A STRUCTURAL APPROACH TO CACSD

Magnus Rimvall
Institute for Automatic Control
Swiss Federal Institute of
Technology (ETH)
CH-8092 Zurich, Switzerland
01/256'28'42

François E. Cellier
Department of Electrical and
Computer Engineering
University of Arizona
Tucson, AZ 85721, U.S.A.
(602) 621-6192

In this paper different structural aspects of the new CACSD-package IMPACT are presented. In a first chapter, the different data structures needed in a general control package are presented using examples from IMPACT. In a second chapter, the need for a structured command interface is discussed. In a last sector, we elaborate on the advantages of using well structured implementation languages like Ada for CACSD-applications.

1. INTRODUCTION

Many CACSD-packages perform their operations on one single data structure: the complex matrix [4], [10]. As long as we want to treat linear systems in the time domain, this structure is adequate as each system can be described by four such matrices. On the other hand, if we for example work in the frequency domain, we would like to describe our systems by transfer-function matrices. This four-dimensional structure can not easily be represented by two-dimensional matrices. Therefore, the new CACSD-Package, IMPACT (Interactive Mathematical Program for Automatic Control Theory), supplies the user with several data structures common in control theory, e.g. polynomial and transfer-function matrices, system descriptions, domains and trajectories. Moreover, IMPACT differs from other packages not only through the supported data-structures, it also offers an extremely versatile user interface. From a computer engineering point of view, IMPACT gives a new dimension to CACSD by being the first package to be implemented in Ada [2].

IMPACT is presently being implemented at the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland [3], [8]. At this time, a kernel (controlling the interactive user dialogue) and a data administrator (handling the dynamically used data structures) exist. In the present phase, the necessary control algorithms are developed/collected and included into IMPACT. The package is already internally used at ETH and will soon be generally available.

2. DATA STRUCTURES IN IMPACT

One of the most serious drawbacks of many control packages is their lack of adequate data structures, many CACSD-packages support the complex matrix as their only data structure. On the other hand, control scientists usually work with structures like polynomial matrices, transfer-functions and linear system descriptions. For this reason, and because the absence of proper data structures in a large software package cannot easily be remedied afterhand (as such a remedy would require extensive changes to the central data structures, and thereby a recoding of large sections of the package), great attention has been given to the initial design of the data structures in IMPACT. In this chapter, these structures will be presented together with a description on how such structures are interactively created (see also [7]).

After calling the program IMPACT, the user will find himself in an interactive environment, where he can create variables of different kinds and enter commands to be executed. The available data structures range from simple scalars over matrices and polynomial matrices to complex system descriptions. The form of the commands used to create these structures is similar to that of MATLAB [5]. If we wanted to create a 2 by 2 matrix, we would write:

```
TWO_TWO =  <1, 3
           5, 7>;
```

Moreover, for more complex structures not available in MATLAB, a similar syntax is used. For example, the input line

```
Q = < 1^2^1 , 2^1 >
```

will result in the polynomial row vector

```
Q(p)     =
   1. + 2.*p + 1.*p**2                    2. + 1.*p
```

Alternatively, a longer but better readable way of entering the polynomial matrix Q would be to first define the variable P as

```
P = <^1>;
```

Thereafter Q can be entered as

```
Q  =  <1 + 2*P + P**2, 2 + 1*P >;
```

Until now, all polynomial matrices have been entered by specifying all non-zero coefficients of the polynomial elements, that is in non-factorized form. Structures in this form can of course be transformed to a factorized form:

```
QF  = FACTOR (Q)
```

will transform the matrix Q to

```
QF(p)   =
    (p + 1.)*(p + 1.)                     (p + 2.)
```

It is also possible to enter factorized polynomial matrices directly:

```
QF = <-1¦-1, ¦-2>
```

