SIMULATION OF SYSTEMS '79
L. Dekker, G. Savastano, G.C. Vansteenkiste (eds.)
© North-Holland Publishing Company, 1980

THE COSY SIMULATION LANGUAGE

Francois E. Cellier Antonio P. Bongulielmi

Institute for Automatic Control
The Swiss Federal Institute of Technology Zurich
ETH - Zentrum
CH-8092 Zurich
Switzerland

COSY is a new simulation language for the simulation of COmbined continuous and discrete SYstems. With its continuous subsystem, COSY represents a new CSSL-type simulation language. COSY-coded continuous simulation programs look quite similar to CSMP-coded programs. As a matter of fact, the COSY definition even constitutes a superset of CSSL since many additional structuring elements are incorporated (e.g. for comfortable modeling of variable structure systems. For the discrete subsystem, a combined process- and event orientation has been used. Discrete COSY programs will, thus, look similar to SIMULA-67 or GPSS-V programs. In addition to these two subsystems, new elements have been incorporated into the language definition to allow for coding combined continuous and discrete systems as well. A PASCAL-coded preprocessor translates COSY input into GASP-V executable code where GASP-V is an ANSI-FORTRAN-IV coded subroutine package presented at the last IMACS Congress in Delft [3,5].

1. INTRODUCTION

Few simulation software packages/languages exist for the simulation of combined continuous/discrete systems. As shown in [4], most of them are extensions to previously defined packages/languages for discrete simulation which in most cases have been developed by scientists with a background in Operations Research. For this reason, most of these software systems offer only limited facilities for modeling of continuous systems, not taking into account much research being devoted to the development of the CSSL-type software which led to so powerful language definitions as ACSL.

Even the accessible CSSL software suffers, however, from notable shortcomings. There has recently been much research devoted to the definition of formal languages. These ideas have not properly been applied to the definition of CSSL software so far.

COSY tries to surmount these shortcomings of earlier simulation software by combining the powerful numerical capabilities of the successful GASP software with generous structuring facilities for both program and data, and with user-friendly model description capabilities. These allow for modeling of complex situations in a comfortable and also safe manner. COSY is a context-free, deterministic, left-to-right language. Its syntax is fully described by syntax diagrams, and also by using the EBNF-notation as proposed by WIRTH [15].

2. THE BASIC LANGUAGE ELEMENTS OF COSY

The COSY language consists primarily of the well known elements of continuous and discrete simulation languages. Few additional elements have been incorporated to weld these two subsystems together.

2.1. The state-event and its associated state-conditions

The only essential new element is the state-condition describing conditions of the continuous subsystem status required to branch to the discrete subsystem. A typical situation is illustrated in the following:

> When the angular velocity of a DC-motor crosses a threshold of 1 500 RPM in the positive direction, the motor has to be loaded.

The crossing of the threshold by the velocity is a typical state-condition, whereas loading the motor is the associated state-event.

The state-condition, in the above problem, is coded using a 'CONDIT'-statement in the continuous subsystem:

```
CONTINUOUS
  ...
  CONDIT EV1: OMEGA CROSSES 1500.0 POS
              TOL=1.0E-3 END;
```

and the reaction to this is coded by an event description in the discrete subsystem:

```
DISCRETE
  EVENTS
    EV1: TL := 200.0 END ;
    ...
```

(the torque load (TL) is to be reset to 200.0). The CONDIT-statement is similar to a CSMP FINISH condition, except that the time of the crossing is iterated until a prespecified tolerance is met (TOL=1.0E-3), and in that the simulation run is not terminated, but control is handed over to the discrete simulation system. After event handling, as described by the discrete subsystem, control is returned to the continuous subsystem where the new value of TL will be used somewhere in one or several equations on the right hand side of the equal sign.

2.2. Operations of the continuous subsystem on the discrete subsystem

There are none.

2.3. Operations of the discrete subsystem on the continuous subsystem

It is most commonly found that not only parameters of the continuous subsystem (as the torque load TL above) change their values at event times but that some of the equations are replaced by others. This situation can be taken care of by the following language elements:

a) The "one-out-of-n" situation

There are n possible "models" out of which one is always active. This situation can best be expressed by a CASE-statement:

```
CASE NMOD OF
```

where NMOD is an integer number pointing to the currently active model. This language element is used in general to describe n different functional ways of behaviour of one model component, e.g. n continuous branches of a discontinuous (but piece-wise continuous) functional block.

b) The "k-out-of-n" situation

Another frequently found situation is illustrated by the following example:

> There are n cars in a system, out of which k are moving around and (n-k) are parked somewhere.

This situation can be represented by the following syntactical construct:

```
FOR I:=1 TO N DO
  IF CAR[I] THEN
```

where CAR is a boolean array with the values "true" for cars moving around and "false" for parked cars.

For n = 1 this case degenerates to a simple IF-clause.

Another way of coding a "k-out-of-n" situation in a more flexible manner is to describe its single component as a full-MODEL of which MODEL-instances can be CREATEd, SUSPENDed, RESUMEd, and DELETEd.

c) Example

Let us consider a mechanical system with a dry friction torque (TFR) modeled somewhere in the system. The functional relationship which models the friction torque (TFR) as a function of the angular velocity (OMEGA) and of the driving torque (T) can be shown by the following graph:

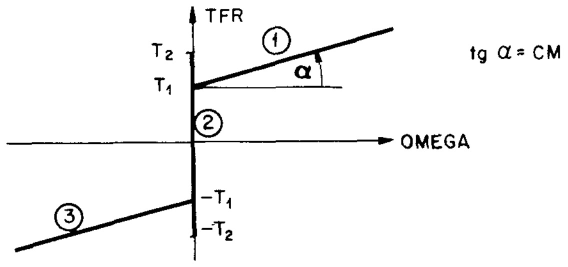

Fig.1: Dry friction torque versus angular velocity

In this example we face the typical "one-out-of-n" situation, where n = 3 are the three continuous branches of the discontinuous TFR-function. Each of them is represented by a different equation and by a different set of state-conditions.

