A New Model Language for Continuous Systems

Elmqvist, Hilding

1977

Document Version:
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Elmqvist, H. (1977). *A New Model Language for Continuous Systems*. (Technical Reports TFRT-7132). Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
1

Dokumentutgivare
Lund Institute of Technology

Handläggare
Hilding Elmqvist

Författare
Hilding Elmqvist

Dokumentnamn
REPORT

Dokumentbeteckning
LUTFD2/(TFRT-7132)/1-112/(1977)

Utgivningsdatum
Dec 1977

Ärendebeteckning
06T6

10T4

Dokumenttitel och undertitel

A new model language for continuous systems

Referat (sammandrag)

A model language for continuous dynamical systems is proposed. The model equations can be entered as they are. They need not be converted to assignment statements. There is a concept, cut, which corresponds to connection mechanisms of complex type, and there is also a simple way of describing the connection structure of a system. These notions make it possible to conveniently describe models in a hierarchical fashion.

The equaitons of the model are sorted and they are converted to assignment statements using formula manipulation. These manipulations are dependent on the operations to be done on the model.

Referat skrivet av
author

Förslag till ytterligare nyckelord
44T0

Klassifikationssystem och -klass(er)
50T0

Indextermer (ange källa)
52T0

Omfång
112 pages

Övriga bibliografiska uppgifter
56T2

Språk
English

Sekretessuppgifter
60T0

ISSN
60T4

ISBN
60T6

Dokumentet kan erhållas från
Department of Automatic Control
Lund Institute of Technology
P.O. Box 725, S-220 07 LUND 7, SWEDEN

Mottagarens uppgifter
62T4

Pris
66T0

DOKUMENTDATABLAD enligt SIS 62 10 12

SIS-
DB 1

Blankett LU 11:25 1976–07

A new model language for continuous systems

Hilding Elmqvist

Department of Automatic Control
Lund Institute of Technology

Contents

1. INTRODUCTION

Many programs for simulation of dynamical systems on digital computers were developed during the sixties. The first languages were designed around concepts that were familiar to users of analog computers. Such languages are called block diagram languages. They used concepts such as integrators, summers and potentiometers. The user had to convert his model to a block diagram of these elementary subsystems. It was then simple to input the model in this form to the computer. Some of the advantages with this technique compared to analog simulation was that there was no need to scale the problem with respect to time and amplitude. The security against badly specified models was increased because the model was not specified as connections on a patch board as for analog computers. The documentation was also better.

The transformation of the model to block diagram form was not necessary. Programs were developed which accepted differential equations directly. Such languages are called equation oriented languages. One of these languages was MIMIC which was widely used. The Simulation Councils in the USA proposed another language in 1967 (Strauss, 1967). Their language was called CSSL (Continuous System Simulation Language) and has been implemented on many computers. The program CSMP-360 (Continuous System Modelling Program) was developed at about the same time.

In equation oriented languages the model is specified as assignment statements of FORTRAN type. A special integration operator is used for differential equations. The equations do not have be given in any special order because they are sorted by the programs. It is a problem for the modeler that the equations have to be given as assignment statements because it is sometimes difficult to determine which variable to solve for in an equation. This problem is further discussed in chapter 2.

In equation oriented languages submodels can be handled by using a Macro concept. Examples are given in chapter 2.

An other commonly used way to specify models is to write a subroutine or procedure in an algoritmic language, which computes the derivatives of the state variables.

After 1967 progress has essentialy been made in two areas: interactive programs and combined continuous - discrete simulation.

Simulation is a good example of the need for interactive computing. The most well known interactive programs are the DARE programs. The program SIMNON has been developed by the author (Elmqvist, 1975,1977). Other examples of interactive programs are ISIS, BEDSOCS and SIM.

The interest for simulation of systems modelled by both ordinary differential equations and discrete events has increased, see Fahrland (1970). One reason for this is the desire to simulate processes controlled by digital computers. Several programs have been developed, e.g. GASP-IV, GSL and CADSIM. Models described by ordinary differential equations and difference equations can be used in SIMNON. Simulation of systems described by ordinary differential equations, partial differential equations and discrete events can be done in GASP-V.

It seems that the development of programs for digital simulation have been much influenced by the available software technology. This has implied that little has been done with model languages. The time has now come to use the advances of computer science in programs which recognizes the demands by the user.

The situation for the modeller can vary widely. A difficult case is when trying to obtain an accurate model for a large complex system for which no prior model exists. The modeller first has to find the structure of the system and then split it up into modules with simple connections. This is necessary because it is practically impossible for a single person to grasp a large system at the same time as the details are described. It must also be decided which phenomena that are interesting for the model and which quantities to be included in the model.

In presently available languages the connections between subsystems are done with variables. There are no concepts which correspond to the much more complex connection mechanisms that occur in physical systems such as shafts, pipes, electrical wires, etc. The connections of submodels would be much simplified if such mechanisms were available. The details of the connection mechanisms, such as the variables involved, do not have to be considered at the time the structure of the system is described. This is an important consequence for the modeller who has models available for the included subsystems. If it is known that the models are compatible with respect to the included phenomena, the degree of complexity and the connections then the model building is reduced to a description of the structure of the system. One example of this situation is the engineer who selects available modules to form a complex system. If there are models for the modules then it is a simple matter to check the performance of the system.

This report contains a proposal for a model language for continuous dynamical systems. The characteristics of this language are the following. The differential equations and the algebraic equations can be introduced as they are. They need not be converted to assignment statements. There is a concept, cut, which correponds to connection mechanisms of complex type and there is also a simple way of describing the connection structure of a system.

The connections between submodels introduces constraints on the variables in the cuts. This can in same cases lead to a reduction of the number of states for the system. The parallel connection of two capacitors is a typical example. Each capacitor is separately described by one state variable. However, the total system will have only one state. Many of the available integration algorithms require that the derivatives of the states can be computed as a function of the states. This is not possible for such systems using the basic equations. In many cases it is possible if the model is augmented by some of the equations differentiated.

An other way of attacking this problem is to develop integration methods that can handle such systems directly. Such methods are available, see chapter 5. For electrical networks and mechanical systems there are special methods to obtain the model in state space form. Such transformations are not needed if integration methods of this type are used.

An important characteristic of the language is that the model is independent of the operations to be done. It could e.g be simulation or different types of static computations. The equations are transformed in different ways depending on what is unknown. The transformation can frequently be done in a way that the variables can be solved one at the time from the equations. When systems of equations occur, which have to be solved simultaneously, they are often small and in many cases linear. There are methods to find in which order the variables should be solved and from which equations. These methods also indicates systems of equations. They only use the structure of the equations, i.e. if a variable appears in an equation or not.

If an equation is linear in the unknown variable it is easy to get the corresponding assignment statement by formula manipulation. Linear systems of equations can also be solved by formula manipulations. Nonlinear equations generally have to be solved by iterative technique.

When solving systems of equations by iterative technique the Jacobian is often needed. The computations may be speeded up if it is computed by symbolic differentiation.

The methods for sorting and manipulation of the equations have consequences not only for the numerical computations. The resulting assignment statements and systems of equations can be shown to the user in symbolic form. This is very interesting because it shows the cause-effect relationship between variables. It also has a positive psycological effect to see exactely the equations that were generated from the model in the high level language.

Gear and Runge at University of Illinois have developed a program for simulation of dynamical systems (see Gear(1972) and Runge(1975)). Their program accepts the model equations as they are without convertion to assignment statements. Cuts or terminals can be introduced as a set of variables describing a connection mechanism. The model structure can be entered using a display and a light pen. A figure can be associated with each submodel. When a submodel is incorporated its figure is placed at a specified point on the display. The connections of the submodels are done by drawing lines between the terminals of the submodels. It is also possible to connect the submodels using alphanumeric instructions. The integration of the equations is done with an implicit routine for differential-algebraic systems (see Brown and Gear (1973)).

The use of equations instead of assignment statemets have been discussed for analysis of static systems. Many of the algorithms for transformation of the equations have been developed for chemical computations. Design computations on thermal power plants (Volgin et.al., 1975) is an other example. The corresponding problem for models of economical systems with difference equations is discussed by Drud (1975). Aarna (1976) has a theoretical discussion of transformation of the equations.

Chapter 2 illustrates some of the problems with present model languages like CSSL, CSMP, DARE and SIMNON. The drawbacks discussed has served as a motivation for the proposed model language. This language is described in chapter 3. Chapter 4 discusses different types of operations on the model. Methods for doing these operations are given in chapter 5. Chapter 6 illustrates the use of the model language to describe some different types of systems. The appendix contains a description of the syntax notation used and the syntax of the language.

2. SOME PROPERTIES OF PRESENT SIMULATION LANGUAGES

To use a language of the CSSL-type the model equations must be rewritten as assignment statements. When a model is derived from physical principles it is frequently not trivial to know what variables should be solved for. The assignment statement is also a worse form of documentation. In some cases the equations have to be transformed in different ways depending on the environment of a subsystem.

This chapter contains two examples which illustrates the advantedges of describing a model with equations.

Example 2.1

Consider the network in Fig. 2.1.

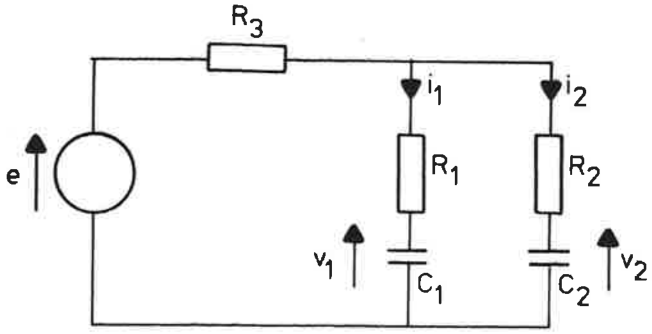


Fig 2.1

A model for this system is

$$C_1 v_1' = i_1$$

$$C_2 v_2' = i_2$$

$$e = R_3(i_1+i_2) + R_1 i_1 + v_1$$

$$e = R_3(i_1+i_2) + R_2 i_2 + v_2$$

To enter the model into a simulation language like CSSL the linear system of equations involving i_1 and i_2 must be solved by hand. This is necessary in order to use CSSL effectively because CSSL only has facilities for solving systems of equations by iterative technique.

An algorithm for finding the derivatives is shown below.

$$e := \ldots$$

$$i_1 := \frac{1}{R_1R_2 + R_1R_3 + R_2R_3}(R_2e - (R_2+R_3)v_1 + R_3v_2)$$

$$i_2 := \frac{1}{R_1R_2 + R_1R_3 + R_2R_3}(R_1e + R_3v_1 - (R_1+R_3)v_2)$$

$$v_1' := i_1/C_1$$

$$v_2' := i_2/C_2$$

It can be observed that the original model is easy to write down and easy to check. The transformed model on the contrary is not at all as easy to check and not as easily readable. A small change in the equations may also imply large changes in the assignment statements. It is, however, possible to make a computer discover systems of equations and solve them by formula manipulations. These manipulated equations can then be used for computations and also be printed for the user.

[]

Example 2.2

This example shows the problem that the needed manipulations of the equations may depend on the environment.

Suppose that the low pass filter in Fig 2.2 is a component of a system.

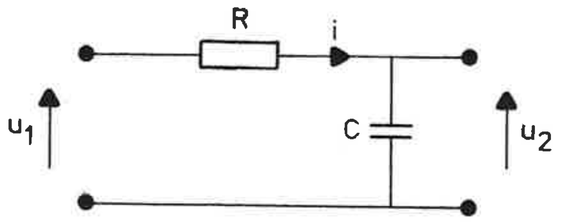


Fig 2.2

A model is

$$u_1 - Ri = u_2$$
$$Cu_2' = i$$

The output gate is assumed to be open. Using the Macro facility of CSSL this system can be modelled as

```
MACRO FILTER [U2,I = U1,R,C]
  I=(U1-U2)/R
  U2=INTEG[I/C,0]
END
```

Assume that the low pass filter is used in the circuit in Fig 2.3.

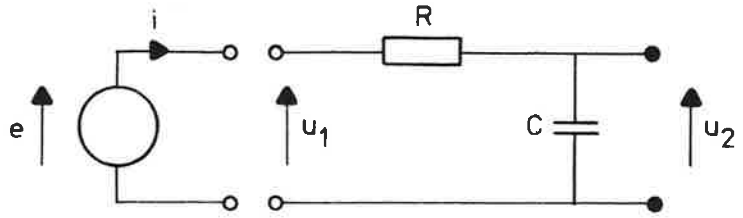


Fig 2.3

The driving voltage is

```
e=sin(t)
```

The system can then be described as

```
E=SIN(T)
U2,I=FILTER[E,R,C]
```

This system description is expanded to the equations

```
E=SIN(T)
I=(E-U2)/R
U2=INTEG[I/C,0]
```

which describes the system correctly.

Consider now the system in Fig 2.4.

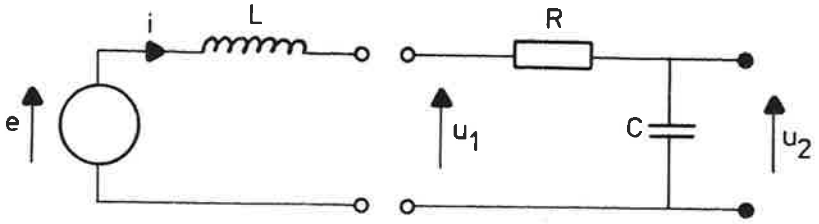


Fig 2.4

The aditional equation is

$$u_1 = e - Li'$$

A description of this system could be

```
E=SIN(T)
I=INTEG[(E-U1)/L,0]
U2,I=FILTER[U1,R,C]
```

The expansion of the Macro gives

```
E=SIN(T)
I=INTEG[(E-U1)/L,0]
I=(U1-U2)/R
U2=INTEG[I/C,0]
```

Two equations have I in their left hand part. This is not allowed in CSSL. However, if the statements are considered as equations they are correct. The Macro FILTER can not be used in this case. It has to be modified as

```
MACRO FILTER2[U1,U2 = I,R,C]
  U1=U2+R*I
  U2=INTEG[I/C,0]
END
```

The system description can now be done as

```
E=SIN(T)
U1,U2=FILTER2[I,R,C]
I=INTEG[(E-U1)/L,0]
```

which is expanded to

```
E=SIN(T)
U1=U2+R*I
U2=INTEG[I/C,0]
I=INTEG[(E-U1)/L,0]
```

These statements constitute a legal model in CSSL.

The third case to be studied is the circuit in Fig 2.5.

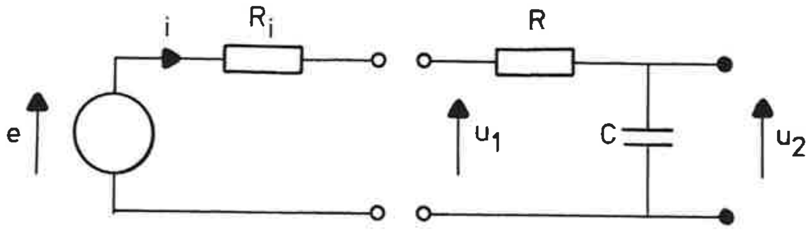


Fig. 2.5

The aditional equation is

$$u_1 = e - R_i i$$

Using the Macro FILTER the system description becomes.

```
E=SIN(T)
U1=E-RI*I
U2,I=FILTER[U1,R,C]
```

This is expanded to

```
E=SIN(T)
U1=E-RI*I
I=(U1-U2)/R
U2=INTEG[I/C,0]
```

These equations can not be sorted for sequential execution. In the second equation U1 is a function of I and in the third equation I is a function of U1. These two equations have to be solved simultanuously. There is an iteration operator in CSSL which can be used. In this case, however, the system of equations are linear and the solution is

$$i = \frac{e - u_2}{R + R_i}$$

$$u_1 = \frac{Re + R_i u_2}{R + R_i}$$

[]

The examples show some benefits with using equations instead of assignment statements when modeling. It is required that the equations can be manipulated into different forms. Linear systems of equations frequently occurs. These could in some cases be solved before computations are performed.

3. MODEL LANGUAGE

This chapter contains a description of a proposed model language. The first six sections describes the basic elements of the language such as submodels, equations, cuts, paths and connection statements. Section seven is devoted to a discussion of some additional features of the language such as conditional statements, indexed elements, loop statements, difference equations, discrete events and model validity.

