  -0.6348 - 0.6916i
   2.2695 + 0.0000i
```

We notice that one eigenvalue has a positive real part, indicating that the system is unstable.

With the same ease, the matrix environment allows us to interactively define small algorithms. For example, it is possible to implement simple integration algorithms for small simulations. The following example illustrates the use of MATLAB[16] to simulate a small linear system, the Cedar Bog Lake[17], using forward Euler integration:

```
// Simulation of Cedar Bog Lake
// ----------------------------------------------
// 1 // Define system order and final time
n = 5; tmax = 10;
//   // Define state matrix
a = [ -4.03,   0.,     0.,     0., 0.
       0.48, -17.87,   0.,     0., 0.
       0.,     4.85,   4.65,   0., 0.
       2.55,   6.12,   1.95,   0., 0.
       1.,     6.9,    2.7,    0., 0. ]
//   // Define input vector
b = [ 1.;      0.;     0.;     0.; 0. ]
//   // Define initial conditions
x0 = [ 0.83; 0.003;   0.0001; 0;  0. ]
// 2 // Eigenvalue computation
e = EIG(a),
//   // Ignore unstable modes
FOR i=1 : n, ...
  IF REAL(e(i)) >= 0, e(i) = -0.01; END, ...
END
//   // Determine critical step size
dtk = NORM( 2*REAL(e) ./ ( e .* CONJ(e) ),'INF')
// 3 // Forward Euler, using small steps (dt=dtk/5)
dt = dtk/5
//   // Initial computations
x = x0; xst = [ 0 ; x0 ] ;
f = EYE(n) + dt*a; g = dt*b;
nn = ROUND(tmax/dt); kk = ROUND(nn/50);
k = 1; IF kk = 0, kk = 1; END
// 4 // Simulation
FOR i=1 : nn, ...
  t = dt*i; u = 95.9*(1 + 0.635*SIN(t-dt)); ...
  x = f*x + g*u; EXEC('store.mtl');
END
// 5 // Printout
EXEC('printout.mtl')
```

The code is fairly selfexplanatory. Cedar Bog Lake is a *5th* order linear continuous time single input/single output system of the type:

$$dx/dt = \mathbf{A}*\mathbf{x} + \mathbf{b}*u$$

that we wish to integrate from time *0* to time *10* using forward Euler integration. For this purpose, we need to determine the maximum step size (*dtk*) which will keep the integration numerically stable. The appropriate algorithm is illustrated in *Figure 1*.

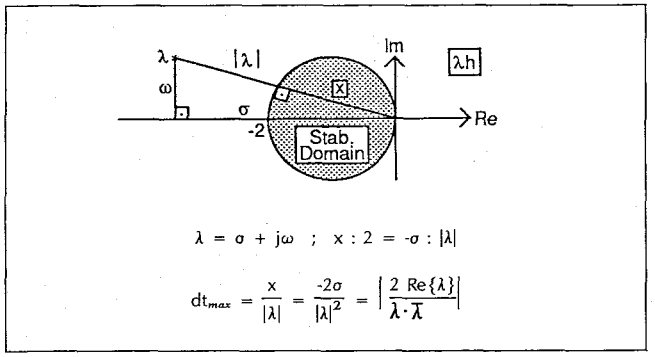


Figure 1. Evaluation of critical step size for forward Euler integration.

We decided to use a step size that was five times smaller than the maximum allowed step size. We then replaced the continuous time system plus the forward Euler integration scheme:

$$dx/dt = \mathbf{A}*\mathbf{x} + \mathbf{b}*u$$

$$x_{k+1} = x_k + \Delta t*dx_k/dt$$

by the equivalent discrete time system:

$$\mathbf{x}_{k+1} = [\ I^{[n]} + \Delta t*\mathbf{A}\ ]*\mathbf{x}_k + [\ \Delta t*\mathbf{b}\ ]*u_k$$

which denotes a set of difference equations of the form:

$$\mathbf{x}_{k+1} = \mathbf{F}*\mathbf{x}_k + \mathbf{g}*u_k$$

These equations are then iteratively solved over *nn* steps, and the results are stored once every *kk* steps for later printout.

Although this example is not very interesting from a simulation point of view, it shows the flexibility of the MATLAB command language and the relative ease with which new algorithms can be developed and tested. Note that only the first of the five sections is needed in control environments with predefined integration algorithms (such as CTRL-C or IMPACT). The same problem could be coded in CTRL-C[12,22] as follows:

```
// Simulation of Cedar Bog Lake
// ----------------------------------------------
// 1 // Define state matrix
a = [ -4.03,   0.,     0.,     0., 0.
       0.48, -17.87,   0.,     0., 0.
       0.,     4.85,   4.65,   0., 0.
       2.55,   6.12,   1.95,   0., 0.
       1.,     6.9,    2.7,    0., 0. ]
//   // Define input vector
b = [ 1.;      0.;     0.;     0.; 0. ]
//   // Define output matrix
```