This situation can be coded as shown in Fig.2. Using this formalism for describing a combined system, the resulting description is not much more complicated than using a normal CSSL-type language, but it allows the preprocessor to produce properly executable run-time code.

```
SYSTEM
  CONTINUOUS
    ....
    ....
    MODEL DRYFRICTION (TFR <- T, OMEGA):
    (* COMMENT: <- SYMBOLIZES A LEFT ARROW AND IS USED
       TO SEPARATE INPUT FROM OUTPUT VARIABLE LISTS *)
      CASE NL OF
        1:  TFR = T1 + CM*OMEGA:
              CONDIT MOD2: OMEGA CROSSES 0.0 NEG  TOL=1.0E-3  END
            END;
        2:  TFR = T;
              CONDIT MOD1: T CROSSES T2 POS  TOL=1.0E-3  END;
              CONDIT MOD3: T CROSSES -T2 NEG  TOL=1.0E-3  END
            END;
        3:  TFR = -T1 + CM*OMEGA;
              CONDIT MOD2: OMEGA CROSSES 0.0 POS  TOL=1.0E-3  END
            END
    END (* DRY FRICTION *);
    ....
  END (* CONTINUOUS SUBSYSTEM *);
  DISCRETE
    EVENTS
      MOD1:  NL := 1  END;
      MOD2:  NL := 2;  OMEGA := 0.0  END;
      MOD3:  NL := 3  END
    END (* STATE-EVENT DESCRIPTION *)
  END (* DISCRETE SUBSYSTEM *)
END (* SYSTEM DESCRIPTION *);
```

Fig.2: Combined description of a dry friction torque

The mode selector switch NL determines the three continuous branches of the discontinuous dry friction torque. Continuous simulation, as described by the CONTINUOUS block, goes on until one of the state-conditions associated with the currently active mode is met. At this moment, integration is interrupted, and control is transferred to the discrete subsystem modeled by the DISCRETE block. The associated state-event is executed which basically changes the selector switch NL, in the above example, to point to another mode. Now, the control is transferred back to the continuous subsystem, and the integration algorithm is restarted to integrate the model over the next inter-event time span.

3. ATTRIBUTES OF COSY AND THE BASIC CONCEPTS

3.1. Flexible Structures

COSY is generally applicable. It has the same wide range of applicability as GASP-V [3,5].

It provides facilities for modular programming. The user can declare a part of the system's description as an autonomous submodel which communicates with its environment through a programmable interface (a list of formal parameters). This can e.g. be realized by a "MODEL"-element as shown in Fig.2 above. In the context of its environment, a MODEL behaves in the same way as the PROCED construct proposed in the CSSL definition. It is a sandwich statement which is regrouped as a whole within the other parallel statements. As in the case of the PROCED construct, the formal parameters must be separated in lists of inputs and outputs of the MODEL. They are required only to enable proper sorting of the MODEL with respect to its environment. Global constants and state variables need not be listed, and can be accessed implicitly as long as the MODEL is not to be precompiled and stored in object form for later reuse. Contrary to the PROCED construct, the statements of the MODEL are again parallel code, that is, they are sorted among each other. As a matter of fact, all modeling elements apply to a MODEL in the same way as to the whole continuous subsystem. It is, in particular, possible to define MODELs in a hierarchical manner. This language element is not identical with the CSSL-type MACRO either, as shall subsequently be shown. As a matter of fact, MODEL is a new language element which is not accessible in today's CSSL-type languages, and which is most useful for structuring problems, especially when a team of several scientists is involved in modeling a complex system jointly.

It is possible to code NOSORT and PROCED sections in COSY, but they are much less "useful" than in a so called "continuous" simulation language since the compiler takes care that no illegitimate discontinuities are coded in such a section. (The main advantage of PROCED and NOSORT sections as praised by CSSL software is the possibility to code discontinuities by writing IF ... THEN ... ELSE. Precisely this must not be done in a combined system simulation language since it deprives the compiler of any fair chance to generate numerically well-conditioned run-time code.)

There are five legitimate ways to code discontinuities in COSY:

a) By using precoded discontinuous functions offered by the language (like the GASP-functions of [3],

b) by modeling discontinuities as time- and state-events in the discrete subsystem with associated CONDIT statements in the continuous subsystem,

c) by coding subroutines which are declared to be parallel, and which, in fact, enlarge the set of (a),

d) by coding "full" MODELs, and

e) by coding MODULEs which are kept as source modules in a symbolic library.

The usual ways to code discontinuities are (a) and (b), whereas (c) and (d) require some sophistication, and are not recommanded to the unskilled user. Mainly (c) is a quite dangerous way since this is the only possibility to cheat the system (!). (e) is not really an additional concept, but rather an extension to improve modularity.

3.2. Extendability

The user of the software can extend the available simulation operators by his own problem-specific ones (open-ended operator set). Such language extensions can take place an four different levels.

On a very basic level, the language operators can be extended by coding FORTRAN subroutines. On a second level, the language operators can be extended by formulating MODELs. These can also be preprocessed into subprograms. In the definition of such MODELs (so called "full MODELs"), the user must now include all interacting variables and constants as formal parameters of the MODEL. Variables not included must be internally declared. They are local to the MODEL . A precompiled MODEL can be called in by declaring it to be an EXTERNAL MODEL. This is very similar to calling EXTERNAL SUBROUTINES. However, to allow for proper bookkeeping, the user must, in addition, specify how many state variables (for differential and difference equations) and how many history functions (requiring a unique identifier each) are internally used in the MODEL definition body.