The description of the language is done as a combination of discussion, examples and syntax description. The syntax is successivly improved. The syntax notation used is described in the appendix. The complete syntax of the language is also given in the appendix.

3.1 Submodels

When developing models for large systems it is advisable to split the system up into a set of well defined subsystems. This subdivision is often done according to the physical structure of the system. Examples of such subsystems are pumps, valves, heat exchangers, tanks, pipes, reactors, destillation columns, motors, generators, transistors, amplifiers, filters, etc.

When a subsystem is isolated the boundaries of the subsystem are first determined. Such a boundary is in fact inherent when defining the basic physical laws. Compare for example the use of "control surfaces" in continum mechanics. To describe the interaction of the subsystem with its environment it is necessary to introduce variables which describes what happens at the boundaries. Such variables are called cut-variables or terminal variables. A typical example from rigid body mechanics is the necessity of introducing reaction forces as cut variables when a part of the rigid body is considered. To describe the model it is also necessary to introduce variables which account for storage of mass energy and momentum in each subsystem. Such variables are called local variables. The cut-variables and local variables are used in the equations describing the subsystem.

The development of the model is greatly simplified by splitting up the system into subsystems. Each subsystem can then be developed separately. It is sufficient to consider the internal behavior of the subsystem and the interaction with its environment. A clear subdivision of the system is also necessary when different persons develop models for different subsystems. The subdivision also increases the possibilities for verifying the models separately and makes the submodels more self-documented.

The language for model description should make it possible to represent the structure of the system in a simple way. There should also be a way to replace a submodel depending on the model-complexity that is wanted. One of the most important advantages with the submodel-concept is the possibility to create model-libraries.

The division of a system into subsystems is done with succesive refinement until all subsystems are so simple that they can be described by equations. The system thus has a hierarchical structure (tree structure) of subsystems.

Example 3.1

A system S1 is considered as composed by three subsystems S2, S3 and S4. The system S3 is split up into S5 and S6. The situation is pictured in Fig 3.1.

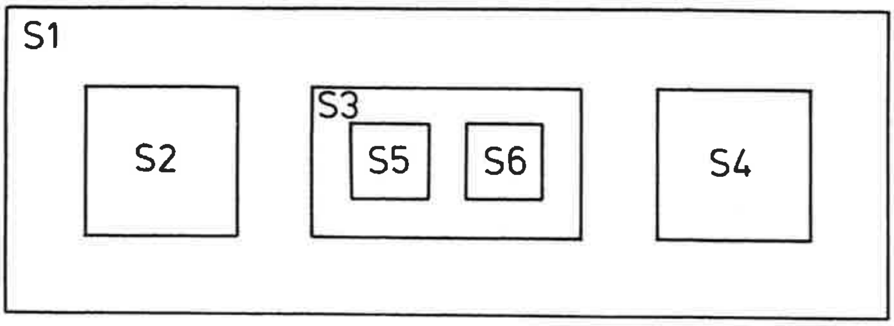


Fig 3.1

This structure can be represented as a tree as shown in Fig 3.2.

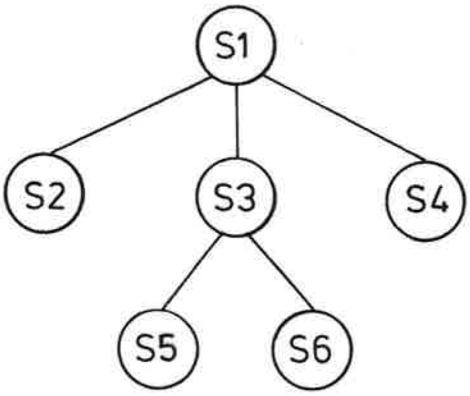


Fig 3.2

[]

The description of a model must include
- the hierarchical structure of the submodels
- the connection structure of the submodels
- the equations

It should be possible to use a submodel when modelling different systems. This implies that each submodel should include a description of its internal structure. The hierarchical structure can be described in the same way as the block structure of Algol.

Example 3.1 (continued)

One way of describing the hierarchical structure of S1 is shown below.

```
model S1
  model S2
  ...
  end
  model S3
    model S5
    ...
    end
    model S6
    ...
    end
  ...
  end
  model S4
  ...
  end
...
end
```

[]

The following pattern for a model is proposed.

```
model <model identifier>

  declaration of submodels
  declaration of variables and connection mechanisms
  equations and description of connection structure

end
```

This way of describing the model hierarchy has two serious drawbacks. If two subsystems have the same model it must be duplicated. The other problem is that a submodel could be a part of a completly different model as well and perhaps be a member of a library.

A way of avoiding these problems is to declare a model type which can be used to generate several models with a submodel statement.

Example 3.1 (continued)

Assume that the systems S2 and S5 have the same model M. A way to simplify the description of S1 is shown below.

```
model type M
...
end
model S1
  submodel (M) S2
  model S3
    submodel (M) S5
    model S6
    ...
    end
  ...
  end
  model S4
  ...
  end
...
end
```

[]

The model type declaration has the same structure as a model declaration.

The submodel statement has the following form.

```
submodel [(<model type identifier>)] {<model identifier>
        [(<parameter list>)]}*
  <parameter  list>::={<number>}*/{<parameter>=<number>}*
```

Ex.

```
submodel Tank Pipe
submodel Tank(A=5 H=10)
```

```
submodel (Tank) Tank1(A=5) Tank2(A=20)
submodel (resistor) R1(5.6) R2(100)                      []
```

The <model type identifier> is given within parentheses followed by the <model identifier>'s. If no <model type identifier> is given it is assumed that it is the same as the <model identifier>.

A parameter list can follow after the <model identifier>. This list is used to set or change default-values for parameters. The parameter list has two forms. The values of the parameters can be given together with the corresponding name of the parameters or they can be given alone in the same order as they are declered in the submodel.

It should be possible to reference submodels (and their variables) on all lower levels. Since it is possible that several models have the same <model identifier> there must be a way to distinguish them. One way of doing this is to follow the path in the submodel tree down to the actual submodel. For this purpose there is a ::-notion which can is used in the following way.

```
<model identifier> {:: <model identifier> }*
```

3.2 Interdependence between submodels

It is possible to distinguish between two types of influence on a submodel from the environment.

In the first case the influence from the invironment comes through distinct mechanisms as e.g. shafts, wires and pipes. It is then practical to introduce variables that describes the coupling through the mechanisms. Such variables are called terminal variables. The coupling between different submodels can then be described by giving relations between the terminal variables in a superior model. This type of coupling is called explicit coupling.

The other case of influence can be thought of as coming from a higher level. Examples of this type of influence are the temperature and pressure of the athmosphere and the temperature of an amplifier influencing all its components. The gravitation field and electrical fields are also examples of this type of coupling. This type of influence can in some cases be described by letting the submodels use common variables declared in the superior model. This type of coupling is called implicit.

3.3 Variables

The behavior of a system is often conceived as the variation of certain quantities. When a model is developed a number of quantities are selected to appear in the model. This selection depends on the complexity of the model. The model contains variables which have correspondance with these quantities. A variable is a function of the time and has an associated name.

Some variables are constant under each computational operation and are called parameters. These are declared in the model by the statement

parameter {<variable> [=<number>] }*

Parameters can be assigned from superior models or interactivly. They can also be computed by static computations or by optimization. If a parameter is not assigned from the outside the default value in the declaration is assumed.

It is also possible to declare variables whose values can not be changed, constants, by the statement

constant {<variable>=<number>}*

The time varying variables are divided into two categories: local variables and terminal variables. The terminal variables describe the interdependence between a submodel and its environment. These types of variables are declared by:

local {<variable>}*
terminal {<variable>}*

There are two special types of terminal variables: input- and output-variables. The value of an input-variable must be given from an equation not included in the same submodel as the declaration. The converse is true for an output-variable. These two types of variables has been introduced to increase the security against bad incorporation of submodels. The declaration of these variables is done with the statements:

input {<variable>}*
output {<variable>}*

Terminal variables are implicitly declared when declaring cuts (see section 3.5).

Models are often developed in a way that it should be possible to use them in different environments. It may then occur that some connection mechanisms are not used. For that reason there is a possibility to give default values to terminal variables. The default value is used if the terminal variable is not externally referenced.

default {<variable> = <number>}*

Submodels can be implicitly connected if they use the same variables. This is accomplished by declaring these variables as *internal* in a superior model. For security reasons the variables must be declared as *external* in the submodels themselves.

internal {<variable>}*
external {<variable>}*

A way of connecting submodels is to give equations relating terminal variables in the submodels from a superior model. Since different variables in different submodels can have the same identifier there must be a mechanism to reference them. A suitable mechanism for that is the dot-notation:

<model identifier>.<variable>

3.4 Equations

When developing a model for a physical system one uses fundamental laws such as mass balance equations, energy balance equations and fenomenological equations. These are either algebraic- or differential-equations which relates certain variables to each other.

There are often conditions in the equations which can be easily entered with the if-then-else construction of Algol. The following form is thus proposed for equations.

<expression> = <expression>

The syntax of the expression is the same as in Algol. The equations can contain ordinary function procedures written in some algoritmic language.

It is also useful to be able to use ordinary procedures written in algoritmic language. In order to allow manipulation of the equations it must be known which variables that are input and which are output for the procedure. A suitable notation for procedure calls is the one used in CSSL and CSMP:

{<variable>}* = <procedure identifier> ({<varible>}*)

Ex. y1 y2 y3 = Proc(u1 u2)

A notation for time derivatives is required to enter differential equations. The following notations are proposed:

first derivative: x' , der(x) , der(x,x0)
second derivative: x'' , der2(x) , der2(x,x0) ,
der2(x,x0,dx0)

etc.

3.5 Cuts and connections

When connecting submodels it is natural to look at the submodel in the same way as the corresponding subsystem. One then wants to work with the physically existent mechanisms that connect the subsystems. Every such mechanism have associated certain variables which are used internally in the equations and which describes the interdependence with other submodels.

Examples of such mechanisms and their associated variables are:

shaft: angle, momentum
pipe: flow-rate, pressure, temperature
electrical line: voltage, current

For the reasons given above there should be a way to name groups of variables in order to simplify the connections. Such groups of variables are composed when defining the boundaries of subsystems by introducing cuts between them. Cuts are declared in the following way (compare above):

```
cut shaft(angle, momentum)
```

The basic concepts are introduced by means of an example.

Example 3.2

Suppose there are two subsystems S1 and S2 which are connected by a pipe with a flow of some liquid, see Fig 3.3. In order to be able to describe the systems separately a cut is defined somewhere along the pipe. The relevant variables to introduce in the cut can e.g. be flow rate (Q), pressure (P) and temperature (T).

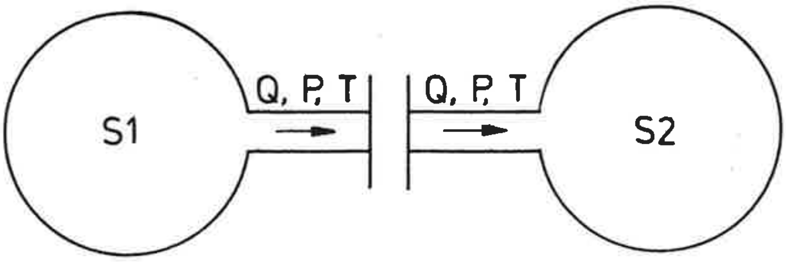


Fig 3.3

The two submodels will contain the cut-declarations:

```
cut outlet(Q ,P, T)
cut inlet(Q, P, T)
```

The variables Q, P and T are terminal variables in both of the submodels. The submodels can then be connected from a superior model in the following way.

```
S1.Q = S2.Q
S1.P = S2.P
S1.T = S2.T
```

Terminal variables are often defined in such a way that connection of subsystems means setting the corresponding variables equal. For this reason there is a special operator, called at, which operates on cuts and which can be used in the following way.

```
S1:outlet at S2:inlet
```

This statement has the same effect as the equations above.

Note that S1.Q is defined as the flow out of S1 but S2.Q is the flow into S2. This problem with reference directions will be solved later.

[]

The discussion in example 3.2 is now summarized. An elementary way to declare a cut is with the statement:

<u>cut</u> <cut identifier> ({<variable>}*)

Submodels can then be connected via the cuts with the <u>connection statement</u>.

<model identifier>:<cut identifier>
{ <u>at</u> <model identifier>:<cut identifier> }*.

The corresponding variables in all cuts are set equal in this way. The same cut can appear in several connection statements. A colon-notation is used when referencing the cuts.

In some cases the connection of submodels do not imply that the cut variables are set equal. This is examplified below.

<u>Example</u>

Consider the electrical circuit in Fig 3.4.

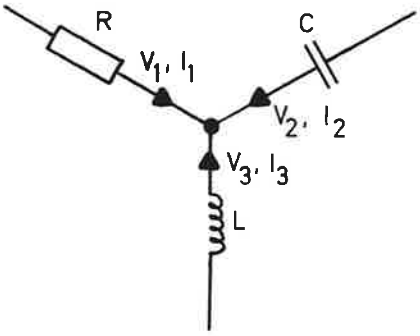


Fig 3.4

The constraints at the connection node are

$V1 = V2 = V3$
$I1 + I2 + I3 = 0$

Only the first equation is of the type discussed earlier.

In order to handle the connection one could of course define a small subsystem with three cuts containing the second equation. This is, however, cumbersome since the number of connected components can vary. A better way is to introduce a new type of variables. The sum of such variables is defined to be zero at a connection point.

Suppose that in all the submodels R, L and C there is defined a cut wirel as

```
  cut wirel (V / I)
```

The / has been used to indicate that I is a variable of the second type. The connection statement

```
  R:wirel at L:wirel at C:wirel
```

then would be equivalent to the following equations

```
  R.V = L.V
  L.V = C.V
  R.I + L.I + C.I = 0
```

[]

Example 3.4

A number of levers are connected as shown in Fig 3.5.

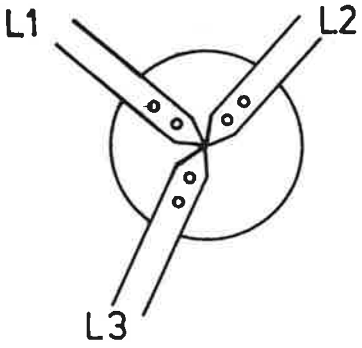


Fig 3.5

If all the levers have a cut endl declared as

```
cut endl (X,Y,Z / Fx,Fy,Fz, Mx,My,Mz)
```

then the connection can be expressed as

```
L1:endl at L2:endl at L3:endl
```

This statement is equivalent to

```
L1.X = L2.X ; L2.X = L3.X
L1.Y = L2.Y ; L2.Y = L3.Y
L1.Z = L2.Z ; L2.Z = L3.Z

L1.Fx + L2.Fx + L3.Fx = 0
L1.Fy + L2.Fy + L3.Fy = 0
L1.Fz + L2.Fz + L3.Fz = 0

L1.Mx + L2.Mx + L3.Mx = 0
L1.My + L2.My + L3.My = 0
L1.Mz + L2.Mz + L3.Mz = 0
```

[]

The examples show that it is practical to introduce two types of cut variables. The notation across variable is sometimes used in the litterature for the variables that are equal in the cuts. The variables that are summed to zero are called through variables.

If the / -sign is used to separate these two types of variables the cut declaration gets the form:

```
cut <cut identifier> ( [<variable>]* / [<variable>]* )
```

The connection statement is the same, only its interpretation is changed.

By introducing through variables there will be a way of handling *reference directions*. An adequate way is to define a common reference direction for all through variables in all cuts. If some variable has the opposite direction it is proceeded by a minus sign in the cut declaration.

Special care must be taken not to introduce *redundant equations* relating through variables. For example *one* of the node equations for current is redundant when connecting electrical components. One way of solving this problem when modelling is of course to introduce a dummy through variable in one of the submodels. The language permits that this dummy variable is replaced by a dot in the cut declaration. When that cut is connected no equation is generated for the corresponding through variables.

Ex. *cut* A(Va / I) B(Vb / .)

Across variables can also be replaced by a dot in the cut declaration. This is sometimes practical when using standardized cuts to show that a submodel is independent of some variable in a cut.

It is sometimes useful to declare cuts without variable specification. If e.g. a model contains submodels and some cuts in the submodels should be available as cuts on the higher level then a cut can be declared as

cut <cut identifier>

This cut can then be connected as

<cut identifier> *at* <model identifier>:<cut identifier>

This notation is a way of avoiding cut references in several model levels.