```
c = EYE(5)
//   // Define direct coupling vector
d = ZROW(5,1)
// // Define initial conditions
x0 = [ 0.83;   0.003; 0.0001; 0;  0. ]
//   // Define a time base
t = 0 : 0.2 : 10;
//   // Compute the input signal over the time base
u = 95.9*(ONES(t) + 0.635*SIN(t));

// 2 // Simulation using Adams integration
SIMU('IC',x0)
y = SIMU(a,b,c,d,u,t)

// 3 // Representation of output
PLOT(y')
```

Both of the preceding small examples illustrate the close conceptual and algorithmic connection between matrix environments and simulation languages. We should therefore ask ourselves the following questions:

- Several simulation languages (e.g. ACSL[14]) allow the user to compute the eigenvalues/eigenvectors of a system by automatically linearizing the system from the Jacobian matrix around any working point and returning the result in a matrix form. Are there any other uses for matrices in general simulation languages?
- In matrix environments, it is possible to perform rudimentary simulations, and thereby utilize the interactive environment as a flexible experimental frame. Would a similar interactive environment also be useable in simulation languages?

3. MATRIX CONSTRUCTIONS AS A MODELING INSTRUMENT

Most simulation languages allow for the declaration and use of matrices within the model description. However, although these matrices can be used at the discrepancy of the user, few languages utilize them to enhance their modeling capabilities. This is very unfortunate, as matrices can be used to simplify the solution to several modeling problems.

Consider the table-lookup function. Although this is a good example of a multi dimensional data structure, practically all simulation languages force the user to enter the function values as a linear list. A more readable approach, which also preempts hard-to-find dimensional errors in the specification, would be to use a somewhat modified matrix structure (internally represented as a matrix):

```
FUNCA = [  0.0  |  0.0
           0.5  |  0.1
           1.0  |  0.33
           1.5  |  0.67
           2.0  |  0.9
           2.5  |  1.0 ] ;
```

This could easily be extended to two dimensions:

```
FUNC__2 = [        |  0.0,  0.5,  1.0
'comment   ---------+-------------------------
             2.0   |  0.0,  0.1,  0.9
             2.5   |  0.1,  0.5,  1.3
             3.0   |  0.6,  1.2,  2.1 ] ;
```

Although no longer displayable in one plane, three dimensional tables could use a similar notation by concatenating two dimensional tables with the value of the third dimension in the free upper left hand field.

Using the philosophy of IMPACT, we could alternatively define *domains* for the independent variables, and a vector/matrix/tensor for the dependent variable, which are then packed into a new data structure of type *table*. Thus in IMPACT, we could code the above example of a two dimensional table as follows:

```
x__dom   =  LINDOM(2,3,0.5);
y__dom   =  LINDOM(0,1,0.5);
z__value =  [ 0.0,  0.1,  0.9
              0.1,  0.5,  1.3
              0.6,  1.2,  2.1 ] ;
func__2  =  TABLE(z__value,x__dom,y__dom);
```

This allows us to redefine the element operations on tables:

```
z = func__2(x,y)
```

here does not denote the matrix element with indices x and y as this would be the case for regular matrices, but rather the value of the two-dimensional function *func__2* interpolated at point (x,y).

Scientists working in systems theory and automatic control are used to represent their linear systems in the time domain by a set of matrices:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -2 & -3 & -4 \end{bmatrix} \quad b = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \end{bmatrix} \quad d = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

which are used to describe the system

$$\begin{aligned} dx_1/dt &= x_2 \\ dx_2/dt &= x_3 \\ dx_3/dt &= -2x_1 - 3x_2 - 4x_3 + u \\ y_1 &= x_1 \\ y_2 &= 2x_2 + u \end{aligned}$$

where *u* is some input signal, and *y* is the resulting output vector. On the other hand, simulationists traditionally describe their, sometimes identical, systems through differential equations and integrational blocks. Using the matrix structures from control theory as an extension to the traditional simulation elements, we can enhance the readability of the simulation program:

```
a = [  0,   1,  0
       0,   0,  1
      -2,  -3, -4 ] ;
b = [  0;   0;  1 ] ;
c = [  1,   0,  0
       0,   2,  0 ] ;
d = [  0;   1   ] ;
u = ...
x = INTEG(a*x + b*u,x0);
y = c*x + d*u
```

Matrix constructions can also be used to simplify more complex modeling structures. Standard continuous simulation languages are sometimes (ab)used to simulate systems described by partial differential equations. A simple, one dimensional diffusion problem can thereby be described by:

```
DIMENSION u(50), dudt(50), uic(50)
u = INTVC(dudt,uic)
PROCEDURAL (dudt=u)
    dudt(1) = c*(uleft  - 2.0*u(1)   + u(2))
    DO 5 i=2,49
    dudt(i) = c*(u(i-1) - 2.0*u(i)   + u(i+1))
5.. CONTINUE
    dudt(50)= c*(u(49) - 2.0*u(50) + uright)
END
```

However, using matrix manipulations and the predefined, automatically scaled structures *ONES* (matrix with all ones), *EYE* (unity matrix), *DIAG(v,k)* (diagonal matrix with all elements of the vector *v* on the *Kth* diagonal), and *E(i)* (the *ith* unity vector), the whole diffusion model can be reduced to:

```
INITIAL
    n = 50
    d = ONES(n-1,1)
    a = DIAG(d,-1) - 2*EYE(n) + DIAG(d,1)
    ...
DERIVATIVE
    u = INTEG(dudt,uic)
    dudt = c*( a*u + E(1)*uleft + E(n)*uright )
    ...
```

In addition to describing systems in the time domain by matrices, control engineers also often represent systems in the frequency domain by transfer function matrices (matrices containing rational functions). Correspondingly, most simulation languages provide block functions for elements described in the frequency domain. For example, the transfer function

$$G(s) = \frac{1.0}{1 + b*s}$$

is generally available as a function REALPL:

```
y = REALPL(b,x,ic)
```

However, some control environments allow the construction of general transfer functions. For example, to define the transfer function

$$G(s) = \frac{2s + 3}{s^3 + 9s^2 + 5s + 9}$$

IMPACT[4, 18, 19, 20] allows the user to enter

```
g = [3^2]/[9^5^9^1]
```

or a little better readable:

```
s = [^1];
p = 2*s + 3;
q = s**3 + 9*s**2 + 5*s + 9;
g = p/q
```

which is even only a special case, since IMPACT allows to define transfer function matrices for the description of multivariable systems. This notation could easily and profitably be incorporated into simulation languages as well.

We conclude that:

Matrix Constructions Can Considerably Enhance the Readability, Security and/or Flexibility of a Simulation Modeling Language.

According to Korn and Wait[11], simulation is experimentation with models. Consequently, each simulation program should consist of two separate parts:

- A coded description of the model which we call the model representation inside the simulation program;
- A coded description of the experiment to be performed on the model which we call the experiment representation inside the simulation program.

So far, we have only looked into the model representation. In the next section of this paper, it will be analyzed how matrix representations can aid the experiment description as well.

4. MATRIX CONSTRUCTIONS FOR EXPERIMENT DESCRIPTIONS

Early simulation packages (e.g. CSMP[7]) were designed for batch operation only. A "simulation job" normally consisted of a deck of punched cards describing not only the model to be simulated (in the form of differential and algebraic equations), but also the experiment to be performed (the run length of the simulation, the variables to be printed/plotted on output, etc.). When performing complex experiments on a given model, this environment is not flexible enough; the whole simulation (including translating and linking) had to be repeated after every, however small, change (for example for rescaling a plot or for changing a single parameter value).

In more modern packages, a "run time monitor" allows the user to interactively invoke simulations, change parameter values, create numerical or graphical output, and so on. However, the run time commands available are normally too primitive (exception: DESCTOP[10]); no structural language elements are available (except for rudimentary macros), and therefore, no procedural programming can be done on this interactive level.

Most simulation languages available as of today offer fairly elaborate facilities for model descriptions, but rudimentary techniques for experiment descriptions. The standard simulation experiment is as follows: starting with a complete and consistent set of initial conditions, the change of the various variables of the model (state variables) over time is recorded. This experiment is often referred to as determining the *trajectory behavior* of a model. Indeed, when the term "simulation", as this is often done, is used to denote a solution technique rather than the ensemble of all modeling related activities, "simulation" can simply be equated to the determination of trajectory behavior. Most currently available simulation programs offer little beside efficient means to compute trajectory behavior.

Contrasting this rather limited interactive interface, the command interfaces of the matrix environments are considerably more flexible. As we already have seen in the introductory examples, small programs (algorithms) can be interactively defined and executed. Tests and decisions can be made reading the commands either from the terminal (interactive use), from a file (batch), or a combination of both (interactive execution using predefined macros/procedures).

Unfortunately, few practical problems present themselves as pure simulation problems. For example, it happens often that the set of starting values is not specified at one point in time. Such problems are commonly referred to as *boundary value problems* as opposed to the *initial value problems* discussed previously. Boundary value problems are not naturally simulation problems in a puristic sense (although they can be converted to initial value problems by a technique called *invariant embedding*). A more commonly used technique for this type of problems, however, is the so-called *shooting technique*. It works as follows:

(1) Assume a set of initial values.

(2) Perform a simulation.

(3) Compute a *performance index*, e.g. as a weighted sum of the squares of the differences between the expected boundary values and the actually computed boundary values.

(4) If the value of the performance index is sufficiently small, terminate the experiment, otherwise interpret the unknown initial conditions as parameters, and solve a *nonlinear programming problem*, looping through (2)...(4) while modifying the parameter vector in order to minimize the performance index.

As can be seen, this "experiment" contains a multitude of individual simulation runs.

A very similar type of simulation experiment results from unknown model parameters which are to be estimated in order to obtain an optimal match between the outcome of a simulation experiment and a set of measurements taken from a real plant to be modeled (parameter fitting experiment). For this purpose, we could try to minimize the following performance index:

$$PI = \int_0^{t_f} \{\sum_{i=1}^{n} (x_i - \xi_i)^2\}\, dt$$

where the x_i are the simulation trajectories, and the ξ_i are the measurement trajectories, assuming that n different trajectories are available from measurements to estimate a number of parameters, say n_p parameters.

The modeling portion of this problem can be coded easily by adding the derivative of the performance index to the set of differential equations:

```
pi = INTEG((x - xcap)' * (x - xcap), 0)
```

where the apostrophe (') has been used to denote the transposition of the vector $x\text{-}x_{cap}$. Of course, this requires that the measurement trajectory vector x_{cap} be imported into the simulation, a feature not commonly provided by current simulation languages.

The experiment portion of the problem requires the performance index *PI* to be minimized over the parameter vector p, assigning an initial value of *1* to all n_p parameters. This could be coded as follows:

```
FUNCTION perfindex(par);
BEGIN
    p = par; SIMULATE; RETURN pi;
END perfindex;

start  = ONES(np,1);
result = OPTIMIZE(IC=start,
         FUNCTION=perfindex, ERROR=1.0E-3)
```

The *OPTIMIZE* function represents a precoded nonlinear programming package containing a variety of different optimization algorithms. For each function evaluation, *OPTIMIZE* calls the user-coded function *PERFINDEX* which sets the simulation parameter vector p equal to the optimization parameter vector *par*, then performs one simulation (execution of the simulation model), and returns the resulting performance index *pi* to the optimization algorithm for further evaluation. *result* contains the optimized parameter vector. Again, very few simulation languages contain such a facility (exception: DARE-INTERACTIVE[3]). This is partly due to the fact that the CSSL'67 standard did not foresee the need to execute simulations as subprograms, a feature that is e.g. provided in DARE-P[23], DARE-INTERACTIVE[3], and DESCTOP[10].

Another way to solve this problem might be to leave the computation of the performance index entirely out of the model description, and replace the function *PERFINDEX* by the following code:

```
FUNCTION perfindex(par);
BEGIN
    p = par;
    SIMULATE;
    pi = NORM(x-xcap);
    RETURN pi;
END error;
```

To elaborate on yet another example, assume that an electrical network is to be simulated. The electrical components of the network have tolerance values associated with them. It is to be determined how the behavior of the network changes as a function of these component tolerances. An algorithm for this problem could be the following:

(1) Consider those components that have tolerances associated with them to be the parameters of the model. Set all parameters to their minimal values.

(2) Perform a simulation.

(3) Repeat (2) by toggling all parameters between their minimal and maximal values until all "worst case" combinations are exhausted (assuming that the worst cases occur at the ends of the tolerance intervals, an assumption that is not necessarily justified in arbitrary nonlinear problems). Store the results from all these simulations in a data base.

(4) Extract the data from that data base, and compute an envelope of all possible trajectory behaviors for the purpose of a graphical display.

As in the previous example, the experiment to be performed consists of many different individual simulation runs. In this case, there are exactly 2^n runs to be performed where n denotes the number of parameters.

Let us assume we have a single output system with *5* parameters. We shall write the parameters with their associated tolerances into a parameter tolerance matrix:

```
ptol = [ 8.,   12.
         2.5,  3.5
         1.,   1.5
         7.2,  7.6
         9.5,  15.  ];
```

where each row contains the minimum and maximum values of one parameter. The proposed experiment can be coded as follows:

```
n = 5;
FOR i=1:2**n,
    p = ptol(1, : );
    SIMULATE;
    ystr(i) = y;
    indx = n;
    ii = ROUND(i/2);
    WHILE i - 2*ii = 0,
        indx = indx - 1;
        i = ii;
        ii = ROUND(i/2);
    END,
    ptol(indx,[1,2]) = ptol(indx,[2,1]);
END
ymax = MAX(ystr);
ymin = MIN(ystr);
PLOT([ymax ; ymin ])
```

We start by extracting the first column of *ptol* into the parameter vector *p*. We then simulate our system. The resulting output *y* is a variable of type *trajectory*. We store the output in *ystr*, a variable of type *trajectory vector*. We then toggle the rows of *ptol* one at a time until all combinations of parameter values have ended up once in the first column of *ptol*. We meanwhile have obtained a trajectory vector *ystr* of length 32.

The *MAX* function is a very powerful operator. When applied to a regular *nxm* matrix, it returns a row vector of length *m* containing the largest (i.e. most positive) elements of each column of the matrix. When applied to a vector, it returns the largest element of that vector. Thus, the largest element of a matrix can be computed by:

```
largest_element = MAX(MAX(matrix))
```

When applied to a *trajectory matrix*, the function returns a row *trajectory vector* that contains the upper envelopes of all trajectories in one trajectory column. When applied to a *trajectory vector*, it returns a single trajectory containing the upper envelope of all trajectories in the trajectory vector. Thus, the upper envelope of all trajectories in a trajectory matrix can be computed by:

```
trajec_envelope = MAX(MAX(trajec_matrix))
```

When applied to a single trajectory, *MAX* returns a scalar, namely the largest (most positive) value of that trajectory, thus:

```
high = MAX(MAX(MAX(trajec_matrix)))
```

can be used for scaling purposes. The *MIN* function computes smallest values and lower envelopes correspondingly.

Further cases where procedural experimental frames might be used include statistical replication analysis, interactive specification of input signals, and calculation and manipulation of steady state and eigenvalue/eigenvector information. Also, it should be possible to directly connect the simulation system to a control environment.

We summarize:

An Interactive Run Time Monitor Supporting Procedural Language Constructs and Matrix Data Structures not only Simplifies the Control over the Model, but also Opens Up a Range of Interesting Experimental Possibilities.

5. OVERLOADED OPERATIONS AS MODELING TOOL

In this section, we will discuss how the "overloading" of operators can make modeling environments simpler, and yet more flexible. The term "overloading" derives from the computer language Ada[1], where it describes the facility to code several *different* functions/procedures using the same name. The individual function/operation is characterized by the types of its input parameters in the function call. This, for example, enables the user to define a data structure "matrix" and a function "*" operating on two matrices that contains a matrix multiplication algorithm. Thereafter, the Ada operator "*" is overloaded to work on matrices as well as on scalars. The aforementioned *MAX* function is an excellent example of a heavily overloaded function.

As present simulation languages do not support overloading, a user wishing to perform e.g. a matrix multiplication is left with two choices: to program a multiplication algorithm within the modeling environment or to call an external numerical routine for the operation. In CSSL-IV[16], the only CSSL simulation language including matrix manipulation algorithms, this call would take the form:

```
"DECLARATION OF MATRICES"
    DIMENSION A(4,4), B(4,4), RES(4,4)
"PERFORM MATRIX MULTIPLICATION"
    RES = MMULTM(A,B,4,4)
```

However, this complicated construction is *completely unnecessary*. Since the model description is translated (normally into FORTRAN), simpler constructions could be used. The translator *knows* the dimensions of *A, B,* and *RES,* and thereby the parameters of the call to MMULTM, making the dimension declarations of the call redundant. It would be perfectly feasible to leave it up to the translator to generate the required declarative statements and the function call out of the user statement:

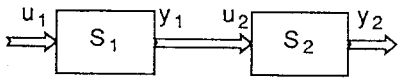
```
RES = A*B
```

This overloading of the multiplication operator also preempts user errors in the parameter specification to the (somewhat clumsy) function call.

Several current simulation languages provide an operator (often called INTEG) for scalar integration, and another operator (INTVC) for vector operations. With the same argument as for the matrices, one of these operators is redundant (see the examples using INTEG in the previous sections). There exists no good reason why the INTEG operator cannot be overloaded to mean two different things depending on the data types of the input parameters. If the underlying run time language does not support the overloading concept, and thereby really must make this distinction, it would be much better to let the translator generate appropriate code for each case.

We know that, given two systems in transfer function form, the cascading of these systems is described by a multiplication of the transfer function matrices in reverse order. Generalizing this concept, a reverse order multiplication of two systems (using now an arbitrary system representation) can be *defined* to mean a cascading of the two systems, while the addition of two systems can be defined to denote a parallel connection of the two systems. These rules can then be used to describe subsystem interconnections whenever these dynamic subsystems are described in a modular form. For example, let us assume that S_1 and S_2 are two modules with several inputs and outputs,

and that the outputs y_1 of the subsystem S_1 are connected to the inputs u_2 of the subsystem S_2. Accordingly, the interconnection:



can be described by the following program:

```
SYSTEM s1 (y1 = u1)
...
END s1
SYSTEM s2 (y2 = u2)
...
END s2
s = s2 * s1
```

However, the "*" operator can be even further overloaded. We will define the statement:

```
y2 = s2*s1*u1
```

to denote a *simulation* of the combined system *(s2*s1)* using *u1* as input trajectory vector. Thereby, the *u1* trajectories are imported into the simulation where the expression *u1(3)(t)* denotes the interpolation of the trajectory $u1_3$ over the domain value t (the current time). As can be seen, single trajectories and one dimensional tables are almost the same thing, except that the former generalize into trajectory vectors and matrices, while the latter generalize into two and three dimensional tables. The resulting output trajectory (vector) *y2* is finally sampled over the same domain as *u1*.

In contrast, the statement:

```
y2 = s2*(s1*u1)
```

involves two separate simulation runs. First, the subsystem *s1* is simulated using u1 as an input trajectory vector. The result of this operation is an auxillary trajectory vector sampled over the same domain. This auxillary trajectory vector is then used as the input to subsystem *s2* which is simulated next. The result of that simulation is finally the trajectory vector *y2* which is once more sampled over the same domain.

Numerically, the two results will of course be different, but conceptually, the *associative law of multiplication* holds for this extended overloading operation.

Notice that the *domain* information, which is an intrinsic part of the trajectory data structure, plays here the role of the *time base*, i.e. the "communication interval" of the classical CSSL language. Notice furthermore that there is no reason why the domain values would have to be equidistantly spaced which is yet another advantage in comparison with the classical CSSL specification.

The notation, which was presented in this section of the paper, has been borrowed from the IMPACT[4, 18, 19, 20] language.

In classical CSSL type languages, *simulation* is being viewed as an activity in its own right. The *simulation program* reads an input "card deck", and produces results in the form of a listing file and/or plot file. Simulations are main programs that cannot form a part of a larger and encompassing task.

Some CSSL like languages, in particular DARE[3, 23] and DESCTOP[10] extend this concept. They allow simulations to be executed as *subprograms* called by an encompassing task which is called the *logic block* (i.e. the experiment description) of the program. Results are stored in a (very primitive) data base for graphical postprocessing. However, previous simulation results cannot be reloaded into another simulation run, e.g. as input trajectories (exception: DARE INTERACTIVE[3]).

Control environments recognize the fact that flexibility is obtained by treating *simulation* as a *function* that maps an input data structure of type *trajectory vector* into an output data structure of the same type. This concept has been demonstrated in this paper in the CTRL C[12, 22] function *SIMU* used in the second version of the Cedar Bog Lake simulation.

IMPACT[4, 18, 19, 20] goes yet another step beyond by interpreting *simulation* as a *binary operator* that maps one data structure of type system and one data structure of type *trajectory vector* into a third data structure of type trajectory vector. This notation is completely general, and indeed, IMPACT contains mechanisms to describe nonlinear systems as well as linear systems. This facility will be demonstrated in the next section of this paper.

We conclude:

In a Simulation Language with Predefined Matrix Data Structures and Modular Program Elements, Standard Operators Can Be Overloaded. This Enhances the Readability and Simplifies the Language by Making Duplicate Operators for each Structural Element Redundant.

6. A COMPLETE EXAMPLE

When solving a finite time Riccati differential equation, one common approach is to integrate the Riccati equation backward in time from final time t_f to initial time t_0 , because the "initial condition" of the Riccati equation is stated as $\boldsymbol{K}(t=t_f) = 0$, and because the Riccati equation is numerically stable in backward direction only. The solution $\boldsymbol{K}(t)$ is stored away during this simulation, and then reused (in reversed order) during the subsequent forward integration of the state equations with given $\boldsymbol{x}(t=t_0)$. Some of the available CSSL's allow to solve this problem (mostly in a very indirect manner), others simply cannot be used at all to tackle this problem.

How can one handle this problem in CTRL-C[12, 22]? The first simulation is non linear (and autonomous), the second is linear (and input dependent) but time varying; thus, we cannot use the *SIMU* function in either case. CTRL-C provides for a second means of simulation though. In newer releases of CTRL-C, an interface to the well known simulation language ACSL[14] was introduced. This interface allows to make use of the modeling and simulation power of a full fledged simulation language, while one is still able to control the experiment from within the more flexible control environment. Several control environments follow this path, and it might indeed be a good answer to our problem if the two languages that are combined in such a manner are sufficiently compatible with each other, and if the interface between them is not too slow. Unfortunately, this is currently not yet the case with any of the control environments that use this route.

Let us illustrate the problems. We start by writing an ACSL program that implements the matrix Riccati differential equation:

$$d\mathbf{K}/dt = -\mathbf{Q} + \mathbf{K}*\mathbf{B}*\mathbf{R}^{-1}*\mathbf{B}'*\mathbf{K} - \mathbf{K}*\mathbf{A} - \mathbf{A}'*\mathbf{K} \; ; \; \mathbf{K}(t_f) = 0$$

Since ACSL does not provide for a powerful matrix environment, we have to separate this compact matrix differential equation into its component equations. (ACSL does provide for a vector-integration function, and matrix operations such as multiplication and addition could be (user-)coded by use of ACSL's MACRO-language. However, this is a slow, and inconvenient replacement for the matrix manipulation power offered in languages such as CTRL-C.) Furthermore, since ACSL does not handle the case: $t_f < t_0$, we must substitute t by:

$$t^* = t_f - t_0 - t$$

and integrate the substituted Riccati equation:

$$d\mathbf{K}/dt^* = \mathbf{Q} - \mathbf{K}^*\mathbf{B}^*\mathbf{R}^{-1}{}^*\mathbf{B}'^*\mathbf{K} + \mathbf{K}^*\mathbf{A} + \mathbf{A}'^*\mathbf{K} \quad ; \quad \mathbf{K}(0) = 0$$

forward in time from $t^*=0$ to $t^*=t_f-t_0$. Through the new interface (A2CLIST), we export the resulting $K_{ij}(t^*)$ back into CTRL-C where they take the form of ordinary CTRL-C vectors. Also in CTRL-C, we have to manipulate the components of $\mathbf{K}(t)$ individually, as $\mathbf{K}(t)$ is a trajectory matrix, that is: a three dimensional structure. However, CTRL-C handles only one dimensional structures (vectors), and two dimensional structures (matrices), but not three dimensional structures (tensors). Back substitution can be achieved conveniently in CTRL-C by simply reversing the order of the components of each of the vectors as follows:

```
[n,m] = SIZE(kij)
nm    = n*m
kij   = kij(nm:-1:1)
```

Now, we can set up the second simulation:

$$d\mathbf{x}/dt = (\mathbf{A} - \mathbf{K}(t)^*\mathbf{B})^*\mathbf{x} \quad ; \quad \mathbf{x}(t_0) = \mathbf{x}_0$$

What we would like to do is to ship the reversed $K_{ij}(t)$ back through the interface (C2ALIST) into ACSL, and use them as driving functions for the simulation. Unfortunately, ACSL is not (yet!) powerful enough to allow us to do so. Contrary to the much older CSMP-III system, ACSL does not offer a dynamic table load function (CALL TVLOAD). Thus, once the $K_{ij}(t)$ functions have been sent back through the interface into ACSL, they are no longer trajectories, but simply arrays, and we are forced to write our own interpolation routine to find the appropriate value of $\mathbf{K}$ for any given time t. After all, it turns out that the combined CTRL-C/ACSL software is indeed capable of solving the posed problem, but not in a very convenient manner. This is basically due to the fact that ACSL is not (yet!) sufficiently powerful for our task, and that the interface between the two languages is still kind of awkward. Because of the weak coupling between the two software systems, it might indeed have been easier to program the entire task out in ACSL alone, although this would have meant to do without any of the matrix manipulation power offered in CTRL-C.

How about IMPACT[4, 18, 19, 20]? In IMPACT, it was decided not to rely on any existing simulation language, but rather to build simulation capabilities into the control environment itself. This is partly because of the fact that (as the above example shows) the currently available simulation languages are really not very well suited for our task, and partly due to our decision of employing Ada as implementation language. As currently no CSSL has been programmed in Ada, we would have had to rely on the "pragma concept" (which is Ada's way to establish links to software coded in a different language). However, we tried to limit the use of the pragma concept as much as possible as this feature does not belong to the standardized Ada kernel (and thus, may be implementation dependent).

Until now, only the use of linear systems in IMPACT was demonstrated. However, non-linear systems can be coded as special macros (called SYSTEM MACRO's). The Riccati equation can be coded as follows:

```
SYSTEM ricc__eq(a,b,q,rb)  RETURN k  IS
k = ZERO(a);
BEGIN
    k` = -q + k*b*rb*k - k*a - a' *k;
END ricc__eq
```

The back-apostrophe operator denotes the derivative, since the forward-apostrophe operator has been reserved to denote transposition.

The state equations can be coded as follows:

```
SYSTEM sys__eq(a,b,rb,x0)  INPUT k  RETURN x  IS
x = x0;
BEGIN
    x` = (a - rb*k)*x;
END sys__eq
```

The total experiment can be expressed in another macro (of type FUNCTION MACRO):

```
FUNCTION fin__tim__ricc(a,b,q,r,xbeg,time__base) IS
BEGIN
    back__time = REVERSE(time__base);
    rb = r\b';
    k1 = ricc__eq(a,b,q,rb)*back__time;
    k2 = REVERSE(k1);
    x = sys__eq(a,b,rb,xbeg)*k2;
    RETURN <x,k2>;
END fin__tim__ricc;
```

Notice the difference in the call of the two simulations. The first system (ricc__eq) is autonomous. Therefore, simulation can no longer be expressed as a multiplication of a system macro with a (non existent) input-trajectory vector. Instead, the system macro here is multiplied directly with the domain variable, that is: the time base. The second system, on the other hand, is input dependent. Therefore, the multiplication is done (as in the case of the previously discussed linear systems) with the input trajectory. *FIN__TIM__RICC* can now be called just like any of the standard IMPACT functions (even nested). The result of this operation are two variables, y and k, of the trajectory vector and trajectory matrix type, respectively.