On a third level, the language provides for a "MACRO"-facility. Formally this looks very similar to the previously presented "MODEL"-facility. It is, however, treated differently by the compiler. All MACRO calls are first replaced by their MACRO definition bodies, before any further preprocessing (like sorting) takes place. In this way, the statements which form a MACRO definition can be spread throughout the system's description, once an executable sequence of statements has been found. On the contrary, MODELs are only sorted internally, whereas the body remains together as an entity. Consequently, MODELs can be precompiled and stored in compiled form, whereas MACROs must always be kept in source form in a "symbolic" library. The MACRO-facility is needed since it is often not possible to avoid mixing equations from different MACROs to obtain an executable sequence of statements. Thus, the MACRO construct grants a higher degree of modularity compared to the MODEL construct, but it requires each MACRO definition body to be preprocessed together with the environment in which it is used.

Since the MACRO replacement must preceed all further preprocessing activities, the MACRO feature need not form an intrinsic part of the language definition. It is taken care of by a separate MACRO handler which is called prior to preprocessing. Advantages of this solution have been discussed in [2]. In this way, one can be more generous in the capabilities offered by the MACRO definition language (like offering interpretative MACRO handling) while saving core memory requirements. The additionally required computation costs are comparatively small.

On a fourth level, the language provides a programmable topological input description combining the advantages of a network formulation with those of an equation oriented language. This can actually be thought of as an extension to the previously discussed MACRO construct. When coding a MACRO, the modeler must declare which are its inputs and which are its outputs. This has some disadvantages, as will be illustrated in the following example.

Let us consider a small electrical network as depicted in Fig.3. The RC-circuit is to be modeled by a MACRO.

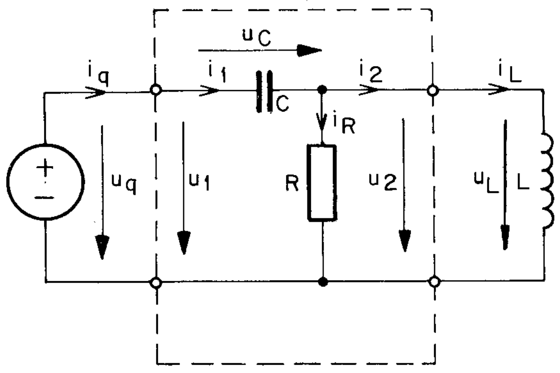

Fig.3: RLC-network with voltage source

Under the assumption that all differential equations are to be solved for state derivatives (which is reasonable, since integration is numerically much better conditioned than differentiation), there exists only one valid formulation for the required MACRO. This is coded in Fig.4.

```
MACRO RC1 (U2, I1 <- U1, I2, R, C);
  MACVAR
    STATE  UC;
    ALGEBR  IR;
  MACCONTIN
    UC' = I1/C;
    I1 = I2 + IR;
    IR = U2/R;
    U2 = U1 - UC
  MACEND (* CONTINUOUS *)
MACEND (* RC1 *);
....
CONTINUOUS
  ....
  UQ = F(TIME);
  RC1 (UL, IQ <- UQ, IL, R, C):
  IL' = UL/L;
```

Fig.4: Model of a RLC-network with voltage source

UQ must be specified as an input to the MACRO since it is an externally computed control signal. Also, IL must be an input to the MACRO since it is a state variable of the system which, consequently, cannot be a computed quantity.

Let us now replace the voltage source by a current source as depicted in Fig.5.

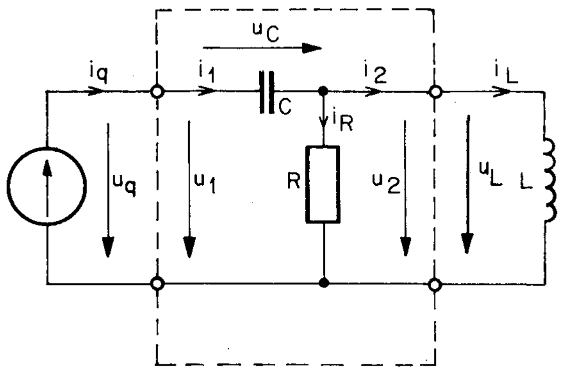

Fig.5: RLC-network with current source

Again, just one valid model can be specified for the MACRO which is depicted in Fig.6.

```
MACRO RC2 (U1, U2 <- I1, I2, R, C);
  MACVAR
    STATE  UC;
    ALGEBR  IR;
  MACCONTIN
    UC' =I1/C;
    IR = I1 - I2;
    U2 = R*IR;
    U1 = U2 + UC
  MACEND (* CONTINUOUS *)
MACEND (* RC2 *);
....
CONTINUOUS
  ....
  IQ = F(TIME);
  RC2 (UQ, UL <- IQ, IL, R, C);
  IL' = UL/L;
```

Fig.6: Model of a RLC-network with current source

This time, the source current (IQ) must be an input to the MACRO since it is an externally computed control signal.