This type of cut is used to make connections between submodels on different hierarchical levels. It is also useful to introduce cuts to simpilify the connection of submodels on the same level. When connecting electrical components the concept node is used. The connection points or nodes are given names or numbers and each component is connected between a number of nodes.

A node is declared as

node <node identifier>

The only difference from a cut is that a node can not be referenced from the outside of the model.

Hierarchical cuts are sometimes useful. When a number of submodels are joined together in a superior model it may be natural to join the externally available cuts into larger cuts. Another form of the declaration statement for cuts is thus

cut <cut identifier> ({<cut identifier> /
<node identifier>}*)

This declaration is further generalized in the syntax in appendix.

It is possible to use temporary cuts in the connection statement. These are formed in the same way as in the declarations. In this way one gets a natural way to connect components between nodes. This is demonstrated in the following example.

Example 3.5

Suppose there is a resistor defined as

```
model resistor
  cut wire1(V1 / I) wire2(V2 / -I)
  cut conn(wire1, wire2)
  parameter R
  R*I = V1-V2
end
```

Such resistors can be incorporated in a network as

```
model network
  submodel (resistor) R1 R2
  node N1 N2 N3
  R1:conn at (N1,N2)
  R2:conn at (N2,N3)
  .
  .
end
```

[]

It is also possible to use temporary cuts to connect a submodel between other specified submodels.

Example 3.5 (continued)

Specifying a resistance in a network can be done as

```
R1:conn at (C1:wire2, L1:wire1)
```

[]

The concept *main cut* is introduced to simplify the connection statements. The main cut of a model is specified by prefixing the cut declaration with the word *main*. When referencing the main cut only the <model identifier> need to be given.

Example 3.5 (continued)

The declaration of the cut conn in model resistor is modified as

```
main cut conn (wire1,wire2)
```

The statement

```
R1 at (N1,N2)
```

then becomes equivalent to

```
R1:conn at (N1,N2)
```

[]

3.6 Description of structures

The previous sections have shown how the relations between variables in different submodels can be given either directly via the dot-notation or by using cuts and the at-operator. The at-operator allows models to be connected in arbitrary structures. The connection statements, however, often becomes hard to read and do not contain the structure of the model themselves. This section treats an alternative way of describing the coupling between submodels.

It is often natural to say that a number of systems are coupled after each other in some sence. This is e.g. the case when something flows through several systems. Such a structure is naturally described as

```
System1 to System2 to System3
```

There can be several paths through a system which one should be able to follow separately as e.g.

```
connect (water) S1 to S5 to S7
connect (steam) S2 to S5 to S9 to S3
```

Only simple paths through a system are considered since there is only a left hand side and a right hand side of a model identifier written on a line. In order to use the to-operator there must be declared paths through the systems. This is done as

```
path <path identifier> (<cut> - <cut>)
```

```
Ex.     path water (inlet - outlet)
```

If a path is branched or if several paths are joined together in a model this can be expressed using hierarchical cuts. Several paths can be declared in a model. To reference them from the outside it is possible to use the notation

<system identifier>..<path>

It is possible to introduce a main path through a model in the same way as main cuts. The main path is specified by prefixing the path declaration by the word main. The reference to a main path can be done by just using the model identifier. Cuts and paths can have the same identifier. The path reference is chosen if both a cut- and path-reference is legal and just the model identifier is given.

Description of a stucture is naturally done by following paths through the models one at a time. This is simplified if the connection statement is preceeded by

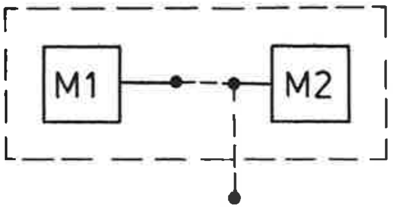
connect (<path identifier>)

Path references in the connect statement which are done with just the model identifier do not refer to the main paths in this case. They refer to the named path. The same holds for cut references.

Ex.

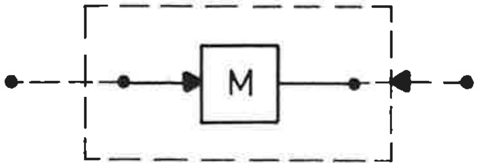
```
  connect (water) Tank1 to Pipe to Tank2
is equivalent to
  Tank1..water to Pipe..water to Tank2..water                    []
```

Generally the connection of the submodels is done with a connection statement. This statement can be compared with an ordinary arithmetic expression. The operands in the statement are cuts and paths which are specified according to the rules given before. The operators are at, reversed, to, from, par and loop. The operators are illustrated in Fig 3.6 - 3.11.



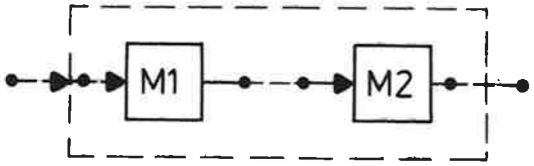
M1 at M2

Fig 3.6



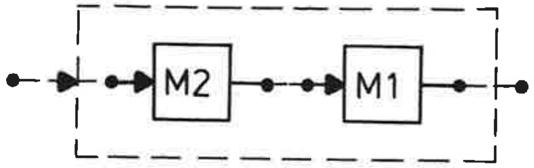
reversed M

Fig 3.7



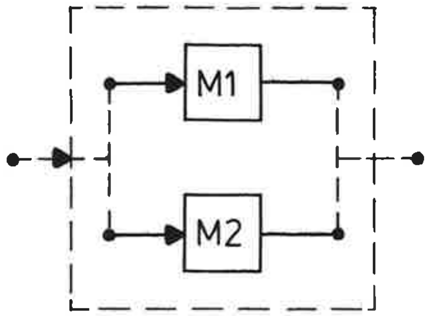
M1 to M2

Fig 3.8



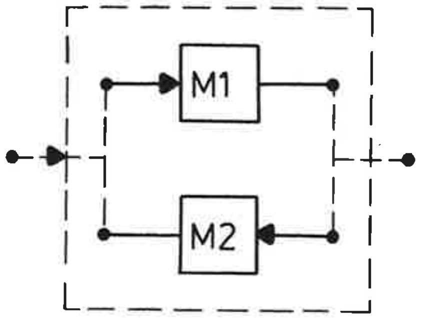
M1 from M2

Fig 3.9



M1 par M2

Fig 3.10



M1 loop M2

Fig 3.11

Note that the operator loop is equivalent to par reversed.

The evaluation of the connection statement is done from left to right if not otherwise stated by parantheses. A simple way of explaining the connection statement is to show how it will be treated. The statement can be evaluated using a stack. The operations delivers values, either cuts or paths. The actual coupling between submodels occurs as side effects which can be described as at-operations. The value of a connection statement is not used. Table 1 describes the operators. The notations C1, C2, etc. has been used for cuts and the notations (C1 - C2), etc. has been used for paths.

In order to shorten the connection statements e.g. when describing electrical networks the following alternative notations are proposed.

at	=
reversed	\
to	-
from	<
par	//

Table 3.1. Evaluation rules for connection operators

	Operation	Result	Effect
1.	C1 *at* C2	C2	C1 *at* C2
2.	*reversed* (C1 - C2)	(C2 - C1)	none
3.	(C1 - C2) *to* (C3 - C4)	(C1 - C4)	C2 *at* C3
	C1 *to* (C2 - C3)	C3	C1 *at* C2
	(C1 - C2) *to* C3	C1	C2 *at* C3
4.	(C1 - C2) *from* (C3 - C4)	(C3 - C2)	C1 *at* C4
	C1 *from* (C2 - C3)	C2	C1 *at* C3
	(C1 - C2) *from* C3	C2	C1 *at* C3
	C1 *from* C2	none	C1 *at* C2
5.	(C1 - C2) *par* (C3 - C4)	(C1 - C2)	C1 *at* C3
			C2 *at* C4
	C1 *par* C2	C1	C1 *at* C2
6	(C1 - C2) *loop* (C3 - C4)	(C1 - C2)	C1 *at* C4
			C2 *at* C3

Example 3.6

Consider the electrical network in Fig 3.12.

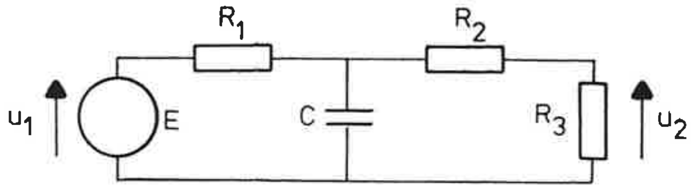


Fig 3.12

For this system it is easy to write down the equations directly. One model is the following.

```
model network
  input u1
  output u2
  local i1 i2 vc
  parameter R1 R2 R3 C

  u1=R1*i1+vc
  vc=R2*i2+R3*i2
  u2=R3*i2
  C*vc'=i1-i2

end
```

An other approach when modelling the network is to develop a library of electrical components which is then connected together. This system contains three different components: resistors, a capacitor and a voltage source.

The following library of models would then be useful.

```
model resistor
  cut A(Va / I) B(Vb / -I)
  main path P (A - B)
  local V
  parameter R

  V = Va-Vb
  R*I = V

end


model capacitor
  cut A(Va / I) B(Vb / -I)
  main path P (A - B)
  local V
  parameter C
```

```
    V = Va-Vb
    C*V' = I

  end


  model voltage
    cut A(Va / -I) B(Vb / .)
    main path P (A - B)
    input V

    Va = V
    Vb = 0

  end
```

Using this library of components, the network can be described in the following way.

```
  model network
    submodel(resistor) R1 R2 R3
    submodel(capacitor) C
    submodel(voltage) E

    R1 to ( C par ( R2 to R3 ) ) par E

  end
```

3.7 Additional features of the language

The previous sections of this chapter have described the basic elements of a model language for continuous dynamical systems. This section is devoted to a brief discussion of some additional features which would be useful when modelling systems. The list of features is by no means complete. More experience of the use of the language is needed to define it completely.

Conditional statements

The model of a system depends on the phenomena of interest. When collecting submodels to form a complete model it is very important that the submodels are compatible in this respect. There could thus be several models of a system in a submodel library which describes different aspects of a system. However, in many cases the differences between the models are small. It could be a matter of which approximations are made. In this case it would be natural for the modeller to include conditional statements in a model. Different models can then be selected by using some kind of structural parameters.

Some of the cases can be handled by the if-then-else construction in the equations. However, even the declarations can be conditional. The problem can be solved by using an if-then-else statement or a case statement.

Consider the simulation problem. If the conditions only depend on parameters the set of equations and variables are the same during one simulation run. This means that the transformation of the equations, which is discussed in later chapters, only has to be done once before the simulation starts.

In some cases it is natural to let the model equations depend on the operating region of the model. If the solution crosses the boundaries during a simulation then the integration algorithm must compute the crossing point and then the new model should be determined. The transformation of the equations must eventually be done at such points.

Indexed elements and loops

It is obvious that a model language should include indexed variables such as vectors and matrices. It should also be possible to operate on them using a generalized assignment statement.

There are examples when it is desirable to index cuts. Consider for example a mechanical system which is built up from levers. Each lever has a number of holes and the levers can be connected by bolts through the holes. To describe such a system it is convenient to make only one model of a lever, declaring an indexed cut hole[n] which could then be referenced as e.g. lever1:hole[3]. The equations in the model will not be the same for different number of holes. However, it is easy to incorporate the equations using a loop statement.

The following example show the use of indexed submodels and the use of a loop statement.

Example 3.7

Consider the problem of modelling a heat exchanger. A heat exchanger is probably most easily described by partial differential equations. Sufficiently good approximations can, however, be obtained by deviding the heat exchanger into a number of sections each described by ordinary differential equations (see Fig 3.13).

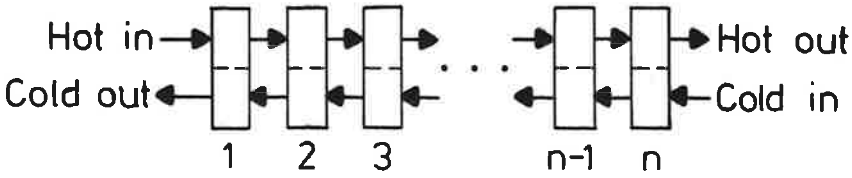


Fig 3.13

The degree of approximation depend of the number of sections. This means that the connection of the sections should be done in a way that it is easy to change the number of sections. This number can also be large. These facts indicate that there should be some loop statement to use when connecting the sections. One way of modelling the heat exchanger is shown below.

```
model type section
  path hot ( ... )
  path cold ( ... )
  ...
end

model heatexchanger
  structure parameter n
  submodel section[n]
  cut hotin hotout coldin coldout
  path hot (hotin - hotout)
  path cold (coldin - coldout)
  ...
  for i:=1 to n do
    begin
    connect (hot) section[i] to section[i+1]
    connect (cold) section [i+1] to section[i]
    end

  hotin to section[1]..hot
  section[n]..hot to hotout
```

```
    coldin to section[n]..cold
    section[1]..cold to coldout

  end
                                                []
```

Difference equations and discrete events

The demand for combined continuous - discrete simulation languages has increased. One of the reasons is the need to simulate computer controlled processes. A computer and its programs can be modelled as a discrete event model. It is, however, interesting to consider the special case difference equations or discrete time models since the basic concept when designing digital controllers is difference equations. Difference equations are also used to model economical systems. A discrete time model could have the same structure as a continuous model and the same facilities to transform the equations could be incorporated.

The discrete event models appear at a superior level to continuous and discrete time models. Discrete events can be triggered by e.g. a variable passing a limit in a continuous model. The event on the other hand can change variables and even change the equations of a continuous model. One way of handling this situation would be to make an interface between the discussed model language and languages for discrete event simulation as e.g. Simula.

The problem of handling changes of model equations is easily done with the structural parameters appearing in the if-then-else-, case- and for-statements as discussed previously. There should be some mechanism to manipulate such parameters from a discrete event model.

Model validity

The proposed model language simplifies the creation of model libraries which can be used by different persons. Since a model is not a complete description of the real world erroneous results can be obtained by using a model in a wrong way. To overcome this problem the models must have good documentation. In some cases the test for suitability can be done automatically. This is the case with the numerical region of validity. It should be possible to express that a model is valid only if certain conditions on the variables are fullfilled. For this purpose the following statement is proposed.

valid <Boolean expression>

Conditions can be given on parameters, variables and derivatives. Conditions on the derivatives can be used to state that a model is valid in a certain frequency range.

4. OPERATIONS ON THE MODEL

4.1 Mathematical notation

The total model is composed of three types of equations.

- The equations in the submodels
- Equations of the type
  $v_i = v_j$
  for across variables
- Equations of the type
  $\pm v_i \pm v_j \pm \ldots = 0$
  for through variables

The two last types of variables are introduced by the cut and path operations.

In order to get a simple mathematical notation for the model all higher order derivatives are eliminated. This is done by introducing auxiliary variables and extra equations.

From a system theoretical point of view it is interesting to distinguish variables that are considered as inputs and outputs for the total system.

A mathematical notation for the models described in the model language is

$$f(t, x', x, z, u, y, p) = 0 \qquad (4.1)$$

where
t - time
x - variables that appears differentiated

u - inputs
y - outputs
p - parameters
z - other variables

Derivatives on u and y has been eliminated by introducing auxiliary variables.

4.2 Linearization

There is a well developed theory which treats linear systems. It is thus interesting to develop linearized models from the basic equations. Suppose the model should be linearized along a reference path defined by the functions $x_0(t)$, $z_0(t)$, $u_0(t)$ and $y_0(t)$. Introduce the deviations

$$Dx(t) = x(t) - x_0(t) \tag{4.2}$$

etc.

Insertion into the model (4.1) gives.

$$f(t,\ x_0'+Dx',\ x_0+Dx,\ z_0+Dz,\ u_0+Du,\ y_0+Dy,\ p) = 0$$

Linearization gives

$$f(t,x_0'(t),x_0(t),z_0(t),u_0(t),y_0(t),p) +$$
$$\frac{df}{dx'}(.)Dx' + \frac{df}{dx}(.)Dx + \frac{df}{dz}(.)Dz + \frac{df}{du}(.)Du + \frac{df}{dy}(.)Dy = 0$$

The arguments of the Jacobians are the same as for f. If new notations are introduced the linear model can be written.

$$A(t)Dx' + B(t)Dx + C(t)Dz + D(t)Du + E(t)Dy + F(t) = 0$$

If $x=x_0$, $z=z_0$, $u=u_0$ and $y=y_0$ is a solution to the original model then $F(t)=0$. In some cases the matrices are constant. The linear model is then:

$$ADx' + BDx + CDz + DDu + EDy = 0 \tag{4.3}$$

4.3 State equations

The original model can be simulated directly. This will be demonstrated in chapter 5. Many integration methods are developed for the system

$$x' = f(x,t)$$

Other methods for analysing dynamical systems also use this form. These are reasons for transforming the model to state space form if possible.