The basic matrix operations addition, subtraction and multiplication may be used on polynomial matrices if the basic dimensional rules are fulfilled. However, only in special cases can the inverse of a polynomial matrix be described in the form of another polynomial matrix. On the other hand, the inverse of any polynomial matrix can be described by another structure very useful in control theory - the transfer-function matrix. For example, an element-by-element division of two polynomial row vectors will result in one transfer-function vector. E.g.

```
NUMER = <P,1>;
TRAFUN = NUMER ./ Q
```

will result in the structure

```
TRAFUN(p) =
          p                              1
  ------------------             ---------
  1. + 2.*p + 1.*p**2            2. + 1.*p
```

which of course also can be obtained by

```
TRAFUN = < ^1/(1^2^1), 1/(2^1)>
```

Whereas polynomial and transfer-function matrices can be used in IMPACT to describe linear control systems in the frequency-domain, a system description containing four matrices in the form

$$\dot{x} = A*x + B*u$$
$$y = C*x + D*u$$

can be used to describe linear systems in the time-domain. Given the component matrices, the function LCSYS will form a continuous linear system description

```
CSYS1 = LCSYS(A,B,C)
```

whereas LDSYS will form a discrete linear system description with a sampling rate of DT:

```
DSYS1 = LDSYS(F,G,H,DT)
```

The D matrix was here assumed to be a null matrix of correct dimensions. However, if the user wants to define a D-matrix, this can be entered through the use of default redefinition:

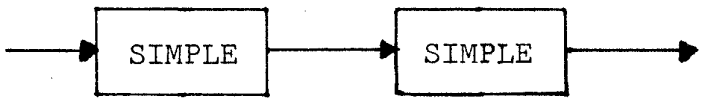
```
CSYS2 = LCSYS(A,B,C //D=DD)
```

will include the matrix DD as the direct-path matrix.

Mathematical operations on system descriptions have been defined such that the physical meaning of the operation is the same as on transfer-function matrices. For example, in the frequency-domain a multiplication of two transfer-functions correspond to a cascading of the two systems. Similarly, if a system of 2nd order has been defined through the matrices

```
A = <1, 1
     0, 1>;
B = <0
     1>;
C = <1, 0>;
SIMPLE = LCSYS(A,B,C);
```

the operation




```
CASC = SIMPLE * SIMPLE
```

will result in a system of order 4 with the component matrices

```
CASC.A = <1, 1, 0, 0
          0, 1, 0, 0
          0, 0, 1, 1
          1, 0, 0, 1>
CASC.B = <0
          1
          0
          0>
CASC.C = <0, 0, 1, 0>
```

Note that the dimension of the system matrix is doubled, just as the order of the physical system.

The IMPACT-structure domain contains a sequence of discrete values. A domain can for example consist of increasing discrete values to be used to form the independent variable of a table.

```
TIME = LINDOM(0.,50.,0.1)
```

would thus define a sequence TIME with 501 elements, the first of which has the value 0 and the last the value 50, using an increment of 0.1.

A trajectory is a table of function values which uses a domain as independent variable. Such a table results from a variety of operations performed on domains. E.g. would the operation

```
TRA = SIN(TIME)
```

result in a table where each entry contains an independent variable copied from the domain TIME and the sine-value thereof.

Mathematical operations are defined on trajectories using the same domain, e.g. would the operation

```
TRB = TRA + COS(TIME)
```

once again be a table with one row of values as function of the independent variable TIME.

All graphical functions return a trajectory as result, this trajectory can then be plotted with the command PLOT.

```
PL1 = BODE(1/<9^5^9^1> //DOMAIN=LOGDOM(.1,1000.,100))
```

or a little more verbose but better readable

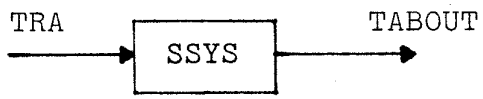
```
S = <^1>;
G = 1 / (S**3 + 9*S*S + 5*S + 9);
FREQ = LOGDOM(.1,1000.,100);
PL1 = BODE (G//DOMAIN=FREQ)
```

will compute a Bode-diagram of the system

$$G(s) = \frac{1}{s^{**}3 + 9.*s^{**}2 + 5.*s + 9.}$$

plot it and store this diagram as a trajectory.