As one can see, two different MACROs are needed to describe one and the same module. In both MACROs the same equations are represented, but rearranged to meet the demands of the required inputs and outputs. This simple example illustrates that the MACRO element is not really modular either. For this reason, we define a new language element, which we call a "MODULE", as shown in Fig.7a.

```
MODULE RC (U1, U2, I1, I2, R, C);
  VAR
    STATE  UC;
    ALGEBR  IR;
  CONTINUOUS
    UC' = I1/C;
    I1 = IR + I2;
    U2 = R*IR;
    U2 = U1 - UC
  END (* CONTINUOUS *)
END (* RC *);
```

Fig.7a: MODULE for a RC-circuit

This MODULE can be used in both networks. If a voltage source feeds the RC-circuit, the MODULE can be called as shown in Fig.7b.

```
CONTINUOUS
  ....
  UQ = F(TIME);
  RC (UL=U2, IQ=I1 <- UQ=U1, IL=I2, R, C);
  IL' = UL/L;
```

Fig.7b: Model of a RLC-network with voltage source

If the RC-circuit is fed by a current source, the same MODULE can be used as shown in Fig.7c.

```
CONTINUOUS
  ....
  IQ = F(TIME);
  RC (UQ=U1, UL=U2 <- IQ=I1, IL=I2, R, C);
  IL' = UL/L;
```

Fig.7c: Model of a RLC-network with current source

In a MODULE, equations may be solved for any variable, as long as their number is correct (problem neither under- nor overspecified), and as long as no contradictory assumptions are made. The same variable may appear several times to the left of the equal sign as U2 in Fig.7a. Formal parameters of a MODULE definition need no longer be separated into inputs and outputs. Only upon usage of a MODULE, one has to specify which are its inputs and which are its outputs . The logical mapping of actual to formal parameters is no longer implicit, but is specified by explicite assignments. UL=U2, for example, specifies that the actual parameter UL is to replace the formal parameter U2 of the MODULE definition header.

The MODULE definition language is more general than the MACRO definition language in two senses.

1) Besides replacement of parameters, it uses formulae manipulation to reorganize the statements.

2) A MODULE definition may contain an INITIAL block, a TERMINAL block and also a DISCRETE block in addition to the CONTINUOUS block. When the MODULE is called from within the continuous subsystem, these blocks will automatically be transferred to their correct locations.

The MODULE definition language is less general than the MACRO definition language in that it does not allow any interpretative execution, as we want to allow it for MACROs.

The MODULE definition language is similar to the MACRO definition language in that all involved activities must be performed prior to any further preprocessing. Also, the MODULE handler is separated from the preprocessor. It forms one program together with the MACRO handler.

This idea has first been formulated by Elmqvist [6] and by Runge [12]. Both scientists came to quite similar constructs independently. In both languages, DYMOLA [6] and MODEL [12], there exist language elements comparable to the MODULE presented herein. DYMOLA proceeds by using structural analysis of the equations and formulae manipulation techniques, whereas MODEL leaves the statements as they are, and uses implicit numerical integration techniques during execution, as has been done for years in linear network analysis programs. Implicit integration is somewhat more general since there exist legitimate system's descriptions which cannot be rearranged to form an executable set of statements (if equations cannot be solved for a particular variable in a closed form, or if algebraic loops are involved which inhibit proper grouping of statements to form an executable sequence). Using formulae manipulation is, on the contrary, somewhat more robust since illegitimate models will be automatically detected whereas this is not necessarily the case when implicit integration is used. Implicit integration will result in lower compilation and higher execution cost compared to the proposed solution technique.

A MODULE can be thought of as a network element with as many legs as there exist formal parameters of the MODULE. Elmqvist and Runge describe their "MODULEs" in a quite similar way to the one proposed except that their "MODULEs" are only intended for purely continuous simulation. However, both apply a programmable topological description to "plug" different MODULEs together, whereas we use an equation oriented approach for that purpose as for the description of the MODULEs. (This feature exists as an option in DYMOLA as well.)

Fig.8 shows how the simple RLC-network can be further decomposed.

```
MODULE RES (U, I, R);
  U = R*I
END (* RES *);
MODULE CAP (U, I, C);
  U' = I/C
END (* CAP *);
MODULE IND (U, I, L);
  I' = U/L
END (* IND *);
MODULE RC (U1, U2, I1, I2, R, C);
  VAR
    STATE  UC;
    ALGEBR  IR;
  CONTINUOUS
    CAP (UC, I1, C);
    I1 = IR + I2;
    RES (U2, IR, R);
    U2 = U1 - UC
  END (* CONTINUOUS *)
END (* RC *);
....
CONTINUOUS
  ....
  UQ = F(TIME);
  RC (UL=U2, IQ=I1 <- UQ=U1, IL=I2, R, C);
  IND (IL=I <- UL=U, L);
```

Fig.8: RLC-network further decomposed

In this program, the user must still know that the IND-MODULE has to compute the current (IL) and not the voltage (UL) to obtain a set of equations in integral form.

He can, however, also automate this procedure by specifying an additional MODULE OUT which has no inputs and which defines as outputs precisely those variables needed for printout (e.g. IL and UL). This is demonstrated in Fig.9.

```
MODULE OUT (UL, IL);
  VAR
    ALGEBR  UQ, IQ;
    REAL  C, L, R;
  CONTINUOUS
    UQ = F(TIME);
    RC (UQ, UL, IQ, IL, R, C);
    IND (UL, IL, L)
  END (* CONTINUOUS *)
END (* OUT *);
CONTINUOUS
  OUT (UL, IL <- )
END (* CONTINUOUS SUBSYSTEM *);
```

Fig.9: RLC-network finally modular

As one can see, the user must specify the inputs and outputs at MODULE calls only if he uses them directly in the continuous subsystem, but not if they are used within another MODULE definition. This is evident since, in the latter case, one can first replace MODULE calls by their MODULE definition bodies and then handle the already expanded MODULE in globo. That is, MODULEs when called from MODULEs are treated like MACROs. In the above modeling technique, the continuous subsystem will consist of one single statement only to call the root-MODULE "OUT", and the user is entirely relieved of solving any equation for particular variables. This modeling technique combines the flexibility and universality of equation oriented languages with the convenience of network modeling techniques.

