A Structured Model Language for Large Continuous Systems

Elmqvist, Hilding

1978

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

Link to publication

Citation for published version (APA):
Elmqvist, H. (1978). *A Structured Model Language for Large Continuous Systems*. [Doctoral Thesis (monograph), Department of Automatic Control]. Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
1

CODEN: LUTFD2/(TFRT-1015)/1-226/(1978)

Hilding Elmqvist

A Structured Model Language for Large Continuous Systems

A Structured Model Language for Large Continuous Systems

av

Hilding Elmqvist

Civ ing, Sm

Akademisk avhandling som för avläggande av teknisk doktorsexamen vid Tekniska Fakulteten vid Universitetet i Lund kommer att offentligt försvaras i sal M:A, Maskinhuset, Lunds Tekniska Högskola, onsdagen den 31 maj 1978 kl 10.15.

Thesis for the degree of "Teknologie Doktor"
in Automatic Control
at Lund Institute of Technology.

To Irene

Printed in Sweden
Sigma-Tryck
Lund 1978

Dokumentutgivare Lund Institute of Technology	Dokumentnamn REPORT
	Dokumentbeteckning LUTFD2/(TFRT-1015)/1-226/(1978)
Handläggare Karl Johan Åström	Utgivningsdatum May 1978
	Ärendebeteckning
Författare Hilding Elmqvist	

Dokumenttitel och undertitel

A Structured Model Language for Large Continuous Systems

Referat (sammandrag)

A model language, called DYMOLA, for continuous dynamical systems is proposed. Large models are conveniently described hierarchically using a submodel concept. The ordinary differential equations and algebraic equations need not be converted to assignment statements. There is a concept, cut, which corresponds to connection mechanisms of complex types, and there are facilities to describe the connection structure of a system. A model can be manipulated for different purposes such as simulation and static calculations. The model equations are sorted and they are converted to assignment statements using formula manipulation. A translator for the model language is also included.

Referat skrivet av
Author

Förslag till ytterligare nyckelord
nonlinear systems, compiler, permutations, graph theory

Klassifikationssystem och -klass(er)

Indextermer (ange källa)
Mathematical models, Simulation languages, Computerized simulation, Nonlinear systems, Ordinary differential equations, Compilers.
(Thesaurus of Engineering and Scientific Terms, Eng. Joint Council, USA)

Omfång 226 pages	Övriga bibliografiska uppgifter	
Språk English		
Sekretessuppgifter	ISSN	ISBN
Dokumentet kan erhållas från Department of Automatic Control Lund Institute of Technology P O Box 725, S-220 07 Lund 7, Sweden	Mottagarens uppgifter	
Pris		

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 like integrators, summers and potentiometers. The user had to convert his model to a block diagram of these elementary subsystems. It was then simple to enter the model in block diagram 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 is not necessary. Programs which accept differential equations have also been developed. Such languages are called equation oriented languages. The language MIMIC is an early example of an equation oriented language. A standardization effort was made by the Simulation Councils in the USA in 1967. This resulted in the CSSL language (Continuous System Simulation Language), (Strauss, 1967). It has been implemented on many computers. The program CSMP-360 (Continuous System Modelling Program) was developed at about the same time (Brennan, Silberberg, 1968).

In equation oriented languages the model is specified as assignment statements of FORTRAN type. A special integration operator is used to describe differential equations. The equations can be given in arbitrary order because they are sorted by the programs. It is a problem for the modeller 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.

Simulation can of course also be made by using a general programming language and an integration routine for ordinary differential equations. The model is then specified by a subroutine or procedure which computes the derivatives of the state variables. There are several subroutine packages available. A typical example is given in Fick (1975).

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 DARE programs (see Korn, Wait, 1978) are well known interactive programs. The program SIMNON has been developed by the author (Elmqvist, 1975,1977a). Other examples of interactive simulation programs are ISIS (Hang, Sangster, 1975), BEDSOCS (Ord-Smith, Stephenson, 1975) and SIM (van den Bosch, Bruijn, 1977).

The interest for simulation of systems modelled by both ordinary differential equations and discrete events has increased, see Fahrland (1970). One reason is the desire to simulate processes controlled by digital computers. Several programs have been developed, e.g. GASP-IV (Pritsker, Hurst, 1973), GSL (Golden, Schoeffler, 1973) and CADSIM (Sim, 1975). A survey is given in Ören (1977). Models described by ordinary differential equations and difference equations can be used in SIMNON. GASP-V (Cellier, 1976) is a program for simulation of systems described by ordinary differential equations, partial differential equations and discrete events.

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

The situation for the modeller can vary widely. A difficult task is 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 are interesting for the model and which quantities should be included in the model.