If

$$\det\left[\frac{df}{dx'} \quad \frac{df}{dz} \quad \frac{df}{dy}\right] (.) \neq 0$$

then it is possible to find functions F,G and H such that locally

$$\begin{aligned} x' &= F(t,x,u,p) \\ z &= G(t,x,u,p) \\ y &= H(t,x,u,p) \end{aligned}$$

Practically it is often sufficient to permute the equations and to solve variables from x', z and y one at a time. There may of course be systems of equations that have to be solved simultanuously but they are often linear. Methods to find the permutations are discussed in section 5.2. In some cases it is possible to find a state space form even if the determinant vanishes (see section 5.3).

Decomposition

A model in state space form can be written as

$$x' = f(t,x,u,p)$$

if the auxiliary variables and outputs are not conserned.

For control purposes it is sometimes interesting to split up the system into subsystems with the structure in Fig 4.1.

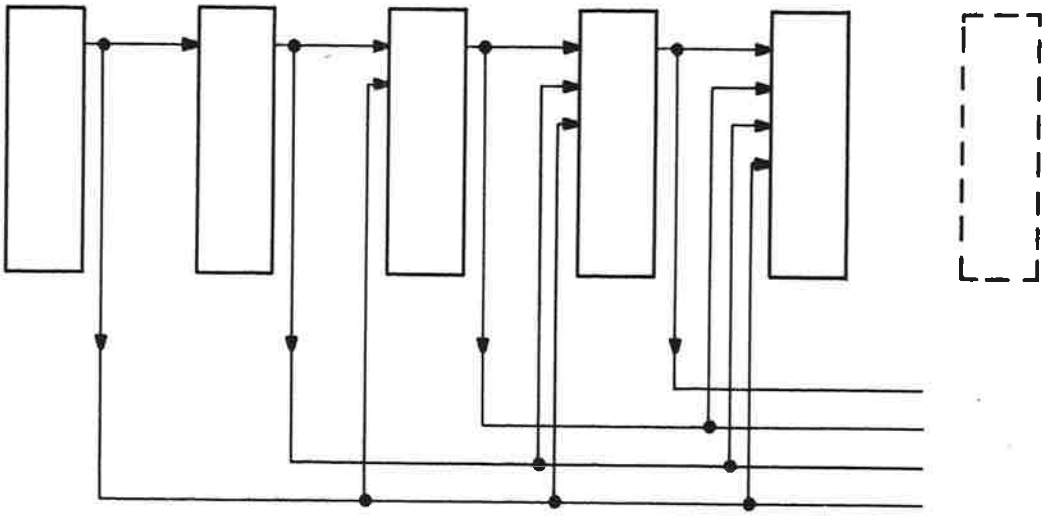


Fig 4.1

It can e.g. be easier to introduce hierarchical control if it is possible to find such a structure. This problem is discussed in Sato, Ichikawa (1967) and Aulin (1969).

The problem can be formulated as to find a permutation matrix P such that

$$P\frac{df}{dx}P^{-1}$$

becomes block triangular. The blocks correspond to the subsystems above. This problem is further discussed in chapter 5.

Structural controllability and observability

It is difficult to see how different variables are influencing each other when the model consists of many equations. When designing regulators heuristically such information is very important. It is not only direct influence which is of interest because the influence of a variable can sometimes be seen only in its derivatives. The information can be presented as the lowest derivative which is influenced. Information of this kind can easily be obtained with a computer and presented to the user as e.g. tables.

4.4 Computations

The original model (4.1) may be used to obtain other models or transformed models. Simplified models may for example be generated by neglecting certain dynamics. This may be done simply by replacing dynamic equations by static equations. The original model may also be used to calculate equilibrium values, to obtain linearized equations and equations for the inverse of the system. Some of these calculations are briefly described below.

The description of the computations are done using the model notation

$$f(t, x', x, z, u, y, p) = 0$$

Simulation

For simulation it is assumed that the inputs u, the parameters p and an initial value of x are known. The purpose is to calculate the time responses of x, z and y.

If x is not a state vector there may be conflicts between the equations and the given initial values. The problem can then in some cases be defined such that the equations are only valid for $t>t_0$. This situation correspond to connecting different subsystems with known initial values at $t=t_0+\varepsilon$.

An other way of specifying the problem is to give initial value to only a part of x and use the equations to compute the other part. After that the integration can take place.

Computation of initial values

In many cases it is unnatural to give initial values to x. The variables x is only characterized by the appearance of their derivatives. It should be possible to give initial values on some variables from x, z and y. Introduce the notation x_1, z_1 and y_1 for these variables. Some equations should be selected which after transformation gives the initial value for the rest of x, x_2.

$$x_2(t_0) = g(t_0, x_1(t_0), z_1(t_0), u(t_0), y_1(t_0), p)$$

The structure of the equations gives constraints on the selection of x_1, z_1 and y_1. If x is a state vector the following must also hold.

$$\dim(x_2) = \dim(z_1) + \dim(y_1)$$

The dimension of x_1 can of course be zero. When x_2 is computed then initial value is known for the entire x and the simulation can be done. The equations are in this way assumed to be valid also for $t=t_0$.

Optimization of dynamical systems

Many problems in system theory can be expressed as optimization problems. Typical examples are model fitting, parameter estimation, regulator tuning and optimal control. The optimization problem can be formulated as follows. Given the model

$$f(t, x', x, z, u, y, p) = 0$$

Find values of the parameters such that the criterion

$$J = \int_{t_0}^{t_f} g_1(t,x',x,z,u,y,p)\,dt + g_2(t_f,x',x,z,u,y,p)$$

is minimal subject to

$$h_1(t_f,x'(t_f),x(t_f),z(t_f),u(t_f),y(t_f),p) = 0$$
$$h_2(t_f,x'(t_f),x(t_f),z(t_f),u(t_f),y(t_f),p) \geq 0$$
$$x(t_0) = x_0(p)$$

Some of the parameters p_1 are fixed. The equations h_1 (and f) may contain dependene between parameters. The optimization becomes more effective if there are as few constraints as possible. The first problem is thus to select a set of parameters p_3 such that the rest p_2 can be solved from the equations. Algorithms for this are discussed in section 5.2.

The optimization problem can now be formulated as: Select values on p_3 such that some variable z_i is minimized subject to

$$f'(t,x',x,z,p_3) = 0$$

$$h_1'(t_f,x'(t_f),x(t_f),z(t_f),p_3) = 0$$
$$h_2(t_f,x'(t_f),x(t_f),z(t_f),p_3) \geq 0$$

Many optimization algorithms need the Jacobians

$$\frac{dz_i}{dp_3}(t_f), \quad \frac{dh_1'}{dp_3}(t_f) \text{ and } \frac{dh_2}{dp_3}$$

The two last Jacobians can be computed directly. The derivative of the loss function can be obtained by numerical differentiation. The equations must then be integrated for different values of p_3. The derivative can sometimes be obtained more efficient by solving an adjoint equation for dz_i/dp_3. This equation is obtained by differentiating f' with resect to p_3.

$$\frac{df'}{dx'}\frac{d}{dt}\left(\frac{dx}{dp_3}\right) + \frac{df'}{dx}\frac{dx}{dp_3} + \frac{df'}{dz}\frac{dz}{dp_3} + \frac{df'}{dp_3} = 0$$

The initial value for this differential equation is

$$\frac{dx}{dp_3}(t_0) = \frac{dx_0}{dp_3}$$

The integration of the original equations and the equations for dx/dp_3 is done at the same time. This can be done in a special way to increase the efficience (see Gear, 1972).

Static model

A static model is obtained by setting $x'=0$.

$$f(t,0,x,z,u,y,p) = 0$$

The variables x,z and y should be solved when t,u and p are given.

Static design

In static design certain variables like the operating point are specified. The equations are then used to compute the other variables. This can formally be written as

$$\begin{bmatrix} x_2 \\ z_2 \\ u_2 \\ y_2 \\ p_2 \end{bmatrix} = g(t,x_1,z_1,u_1,y_1,p_1)$$

The static model is clearly a special case of this.

Static optimization

Static optimization of a model is a special case of the dynamical optimization.

4.5 Symbolic manipulation

The transformations discussed so far have been made to effectivize different types of computations. There is also a need to obtain the manipulated equations in symbolic form. This is easily done by the computer. The problem is that the variables used have been chosen to give good descriptions of the subsystems. They can, however, have a bad meaning when considering the total system. Many variables are introduced in order to simplify the connections by using cuts. In that way there are many equations of the type $v_i=v_j$. Before outputting the equations in symbolic form the names of the variables could be modified and some variables may be eliminated by the user.

The equations are best prepared interactivly. The basic equations are obtained from the connections of submodels. If these submodels are used in a correct way the equations are correct. The equations can then be manipulated interactivly in such a way that the correctness is maintained. Some examples of desired operations are

- change variable identifier
- make substitution of a variable
- differentiate an equation

Example 4.1

Consider the equations for the network in example 3.6.

The state equations are obtained if the parameters R_1, R_2, R_3 and C, the state v_c and the input u are assumed known. The equations are solved for i_1, i_2, u_2 and v_c'.

$$i_1 = (u_1-u_2)/R_1$$

$$i_2 = v_c/(R_2+R_3)$$

$$u_2 = R_3*i_2$$

$$v'_c = (i_1-i_2)/C$$

5. COMPUTATIONAL METHODS

This chapter contains a brief discussion of some of the computational methods needed for operation on the model.

5.1 Integration

The basic operation on the model

$$f(t,x',x,z,u,y,p) = 0 \qquad (5.1)$$

is simulation, i.e. solution of x(t), z(t) and y(t) when u(t) and p is known.

Almost all integration algorithms are solving the equation

$$x' = f(t,x) \qquad (5.2)$$

In order to use methods for (5.2) on the model (5.1) it is required that

$$\det \left[\frac{df}{dx'}, \frac{df}{dz} \frac{df}{dy}\right] \neq 0$$

along the trajectory. This condition is not always fulfilled. This is the case in example 5.1.

It is thus interesting to solve (5.1) directly. Algorithms for this can be found in Gear (1971,1972), Brown and Gear (1973), Hachtel, Brayton, Gustavson (1971) and Brayton, Gustavson, Hachtel (1972). These algorithms are implicit multi step methods. In the special case when the order of the method is one the derivative is approximated by a backward difference.

$$x'(t_n) \approx \frac{x(t_n) - x(t_{n-1})}{h}; \qquad h = t_n - t_{n-1}$$

This is inserted into (5.1) to get

$$f(t_n, \frac{1}{h}(x(t_n)-x(t_{n-1})), x(t_n), z(t_n), u(t_n), y(t_n), p) = 0$$

This equation can e.g. be solved by using Newton-Raphson technique. Introduce the notation $x_n = x(t_n)$.

$$\left(\frac{df}{dx'}\frac{1}{h} + \frac{df}{dx}\right)(x_n^{m+1}-x_n^m) + \frac{df}{dz}(z_n^{m+1}-z_n^m) + \frac{df}{dy}(y_n^{m+1}-y_n^m) = -f$$

The matrices df/dx', df/dx, df/dz, df/dy and the vector f all have the argument

$$(t_n,\ \frac{1}{h}(x_n^m-x_{n-1}),\ x_n^m,\ z_n^m,\ u_n,\ y_n^m,\ p) \qquad (5.3)$$

The iteration index is m.

In order to solve x, z and y the following condition must be fulfilled

$$\det\begin{bmatrix}\frac{df}{dx'}\frac{1}{h} + \frac{df}{dx} & \frac{df}{dz} & \frac{df}{dy}\end{bmatrix} \neq 0$$

The Jacobians all have the argumentlist (5.3).

This condition is different from the one that was necessary for transformation to state space form because df/dx' has been replaced by df/dx'/h + df/dx.

<u>Example 5.1</u>

In order to study some of the characteristics of the integration algorithm, the following system is studied.

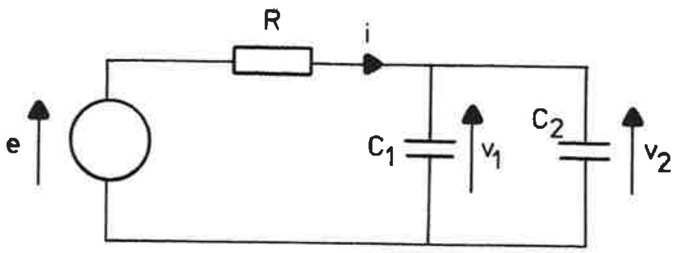


Fig 5.1

A model for this system is

$$e = Ri + v_1$$

$$i = C_1 v_1' + C_2 v_2'$$

$$v_1 = v_2$$

The derivative is approximated by

$$v'(t_n) \approx \frac{v(t_n) - v(t_{n-1})}{t_n - t_{n-1}} \equiv \frac{v^n - v^{n-1}}{h}$$

Simple calculations give the following difference equations.

$$v_1^n = \frac{1}{h + R(C_1+C_2)} (RC_1 v_1^{n-1} + RC_2 v_2^{n-1} + he^n)$$

$$v_2^n = \frac{1}{h + R(C_1+C_2)} (RC_1 v_1^{n-1} + RC_2 v_2^{n-1} + he^n)$$

$$i^n = \frac{1}{h + R(C_1+C_2)} (-C_1 v_1^{n-1} - C_2 v_2^{n-1} + (C_1+C_2)e^n)$$

$n = 1,2,\ldots$

The capacitors will all get the same voltage after one iteration.

$$v_1^n = v_2^n \;; \quad n \geqslant 1$$

The following equation is obtained if $h << R(C_1+C_2)$ (the time constant).

$$v_1^1 = v_2^1 = \frac{C_1}{C_1 + C_2} v_1^0 + \frac{C_2}{C_1 + C_2} v_2^0$$

This is true since charges are moved from one capacitor to the other.

For $n \geqslant 2$ it follows that

$$v_1^n = \frac{1}{h + R(C_1+C_2)} (R(C_1+C_2) v_1^{n-1} + he^n)$$

or equivalently

$$R(C_1+C_2)\frac{1}{h}(v_1^n - v_1^{n-1}) + v_1^n = e^n$$

This difference equation corresponds to the differential equation

$$R(C_1+C_2)v_1' + v_1 = e$$

which is obtained if the capacitors are replaced by one with the capacitance C_1+C_2.

[]

In this particular example it is thus clear that the implicit integration method will give a proper solution and that it is not necessary to convert the system to state space form for simulation.

5.2. Transformation of the equations

Different operations on the model were discussed in chapter 4. This section contains methods to transform the equations to simplify the calculations. The equations can be written in the following form independent of what operation should be done.

$$f(x,y) = 0$$

The known variables are denoted y and the unknown by x. The vectors x and y contains different variables depending on the operation desired.

A numerical solution can be obtained by Newton-Raphson technique. The Jacobian has, however, often a simple structure. It is frequently sparse and many of its nonzero elements are constant. This can be used to make the calculations more efficient. In some cases the system of equations are so simple that the variables can be solved sequentially one at a time. This corresponds to the case when the Jacobian can be made triangular by permuting equations and variables independently. Such transformations are very important if the system of equations should be solved by formula manipulations.

Many of the methods to transform the equations are formulated using graph theory. The basic methods use only the structure of the equations, i.e. whether the elements of the Jacobian are identically zero or not. This information can be put into a bipartite graph. A bipartite graph contains two sets of nodes, which in this case are the equations and the variables. Edges must not connect two nodes from the same set of nodes. An edge between an equation node and a variable node means that the variable is present in the equation.

Example 5.2

Consider the following system of equations

$f_1(x_2,x_4) = 0$

$f_2(x_3) = 0$

$f_3(x_1,x_3) = 0$

$f_4(x_1,x_2,x_3,x_4) = 0$

These equations can be represented structurally by the bipartite graph in Fig 5.2.

