B. Zupancic. Modular hierarchical modelling with SIMCOS language. Math. comput. simul., Vol. 46, pp. 67-76, 1998.

Modular hierarchical modelling with SIMCOS language

B. Zupančič
University of Ljubljana, Faculty of Electrical Engineering, Tržaška 25, 1000 Ljubljana, Slovenia
borut.zupancic@fe.uni-lj.si

Abstract

The paper deals with the modular hierarchical modelling and simulation possibilities of the simulation language SIMCOS. After a brief description of the modular modelling possibilities of modern modelling and simulation tools, the emphasis is put on the concept of precompiled dynamic submodels implemented in the SIMCOS simulation language. The simulation processor distinguishes between static, dynamic, continuous, discrete, delayed and non-delayed modelling constructs. Thus a very powerful simulation system with automatic sorting and runtime self-configuration was obtained. The precompiled modelling constructs can be used with textual or graphical modelling. In addition, a preprocessor for hierarchical modelling similar to the macro concept in modern simulation tools was also implemented.

1. Introduction

The concept of modularity is perhaps the most important concept of each structured program. Using this principle, large simulation models are developed not as single, monolithic units, but as subdivided modules which operate as independent functional components of the overall system. Such a hierarchical decomposition of a model has the following advantages:

- It is possible to focus attention on each component as a small problem. If a component is still too complicated, it can be further divided into smaller components until each module is of manageable size.
- Modules which are too complicated cause details to be forgotten and interactions to be confused.
- Several modellers can work simultaneously on a modelling and simulation project, since they work on different modules whose interactions must be carefully defined in advance.
- It is easy to implement changes and corrections, as they should be made in one or a few simple modules.
- The modular structure enables partial testing of a model, i.e. testing component by component.
- The modular developed model is extremely convenient for documentation purposes.
- The modular structure enables the application of different algorithms to different subsystems (e.g. the model is divided into stiff and non-stiff submodels using stiff and non-stiff integration algorithms).
- The modular concept is very important when the model is simulated on parallel processing systems, where submodels are assigned to different processors.
- The modular hierarchical tool enables the modelling of components on a very physical level. These components can be used at higher levels in different configurations, not just in the configuration for which they were developed.
- Finally, the modular model can be very realistic, reflecting the hierarchical structure inherent in the system. It can even be said that investigation of the natural but somewhat hidden hierarchical structures of real systems influences the development of modern modelling and simulation tools.

Thus strongly influenced by software engineering proposals, modern simulation languages introduce very flexible possibilities for simulating hierarchical models. They enable the use of many preprogrammed submodels from a particular language library and the creation of user-defined

submodels which can first be tested separately and then included in the library. Using such a library, a general purpose simulation tool can become a kind of application-oriented tool.

2. Types of submodel implementation

There are different ways in which submodels are implemented in simulation languages. The MACRO feature has been the most important facility, enabling modular modelling and simulation for a long period. It is still implemented in many modern simulation languages. During processing, all MACRO calls in the source simulation model are replaced by appropriate MACRO definition bodies, and their formal arguments are exchanged with the real ones. In further processing, the sorting algorithm can arbitrarily change the order of all statements, so the MACRO feature does not cause any problems with the sorting algorithm. The main disadvantage of this approach is that, for each structural change, all model components must be reprocessed. So a MACRO feature cannot be treated as a modern hierarchical approach.