In presently available languages the connections between submodels are done with variables. There are no concepts which correspond to the much more complex connection mechanisms that occur in physical systems like 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 has an important consequence for the modeller who has models available for the subsystems included. If it is known that the models are compatible with respect to the phenomena included, 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 thesis describes a <u>proposal for a model language</u> for continuous dynamical systems. The characteristics of this language are the following. The differential equations and the algebraic equations of the model can be introduced in their original form. They need not be converted to assignment statements. There is a concept, <u>cut</u>, which correponds to <u>connection mechanisms</u> of complex type and there is also a simple way of describing the <u>connection structure</u> of a system.

The model language is called DYMOLA (Dynamic-Modelling Language).

The connections between submodels introduces constraints on the variables in the cuts. This can in some cases lead to a <u>reduction</u> of the number <u>of states</u> for the system. The parallel connection of two capacitors is a typical example. Each capacitor is separately described by one state variable. The total system will, however only have 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 many systems which are described by basic physical equations because it happens frequently that the derivatives of the state variables are only defined implicitly. The determination of the derivatives is often a nontrivial task because it may happen that it is necessary to differentiate some of the equations to be able to solve for the derivatives.

There are special methods to transform models of electrical networks to state space form. These methods are based on graph theory, see e.g. Desoer, Kuh (1969). It is often not possible to transform Newton's equations for mechanical systems to state space form directly. The state space form is obtained if a Lagrange function is used.

Another way of approaching this problem is to develop integration methods that can handle such systems directly. Such methods are available, see chapter 5.

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 such a way that the variables can be solved one at the time from the equations. When systems of equations, which have to be solved simultaneously, occur 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 indicate 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 obtain 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 some iterative technique.

When solving systems of equations by iterative technique the Jacobian is often needed. The computations may be speeded up by using 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 psychological effect to see exactly the equations that were generated from the model in the high level language.

A translator for the DYMOLA language has been written in Simula. It accepts a model as input. The output from the program is the model equations. They are sorted and grouped into systems of equations. Equations that are linear in their unknown variable are solved. There are commands to specify what variables are considered as known or unknown.

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 requiring conversion 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 design of chemical processes. Design computations on thermal power plants (Volgin et.al., 1975) is another example. The corresponding problem for models of economical systems with difference equations is discussed by Drud (1975). A theoretical discussion of transformations of the equations is given in Aarna (1976).

This thesis is based on the report Elmqvist (1977b). Chapter 2 illustrates some of the problems with present model languages like CSSL, CSMP and SIMNON. The drawbacks discussed have served as a motivation for the proposed model language. The model language DYMOLA is described in chapter 3. Different types of operations on the model are discussed in chapter 4. Methods for doing these operations are given in chapter 5. An implementation of a translator for the model language is described in chapter 6. Chapter 7 illustrates the use of the model language to describe different types of systems. It also contains the sorted and manipulated equations of the models as they are outputted by the translator. The appendix contains a description of the syntax notation used and the syntax of the language. The listing of the translator is also given in appendix.

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 advantages of describing a model with equations.

Example 2.1

Consider the network in Fig. 2.1.

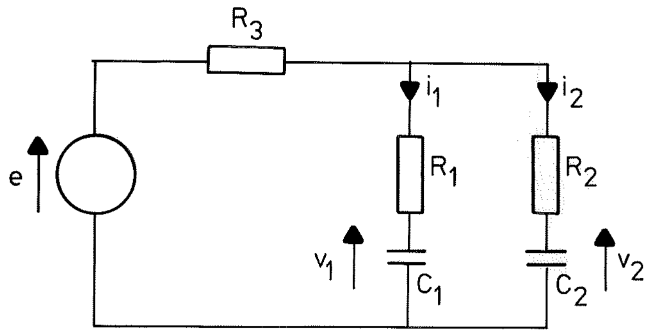


Fig 2.1

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 iteration.

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 illustrates the problem that the manipulations of the equations that have to be made may depend on the environment.

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

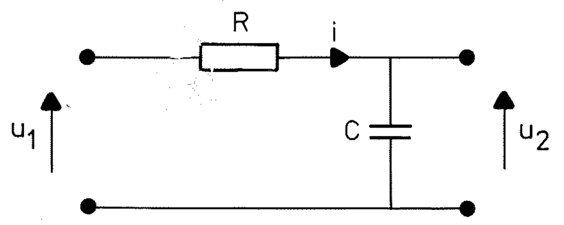


Fig 2.2

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,Ø]
END
```

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

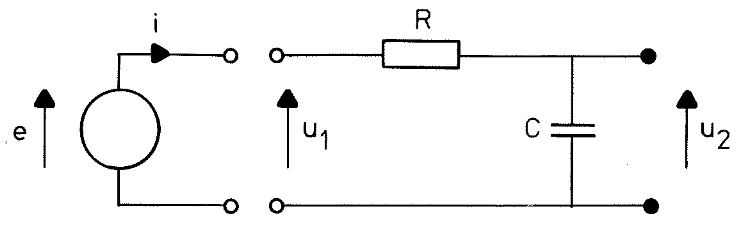


Fig 2.3

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,Ø]
```

which describes the system correctly.