Furthermore, domains and trajectories can be used to simulate system behaviour. If SSYS is any system representation (e.g. a transfer-function matrix or a system description),



TABOUT = SSYS * TRA

will perform a simulation and store away the values of the output signal at the discrete times of the trajectory TRA, thus making TABOUT another trajectory variable specified over the same domain as TRA.

As we live in an imperfect world, control scientists usually have to use non-linear models to describe real systems. Therefore, IMPACT will include structures to describe non-linear systems as well. From such a structure, it will be possible to get a linearized model which can be used in a linear design of an appropriate controller. Thereafter, this controller can be inserted into the non-linear model, and the behaviour of the controlled, non-linear system can be simulated. If the simulated results are not satisfactory, this design sequence can be repeated, using e.g. another criterion for the controller design. IMPACT also provides several possibilities to transform linear system descriptions to nonlinear system descriptions, and vice-versa. Both continuous and discrete systems can be modeled, and sampled data systems are connections of the two.

3. MODE OF INTERACTION

Many modern interactive CACSD-packages are command driven in the sense that the user controls the flow of action of the package using an (often quite complex) command language. Compared with other means of communication, e.g. question-and-answer or menu-driven interaction, a command driven interface is faster, gives the user a greater flexibility and is better suited for the algorithmic kind of problems found in control theory. However, any developer of such a command language must make compromises: if the language is made too rich, the complexity of the system makes it hard to use. On the other hand, if too few language elements are included, the system will not be flexible enough to let the user solve all his problems. In this chapter, we will discuss the need for a highly structured command language in CACSD, and how at the same time elements can be built into the language to facilitate its use by inexperienced users.

As no general CACSD-package can include every conceivable control algorithm, especially not if it is to be used as a tool in scientific research, the user must be supplied with an interface flexible enough to let him extend the package according to his own needs. In particular, it must be possible for the user to assemble existing base algorithms to form more powerful or more general algorithms. Taking the development in software engineering during the last decade into account [9], this is most adequately fulfilled by a highly structured command language containing the necessary flow control elements (e.g. FOR-, WHILE-loops, IF-statements). Such a command language could be developed from scratch, giving the developer full freedom of design, or it could be derived from any existing structured computer language like Algol, Pascal or Ada. For IMPACT, a command language with a syntax similar to that of Ada had been developed. This similarity to Ada brings three advantages:

- Together with the data structures presented in the previous chapter, the command language can be made rich enough to describe almost any control algorithm.

- It can be expected that Ada will become a widely used language in the near future. Therefore, many users will find a very familiar user interface and will require only a short time to get acquainted with the package interface.

- As the user input, and any functions described in the command language, have to be interpreted rather than compiled, the execution of complex algorithms described by command language is rather slow. During the development of new algorithms, this is offset by the time not spent on compilations. However, to obtain shorter execution times, any complex algorithm should be compiled and incorporated into the package itself as soon as it is developed and tested. As the implementation language (Ada) is similar to the command language of IMPACT, such a transition can be made with a minimum of recoding.

As an example of the IMPACT command language, let us consider the problem of solving the Riccati Equation

$$\dot{x} = A*x + B*u \qquad y = C*x \qquad \int_0^\infty (x'*Q*x + u'*R*u)dt \stackrel{!}{=} \text{MIN}$$

After defining the matrices A, B, Q and R, this problem can be solved by a simple algorithm described in [6]:

```
a = <...>; b = <...>; q = <...>; r = <...>;

<v,d> = EIG(<a, -b*(r\b'); -q, -a'>);
k=0; n = DIM(a);
FOR j IN 1 .. 2*n
 LOOP
  IF d(j,j) < 0 THEN k=k+1; v(:,k) = v(:,j); END IF;
 END LOOP;
p  = REAL(v(n+1..2*n,1..k)/v(1..n;1..k));
fc = -r\b'*p
```