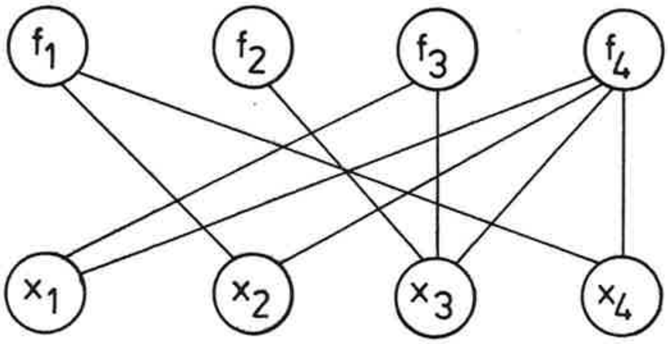


Fig 5.2

Output set

The system of equations will in the following sections be denoted by $f(x) = 0$. To find an output set means that each equation is associated with one and only one variable. This can also be seen as to transform the bipartite graph to a directed bipartite graph such that each equation node has one outgoing edge and each variable node has one incoming edge. The problem is equivalent to finding a permutation matrix which permutes the equations

$g(x) = Pf(x)$

such that

$(\frac{dg}{dx})_{ii} \neq 0$

A necessary condition for the equations to have a solution is that there exits an output set.

Algorithms for finding an output set can be found in Steward (1962) and Wiberg (1977).

Partitioning

Partitioning is used to permute both equations and variables independently in a way that the variables can be solved sequentialy.

Two permutation matrices are wanted, one that permutes the equations, P and one that permutes the variables, Q, i.e.

$$g(y) = Pf(x) \; ; \quad x = Qy$$

They should be chosen in a way that the matrix

$$\frac{dg}{dy} = P \frac{df}{dx} Q$$

becomes block triangular with minimal blocks.

If all the blocks are scalar and all the equations g_i are linear functions of y_i then the equations can be transformed as

$$g_i(y_1,\ldots,y_i) = h_{i1}(y_1,\ldots,y_{i-1}) + y_i h_{i2}(y_1,\ldots,y_{i-1})$$

The variable y_i is easily solved from this equation. All the variables can in this case be solved successivly from the equations. If $h_{i2}=0$ the problem is ill posed.

Nonscalar blocks in the permuted Jacobian correspond to systems of equations that must be solved simultaneously. Special methods can be used if the equations are linear in the unknown variables. Newton-Raphson technique can be used to solve nonlinear equations.

Algorithms for partitioning can be found e.g. in Steward (1965) and Wiberg (1977). Wiberg also gives a comparison between some algorithms. Some of the algorithms first finds an output set. The equations and the variables are then permuted by the same permutation matrix.

Tearing

Tearing is a method to decrease the number of iterated variables when solving systems of equations with iterative technique.

The problem can be formulated as follows. Find a partitioning of the variables and the equations, and permutation matrices P and Q, such that

$$\begin{bmatrix} g_1 \\ g_2 \end{bmatrix} = P\,f \quad ; \; x = Q \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

The system of equations can then be written as

$$g_1(y_1, y_2) = 0$$

$$g_2(y_1, y_2) = 0$$

The criterion for the partioning and permutation can be e.g to make dg_1/dy_1 triangular or block triangular with blocks corresponding to linear systems of equations. The dimension of y_2 should be chosen as small as possible. The equations are solved by iterating over y_2. y_1 is solved from g_1 and substituted into g_2.

Combinatorical problems occurs when the dimension of y_2 is high. Algorithms for tearing has been given in Steward (1965), Lee, Christenson, Rudd (1966), Christenson (1970) and Stadther, Gifford, Scriven (1974). In Ledet, Himmelblau (1970) there is an algorithm that not necessarily gives the minimal dimension of y_2 but could be useful for tearing large systems.

Design variable selection

In some cases the system of equations is underdetermined.

$$f(x) = 0 \; ; \quad \dim f < \dim x$$

One problem then is to select dim x - dim f variables called design variables in such a way that the other variables can be solved as a function of the design variables. In some cases the problem is combined with tearing.

The variables and the equations are permuted and partitioned as follows.

$$g_1(y_1, y_2, y_3) = 0$$

$$g_2(y_1, y_2, y_3) = 0$$

The design variables are denoted by y_3, y_2 are torn variables for iteration. The variables y_1 are solved from g_1. The criterion for the selection can be that dg_1/dy_1 should be triangular and that the dimension of y_2 should be as small as possible.

Lee, Christenson, Rudd (1966) gives an algorithm without tearing which works when dg_1/dy_1 can be made triangular. Christenson (1970) gives an algorithm which includes tearing. Stadther, Gifford, Scriven (1974) allows that dg_1/dy_1 is block triangular and the types of the blocks can be specified.

Sparse matrix technique

Sparse matrix techniques can be used as an alternative to tearing in order to speed up the solution of large systems of equations. When the coefficient matrix or the Jacobian is sparse there are special methods to store the Jacobian and to solve the system of equations. There are e.g. other types of permutations than the ones discussed earlier which can make the computations more effective. A good overview of such methods can be found in Tewarson (1973).

5.3 State equations

Consider the basic model

$$f(t,x',x,z,u,y,p) = 0$$

It was mentioned in chapter 4 that there are many reasons for having the model in state space form. If the states are chosen as x, i.e. the variables which appear differentiated, then it is often possible to use the partitioning algorithm to obtain the state space form. If the dynamical order of the system is less than dim x or if elements of z and y are chosen as states then it is not possible to get the state space form directly. In many cases it is possible if the model is extended by some of the equations differentiated.

Pernebo (1977) gives a general algorithm for finding the state space form of a linear time invariant system. This section outlines an algorithm for nonlinear systems to determine which equations to differentiate if the state variables have been specified.

Assume that the vectors x, z and y are partitioned in two parts in such a way that the state vector is $[x_1 \ z_1 \ y_1]$. The problem is then to transform the model

$$f(t,x_1',x_2',x_1,x_2,z_1,z_2,u,y_1,y_2,p) = 0$$

to

$$\begin{bmatrix} x_1' \\ x_2' \\ x_2 \\ z_2 \\ y_2 \\ z_1' \\ y_1' \end{bmatrix} = F(t,x_1,z_1,y_1,u,u',\dots,p)$$

In order to do this transformation dim x_2 more equations are needed.

The determination of the variables which have to be differentiated is done at the same time as the partitioning. Introduce the notation v_i for the variables and derivatives which are output set in the i:th block. After the i:th block has been processed the situation is

$$v_1 = F_1(t,x_1,z_1,y_1,u,p)$$

$$\cdot$$

$$\cdot$$

$$v_i = F_i(t,x_1,z_1,y_1,u,u',\dots,p,v_1,\dots,v_{i-1})$$

When a block has been found it is tested if any element of v_i appears differentiated in the equations. If not so the algorithm seeks the next block of equations else all the equations in the block is differentiated.

$$v_i' = G_i(t,x_1,x_1',z_1,z_1',y_1,y_1',u,u',\dots,p,v_1,v_1',\dots,v_{i-1},v_{i-1}')$$

Since variables in previous output sets now appears differentiated the corresponding equations have to be differentiated. This step is performed until there is no more new variables appearing differentiated. The variables v_i' can formally be expressed as

$$v_i' = H_i(t,x_1,x_1',z_1,z_1',y_1,y_1',u,u',\ldots,p)$$

All the differentiated equations are added to the model. If the blocks are scalar and m equations have been differentiated then m-1 new variables (not previously appeared derivatives) have been introduced. Since this situation will appear dim x_2 times if the blocks are scalar then dim x_2 new equations are generated which was needed. If an equation is differentiated which has a variable which not appears differentiated as output set then as many new equations as new variables is generated. The only way to obtain dim x_2 new equations if differentiation is performed after the blocks have been found is thus to differentiate variables (equations) which appear differentiated.

The discussion can be applied also to nonscalar blocks after solving the corresponding system of equations formally. Practically all the equations in a block are differentiated if any of the output varaibles appears differentiated. The reason is that the derivatives will appear in a system of equation with the same structure as the original. It is thus not possible to partition this system of equations and solve only for the unknown derivatives. This fact can be seen by studying a block of equations.

$$f(v) = 0$$

These equations are differentiated.

$$f_v(v)v' = 0$$

The derivatives v' should be solved from these equations. The Jacobian with respect to v' is $f_v(v)$, i.e. the same Jacobian as for the original system of equations. They thus have the same structure.

The algorithm is demonstrated on an example.

Example 5.1 (continued)

The model equations of the network in example 5.1 is

$$e = R*i + v_1$$
$$i = C_1*v_1' + C_2*v_2'$$
$$v_1 = v_2$$

The input is e. If v_1 is chosen as state the algorithm gives

$$i = (e-v_1)/R$$
$$v_2 = v_1$$

and the system of equations

$$v_2' = v_1'$$
$$i = C_1*v_1' + C_2*v_2'$$

If the system of equations is solved and the variables i, v_2 and v_2' are eliminated then

$$v_1' = (e - v_1)/R/(C_1+C_2) \qquad []$$

5.4 Formula manipulation

Solution of linear equations

It has been stressed earlier in this report that the equations to be solved can often be solved sequentially, one variable at a time. In fact in many cases the equations are linear in their unknown variable. In such cases the computations can be speeded up by manipulating the equations in order to produce code that directly assigns the value of the variable. More over, it is very interesting to get the manipulated equations written in symbolic format.

Example 5.2

Assume that the variable B should be solved from the equation

$$A + B + C*(D + 2*B) = E*F$$

The symbolic result should be

$$B = (E*F - A - C*D)/(1 + C*2)$$

The equations can also contain functions and the if-then-else construction.

[]

The equations which have one unknown variable can be written in the following form

$$f(x,y) = g(x,y)$$

The variables have been split up into two parts. The unknown variable is denoted x and y is a vector of known variables.

If f and g are linear functions of x the equation can be rewritten as

$$f_0(y) + f_1(y)x = g_0(y) + g_1(y)x$$

The solution of the equation is

$$x = (g_0(y) - f_0(y))/(f_1(y) - g_1(y))$$

The problem is badly posed if the denominator vanishes. This is a numerical problem.

The operations above should be performed directly on the formulas. The problem is then to split up the expression f (and g) in f_0 and f_1. In order to do that the structure of f must be known. In this case it is assumed that it is an <expression> in the sence of Algol-60 (Naur, 1962).

An expression can be represented by a syntax tree. The syntax tree is very important for formula manipulations. The terminal nodes in a syntax tree is variables and constants, other nodes represent operations.

Example 5.3

The expression

A + B + C*(D + 2*B)

has the syntax tree shown in Fig 5.3

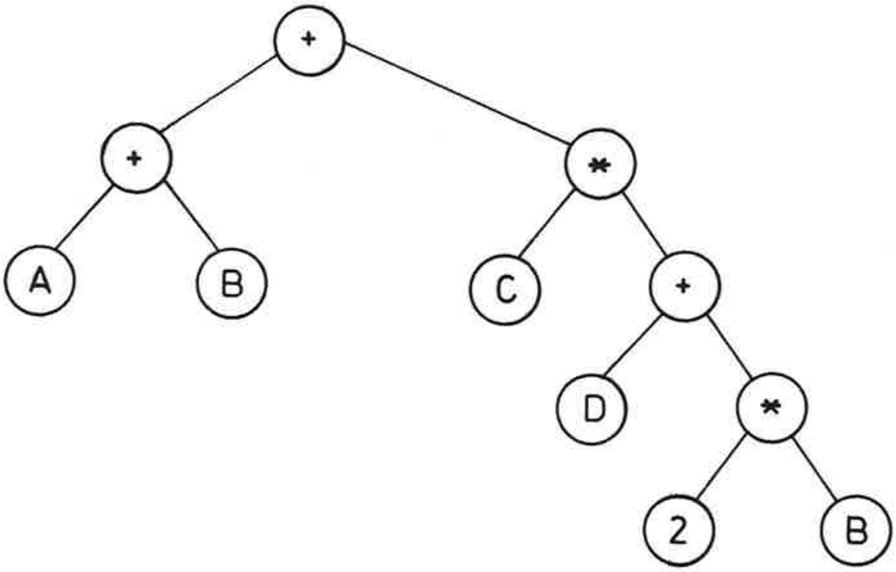
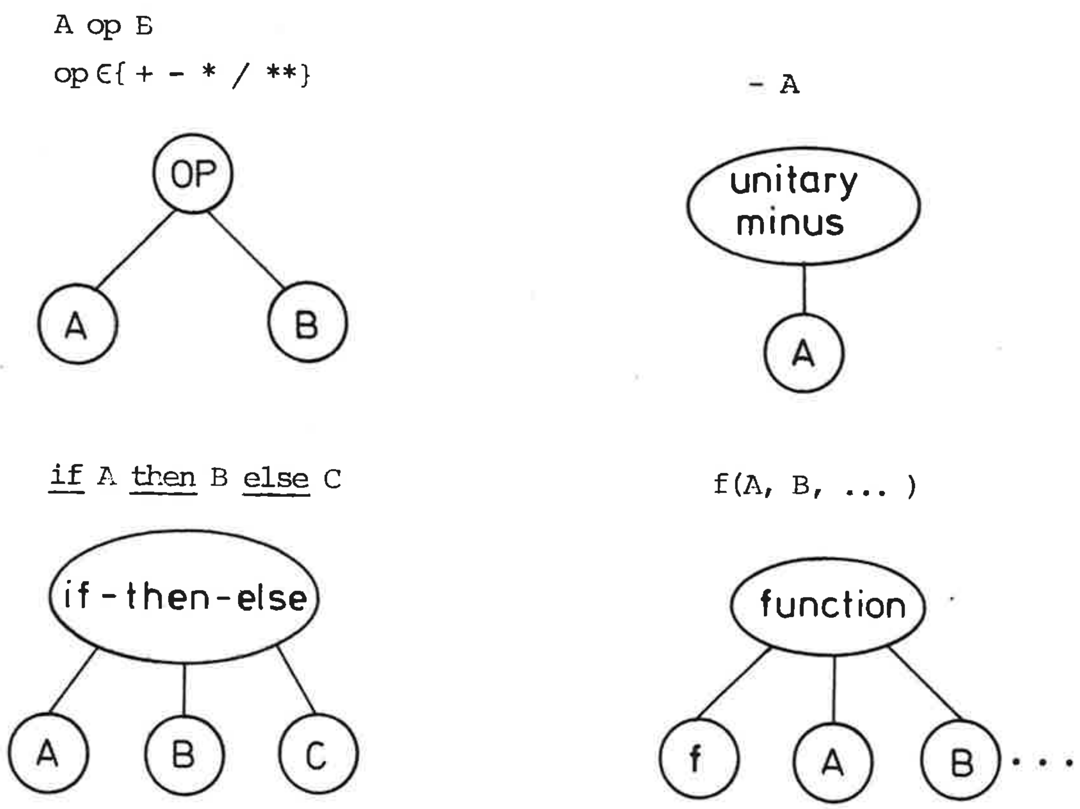


Fig 5.3

Fig. 5.4 shows the types of nodes needed to represent an Algol expression. A syntax tree can be constructed during syntax analysis e.g. top-down analysis or bottom-up analysis, see e.g. Gries (1971). Top-down analysis is easy to program. Each syntactical rule will correspond to a recursive procedure.



A op B

op ∈ { + - * / **}

- A

<u>if</u> A <u>then</u> B <u>else</u> C

f(A, B, ...)

Fig 5.4

Different information can be obtained from the syntax tree by making traversals of it. A traversal of the tree with "suffix walk" will produce Reverse Polish Notation. A symmetric order traversal produces infix notation, i.e. the expression in mathematical notation except for parantheses. These are, however, easily incorporated during the traversal.

The syntax tree is manipulated in order to split up an expression which is linear in some variable. Elementary rules of computation are used to transform the syntax tree to the form shown in Fig 5.5.

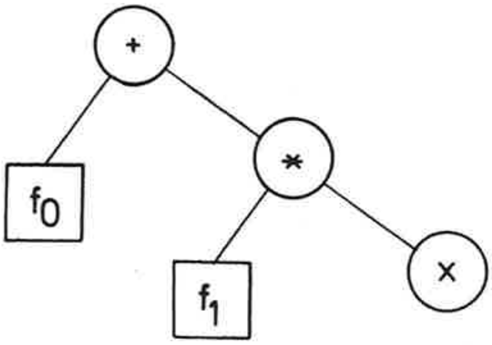


Fig 5.5

The squares indicates the syntax trees for the expressions f_0 and f_1 which do not contain the unknown variable x.