The system engineer is given the possibility to extend the basic language definition itself. For this purpose, the preprocessor has been constructed in such a way that it can be easily augmented to accommodate new ideas. For this task, the most recent compiler building techniques employing structured programming and structured data representation have been applied as described, for instance, by Wirth [13].

3.3. "One-to-one" Correspondence between System and Model

We tried to construct COSY in such a way that the modeler can represent the single building blocks of the physical system under investigation by (eventually composed) building blocks of the language, and to represent the modes in which the elements of the system cooperate with each other by constructs of the language to combine building elements to larger building blocks and, finally, to a functional description of the system. This "one-to-one" correspondence of system and model will certainly lead to a much more methodical way of modeling and, by these means, improve the robustness of the model. Furthermore, there are often several system engineers involved in the formulation of one model of a complex physical system (e.g. an atomic reactor). This modeling approach is essential to allow for a subdivision of the model into single entities which can be constructed and debugged independently, and which cooperate only through interfaces which can be properly described beforehand.

The single modeling element consists partly of structures and partly of data. These are separable within the building block, but both are, nevertheless, codable within the same building block so that one physical building element can be expressed by one building block of the language as well. The building elements of the physical system are connected by topological structures. Corresponding structural elements have been made available in the language as well. These are, furthermore, also descriptive elements of the building blocks themselves to allow for a hierarchical structuring of building blocks.

Physical "modules" can, of course, consist of partly continuous and partly discrete elements. For this reason, it is certainly useful if this situation can be coded in one single building block of the language as well. This demand conflicts with the wish to have a clear separation of the continuous and the discrete subsystem. Therefore, the user can specify MODULEs which possibly consist of both a continuous and a discrete part, whereas the MODULE handler regroups the description in such a way that, on output, all continuous subsystems and all discrete subsystems are merged to form a system's description as the preprocessor should find it to produce numerically well-conditioned run-time code.

When a model of a physical system has been constructed, it remains to describe in terms of language elements, the experiment which is to be performed on the model. It seems essential that different experiments can be carried out without need for redesigning the model of the system, as one would do in a real-world experiment. It could prove useful to let several experiments be executed by one simulation program. For this reason, the language should be designed to allow several EXPERIMENTs to be formulated subsequently. All these EXPERIMENTs form together the MONITOR-segment of the simulation program.

The experiment description is composed of one part describing the control signals (input signals) to the model, and one part describing the quantities which are to be measured and output (output signals). It may be convenient to separate these two parts from each other in that the MONITOR segment only describes modes of control whereas an OUTPUT segment is used to describe graphical representations of the sampled quantities.

The considerations in this section are strongly influenced by the pioneer work in modeling methodology as performed by Zeigler [16] and Oren [10,11]. These were the first scientists who tried (only recently) to conceptualize modeling in a methodological manner.

3.4. Ease of learning Syntax and Semantics

Two main goals are to be achieved: It should be easy to write programs by one's self, and it should be easy as well to read programs coded by somebody else. These two goals tend to compete with each other. To meet the former goal, we want the different elements of the language to use the same syntactical constructs as much as possible. To meet the latter goal, we want to be as flexible as possible in choosing appropriate mnemonics and close to conversational English constructs.

To give an example of the conflicting nature of the two goals, let us consider, once again, the dry friction example stated above (Fig.2). To meet the second goal, we introduced the '='-symbol in the notation of equations of the parallel section and the ':='-symbol in the notation of statements of the procedural section. By these means the inherent difference between parallel and procedural code is clarified, in that, for instance,

```
I := I + 1 ;
```

is a meaningful statement, whereas

```
I = I + 1 ;
```

is a meaningless equation. This rule, thus, helps to improve the readability of programs. However, it complicates, at the same time, the writing of programs since it simply introduces an additional (not necessarily required) syntactical construct to remember.

4. GLOBAL VERSUS LOCAL VARIABLES

In the old days of information processing, computer languages used to be constructed in such a way as to let each independent programming unit have its own set of variables assigned to it. We call this a concept of local variables. A typical example of this type of language is FORTRAN-IV. Each SUBROUTINE has its own variables assigned to it which keep their values even over several calls to the routine. Data communication between SUBROUTINEs is only possible through lists of formal parameters or through COMMON variables.

Good modern computer languages like PASCAL make use of a global variable concept. By these means, the user can declare new variables on each hierarchical level which are then valid in the PROCEDURE in which they are defined as well as in any PROCEDURE called by it. There is no need to include any variable in the list of the formal parameters as long as the PROCEDURE is not called several times by different actual arguments. PROCEDUREs which are on the same hierarchical level can communicate data with each other only by declaring variables an a hierarchically higher level for that purpose.

This elegant concept is very clear and clean from the aspect of information processing. It has, however, two major drawbacks:

a) PROCEDUREs making use of global variables (and direct exits) may not be precompiled. They must be stored, if at all, in source form. In fact, PROCEDUREs are not really meant to be compiled separately.

b) PROCEDUREs making use of these possibilities are much more difficult to describe since they do not communicate data through a distinct interface only ("back-door" programming!).

Such a concept is, therefore, not really modular. However, modularity is an important requisite in a simulation environment. In the design of the concepts of a simulation language, one has to take this aspect into account, and design the language in such a way that provisions exist to precompile those structural blocks which can be stored in compiled form. For this reason, a language like PASCAL is very well suited for the coding of a simulation compiler which is a "closed" program, whereas a FORTRAN-like language is much better suited for the coding of a simulation run-time system which is, principally, to be compiled once, but to which different subprograms are to be added for each application problem.