In this algorithm, we first calculate the eigenvectors and eigenvalues of the Hamiltonian. We store the eigenvalues diagonally in d and the eigenvectors as columns in v. Thereafter we extract the columns corresponding to negative eigenvalues and finally we can calculate the feedback coefficients fc. We note the similarity with Ada, the main differences derive from the notation of MATLAB and include the use of '=' for assignment statements , '<' and '>' to describe the mathematical structures and ':' to form substructures (e.g. to form column vectors out of matrices).

Although the above sequence of statements could be directly entered on the terminal, for repetitive use it should be made into a function. In IMPACT, such a function is as simple to define as to use:

```
Function Riccati(a,b,q,r);
BEGIN -- Riccati
  <v,d> = EIG(<a, -b*(r\b'); -q, -a'>);
  ..
  ..
  p = REAL(v(n+1..2*n,1..k)/v(1..n;1..k));
  RETURN -r\b'*p;
END Riccati;

ffcc = Riccati(aa,bb,qq,rr)
```

In the previous examples, variables have been created without being previously declared. The reason for this disparity to Ada is that in IMPACT command sequences ("algorithms") are entered interactively. It would then be very cumbersome for the

user to be forced to define every single variable in the beginning of every session, especially if he does not exactly know which method to use and which intermediate variables will be needed. On the other hand, explicit variable declarations help detecting programming errors in functions and increases the security of the functions by performing run-time type checks on the parameters. Therefore the header of the Riccati function could be complemented with type declarations, in which case all variables used in the function have to be declared:

```
FUNCTION Riccati(a,b,q,r : MATRIX);
  v,d,p : MATRIX;
  k,n   : INTEGER; --j is an implicitly declared loop counter (cf. Ada)
BEGIN -- Riccati
  ..
```

To further enhance the flexibility of IMPACT functions, so called default parameters can be used. In the last chapter, we saw an example on how to produce a Bode plot:

```
BODE(1/<9^5^9^1>)
```

Using the above call, a Bode plot with 100 points over the default frequency range 0.01 to 100.0 is produced. However, if we wanted to magnify this plot over a smaller range, we could specify the optional (defaulted) parameters LOW_BOUND and/or HIGH_BOUND:

```
BODE(1/<9^5^9^1> //LOW_BOUND=0.1//HIGH_BOUND=10.0)
```

The use of defaulted parameters simplifies the standard calls, yet offers an extensive functional flexibility. Moreover, through the use of mutually exclusive qualifiers, different modes and/or input/output specifications can be selected. Hence, the parameters of the Bode function specifying the frequency bounds can, as we have previously seen, be replaced by a //DOMAIN qualifier (helpful for the user producing several Bode plots over the same domain):

```
FREQ = LOGDOM(.1,10.,100);
BODE (G //DOMAIN=FREQ)
```

With the potent data structures and command language of IMPACT, the advanced control scientist is given a very powerful algorithmic environment, which he can further adapt to his own needs. On the other hand, if first time users are directly presented with the full IMPACT package, they will most certainly be stunned by the complexity of the package. Many CACSD-packages try to resolve this problem by including an interactive HELP facility. However, whereas such help is excellent once you have a general overview of the package and only need information on a particular subject, during an introduction to the package it is as pedagogic as a 200 page reference manual. Of course, IMPACT does support an interactive HELP, but to prevent the initial shock it also gives the user a gradual introduction. A tutorial presents only the simplest language elements, e.g. how to create variables and how to call standard functions. Moreover, as even this might be too complicated for a beginner unfamiliar with the standard concepts of control theory, a query-mode has been introduced. In this mode, the initiative is transferred from the user to the system. Through a guided conversation, the system will determine the correct action to take.