```
x0 = [0;0]; a = [0,1;-2,-3]; b = [0;1];
q = [10,0;0,100]; r = 1;
forw__time = LINDOM(0,10,0.1,METHOD=>'ADAMS',
                            ABSERR=>0.001);
[y,k] = fin__tim__ricc(a,b,q,r,x0,forw__time);
PLOT(y)
```

As can be seen from the above example, the entire *integration information*, in IMPACT, is packed into the *domain variables* which makes sense as these variables anyway contain part of the run-time information (namely the communication points, and the final time). Moreover, this gives us a neat way to separate clearly between the *model description* on the one

hand, and the *experiment description* on the other as suggested by Zeigler[25].

Obviously, this is a much more powerful tool for our demonstration task than even the combined ACSL/CTRL-C software. Unfortunately, contrary to CTRL-C, IMPACT has not yet been released. Roughly the first 75.000 lines of Ada code have meanwhile been coded and debugged. The IMPACT kernel has been completed in 1987. This kernel implements all of IMPACT's language structures (including all of the macro types, the complete query feature, and multiple sessions), but it does not contain all of the foreseen control library functions, the advocated simulation facility, and support of multiple windows.

7. CONCLUSIONS

By the development of new simulation standards, the possible inclusion of matrix elements as standard building elements in the modeling language should be seriously considered. Moreover, the experimental frame of a modern simulation language should be as flexible and versatile as the command languages of matrix/control environments.

REFERENCES

[1] **ANSI/MIL-STD 1815A,** (1983). *Reference Manual for the Ada Programming Language*.

[2] **Augustin, D.C., J.C. Strauss, M.S. Fineberg, B.B. Johnson, R.N. Linebarger and F.J. Sansom,** (1967). "The SCi Continuous System Simulation Language (CSSL)". *Simulation*, **9**, pp. 281-303.

[3] **Cellier, F.E.,** (1986). "Enhanced Run-Time Experiments in Continuous System Simulation Languages". In: *Proceedings of the 1986 SCS Multiconference*, (F.E. Cellier, ed.), SCS Publishing, San Diego, CA, pp. 78-83.

[4] **Cellier, F.E. and C.M. Rimvall,** (1988). "Computer Aided Control System Design: Techniques and Tools". In: *Systems Modeling and Computer Simulation*, (N.A. Kheir, ed.), Appendix A, Marcel Dekker.

[5] **Dongarra, J.J, J.R. Bunch, C.B. Moler, and G.W. Stewart,** (1979). *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics.

[6] **Garbow, B.S., *et al.*,** (1977). *Matrix Eigensystem Routines, EISPACK Guide Extensions*. Springer, Lecture Notes in Computer Science, **51**.

[7] **IBM,** (1972). *Continuous System Modeling Program III (CSMP III) Program Reference Manual*, Program Number 5734-XS9, Form SH19-7001-2, IBM Canada Ltd., Program Product Centre, 1150 Eglington Ave. East, Don Mills 402, Ontario.

[8] **IBM,** (1984). *Dynamic Simulation Language/VS (DSL/VS). Language Reference Manual*, Program Number 5798-PXJ, Form SH20-6288-0, IBM Corporation, Dept. G12/Bldg. 141, 5600 Cottle Road, San Jose, CA 95193.

[9] **Integrated Systems, Inc.,** (1984). *$MATRIX_X$ User's Guide, $MATRIX_X$ Reference Guide, MATRIXx Training Guide, Command Summary, and On Line Help*, Integrated Systems, Inc., Palo Alto, CA.

[10] **Korn, G.A.,** (1987). "Control System Simulation on Small Personal Computer Workstations", *Int. J. Modeling and Simulation*, **8**(4).

[11] **Korn, G.A. and J.V. Wait,** (1978). *Digital Continous System Simulation*, Prentice Hall, 212p.

[12] **Little, J.N. *et al*,** (1984). "CTRL-C and Matrix Environments for the Computer Aided Design of Control Systems", In: *Proceedings of the 6 th International Conference on Analysis and Optimization (INRIA)*, Nice, France, Springer Verlag, Lecture Notes in Control and Information Sciences, **63**.

[13] **Little, J.N. and C.Moler,** (1985). *PC-MATLAB, User's Guide, Reference Guide, and On Line HELP, BROWSE and Demonstrations*, The MathWorks, Inc., 158 Woodland St., Sherborn, MA 01770.

[14] **Mitchell and Gauthier, Assoc.,** (1981). *ACSL: Advanced Continuous Simulation Language - User Guide / Reference Manual*, P.O. Box 685, Concord, Mass.

[15] **Moler, C.,** (1980). *MATLAB, User's Guide*, Department of Computer Science, University of New Mexico, Albuquerque, NM.

[16] **Nilsen, R.N.,** (1984). *The CSSL-IV Simulation Language, Reference Manual*. Simulation Services, Inc., 20926 Germain Street, Chatsworth, CA.

[17] **Pritsker, A.A.B.,** (1979). *Introduction to Simulation and SLAM-II*, 3rd edition, Halsted Press.

[18] **Rimvall, C.M.,** (1983) *IMPACT, Interactive Mathematical Program for Automatic Control Theory, User's Guide*, Dept. of Automatic Control, Swiss Federal Institute of Technology, ETH Zentrum, CH 8092 Zürich, Switzerland, 208 p.

[19] **Rimvall, C.M.,** (1986). *Man Machine Interfaces and Implementational Issues in Computer Aided Control System Design*. Ph.D. Dissertation, ETH Zürich, Diss. ETH No 8200, Dept. of Automatic Control, ETH Zentrum, CH-8092 Zürich, Switzerland, 225p.

[20] **Rimvall, C.M. and Cellier, F.E.,** (1985) "The Matrix Environment as Enhancement to Modeling and Simulation", In: *Proceedings of the 11th IMACS World Conference*, Oslo, Norway, August 5 9, North Holland Publishing.

[21] **Smith, B.T., *et al*,** (1974). *Matrix Eigensystem Routines, EISPACK Guide*. Springer, Lecture Notes in Computer Science, **6**.

[22] **Systems Control Technology, Inc.,** (1984). *CTRL-C, A Language for the Computer Aided Design of Multivariable Control Systems, User's Guide*, Systems Control Technology, Inc., Palo Alto, CA.

[23] **Wait, J.V. and D. Clarke III,** (1976). *DARE-P User's Manual*, Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721.

[24] **Walker, R., *et al.*,** (1982). *$MATRIX_X$, A Data Analysis, System Identification, Control Design, and Simulation Package*, IEEE Control Systems Magazine, December Issue.

[25] **Zeigler, B.P.,** (1976). *Theory of Modeling and Simulation*, John Wiley.