Consider now the system in Fig 2.4.

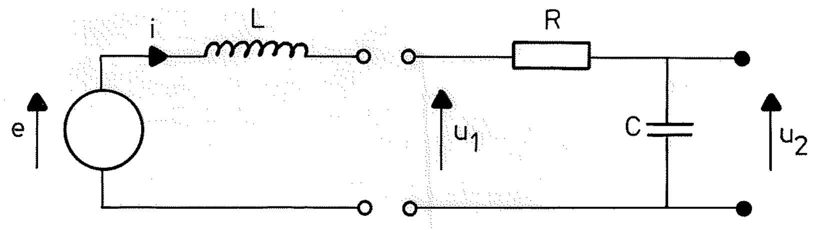


Fig 2.4

The additional equation is

$$u_1 = e - Li'$$

A description of this system could be

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

The expansion of the Macro gives

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

Two equations have the variable I in their left hand parts. 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,Ø]
END
```

The system description can now be done as

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

which is expanded to

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

These statements constitute a legal model in CSSL.

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

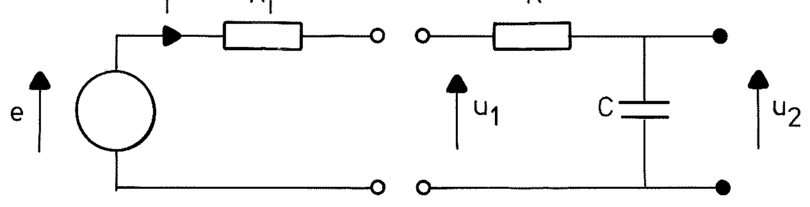


Fig. 2.5

The additional 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,Ø]
```

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 is 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 of the benefits of using equations instead of assignment statements when modelling. It is required that the equations can be manipulated into different forms. Linear systems of equations occur frequently. These could be solved before computations are performed.

3. MODEL LANGUAGE

This chapter contains a description of the model language. The first six sections describe the basic elements of the language such as submodels, equations, cuts, paths and connection statements. Section 7 contains an example. Additional features such as conditional statements, indexed elements, loop statements, difference equations, discrete events and model validity are discussed in section 8.

The language is described by a combination of discussion, examples and syntax. The syntax is developed gradually. The syntax notation used is described in appendix 1. The complete syntax of the language is given in appendix 2.

3.1 Submodels

When models for large systems are developed it is advisable to split the system up into a set of well defined subsystems. The physical structure of the system often suggests suitable subdivisions. 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. A typical example is the use of "control surfaces" in continuum mechanics. To describe the interaction of the subsystem with its environment it is necessary to introduce variables which describe 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.

When a system is split up into subsystems the corresponding submodels can 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 to verify the models separately.

A language for model description should make it possible to represent the structure of the system in a simple way. There should also be a possibility to replace a submodel with another submodel having different complexity. The possibility to create model libraries is another advantage of the submodel concept.

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 of subsystems. Such a structure can be represented as a tree.

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.

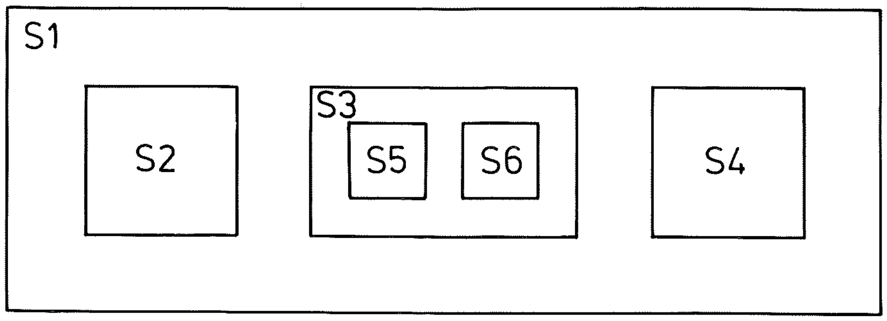


Fig 3.1. A hierarchical submodel structure

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

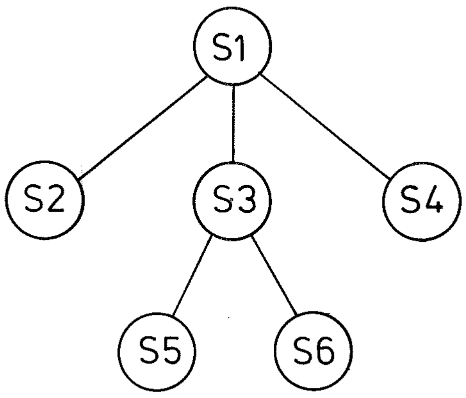


Fig 3.2. Submodel structure represented as a tree

[]