Assuming that an inexperienced user wants to use the for him new function BODE. He would then call the function using the //QUERY qualifier, forcing IMPACT to enter the query mode. Thereafter the user will be asked for values of each parameter. Optional (defaulted) parameters need to be specified only when another default value is to be changed (user input has been underlined):

```
BODE (//QUERY)
BODE>>The Bode function produces one or several Bode plots of system(s)
BODE>>described by transfer-function(s).
BODE>>SYSTEM     (NO DEFAULT): 1/<9^5^9^1)
BODE//LOW_BOUND (DEF=0.01)   : 0.1
BODE//HIGH_BOUND(DEF=100.0) : 10.0
BODE//POINTS     (DEF=100)    :
```

Especially for functions with many parameters, this facility is very useful. Moreover, if the user is uncertain on the meaning of a particular parameter, he can enter a HELP for further information.

```
BODE>>SYSTEM: HELP
BODE>>Please enter a transfer-function (transfer-function matrix)
BODE>>describing the system.
BODE>>
BODE>>When the system is given by a transfer-function matrix,
BODE>>one Bode plot is produced for each transfer-function of the
BODE>>matrix (can be overridden by the //SELECT_COMPONENT qualifier).
BODE>>SYSTEM:
```

If at this point the users description of the system is in the form of a system description and not a transfer-function, one of the options available to the user would be to open another interactive session through the command SPAWN, there use the general HELP facility to find out how a transfer-function is obtained from a linear system description, perform the transformation and return the result to the BODE command:

```
BODE>>SYSTEM: SPAWN

%IMPACT-MESSAGE, Global session 3 is started.
%IMPACT-MESSAGE, All variables are imported as local copies.

>> HELP
 ..
>> RETURN_TRANS(LG)

%IMPACT-MESSAGE, Global session 3 is closed.
%IMPACT-MESSAGE, All local variables are deleted.
%IMPACT-MESSAGE, 1 parameter is passed back to queried function BODE.

BODE//LOW_BOUND (DEF=0.01)   : 0.1
```

The possibility to start a new, "virtual" session can also be used by the more advanced users to open up local workspaces for intermediate calculations (scratchpads), using an environment similar to the virtual processes/windows found on many modern workstations.

4. IMPLEMENTATION CONSIDERATIONS

Until now, most CACSD-packages and CACSD-libraries have been implemented in FORTRAN. This insures a fair portability, as FORTRAN compilers are available on virtually all computer systems and ANSI FORTRAN standards exist. However, due to the diminishing cost of hardware and the soaring cost of software, software engineering aspects like reliability, portability, and maintenance costs will play a more important role in future implementations of larger application packages. Whereas these aspects certainly do not favour FORTRAN, the new computer language Ada was designed with these particular aspects in mind [2]. Being the first larger CACSD-project to be implemented in Ada, IMPACT therefore gives a new dimension to CACSD.

In Ada (as in other languages, e.g. Pascal) the user can define his own structured data types. This is very useful to CACSD-programs, permitting us to implement program structures directly corresponding to the structures used in control theory. For example, we can define a linear system description type as one record containing four different-sized matrices:

```
TYPE syst_descr_type(n_dim, m_dim, p_dim : positive) IS
        RECORD
          state_matrix  : matrix_type(n_dim, n_dim);
          input_matrix  : matrix_type(n_dim, m_dim);
          output_matrix : matrix_type(p_dim, n_dim);
          direct_matrix : matrix_type(p_dim, m_dim);
        END RECORD;
```

The so called discriminants n_dim, m_dim and p_dim specify the order of the system, the number of inputs and the number of outputs, respectively. We notice that the four matrices are of different sizes. Moreover, as Ada allows us to dynamically create any number of variables of this kind (syst_descr_type) with different discriminant values, we will be able to simultaneously work on several different-sized systems without wasting any storage space. Furthermore, the use of such high-level structures makes the program more readable and enhances the robustness of the program, as it is automatically checked for consistency. During run-time, IMPACT will do a type-check on each operation, guaranteeing that you do not for example multiply a system description in the time domain with a transfer-function.