The transformation is easily done using a recursive procedure. This procedure has a syntax tree as input and produces a modified tree with the structure given above. If the input syntax tree has an operator node as its root then the syntax trees for the operands (sons) are first modified by calls of the procedure. Then a new modified tree is constructed using the elementary rules of each operator given in Table 5.1. If the input syntax tree is just a variable or a constant the modification is trivial. This case ends the recursion.

Example 5.4

The expression

$(A + B*X) + (C + D*X)$

is modified with rule 1 to

$(A+C) + (B+D)*X$

This corresponds to a modification of the syntax tree as shown in Fig 5.6.

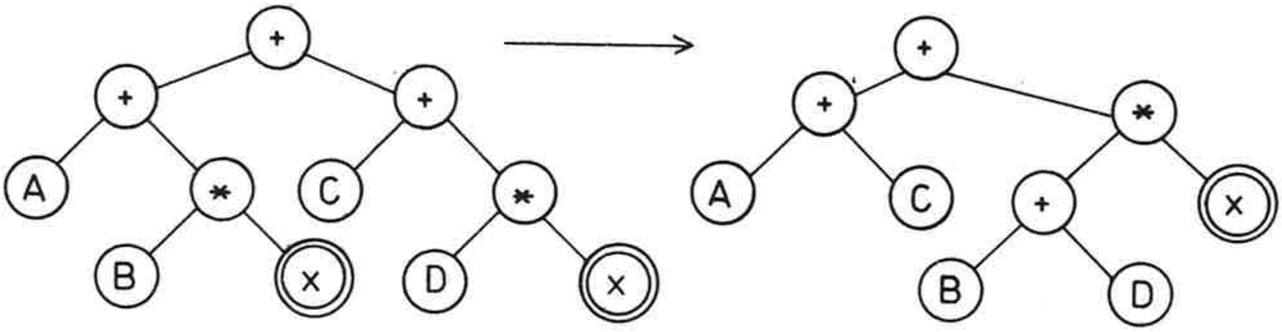


Fig 5.6

The second tree has the wanted structure with the unknown variable in just one node.

When the decomposition of the two expressions of the equation has been done it is trivial to build the syntax tree for the corresponding assignment statement. From this tree is then derived either Reverse Polish Notation or the assignment statement in symbolic form.

A program has been constructed which makes this formula manipulation using the programming language LISP.

Differentiation

Differentiation of the equations of the model is needed to get the linearized model. It is interesting to get the linearized model in symbolic form, thus symbolic differentiation is needed. An other need for differentiation is to speed up the numerical solution of problems involving Jacobians. The Jacobians then do not have to be computed by numerical differentiation.

Symbolic differentiation of an expression is easily done by use of the syntax tree. A modified syntax tree is built up in the same way as when solving linear equations. The difference is the rules of modification. The rules for symbolic differentiation are given in Table 5.2.

Table 5.1 Rules of transformation

1. $(f_0+f_1x) + (g_0+g_1x) = (f_0+g_0) + (f_1+g_1)x$
2. $(f_0+f_1x) - (g_0+g_1x) = (f_0-g_0) + (f_1-g_1)x$
3. $(f_0 + 0x) * (g_0 + 0x) = f_0g_0 + 0x$
   $(f_0 + f_1x) * (g_0 + 0x) = f_0g_0 + f_1g_0x$
   $(f_0 + 0x) * (g_0 + g_1x) = f_0g_0 + f_0g_1x$
4. $(f_0 + f_1x) / (g_0 + 0x) = f_0/g_0 + f_1/g_0x$
5. $(f_0 + 0x) ** (g_0 + 0x) = f_0**g_0 + 0x$
6. $F(f_0 + 0x) = F(f_0) + 0x$
7. $\underline{\text{if}}\ h\ \underline{\text{then}}\ (f_0 + f_1x)\ \underline{\text{else}}\ (g_0 + g_1x) =$
   $(\underline{\text{if}}\ h\ \underline{\text{then}}\ f_0\ \underline{\text{else}}\ g_0) +$
   $(\underline{\text{if}}\ h\ \underline{\text{then}}\ f_1\ \underline{\text{else}}\ g_1)x$

Table 5.2 Rules of differentiation

```
1. (f + g)' = f' + g'
2. (f - g)' = f' - g'
3. (f * g)' = f'*g + f*g'
4. (f / g)' = (f'*g - f*g')/(g*g)
5. (f ** g)' = f**g*(f'*g/f + ln(f)*g')
6. (f(g))' = f'(g)*g'
7. (if h then f else g)' = if h then f' else g'
```

6. EXAMPLES

This chapter contains some examples illustrating the use of the model language.

6.1 Motor, gear and load

Consider the system shown in Fig 6.1.

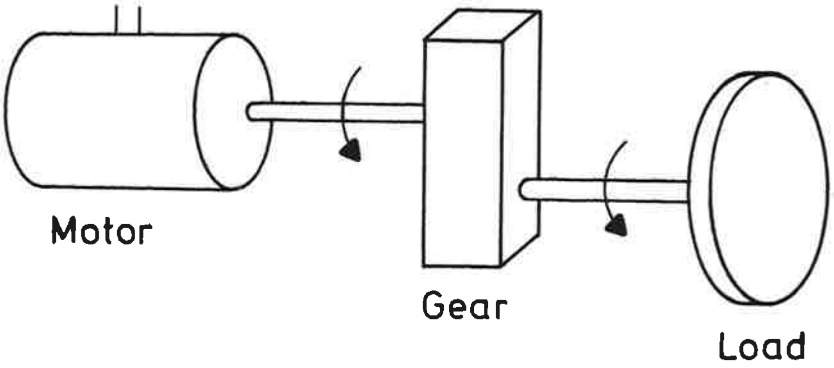


Fig 6.1

It consists of a DC-motor, a gear and a load. A simple model for this system is given below.

This system is interesting because both the motor and the load are decribed as having one state variable each. When connecting them with a stiff gear there appears a constraint which makes the total system first order. If the state e.g. is located to the motor then it is easy to put the equations in state space form. Some of the equations then have to be differentiated (see section 5.3).

```
model motorload

  model motor
    main cut (Omega / Tload)
    input U
    parameter J K R
    local Tel I

    J*der(Omega) = Tel - Tload
    Tel = K*I
    U = R*I + K*Omega

  end

  model gear
    main path ( (Omega1 / T1) - (Omega2 / T2) )
    parameter N

    Omega2 = Omega1*N
    T2 = T1/N

  end

  model load
    main cut (Omega / Tload)
    parameter J1 D1 Tfr

    Tload = J1*der(Omega) + D1*Omega + Tfr*sign(Omega)

  end

  motor to gear to load

end
```

6.2 Tank system

The model of the tank system in Fig 6.2 is shown on the next page.

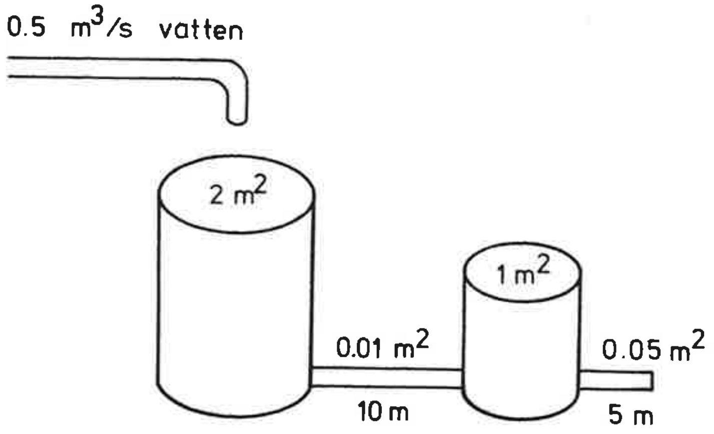


Fig 6.2

Note the use of default statement to describe that the inlet or outlet of a pipe is considered as open if it is not connected. The pressure is set to zero in such a case. If a port of a tank is left unconnected then it should be considered as closed. This will be the fact automatically since through variables have the default value zero.

```
model Tanks

  submodel (Tank) Tank1(Area=2),
                  Tank2(Area=1)
  submodel (Pipe) Pipe1(A=0.01, L=10),
                  Pipe2(A=0.05, L=5)

 (.,.,.,1000/0.5) to
 Tank1 to Pipe1 to Tank2..bottom to Pipe2

end

{-------------------------------------------------------------}

model type Tank

  cut inlet(P0,.,., dens / -qin)
  cut port1(P1, A1, H, dens / qut1)
  cut port2(P2, A2, H, dens / qut2)
  main path Tank(inlet - port1)
  path bottom(port1 - port2)
  parameter A1=1, A2=1 H=0, Area, dens
  constant g=9.81
  local h, vut1, vut2

  P0=0
  Area*der(h) = qin-qut1-qut2
  g*h = P1/dens + (if vut1>0 then vut1**2 else 0)
  qut1 = A1*vut1

  g*h = P2/dens + (if vut2>0 then vut2**2 else 0)
  qut2 = A2*vut2

end

{-------------------------------------------------------------}

model type Pipe

  cut inlet(P1, A, H1, dens / -q)
  cut outlet(P2, A, H2, dens / q)
  main path Pipe(inlet - outlet)
  parameter A, H1=0, H2=0, L, dens
  constant g=9.81
  default P1=0, P2=0
  local v

  L*der(v) + (P2-P1)/dens + g*(H2-H1) = 0
  q = A*v

end
```

6.3 Electrical network

Figure 6.3 shows a model for a transistor used in the logical inverter shown in Fig 6.4. The models are shown on the following pages using a library of electrical components.

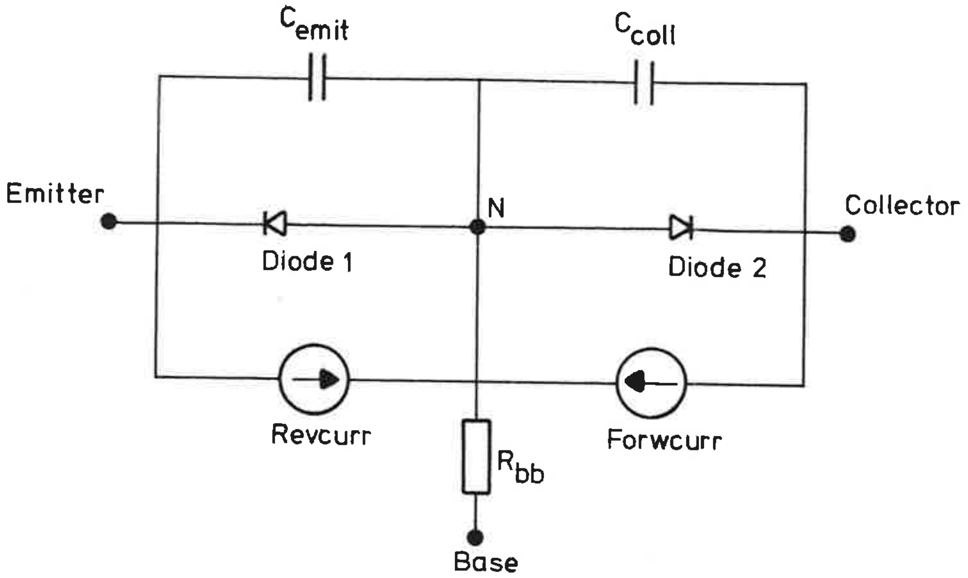


Fig 6.3

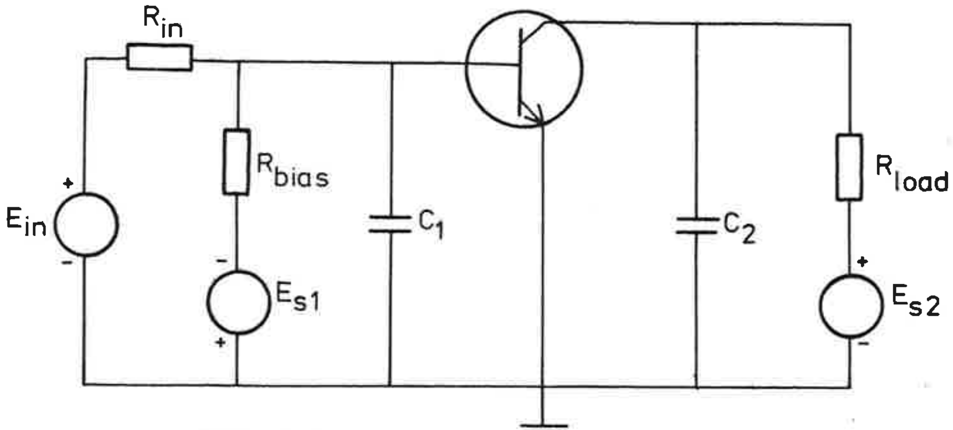


Fig 6.4

```
{ Logical inverter }

model inverter

  submodel Transistor( Rb = 20)
  submodel (voltage) Ein, Es1, Es2
  submodel (resistor) Rin(5.6E3)
                     Rbias(10E3)
                     Rload(1E3)
  submodel (capacitor) C1(3.6E-9)
                     C2(3.4E-9)
  submodel Common

  input U
  output Y

  Common - Ein - Rin - ( (Rbias - Es1) // C1 //
    Transistor..(Base - Emitter) ) - Common

  Common - Es2 - Rload - (C2 //
    Transistor..(Collector-Emitter))

  Ein.V = U
  Y = Rload.VB

  Es1.V = 6
  Es2.V = 6

end
{----------------------------------------------------------}
model type Transistor
  submodel (diode) Diode1(3.5E-9,28),
                    Diode2(7.3E-9,32)
  submodel (current) Revcurr, Forwcurr
  submodel (resistor) Rbb(Rb)
  submodel (varcapacitor) Cemit, Ccoll

  parameter Rb = 30, A1 = 0.47, A2 = 0.998

  cut Base, Emitter, Collector
  cut Transistor(Base,Emitter,Collector)
  node N

  Emitter - ( \ Diode1 // Cemit // Revcurr ) - N
  Collector - ( \ Diode2 // Ccoll // Forwcurr ) - N
  Base - Rbb - N

  Cemit.C = 3E-9 + 6.7E-9*Diode1.I
  Ccoll.C = 2E-9 + 180E-9*Diode2.I

  Revcurr.I = A1*Diode2.I
  Forwcurr.I = A2*Diode1.I
```

```
end
{----------------------------------------------------------}
model type varcapacitor
  cut A(VA / I)
  cut B(VB / -I)
  main path P(A - B)
  local V,Q
  input C
  V = VA-VB
  Q = C*V
  Q' = I
end
{----------------------------------------------------------}
model type diode
  cut A(VA/I)
  cut B(VB/-I)
  main path diode(A - B)
  parameter I0, K
  local V
  V = VA-VB
  I = I0*(exp(K*V) - 1)
end
{----------------------------------------------------------}

{ Library of basic electrical components }

model type resistor
  main path resistor( (Va / I) - (Vb / -I) )
  local V
  parameter R
  V = Va-Vb
  R*I = V
end
{----------------------------------------------------------}
model type capacitor
  main path capacitor ( (Va / I) - (Vb / -I) )
  local V
  parameter C
  V = Va-Vb
  C*der(V) = I
end
{----------------------------------------------------------}
model type coil
  main path coil ( (Va / I) - (Vb / -I) )
  local V
  parameter L
  V = Va-Vb
  L*der(I) = V
end
{----------------------------------------------------------}
model type voltage
  main path voltage ( (Va / I) - (Vb / -I) )
  input V
```

```
  V = Va-Vb
end
{-----------------------------------------------------------}
model type current
  main path current ( (V / I) - (V / -I) )
  input Iin
  I = Iin
end
{-----------------------------------------------------------}
model type Common
  main cut Common (V / .)
  V = 0
end
```

6.4 Electrical energy distribution

Consider a power system consisting of two synchronous generators, three transmission lines and loads as shown in Fig 6.5 (see Elgerd, 1971).