In the long run, it would be more consistent with our ideas to use a PASCAL-like general task language for which the PROCEDURE concept has been enlarged to a PROCESS- or CLASS concept (which again would be modular) also for the simulation run-time system. However, although these concepts have been discussed on many occasions, and although there exist implementations of such features (SIMULA-67 , MODULA [14], PORTAL [8,9]), there does not exist any such language to date which is widespread and which has been generally accepted. SIMULA-67 has certainly found many "disciples", but even for this language there does not exist any good library of carefully debugged CLASSes for specific applications like numerical integration which could, for example, compete with the Kahaner implementation of the Gear algorithm. Such libraries exist to date only for FORTRAN-IV and for ALGOL-60.

Moreover, the main disadvantage of using FORTRAN-IV, namely its unsatisfactory programming safety, is not so critical in our application since the target language code of the users' programs is machine generated and not hand coded.

MODULEs and MACROs must be stored in source form, as we have seen. they can, therefore, easily use a global variable concept. Variables which are locally declared obtain a new unique name each time the MODULE (MACRO) is expanded.

Subprograms look syntactically similar to PASCAL PROCEDUREs, to grant safe programming, but effectively they are rather similar to FORTRAN SUBROUTINEs to satisfy the requirements of modularity.

It may, furthermore, also be useful to precompile MODELs. Since the statements of a MODEL are only sorted internally, but remain together as a block, this is feasible. The resulting code is again similar to a FORTRAN SUBROUTINE , except that the number of internally used state variables and predefined functions must be declared when a MODEL is called in as an EXTERNAL MODEL. Consequently, MODELs must also use a concept of local variables for that purpose.

Since this will, however, not be the normal case, we allow MODELs to use global variables in general. If a MODEL is to be precompiled, it must be declared to be a "full" MODEL which is not allowed to use global variables.

5. RESTRICTIONS OF COSY

It is evident that only such features can be offered in COSY which are codable in GASP-V as well. This imposes some restrictions on the definition set of COSY which are to be discussed.

The data structuring capabilities of COSY are as limited as those offered in FORTRAN-IV. The TYPE statement may be used only to define RECORDs of file entries where a RECORD may consist of a variable number of attributes which are adminstered by GASP-V as forward and backward linked linear lists. No more complex RECORD structures are available, and also the pointer variables to link RECORDs in a programmable manner are not accessible to the user. No symbolic TYPEs can be defined. All user variables must be declared to belong to one of the (admittedly high number of) predefined TYPEs:

ALGEBR, BOOLEAN, COLLECT, DSTATE, FACILITY, GATE, HISTOGRAM, INTEGER, LOGIC, MEMORY, MODEL, PRINTPLOT, PROCESS, RANDOM, REAL, SAMPLE, SEVENT, SETFILE, STATE, STORAGE, TABLEPRINT, TEVENT, TIMEPERS

or ARRAYs of them. No SETs are available either.

Routines, although looking very much like PASCAL PROCEDUREs, are really SUBROUTINEs in that variables which are locally declared keep their assigned values over several calls. Routines may not be called recursively, and it is forbidden to leave them by a GO TO statement (which is a very doubtful option even in PASCAL (!)).

Discrete PROCESSes are not as versatile as they could be. A more axiomatic approach to their semantics would be useful, but FORTRAN SUBROUTINEs are a very unwieldy carrier of such a language element, and without heavy constraints not applicable at all.

For these reasons, one may conclude that other languages like SIMULA-67 or PORTAL [8,9] are still more useful for purely discrete simulation problems than COSY, although these languages offer much less language elements directly dedicated to simulation, and although their use will, consequently, result in longer, less safe, and less readable application code.

These obvious shortcomings of COSY cannot be overcome as long as there does not exist any other generally accepted language which could replace FORTRAN-IV as a target language for COSY. We have seen that either FORTRAN-IV or PASCAL are not suitable for this purpose. ALGOL-60 has drawbacks similar to PASCAL. PL/I would be a possible candidate, but many people consider its definition set too large and its semantics not axiomatic enough to make this language very enlightening either. PL/I programs coded by others are usually not easily readable. This can, for instance, be seen in the PL/I-code generated of SIMPL/I programs, a discrete simulation language on the basis of PL/I. PORTAL [8,9] would, as to our opinion, be a promising candidate, but since this language is not administered either by a comupter manufacturer or by a non-profit organization, we give it little chance to take the barrier of international acceptance in the near future. We do not see any chance to overcome this problem, as long as the computer manufacturers find a market for their hardware without offering appropriate software for it. Only the customer will, finally, be able to force them to <u>sit together</u> to come to an agreement concerning a modern, widely supported general task language for which not only an efficient and well tested compiler is developed, but for which also cautiously debugged mathematical application software is made available.

6. EXAMPLE -- PILOT EJECTION STUDY

This is a commonly cited benchmark problem for "continuous" simulation. According to our terminology, it belongs to the class of combined problems [4].

<u>System description:</u> The pilot ejection system, when activated, causes the pilot and his seat to travel along rails at a specified exit velocity VE at an angle THETAR (measured in radians) or THETAD (specified in degrees) backward from vertical. After traveling a vertical distance Y1, the seat becomes disengaged from its mounting rails and, at this point, the pilot is considered out of the cockpit. When this occurs, a second phase of operation begins during which the pilot trajectory is influenced by the force of gravity and atmospheric drag.

This problem belongs to the class of combined systems since the model is discontinuous at the moment when the ejector seat is disengaged from its mounting rails. Even the number of state equations is time dependent. The system is of second order (NNEQD := 2) during the first phase, but it is of fourth order (NNEQD := 4) during the second phase of the simulation run. During the first phase, there exists a state-condition to establish the condition when to branch to the second phase. During the second phase, this state-condition is no longer active.