Through the use of data abstraction, we can hide the details on e.g. data types or algorithm implementations from other parts of the program. This feature is a cornerstone in Ada's aim at robust programming, and should be used to modularize larger software projects. Data abstraction is used quite extensively in IMPACT. For example, procedures containing mathematical algorithms are "hidden" from the programmer not working directly on these algorithms. Such a programmer can of course use all these algorithms to perform the mathematical operations, but he has no possibility to change the algorithms, or to access the internal data structures of the algorithms. Moreover, in some cases the whole implementation of the algorithm is hidden, so that the programmer has no way of knowing how the algorithm internally works, and thereby cannot use this knowledge to adapt ("improve") his program in such a way that it relies on the particular implementation used. This means that, even at a very late stage of development, it is possible to replace internal routines (e.g. for numerically better ones), without the risk of influencing the behaviour of other routines.

Ada is per definition portable, there may not exist any sub- and/or super-set of Ada with that name. Furthermore, the language is rich enough to support CACSD-packages; for example Ada's support of recursiveness allows for a much more elegant coding of the IMPACT expression parser than FORTRAN would do. In this way, IMPACT shall be easier maintainable and updatable than most other CACSD-packages.

Ada provides for a unique means of exception handling. The main difference between the Ada exception handler and most conventional (user defined) error handlers is that Ada can handle user-errors (e.g. erroneous interactive input sequences) as well as system-/programming-errors (e.g. division by zero or array-index out of range) in a portable manner. This is heavily used in IMPACT to return the program to a consistent state in the sequel of an interactive input error and/or program error, guaranteeing that the program is not aborted with a loss of all intermediate results.

A current disadvantage of Ada is that no Ada-libraries of e.g. mathematical algorithms are available. However, such libraries are planned and should emerge on the market in the near future [1]. Moreover, Ada allows for a so called "PRAGMA INTERFACE" to access programs written in other languages.

5. SUMMARY

The keyword of this contribution was "structure". Only through structured data elements it is possible to describe all mathematical entities used in control theory. The incorporation of a highly structured command language in a CACSD-package guarantees the flexibility needed in a research environment. A package implemented in a highly structured computer language like Ada can be expected to be more reliable and require less maintenance than a package using FORTRAN. Examples from the new CACSD-package IMPACT have given illustrations to these "structural" aspects of CACSD.

6. REFERENCES

[1] ACM Ada letters, 4 (1984) Iss.3 p.68.

[2] ANSI/MIL-STD 1815 A, Reference manual for the Ada programming language (January 1983).

[3] Cellier, F.E. and M. Rimvall, Computer Aided Control Systems Design 1984., in Ameling, W. (Ed.), Proc. First European Simulation Conference ESC'83, (Informatik Fachberichte, Springer Verlag, 1983).

[4] Little, J.N. et alia, CTRL-C and matrix environments for the computer aided design of control systems, in Proc. 6'th International Conference on Analysis and Optimization (INRIA), (Lecture notes in Control and Information Sciences 63, Springer Verlag, 1984).

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

[6] Potter, J.E., Matrix quadratic solutions, SIAM Journal of Applied Mathematics, 14 (1966) 496-501.

[7] Rimvall, M., IMPACT, Interactive Mathematical Program for Automatic Control Theory, A Preliminary User's Manual, Institute for Automatic Control, ETH Zurich, Switzerland (1983).

[8] Rimvall, C.M, Cellier, F.E, IMPACT - Interactive Mathematical Program for Automatic Control Theory, in Proc. 6'th International Conference on Analysis and Optimization (INRIA), (Lecture notes in Control and Information Sciences 63, Springer Verlag, 1984).

[9] Scientific American, Special issue on Computer Software, (September 1984).

[10] Walker, R. et alia, MATRIX$_x$, A Data Analysis, System Identification, Control Design, and Simulation Package, IEEE Control Systems Magazine (December 1982).