The MODUL can be treated as a generalisation of the MACRO feature. Whereas MACRO implies that its inputs and outputs are known in advance, i.e. the cause and the consequence are fixed, this is not always necessarily the case. For example, the same model - an electrical component - can be described by the equation $I=U/Z$, $U=IZ$ or $Z=U/I$, where I is the current, U is the voltage and Z is the impedance of this component. So the MACRO feature requires three possible submodels for the same physical law. Using the MODUL submodel, both equations can be implemented by one submodel because the formula manipulation routine also sorts the equation in the 'horizontal direction', depending on the context in which the equation is used. MODUL is rarely used in simulation languages (COSY - [2]), and it was perhaps mainly used as a subject for theoretical investigations. A similar concept implemented in DYMOLA [5] gives this modelling tool a very modern object orientation and significant support in the modelling phase. Physical components can be described by physical laws. Each equation is solved for the appropriate unknown variable and sorted within a simulation model. So the well-known description $\dot{x} = f(x,t)$ of all CSSL simulators [9] is automatically derived. DYMOLA enables the implementation of component submodels on the true object and physical level, where components can be used in arbitrary configuration, not only in the configuration for which they were primarily developed. Of course, there are problems with nonlinearities, structural singularities, algebraic loop etc. These problems are only partly solved in DYMOLA. DYMOLA can be used to model combined systems (e.g. block diagrams, mechanical systems, electrical systems, hydraulic systems, ...). The connections of these components are not established through inputs and outputs (as in almost all CSSL simulators), but through more general connection structures called cuts. This approach allows us to couple components which influence each other. This is a more complicated situation than when "two operational amplifiers" (e.g. two blocks in a block diagram) are connected.

Separately compiled submodels in some aspects provide an even more efficient submodel feature in new generation simulation languages. They are defined in a manner similar to subprograms in general purpose programming languages. These submodels are separately compiled and enable true hierarchical modelling and simulation structures, since they can be nested to any depth. The production of a simulation tool supporting separately compiled hierarchical structures is a very exacting and complicated task. Central manipulation of state variables and their derivatives or predictions and self-configuration, meaning that each submodel reserves its storage requirements and establishes the necessary pointers, must be performed. But the most difficult task is to define the hierarchical data base properly and to implement the sorting algorithm correctly [8]. The calculations which implement a submodel containing other submodels and/or blocks with delay attributes must, due to the sorting algorithm, be performed in several stages. Implementation is particularly complicated as submodels can have several inputs and outputs. SYSMOD and COSMOS [1], [6] are, in the sense of hierarchical model structures, very powerful. Influenced by software engineering proposals, they enable dynamic creation of submodels during the simulation. OOSlim [8] combines the object-oriented approach with the concept of precompiled modules. The sorting procedure is implemented in several phases. The precondition phase takes place at compile

time, ensuring the marking of the most obvious errors. The level evaluation phase takes place at run time and gives each instance of a hierarchical model a proper weight (i.e. level). The so-called post condition phase takes place at run time and handles algebraic loops. The final phase is the sorting procedure, which takes place at run time and sorts components of the model according to their level (i.e. weight).

The implementation of so-called segments (e.g. ESL - [4]) seems to be a convenient feature of some modern simulation languages, making an additional contribution to the model definition modularity. Segments are in fact those parts of a model which are simulated separately. They communicate with each other only at defined prescribed time instants. The final goal of segment implementation is to simulate a model in a multiprocessor environment. However, they can be also used on sequential machines, where the multiprocessor environment is emulated.

The meaning of object-orientation is sometimes misunderstand in the modern modular modelling and simulation approach. Object orientation on the modelling level means that the modeller can use submodels of a real process components in the way DYMOLA allows. Of course, it is easier but not necessary to implement such a tool with modern object-oriented programming languages. And if object orientation is extremely important for the modelling level, this is not true for a runtime simulation level. An executable object-oriented simulation program is still inefficient from the computational point of view. So a flat program is still preferred [3].

3. Implementation of modular constructs in SIMCOS simulation language

The SIMCOS simulation language is a CSSL-type equation-oriented language developed at the Faculty of Electrical Engineering in Ljubljana [10], [11], [12]. It works as a compiler. The model, which is coded in CSSL syntax [9] or is described by a graphical block-oriented simulation scheme, is processed by the compiler into FORTRAN modules and a model data base. These modules are further processed by a FORTRAN compiler and linked with appropriate libraries into an executable program. The supervisor program automatically handles all of these procedures and is able, together with a highly interactive user interface, not only to simulate the model but also to perform simple experiments (e.g. change model constants, output specifications, function generators' breakpoints) and complex ones (e.g. parameter studies, optimisation, linearisation, ...). Many integration algorithms (single step, multistep, low order, high order, extrapolation, stiff, ...) cover all common numerical problems, giving the simulation tool an appropriate numerical robustness. Implementing real-time possibilities, the language was expanded with hardware-in-the-loop and man-in-the-loop simulation possibilities. Animation with the aid of AutoCAD defined schemes is also provided.