<u>Experiment description:</u> The experiment is carried out to analyse how large the maximum velocity of the aircraft (VA) may be, as a function of the height above sealevel (H), in order to allow for a secure ejection. An ejection is said to be "secure" if the ejection seat clears the vertical stabilizer of the aircraft which is 9m behind and 3.5m above the cockpit in a distance of at least 2.5m. For this purpose, a first simulation run is carried out with a small velocity (VA := 30.0) at ground level (H := 0.0). If this ejection is successful (according to our rules (!)), the velocity is increased by 15.0 m/s, and the experiment is repeated. If the ejection is not successful, the height is increased by 150.0 m, and the experiment is repeated. In this way we proceed until either the velocity has reached a value of 270.0 m/s or until the height has reached a value of 16 500.0 m above sealevel, which ever occurs first. The experiment cannot start at zero velocity since, for this velocity, the specified model is not valid.

Usually, the simulation is terminated by a CSSL-type FINISH-statement

```
FINISH  X = -9.0 ;
```

to indicate that the critical portion of the pilot trajectory is over. However, in this way, the simulation is continued over the last integration step and may end at a much more negative value of X. Since we are interested in knowing the value of Y at X = -9.0 to decide upon success or failure of the ejection, we must compute this value with some accuracy. For this reason, a continuous simulation language will have to restrict the integration step-size artificially whereas, in a combined system simulation language, we can replace the FINISH-condition by another state-CONDITion to locate the critical point, X = -9.0, precisely. The associated state-event can then be used to terminate the simulation run.

This specific experiment description has been taken from [7]. except that all data have been converted to metric units.

Output description: After each successful ejection, the actual values of the ejection level (H), and of the aircraft velocity (VA), are to be stored for later graphical representation of H as a function af VA. During the last run, we want, furthermore, to store X and Y. These values are to be plotted versus time.

Fig.10 shows the listing of a possible COSY program for this benchmark problem.

```
(*********************************************)
(*                                           *)
(*        COMBINED SYSTEM SIMULATION         *)
(*                                           *)
(*           PILOT EJECTION STUDY            *)
(*                                           *)
(*********************************************)


PROGRAM PILOTEJECT (INPUT,OUTPUT);

PROJECT 56 BY $CELLIER$;


MACRO ROTATE (X, Y <- V, THETA, CONST FLAG);
  MACIF vFLAGv = 1 THEN
  MACBEGIN
    X = V*COS (THETA);
    Y = V*SIN (THETA)
  MACEND
  MACELSE
  MACBEGIN
    X := V*COS (THETA);
    Y := V*SIN (THETA)
  MACEND
MACEND  (* ROTATE *);


LABEL
  100;


CONST
  REAL
    VE     = 12.0   (* M/S *),
    M      = 100.0  (* KG *),
    Y1     = 1.2    (* M *),
    THETAD = 15.0   (* DEGREES *),
    CD     = 1.0    (* --- *),
    S      = 1.0    (* M*M *),
    G      = 9.81   (* M/(S*S) *);


VAR
  REAL  DIST, H, HELP, RHO1, THETAR, VA, VECOS, VESIN;
  STATE  X, Y, V, THETAS;
  ALGEBR  D, VX, VY, GX, GY;
  INTEGER  PHASE;
  BOOLEAN  LAST;
  SEVENT  DISENGAGE, OVER;
  MODEL  INOROUT;


STORE
  H, VA, X, Y;

FUNCTABLE
  SPLINE RHO = (    0.0 : 1.293) (  300.0 : 1.256) (  600.0 : 1.220)
               ( 1200.0 : 1.152) ( 1800.0 : 1.082) ( 3000.0 : 0.955)
               ( 4500.0 : 0.815) ( 6000.0 : 0.676) ( 9000.0 : 0.476)
               (12000.0 : 0.319) (15000.0 : 0.196) (18000.0 : 0.122);


MONITOR
  INITCOND  X = 0.0,' Y = 0.0;
  LAST := FALSE;  STOREOFF;
  THETAR := THETAD/57.2957795;
  ROTATE (VECOS, VESIN <- VE, THETAR, 2);
  H := 0.0;  VA := 30.0;
  SIMULATE  FROM 0.0 TO FINISH  COMINT = 0.2
END  (* EXPERIMENTAL FRAME BLOCK *;

SYSTEM

  INITIAL
    HELP := VA - VESIN;
    V := SQRT (HELP*HELP + VECOS*VECOS);
    THETAS := ATAN (VECOS/HELP);
    RHO1 := 0.5*CD*S*RHO (H);
    PHASE := 1;  NNEQD := 2
  END  (* INITIAL SECTION *);

  CONTINUOUS
    ROTATE (VX, VY <- V, THETAS, 1);
    X' = VX - VA;  Y' = VY;
    MODEL INOROUT ( <- );
      CASE PHASE OF
      1:   CONDIT DISENGAGE: Y CROSSES Y1 POS TOL=1.0E-3 END
         END;
      2:   ROTATE (GX, GY <- G, THETAS, 1);
           V' = -D/M - GY;
           THETAS' = -GX/V;
           D = RHO1*V*V;
           CONDIT OVER: X CROSSES -9.0 NEG TOL=1.0E-3 END
         END
    END  (* MODEL IN-OR-OUT *)
  END  (* CONTINUOUS SUBSYSTEM *);

  DISCRETE
    EVENTS
      DISENGAGE:  PHASE := 2;  NNEQD := 4  END;
      OVER:  FINISH  END
    END  (* STATE-EVENT DESCRIPTION *)
  END  (* DISCRETE SUBSYSTEM *);

  TERMINAL
    IF  NOT LAST  THEN
    BEGIN
      IF  ((H <= 16500.0) AND (VA <= 270.0))  THEN
      BEGIN
        IF  Y > 6.0  THEN
        BEGIN
          DIST := Y - 2.5;
          FORTRAN
            WRITE (OUTPUT,100)  X, Y, DIST
            100 FORMAT (5H X = ,E12.4,5H Y = ,E12.4,8H DIST = ,E12.4)
          END  (* FORTRAN *);
          CROSSPLOT;  VA := VA + 15.0
        END ELSE
          H := H + 150.0
      END ELSE
      BEGIN
        LAST := TRUE;  STOREON
      END;
      RERUN
    END
  END  (* TERMINAL SECTION *)
END  (* SYSTEM DESCRIPTION *);


OUTPUT
  TITLE $PILOT EJECTION STUDY$;
  LIST (CROSSF) VA, H;
  PLOT (CROSSF) VERSUS VA: H;
  FACTOR  XFAK = 2.0  YFAK = 2.0;
  GRAPH (CROSSF) VERSUS VA: H;
  GRAPH (TIMEF) X, Y
END  (* OUTPUT BLOCK *)

END .
```