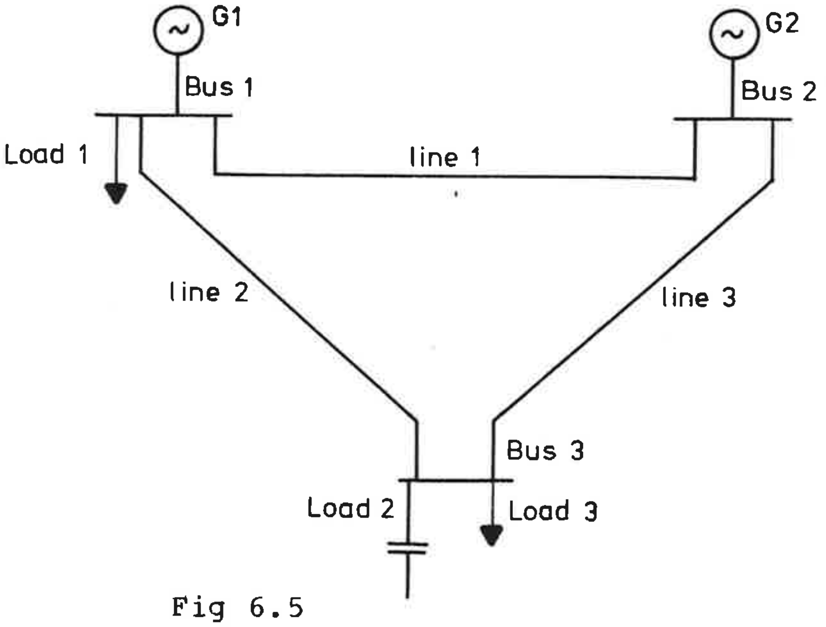


Fig 6.5

A model is shown below.

The voltages and currents are assumed to be sinousoidal with slowly varying amplitude and phase. This means that the transmission lines and the loads can be considered as static systems and that the jw-method can be used for calculating the load flow. This system is an example where the variables in the model are complex. They are split up in their real and imaginary parts in this model.

It should be noted that a large nonlinear system of equations appears which corresponds to the load flow analysis. The system is an example where it is not suitable to give initial conditions on the state variables. These are computed from other variables. Different operations then have to be performed on the model. It is also interesting to model discrete events as e.g. disconnection of faulty loads. Such models are not considered here.

```
model powersystem

{ Distribution of electrical energy }

  submodel (generator)
    G1(Xd=0.054, H=30, f0=50, D=0)
    G2(Xd=0.054, H=300, f0=50, D=0)
  submodel (Line) Line1(0.05), Line2(0.05), Line3(0.05)
  submodel (Load) Load1, Load2(Zx=0), Load3

  node Bus1, Bus2, Bus3

  G1 at Bus1
  G2 at Bus2

  Line1 from Bus1 to Bus2
  Line2 from Bus1 to Bus3
  Line3 from Bus2 to Bus3

  Load1 at Bus1
  Load2 at Bus2
  Load3 at Bus2

end

{-----------------------------------------------------------}

model type generator

{ A model for an electrical generator in three phase
  symmetric operation }

  parameter E, Xd, H, f0, D, Pt
  constant PI=3.14159
  main cut generator(Vx, Vy / -Ix,-Iy)
  local delt, Pg, V

{ The electrical model for the generator is a voltage source
  behind a reactance:
        E = j*Xd*Ig + V
  The magnitude of E is dependent on the field current,
  but is assumed constant in this case. The phase angle
  of E is the relative angular rotor position delt. }

  Ex = E*cos(delt)
  Ey = E*sin(delt)

  Ex = -Xd*Iy + Vx
  Ey = Xd*Ix + Vy

{ The power delivered from the generator is
        Pg = Re(E*Ig*) }
```

```
  Pg = Ex*Ix + Ey*Iy

{ If the mechanical input power Pt is not equal to Pg then
  the phase angle of E will vary. A model for this is the
  so called Swing equation: }
  delt'' *H/(PI*f0) + delt'*D = Pt - Pg

{ The magnitude of the terminal voltage is: }
  V = sqrt(Vx**2 + Vy**2)

end

{----------------------------------------------------------}

model type line

{ Model for a transmission line }

  cut A(Vx1, Vy1 / Ix, Iy)
  cut B(Vx2, Vy2 / -Ix, -Iy)
  main path line(A - B)
  parameter XL

{ The transmission line is modelled by just a reactance:
        V1 = j*XL*I + V2 }
  Vx1 = -Iy*XL + Vx2
  Vy1 = Ix*XL + Vy2

end

{----------------------------------------------------------}

model type Load

{ The load is modelled by an impedance }

  main cut Load(Vx, Vy / Ix, Iy)
  local P, Q, V
  parameter Zx, Zy

{  V = Z*I }
  Vx = Zx*Ix - Zy*Iy
  Vy = Zx*Iy + Zy*Ix

{ The energy load is:
        S = V*I* }
  P = Vx*Ix + Vy*Iy
  Q = Vy*Ix - Vx*Iy

{ The terminal voltage is }
  V = sqrt(Vx**2 + Vy**2)

end
```

6.5 A drum boiler - turbine model

A model of a heat power station has been developed by Lindahl (1976) and used in Simnon (Elmqvist, 1975, 1977). The structure of the system is shown in Fig 6.6 and the model is given on the following pages.

It should be noted how easy the structure of the model is entered using the model language. It is also very easy to understand each submodel because of the well defined cuts.

The pressure equations for steam will constitute a nonlinear set of equations. There are functions HSP, RHP, etc. which defines the state of steam and water by interpolating in the Moliere diagram.

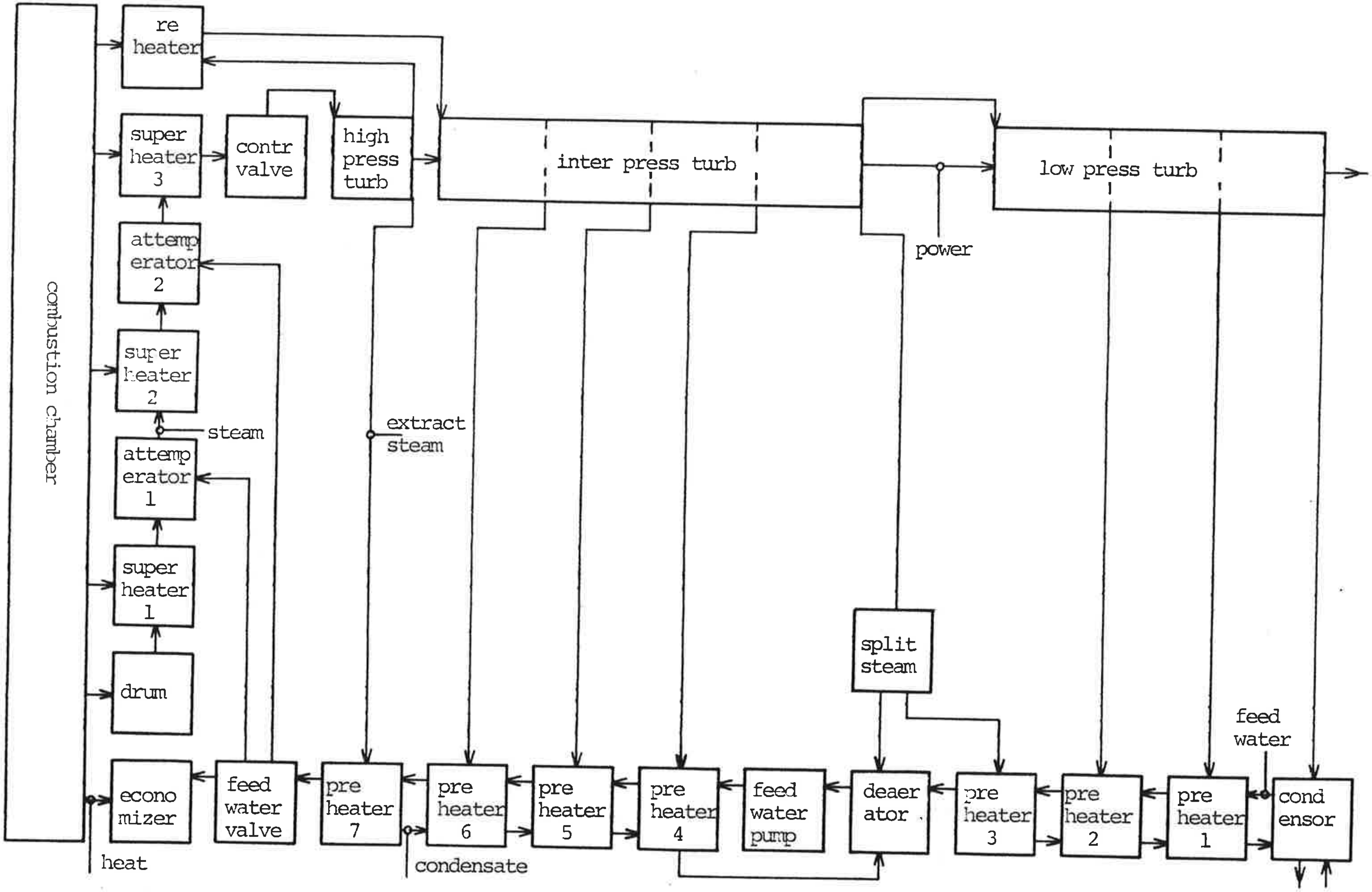

combustion chamber
re heater
super heater 3
contr valve
high press turb
inter press turb
low press turb
attemp erator 2
super heater 2
steam
attemp erator 1
super heater 1
drum
extract steam
power
split steam
feed water
econo mizer
feed water valve
pre heater 7
pre heater 6
pre heater 5
pre heater 4
feed water pump
deaer ator
pre heater 3
pre heater 2
pre heater 1
cond ensor
heat
condensate