The SIMCOS concept did not enable the implementation of a unified concept of the above-described precompiled hierarchical constructs. So we implemented

- precompiled dynamic blocks, and
- hierarchical submodels.

as two different additional possibilities.

4. Precompiled dynamic blocks

The concept with precompiled dynamic blocks was implemented by several modifications to the SIMCOS simulation compiler, the runtime system and the graphical editor preprocessor.

Features of the simulation compiler

Modifications of the simulation compiler were mainly performed in a module for analysing function calls. The modified compiler distinguishes between different function calls or blocks. Fig. 1 shows the basic grouping of modelling blocks.

Fig. 1. Basic grouping of modelling blocks

Blocks can be static or dynamic. Static blocks which do not posses continuous or discrete states can be simple, function generators or multivariable ones. Simple blocks are mainly nonlinearities and standard signals. They are called as FORTRAN FUNCTIONs, and the calls are not modified during SIMCOS compilation. Function generators, due to the CSSL standard requirements [9], are appropriately preprocessed during compilation into a breakpoints data file, into program code for reading these points at runtime before the simulation run, and into a function call for function generators' output evaluation in the DERIVATIVE runtime part. As FUNCTION in FORTRAN can only return a single value, a block with several outputs (multivariable block) can be implemented only by converting the FUNCTION call into a SUBROUTINE call during SIMCOS compilation. Fig. 2 shows an example in which Cartesian coordinates are transformed into the polar ones. The block is appropriately sorted before conversion from FUNCTION to SUBROUTINE call.

SIMCOS source definition module	After SIMCOS compilation in derivative
ARRAY POLAR(2), CART(2)	DIMENSION POLAR(2), CART(2)
CART(1)= ...	CART(1)= ...
CART(2)= ...	CART(2)= ...
POLAR=CONV(CART)	CALL CONV(POLAR, CART)

Fig. 2. Compilation of the call of a multivariable block

All other function calls are treated as calls to dynamic blocks. They can be continuous (with integrator operator) or discrete (with delay operator), univariable or multivariable, non-delayed or delayed. As continuous and discrete blocks are uniquely implemented, processing during SIMCOS compilation is the same for both blocks. The blocks are distinguished for possible future expansions. During compilation, these blocks' calls are modified by extensions with two arguments in function calls. One real variable is added for storing the output variable in cases where such blocks are multiply used. Another integer variable is used for storing the offset (the address) of the particular block in common arrays of states, derivatives and predictions. The latter grouping is important in terms of the sorting procedure. Non-delayed blocks are blocks which respond instantly to an input change. Delayed blocks are blocks whose outputs do not change instantly with inputs (e.g. integrators, discrete delays, appropriate transfer functions, zero order hold,...) and which can be used for disconnectiong model loops. Fig. 3 shows this grouping.