Fig.10: COSY program for the pilot ejection problem

Fig.11 shows the resulting graph of the ejection level depicted versus the aircraft velocity.

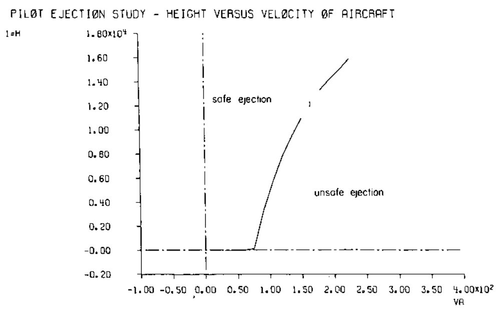

Fig.11: Safe and unsafe ejection

This example can illustrate only a small part of the features available in COSY, but the limited space does prevent us from either presenting a more complex example or from presenting several different examples.

7. CONCLUSIONS

In this article we tried to outline the methodological aspects which led to the development of the new simulation language COSY for a comfortable and safe simulation of both continuous and discrete as well as combined continuous/discrete systems. It is not the goal of this paper to replace a users' manual for COSY.

8. REFERENCES

[1] A.P.Bongulielmi: (1978) "Definition der allgemeinen Simulationssprache COSY". Semesterwork, Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich. To be obtained on microfiches from : The main library, ETH - Zentrum, CH-8092 Zurich, Switzerland . (Mikr. S637).

[2] F.E.Cellier: (1976) "Macro - Handler for Simulation Packages Using ML/I". Proc. of the 8th AICA Congress on Simulation of Systems, Delft, The Netherlands. Published by North-Holland Publishing Company (Editor: L.Dekker) ; pp. 515 - 521.

[3] F.E.Cellier: (1978) "The GASP-V Users' Manual". To be ordered from: Institute for Automatic Control, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum , CH-8092 Zurich, Switzerland.

[4] F.E.Cellier: (1978) "Combined Continuous/Discrete System Simulation Languages --- Usefulness, Experiences and Future Development". Proc. of the Symposium on Modeling and Simulation Methodology, Rehovot, Israel. Published by North-Holland Publishing Company (Editor: B.P.Zeigler).

[5] F.E.Cellier, Blitz A.E.: (1976) "GASP-V: A Universal Simulation Package". Proc. of the 8th AICA Congress on Simulation of Systems, Delft, The Netherlands. Published by North-Holland Publishing Company (Editor: L.Dekker) ; pp. 391 - 402.

[6] H.Elmqvist: (1978) "A Structured Model Language for Large Continuous Systems". Form: CODEN LUTFD2/(TFRT-1015)/1-226/(1978). Ph.D. Thesis. Lund Institute of Technology, Dept. for Automatic Control, Lund, Sweden.

[7] G.A.Korn,Wait J.V.: (1978) "Digital Continuous-System Simulation". Prentice Hall.

[8] H.Lienhard: (1978) "PORTAL Language Definition". To be ordered from: Landis & Gyr AG, Zug, Switzerland. (Partly in German).

[9] H.Lienhard: (1978) "Die Echtzeitprogrammiersprache PORTAL, eine Uebersicht". Landis & Gyr Mitteilungen 25(1978); pp. 2 - 8.

[10] T.I.Ören: (1978) "Concepts for Advanced Computer Assisted Modeling". Proc. of the Symposium on Modeling and Simulation Methodology, Rehovot, Israel. Published by North-Holland Publishing Company (Editor: B.P.Zeigler).

[11] T.I.Oren, Zeigler B.P.: (1978) "Concepts for Advanced Simulation Methodologies". Simulation, vol. 30 no. 6 : June 1978.

[12] T.F.Runge: (1977) "A Universal Language for Continuous Network Simulation". Form: UIUCDCS-R-77-866. Ph.D. Thesis. University of Illinois at Urbana-Champaign, Dept. of Computer Science, Urbana, Illinois, U.S.A..

[13] N.Wirth: (1976) "Algorithms + Data Structures = Programs". Prentice Hall, Series in Automatic Computation.

[14] N.Wirth: (1977) "MODULA: A Language for Modular Multiprogramming". Berichte des Instituts fuer Informatik, Nr 18. To be ordered from: Institute for Information Processing, The Swiss Federal Institute of Technology Zurich, ETH - Zentrum, CH-8092 Zurich, Switzerland.

[15] N.Wirth: (1977) "What can we do about the unnecessara diversity of notation for syntactic definitions?". Comm. ACM, vol 20 , no 11 : November 1977.

[16] B.P.Zeigler: (1976) "Theory of Modeling and Simulation". John Wiley.