Fig. 6.6

```
 { A non-linear drum boiler-turbine model

Translated to the model language by H. Elmqvist

Reference: S. Lindahl, A non-linear drum boiler -
turbine model, TFRT-3132,
Department of Automatic Control,
Lund Institute of Technology

------------------------------------------------------------}

model powerstation

submodel drum
submodel (superheater) superh1, superh2, superh3
submodel (attemperator) attemp1, attemp2
submodel reheater
submodel controlvalve
submodel (turbsection) highpressturb
submodel interpressturb
submodel lowpressturb
submodel condensor
submodel (preheater) preh1, preh2, preh3, preh4, preh5,
          preh6, preh7
submodel splitsteam
submodel deaerator
submodel feedwaterpump
submodel feedwatervalve
submodel combustionchamber
submodel economizer

cut inwater, outwater
path coolingwater (inwater - outwater)
cut power

connect (heat) combustionchamber to (economizer, drum,
          superh1, superh2, superh3, reheater)
connect (steam) drum to superh1 to attemp1 to superh2 to
          attemp2 to superh3 to controlvalve to
          highpressturb to reheater to interpress turb
          to lowpressturb to condensor
connect (extractsteam) highpressturb to preh7,
          interpressturb to (preh6, preh5, preh4,
            splitsteam to (deaerator, preh3) ),
          low-pressturb to (preh2, preh1)
connect (feedwater) condensor to preh1 to preh2 to preh3
          to deaerator to feedwaterpump to preh4 to
          preh5 to preh6 to preh7 to feedwatervalve
          to (economizer, attemp1, attemp2)
connect (condensate) preh7 to preh6 to preh5 to preh4
          to deaerator,
          preh3 to preh2 to preh1 to condensor
connect (power) highpressturb to interpressturb
```

```
          to lowpressturb to power
connect (coolingwater) inwater to condensor to outwater

end

{----------------------------------------------------------}

model type drum

submodel drumdowncomers
submodel risers

cut feedwater, heat, steam

connect (feedwater) feedwater to drumdowncomers
connect (heat) heat to risers
connect (water) drumdowncomers to risers to drumdowncomers
connect (steam) risers to drumdowncomers to steam

end

  {-------------------------------------------------------}

model type drumdowncomers

cut inwater (Wwr, Hwr, Pd)
cut outwater (Wdc, Hdc, Pdc)
path water (inwater - outwater)
cut insteam (Wsr, Hsr, Pd)
cut outsteam (Ws, Hs, Pd)
path steam (insteam - outsteam)
cut feedwater (Ww, Hw, Pd)
local z, Rw, Vw, Vs, HsP, Rs, RsP, Rwr, Hd
parameter Vdc, Adrum, f, L, D, A, g, Vw0, Vs0

Pd - Pdc = (1+F*L/D)*Wdc**2 / (2*A*Rw) - g*L*Rw

  { mass balance for water }
  { der((Vw+Vdc)*Rw) = }
Aw*der(z)*Rw = Ww + Wwr - Wd
Vw = Vw0 + Adrum*z

  { energy balance for water }
  { der((Vw + Vdc)*Rw*Hd) = }
(Vw + Vdc)*Rw*der(Hd) = Ww*Hw + Wwr*Hwr -Wdc*Hdc
Hdc = Hd

  { mass balance for steam }
  { der(Vs*Rs) = }
-Adrum*der(z)*Rs + Vs*RsP*der(Pd) = Wsr - Ws
Vs = Vs0 - Adrum*z

  { energy balance for steam }
```

```
    { der(Vs*Rs*Hs = }
-Adrum*der(z)*Rs*Hs + Vs*(RsP*Hs + Rs*HsP)*der(Pd)
    = Wsr*Hsr - Ws*Hs

Hs = HSP(Pd)
HsP = HSPP(Pd)
Rw = RHP(Hd,Pd)
Rs = RSP(Pd)
RsP = RSPP(Pd)
Rwr = RWP(Pd)

end

    {-----------------------------------------------------}

model type risers

cut inwater (Wdc,Hdc,Pdc)
cut outwater (Wwr, Hwr, Pd)
path water (inwater - outwater)
cut steam (Wsr, Hsr, Pd)
local Vb, x, xs, TAU, Wsprod, Wmix, A, Rmix, Tm, TmP, TmixP
parameter Vr, Cm, m, K

Pdc - Pd = (1 +f*L/D)*Wmix**2 / (2*A*Rmix) + g*L*Rmix
Rmix*Vr = Vb*Rs + (Vr - Vb)*Rwr

    { mass balance for water }
    { der((Vr - Vb)*Rwr) = }
-der(Vb)*Rwr = Wdc -Wsprod -Wwr
Wwr = (1-x)*Wmix

    { mass balance for steam }
    { der(Vb*Rs) = }
der(Vb)*Rs + Vb*RsP*der(Pd) = Wsprod -Wsr
Wsr = x*Wmix

    { energy balance }
    { der(Cm*m*Tm + (Vr - Vb)*Rwr*Hwr + Vb*Rs*Hs) = }
Cm*m*TmP*der(Pd) - der(Vb)*Rwr*Hwr +
    (Vr - Vb)*(RwrP*Hwr + Rwr*HwrP)*der(Pd) +
    der(Vb)*Rs*Hs + Vb*(RsP*Hs + Rs*HsP)*der(Pd) =
    Q + Wdc*Hdc - Wsr*Hsr - Wwr*Hwr
Tm = Tmix + K*Q**(1/3)
TmP = TmixP

xs = 2*Vb*Rs/(Vr*Rmix)
TAU = Vr*Rmix/Wdc
der(x) = 2/TAU*(xs - x)

Hwr = HWP(Pd)
HwrP = HWPP(Pd)
Rs = RSP(Pd)
```

```
RsP = RSPP(Pd)
Rwr = RWP(Pd)
RwrP = RWPP(Pd)
Tmix = TLP(Pd)
TmixP = TLPP(Pd)
Tdc = THP(Hd, Pd)

end

  {-----------------------------------------------------------}

model type superheater

cut instearn (W, H1, P1)
cut outsteam (W, H2, P2)
path steam ( insteam - outsteam )
cut heat (Q)
parameter Cm , m, K, Vs, f
local Tm, TmH, T2, T2H, R2

P1**2 - P2**2 = f*W**2

  { energy balance }
  { der(m*Cm*Tm + Vs*R2*H2) = }
(m*Cm*TmH + Vs*R2)*der(H2) = Q - W*(H2 - H1)

Tm = T2 + K*W*(H2 - H1)
TmH = T2H + K*W

R2 = RHP(H2, P2)
T2 = THP(H2, P2)
T2H = THPH(H2, P2)

end

  {-----------------------------------------------------------}

model type attemperator

cut insteam (W1, H1, P)
cut outsteam (W2, H2, P)
path steam (insteam - outsteam)
cut feedwater (Ww, Hw, Pw)
input Sw
parameter fw

aw**2 - P**2 = fw*(Ww/aw)**2
aw = Sw**2

  { mass balance }
W1 + Ww = W2

  { energy balance }
```

```
W1*H1 + Ww*Hw = W2*H2

end

  {--------------------------------------------------------}

model type controlvalve

cut instеam (W, H, P1)
cut outsteam (W, H, P2)
path steam ( insteam - outsteam)
input Sv
parameter fv

P1**2 - P2**2 = fv*(W/av)**2
av = VALVE(Sv)

end

  {--------------------------------------------------------}

model type reheater

cut insteam (W1, H1, P1)
cut outsteam (W2, H2, P2)
path steam ( insteam - outsteam )
cut heat (Q)
parameter Cm, m, Vs, K, f
local T2, T2H, Tm, TmH, R2, R2H, R2P

P1**2 - P2**2 = f*W1**2

  { mass balance }
  { der(Vs*R2) = }
Vs*R2T = W1 - W2

  { energy balance }
  { der(m*Cm*Tm + Vs*R2*H2) = }
(m*Cm*TmH)*der(H2) + Vs*(R2T*H2 + R2*der(H2)) =
  Q + W1*H1 - W2*H2

Tm = T2 + K*Q
TmH = T2H

  { der(R2) = }
R T = R2H*der(H2) + R2P*der(P2)

R2 = RHP(H2, P2)
R2H = RHPH(H2, P2)
R2P = RHPP(H2, P2)
T2 = THP(H2, P2)
T2H = THPH(H2, P2)
```

```
end

  {--------------------------------------------------------}

model type turbsection

cut insteam (W1, H1, P1)
cut outsteam (W2, H2, P2)
path steam ( insteam - outsteam )
cut extractsteam (Wp, H2, Pp)
cut inpower (N1)
cut outpower (N2)
path power ( insteam - outsteam )
input S
local H, T2
parameter f, fp, Eh
default N1=0, Wp=0, S=1

P1 = f*W1
P2**2 - Pp**2 = fp*(Wp/ap)**2
ap = S

W1 = W2 + Wp

N2 = N1 + W1*(H1 - H2)
H2 = H + (1 - Eh)*(H1 - H)

H = ISENX(H1, P1, P2)
T2 = THP(H2,P2)

end

  {--------------------------------------------------------}

model type interpressturb

submodel (turbsection) IP1, IP2, IP3, IP4

cut insteam, outsteam
path steam ( insteam - outsteam )
cut extr1, extr2, extr3, extr4
cut extractsteam (extr1, extr2, extr3, extr4)
cut inpower, outpower
path power ( inpower - outpower )
input S1, S2

connect (steam) insteam to IP1 to IP2 to IP3
           to IP4 to outsteam
connect (extractsteam) IP1 to extr1, IP2 to extr2,
           IP3 to extr3, IP4 to extr4
connect (power) inpower to IP1 to IP2 to IP3
           to IP4 to outpower
```

```
IP1.S=S1
IP2.S=S1
IP3.S=S1
IP4.S=S2

end

  {--------------------------------------------------------}

model type lowpressturb

submodel (turbsection) LP1, Lp2, LP3

cut insteam, outsteam
path steam ( insteam - outsteam )
cut extr1, extr2
cut extractsteam (extr1, extr2)
cut inpower, outpower
path power ( inpower - outpower )
input S

connect (steam) insteam to Lp1 to LP2 to LP3
        to outsteam
connect (extractsteam) LP1 to extr1, LP2 to extr2
connect (power) inpower to LP1 to LP2 to LP3
        to outpower

LP1.S=S
LP2.S=S

end

  {--------------------------------------------------------}

model type condensor

cut steam (Ws, Hs, Ps)
cut condensate (Wc, Hc, Pc)
cut feedwater (Ww, Hw, Pw)
path coolingwater ( (W1, H1, P1) - (W1, H2, .) )
parameter Hdiff, Vc, m, Cm, Vcool, Pdiff
local Rw, R2, Tw, T1, T2, HwP, TwP, Tc

Pc = Pw + Pdiff

  { mass balance }
Ws + Wc = Ww

  { energy balance }
  { der(Vc*Rw*Hw + m*Cm*Tm + Vcool*R2*Hw) = }
(Vc*Rw*HwP + m*Cm*TmP + Vcool*R2*HwP)*der(Pw) =
Ws*Hs + Wc*Hc - Ww*HW - W1*(H2-H1)
```

```
H2 = Hw - Hdiff
TmP = TwP

Hw = HWP(Pw)
HwP = HWPP(Pw)
Rw = RWP(Pw)
R2 = RHP(H2, P1)
Tw = TLP(Pw)
TwP = TLPP(Pw)
T1 = THP(H1, P1)
T2 = THP(H2, P1)
Tc = TLP(Pc)

end

  {-------------------------------------------------------}

model type preheater

path feedwater ( (W, H1, P1) - (W, H2, P2) )
cut steam ( Ws, Hs, Ps )
path condensate ( (Wc1, Hc1, Pc1) - (Wc2, Hc2, Pc2) )
local Psat, Tsat, TsatP, T2
parameter Vc, Vw, Hdiff, f

P1**2 - p2**2 = f*W**2

  { mass balance }
Wc2 = Wc1 + Ws

  { energy balance }
  { der(Vc*Rc*Hsat + m*Cm*Tsat +Vw*Rw*Hsat) = }
(Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)*der(Psat) =
     Ws*Hs + Wc1*Hsat - W*(H2-H1) - Wc2*Hc2

Rc = RWP(Psat)
Hsat = HWP(Psat)
HsatP = HWPP(Psat)
Tsat = TLP(Psat)
TsatP = TLPP(Psat)
Rw = RHP(H2, P2)
H2 = Hsat - Hdiff
Hc1 = Hsat
T2 = THP(H2, P1)

end

  {-------------------------------------------------------}

model type splitsteam

path extractsteam ( (W, H, P) - ( (W1, H, P), (W2, H, P) ) )
```

```
W1 = 0.3*W
W2 = 0.7*W

end

  {---------------------------------------------------------}

model type dearator

path feedwater ( (W, H1, P1) - (W2, H2, .) )
cut steam (Ws, Hs, .)
cut condensate (Wc, Hc, Pc)
cut water (Wwater, Hwater, .)
parameter Hstorage, Vw, m, Cm, Pdiff
local Rw, H2P, T2, T2P, Psat

  { mass balance }
W2 = W + Wc + Ws + Wwater

Hwater = if Wwater > 0 then Hstorage else H2

  { energy balance }
  { der( Vw*Rw*H2 + m*Cm*T2 = }
(Vw*Rw*H2P + m*Cm*T2P)*der(Psat) =
  Ws*Hs + W*H1 + Wwater*Hwater + Wc*Hc - W2*H2

Rw = RWP(Psat)
H2 = HWP(Psat)
H2P = HWPP(Psat)
T2 = TLP(Psat)
T2P = TLPP(Psat)
Pc = Psat + Pdiff
Hc = HWP(Pc)
Tc = TLP(Pc)
P1 = Psat

end

  {---------------------------------------------------------}

model type feedwaterpump
path feedwater ( (W, H, . ) - (W, H, P2) )
input Ppump
parameter f

P2 = Ppump - f*W**2

end

  {---------------------------------------------------------}

model type feedwatervalve
```

```
cut infeedwater (W, H, P1)
cut outfeedwater ( (Wd, H, P2), (Wa1, H, P2), (Wa2, H, P2) )
path feedwater ( infeedwater - outfeedwater )
input a
parameter f

P2 = P1 - f*(W/a)**2
Wd + Wa1 + Wa2 = W

end

  {----------------------------------------------------------}

model type combustionchamber

cut heat ( (Q1), (Q2), (Q3), (Q4), (Q5), (Q6) )
input Woil
parameter b10, b20, b30, b40, b50, b60
parameter b11, b21, b31, b41, b51, b61
parameter b12, b22, b32, b42, b52, b62

Q1 = b10 + b11*Woil + b12*Woil**2
Q2 = b20 + b21*Woil + b22*Woil**2
Q3 = b30 + b31*Woil + b32*Woil**2
Q4 = b40 + b41*Woil + b42*Woil**2
Q5 = b50 + b51*Woil + b52*Woil**2
Q6 = b60 + b61*Woil + b62*Woil**2

end

  {----------------------------------------------------------}

model type economizer

path water ( (W, H1, P1) - (W, H2, P2) )
cut heat (Q)
local Tm, T2, T2P, R2
parameter k, f

  { energy balance }
  { der( Cm*m*Tm + Ve*R2*H2 ) = }
(Cm*m*TmH + Ve*R2)*der(H2) = Q + W*H1 - W*H2

P2 = P1 - f*W**2

R2 = RHP(H2, P2)
T2 = THP(H2, P2)
T2H = THPH(H2, P2)
Tm = T2 + k*Q
TmH = T2H

end
```

REFERENCES

Aarna, O. (1976): Continuous systems modeling. TRITA - REG - 7601, Department of Automatic Control, The Royal Institute of Technology, Stockholm, Sweden.

Aulin, B. (1969): Decomposition of systems of high order. TFRT-5046, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden (in swedish).

Brayton, R.K., Gustavson, F.G. and Hachtel, G.D (1972): A new efficient algorithm for solving differential-algebraic systems using implicit backward differentiation formulas. Proc. of the IEEE, vol. 60, No. 1, January 1972, pp 98-108.

Brown, R.L. and Gear C.W. (1973): DOCUMENTATION FOR DFASUB - A program for the Solution of Simultaneous Implicit Differential and Nonlinear Equations. UIUCDCS-R-73-575, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

Christensen, J., (1970): The Structuring of Process Optimization. AIChE Journal, vol. 16, No. 2, March 1970, pp 177-184.

Christensen, J. and Rudd, D. (1969): Structuring Design Computations. AIChE Journal, vol. 15, No. 1, January 1969, pp 94-100.

Drud, A. (1975): Simulation with Large Simultaneous Systems. Proc. Simulation '75, Zurich.

Elgerd, O. (1971): Electric Energy Systems Theory: An Introduction. McGraw-Hill.

Elmqvist, H. (1975): SIMNON - An Interactive Simulation Program for Nonlinear Systems - User's manual. TFRT-3091, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist, H. (1977): SIMNON - An Interactive Simulation Program for Nonlinear Systems. Proc. Simulation '77, Montreux.

Fahrland, D.A. (1970): Combined Discrete Event Continuous Systems Simulation. Simulation, February 1970, 61-72.

Gear, C. (1971): Simultaneous Numerical Solution of Differential-Algebraic Equations. IEEE Tr. Circuit Theory, vol. CT-18, No. 1, 89-95.

Gear, C. (1972): A Generalized Interactive Network Analysis and Simulation System. Proc. First USA-JAPAN Computer Conference, 1972, 559-566.

Gries, D. (1971): Compiler Construction for Digital Computers. John Wiley and Sons.

Hachtel, G., Brayton, R. and Gustavson, F. (1971): The Sparse Tableau Approach to Network Analysis and Design. IEEE Tr. Circuit Theory, vol. CT-18, No. 1, 101-113.

Lee, W., Christensen, J. and Rudd, D. (1966): Design Variable Selection to Simplify Process Calculations. AIChE Journal, Vol. 12, No. 6, 1104-1110.

Lee, W. and Rudd, D. (1966): On the Ordering of Recycle Calculations. AIChE Journal, Vol. 12, No. 6, 1184-1190.

Lindahl, S. (1976): A nonlinear drum boiler - turbine model. TFRT-3132, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Naur, P. (Ed.) (1962): Revised Report on the Algoritmic Language ALGOL 60. Numer. Math. 4, 1962, 420-453; Commun. ACM 6, 1, 1963, 1-17; Comput. J. 5, 4, 1963, 349-367.

Pernebo, L. (1977): Algorithms for linear time invariant systems. Report, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, (to appear).

Runge, T. (1975): Simulation Modeling with GRASS and MAP. UIUCDCS-R-75-712, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

Sato, T. and Ichikawa, A. (1967): On a Decomposition Technique for Multilevel Control Systems. IEEE AC 12, 457-458.

Stadtherr, M., Gifford, W. and Scriven L. (1974): Efficient Solution of Sparse Sets of Equations. Chem. Eng. Science, Vol. 29, 1025-1034.

Steward, D. (1962): On an Approach to Techniques for the Analysis of the Structure of Large Systems of Equations. SIAM Review, Vol. 4, No. 4, 321-342.

Steward, D. (1965): Partitioning and Tearing Systems of Equations. J. SIAM Numer. Anal., Ser. B, Vol. 2, No. 2, 345-365.

Tewarson, R. (1973): Sparse Matrices. Academic Press, New York and London.

Volgin, V., Samoilov, Y. and Usenko, V. (1975): The Optimal Sequence of Thermal Calculation of Thermal Power Plants. Teploenergetika, 22,1 26-29.

Wiberg, T. (1977): Permutation of an Unsymmetric Matrix to Block Triangular Form. Diss., Department of Information Processing, University of Umea, Umea, Sweden.

ACKNOWLEDGEMENTS

I would like to thank my adviser Prof. Karl Johan Astrom for many stimulating discussions and encouragement.

The work has been supported by the Swedish Institute of Applied Mathematics (ITM).

This report has been prepared on a PDP-11 (LSI) computer. The text was edited on a text screen using a page oriented editor written in Pascal. The nice lay out was produced by the RUNOFF program. A terminal with a Diablo printer was used for output. The figures were drawn by Britt-Marie Carlsson.

APPENDIX

1. Syntax notation

The following syntax notation is used in this report.

/	or (separates terms in a list from which one and only one must be chosen
{ }	groups terms together
[ ]	groups terms together and denotes that the group is optional
$\{ \}^{*}$	denotes the repetition of a choice in the group one or more times
$[ ]^{*}$	denotes the repetition of a choice in the group none or more times

It should be noted that the syntax is not complete in some respects. This is the case e.g. with the variable lists. It is possible to use a comma as separator between variables even if this is not included in the syntax. Comments are written within the parantheses { }. New line or ; serves as a separator between statements. If a statement occupies several lines this must be indicated.

2. Syntax for model language

```
<model> ::= model <model identifier> <model body> end
  <model type> ::= model type <model type identifier>
                   <model body> end
  <model body> ::= <submodel part> <declaration part>
                   <statement part>

    <submodel part> ::= [ <model> / <model type> /
                   <submodel incorporation> ]*
      <submodel incorporation> ::=
          submodel [ (<model type identifier>) ]
          { <model identifier> [(<parameter list>)] }*
      <parameter list> ::= {<number>}* /
                   {<parameter>=<number>}*

    <declaration part> ::= [ <variable declaration> /
          <cut declaration> / <node declaration> /
          <path declaration> ]*

      <model spec> ::= <model identifier>
                   [::<model identifier>]*
      <path spec> ::= <model spec> [..<path>] / <path>
      <cut spec> ::= <model spec> [:<cut>] / <cut>
      <variable spec> ::= <model spec>.<variable>/<variable>

      <variable declaration> ::=
          parameter { <variable> [=<number>] }* /
          constant { <variable> =<number> }* /
          local { <variable> }* /
          terminal { <variable> }* /
          input { <variable> }* /
          output { <variable> }* /
          default { <variable> = <number> }* /
          internal { <variable> }* /
          external { <variable> }*

      <cut declaration> ::= [main] cut {<cut identifier>
```

```
   [<cut clause>]}*
 <node declaration> ::= node {<node identifier>
   [<cut clause>]}*

  <cut clause> ::= ( [<cut list>] / [<cut list>] )
  <cut list> ::= { <cut element> }*
  <cut element> ::= <variable> / -<variable> /
   <cut spec> / .
  <cut> ::= <cut clause> / <cut identifier> /
   <node identifier>

 <path declaration> ::= [main] path {<path identifier>
   <path clause>}*
  <path clause> ::= ( <cut spec> - <cut spec> )
  <path> ::= <path clause> / <path identifier>

<statement part> ::= [ <equation> / <procedure call> /
   <connection statement> ]*

 <equation> ::= <expression> = <expression>
 <procedure call> ::= { <variable> }* =
   <procedure identifier> ( {<variable>}* )

 <connection statement> ::=
   [connect (<cut identifier> / <path identifier>)]
   <connection expression>
  <connection expression> ::= <connection secondary>
   { { at|=|to|-|from|<|par|//|loop }
   <connection secondary> }*
  <connection secondary> ::=
   [reversed|} <connection primary>
  <connection primary> ::= <connection operand> /
   ( <connection expression> )
  <connection operand> ::= <cut spec> / <path spec>
```

INDEX REGISTER