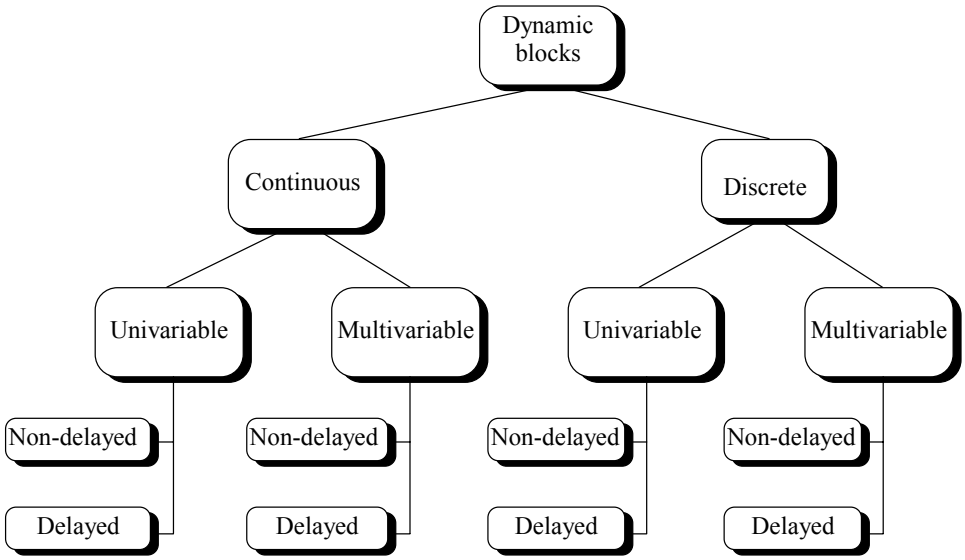

Fig. 3. Grouping of dynamic blocks

All blocks except simple blocks and function generators must be installed in a special installation file, which lists the names of blocks (functions or subroutines) and appropriate grouping codes. So the system is opened for adding new blocks. The implemented concept also enables nested calls. However, nested calls must be properly horizontally sorted, since the sorting algorithm only sorts among rows (blocks). The PID control system shown in Fig.4

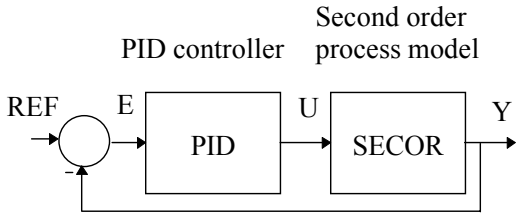

Fig. 4. PID control system with a second order model

can be programmed as:

```
U=PID(REF-Y,KP,KI,KD,TF)
Y=SECOR(U,ZETA,OMEGAN)
```

or

```
U=PID(REF-SECOR(U,ZETA,OMEGAN),KP,KI,KD,TF)
```

The second option means that the sorting is done by a user, since SECOR as a continuous dynamical function with the delayed attribute must first calculate its output.

Fixed dynamic structures can be easily installed as delayed or non-delayed ones. However, using general structures as transfer functions or state space descriptions, the delayed or non-delayed character depends on the selected parameters. As they are only known at runtime during simulation, the only solution would be to implement the sorting procedure at runtime. The implemented

compiler concept did not allow this modification, so the only solution was to implement two dynamic structures for such blocks, with and without delay.

Structure of precompiled dynamic modules and the derivative module

Fig. 5 shows the FORTRAN source code for the second order system.

```
function SECOR(zout,iofset,u,zeta,omegan)
c     second order system
      real u,zeta,omg
      common /Z00OUT/IOUT
      common/Z00000/nstates,x(150),dx(150),x0(150)
      common /Z00009/z00008(10)
      integer z00008
      data nint/2/
      if(Z00008(9).ne.4) goto 10
c     self-configuration and initialization of states
         iofset=nstates
         nstates=nsates+nint
         x0(iofset+1)=0.0
         x0(iofset+2)=0.0
   10 continue
      if(IOUT.eq.1) goto 100
   c  derivatives evaluations
         dx(iofset+1)=x(iofset+2)
         dx(iofset+2)=u*omegan**2-2*zeta*omegan
     *     *x(iofset+2)-omegan**2*x(iofset+1)
      goto 200
   100 continue
   c  storing of outputs
         zout=x(iofset+1)
   200    SECOR=zout
      return
      end
```

Fig. 5. FORTRAN code for the second order system

The program module can be used as a template to program similar continuous dynamical modules. In the first part, self-configuration and the initialisation is performed. Each block counts its own address (offset) iofset within the common vector of states and derivatives, and initialises its states. The following part consists of two operations: derivative evaluations, when a flag is IOUT=0, and output evaluations, when the flag is IOUT=1. These two parts are calculated during two calls to each dynamic module in the module for derivatives evaluations (DERIV). Fig. 6 shows the structure of the derivative evaluation module, which is generated by the simulation compiler.

Declaration of variables
IOUT=1 Calculation of outputs of all dynamical blocks and other assignment statements in the sorted statements order e.g. Y=SECOR(Z00K01,Z00I01,U,ZETA,OMEGAN)
IOUT=0 Evaluation of derivatives of dynamic blocks e.g. Y=SECOR(Z00K01,Z00I01,U,ZETA,OMEGAN)

Fig. 6. Structure of derivative evaluation (program module DERIV)

Like continuous dynamical blocks, which operate with the centralised vectors of continuous states and derivatives, discrete dynamic blocks use a centralised vector of discrete states and predictions. However, updating of discrete states is done for each dynamic block separately and not centralised as in the integration module for continuous states. This enables different sampling intervals for each discrete block. During the first call in the derivative module, the discrete blocks calculate their outputs from the states and inputs. During the second call, the predictions of states and updated values of states (the values of states for the next sampling interval) are calculated.

Features of the graphical editor

Precompiled objects are also suported in the block-oriented graphical editor BLOCK, which enables model definition in a graphical way. As modelling constructs can be very complex, such a modelling approach is very efficient.

Implemented blocks

Table 1 shows the continuous and discrete blocks implemented in SIMCOS. Besides standard dynamic blocks, continuous and discrete industrial PID controllers were included. The blocks have integral windup protection, a switch from automatic to manual and vice versa, and different structures for the inclusion of differential term.

CONTINUOUS blocks		DISCRETE blocks	
Integrator	Y=FIN(U,Y0)|	Discrete delay	Y=DDLY(U,D,STATES,TS)
Vector integrator	Y=VIN(U,N,Y0)|		
Mode control integrator	Y=FKIN(U,Y0,IC,OP)|		
First order system	Y=FLAG(U,K,TAU)|		
Integral system	Y=FINTLG(U,K,TAU)		
Differential system	Y=DIFF(U,KD,TF)		
Lead-lag	Y=FLEDLG(U,K,Z,P)		
Second order system	Y=SECOR(U,ZETA,OMEGAN)		
Dead time	Y=DELAY(U,TD,STATES,TS)		
Transfer function in polynomial form (without delay)	Y=CTF(U,N,B,A)	Transfer function in polynomial form (without delay)	Y=DTF(U,N,B,A,TS)
Transfer function in polynomial form (with delay)	Y=CTFD(U,M,N,B,A)	Transfer function in polynomial form (with delay)	Y=DTFD(U,M,N,B,A,TS)
Transfer function in factorised form (without delay)	Y=CZP(U,N,K,Z,P)	Transfer function in factorised form (without delay)	Y=DZP(U,M,N,K,Z,P,TS)
Transfer function in factorised form (with delay)	Y=CZPD(U,M,N,K,Z,P)	Transfer function in factorised form (with delay)	Y=DZPD(U,D,M,N,K,Z,P,TS)
State space description (without delay)	Y=CSS(U,N,A,B,C,D,X)|	State space description (without delay)	Y=DSS(U,N,A,B,C,D,X,TS)
State space description (with delay)	Y=CSSD(U,N,A,B,C,X)|	State space description (with delay)	Y=DSSD(U,N,A,B,C,X,TS)
Multivariable system in state space (without delay)	Y=MCSS(U,N,M,L,A,B,C,D,X)	Multivariable system in state space (without delay)	Y=MDSS(U,N,M,L,A,B,C,D,X,TS)
Multivariable system in state space (with delay)	Y=MCSSD(U,N,M,L,A,B,C,X)	Multivariable system in state space (with delay)	Y=MDSSD(U,N,M,L,A,B,C,X,TS)
PI controller	U=PI(E,KP,KI)	Sample&hold	Y=SH(U,STATE,TS)
PID controller	U=PID(E,KP,KI,KD,TF)	PID controller	U=DPID(E,Q,STATES,TS)
Industrial PID controller	U=PIDAB(Y,R,U00,MA,KP,TI,TD,TF,KA,GAMA,UMIN,UMAX)	Industrial PID controller	U=DPIDAB(Y,R,U00,MA,KP,TI,TD,TF,KA,GAMA,UMIN,UMAX,TS)

Table 1. Implemented precompiled dynamic blocks

5. Hierarchical modelling

Using a special preprocessor, hierarchical modelling is also available. Well-tested and previously developed programs for particular components can be reused as submodels in higher hierarchical levels. The hierarchical preprocessor is in some ways similar to the MACRO concept in simulation languages. The requirements for hierarchical preprocessor development were as follows:

- CSSL syntax was taken into account.
- The internal variables of submodels must be exchanged with working names so that each hierarchical model can be reused several times.
- The preprocessor must be fast and efficient.
- The user should not be concerned with sorting model statements.

The submodel is defined in SIMCOS syntax with the header

```
submodel NAME (output1,output2,... = input1,input2,...)
                                    ...
end
```

and called with

```
call NAME (output1,output2,... = input1,input2,...)
```

The only limitation is in the PROCEDURAL block. If it is used in a nested submodel, its body can posses only a FUNCTION or SUBROUTINE call.

6. Conclusions

As the concept of the SIMCOS simulation language did not enable implementation of a unified precompiled hierarchical approach, the precompiled dynamic blocks and the hierarchical preprocessor were implemented separately. Therefore, precompiled dynamic blocks do not enable nested structures. Only the block calls can be nested. On the other hand, the hierarchical preprocessor which works similarly to MACRO has no nesting level restrictions. With these extensions, SIMCOS became a very powerful and modular simulation tool for complex dynamic model simulation.

References

[1] N.J.C. Baker, P.J. Smart, Elements of the SYSMOD simulation language, *Proceedings of the 11th IMACS world congress*, Oslo, 1985.

[2] F.E. Cellier, *Combined continuous / discrete system simulation by use of digital computers: Techniques and tools.* Ph.D. Swiss Federal Institute of Technology, Zürich, 1979.

[3] F.E. Cellier, *Continuous System Modeling*, Springer - Verlag, New York, 1991.

[4] R.E. Crosbie, S. Javey, J.S. Hay, J.G. Pearce, ESL - a new continuous system simulation language *Simulation*, vol.44, no.5, pp. 242-246.

[5] H. Elmqvist, *Dymola - Dynamic Modeling Language*, User's Manual, Dynasim AB, Lund, 1994.

[6] D.L. Kettenis, *Issues of parallelization in implementation of the combined simulation language COSMOS*, Ph. D. Thesis. Delft Technical University, 1994.

[7] D. Matko, B. Zupančič, R. Karba, *Simulation and Modelling of Continuous Systems - A Case Study Approach*. Prentice Hall, London, 1992.

[8] M. Ostroveršnik, D. J. Murray-Smith, Modularity in Dynamic System Simulation, Proceedings of CESA'96 IMACS multiconference - Computational Engineering in Systems Applications, Lille, France, (1996), vol. Symposium on Modelling, analysis and simulation 2/2, pp. 660-665.

[9] J.C. Strauss, The SCI continuous system simulation language, *Simulation* no.9, 1967, pp. 281-303.

[10] M. Titovšek, *Modern methods and implementations in continuous systems simulation languages*. MSc. Thesis, Faculty of Electrical and Computer Engineering, University of Ljubljana, 1994.
[11] B. Zupančič, *Digital simulation language synthesis for the computer aided control system design*, Ph. D., Faculty of Electrical and Computer Engineering, University of Ljubljana, 1989.
[12] B. Zupančič, D. Matko, R. Karba, M. Atanasijević Kunc, Z. Šehić, Extensions of the simulation language SIMCOS towards continuous - discrete complex experimentation system, *Preprints of the IFAC Symposium CADCS '91*, University of Wales, Swansea, U.K., (1991): pp. 351-356.