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 to describe 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 description 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 a serious drawback. If two subsystems have the same model the model description must be duplicated. One way to avoid 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 contain the same model M. The previous description of S1 can then be simplified as follows.

```
model S1
  model type M
  ...
  end
  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 syntax.

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

The brackets [ ] denotes that something is optional. The notation { }* denotes repetition one or more times. The complete syntax notation is given in appendix 1.

Ex.

<u>submodel</u> Tank Pipe
<u>submodel</u> Tank(A=5 H=1Ø)
<u>submodel</u> (Tank) Tank1(A=5) Tank2(A=2Ø)
<u>submodel</u> (resistor) R1(5.6) R2(1ØØ) []

The submodel statement

<u>submodel</u> (M) M1 M2 M3

is read

'submodels of type M named M1, M2 and M3'.

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 names of the parameters or they can be given alone in the same order as they are declared 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 ::-notation which can is used in the following way.

<model spec>::=
  <model identifier> [:: <model identifier>]*

In modern programming languages like Simula, Pascal and Algol-68 a dot-notation is used to reference components of data structures. A submodel is often considered as a 'black box'. The reference notation :: was chosen because it looks like a box.

3.2 Interdependence of submodels

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

In the first case the influence from the environment comes through distinct mechanisms e.g. shafts, wires and pipes. It is then practical to introduce variables which describe 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 model which is higher in the hierarchy.

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 gravitational 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.

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 correspondence with these quantities. A variable is a real function of the time and has an associated name. All variables must be declared.

Parameters are basic attributes of a system. They are declared in the model by the statement

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

Parameters can be assigned from superior models or interactively. 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 independent variable is global and is called time. 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 have been introduced to increase the security against bad incorporation of submodels. The declaration of these variables is done using the statements:

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

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

Models are often developed so that they can be used 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 referenced externally.

```
default {<variable> = <number>}*
```

Submodels can be connected implicitly 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 then be declared as external in the submodels themselves. The declarations are

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

One way to connect submodels is to give equations which relate terminal variables of the submodels in a superior model. Since different variables in different submodels can have the same identifier there must be a mechanism to reference them. The dot-notation is a suitable reference mechanism.

```
<variable spec>::= [<model spec>.]<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 phenomenological equations. These are either algebraic or differential equations which relate certain variables.

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 except for variable references. The equations can contain ordinary function procedures written in some algorithmic language.

It is also useful to be able to use ordinary procedures written in an algorithmic 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. The simulation languages CSSL and CSMP have a suitable notation for procedure calls:

```
{<variable>}*   =  <procedure  identifier>({<expression>}*)
```

```
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' or <u>der</u>(x)
second derivative: x'' or <u>der2</u>(x)
etc.

3.5 Cuts and connections

When connecting submodels it is natural to view a submodel in the same way as the corresponding subsystem. One then wants to work with the physical mechanisms that connect the subsystems. Each mechanism is associated with certain variables. These are used internally in the equations and they describe the interdependence with other submodels.

Examples of such mechanisms and their associated variables are:

```
  shaft: angle, torque
  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, torque)
```

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. 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).

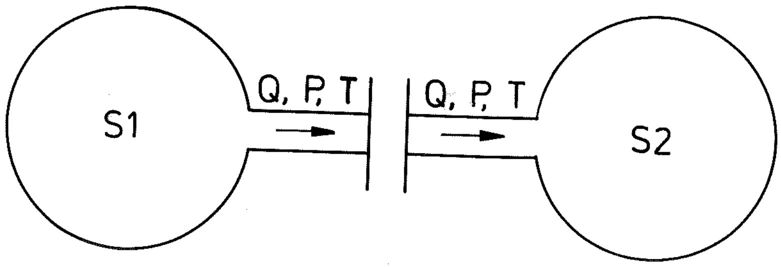


Fig 3.3

The two submodels will contain the cut-declarations:

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

The submodels could be connected from a superior model in the following way.

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

Cut variables are often defined in such a way that connections of subsystems mean that the corresponding variables are set 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:

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

Submodels can then be connected via the cuts with the connection statement.

connect <model identifier>:<cut identifier>
{ at <model identifier>:<cut identifier> }*

The corresponding variables in all cuts are set equal. The same cut can appear in several connection statements. A colon-notation is used when referencing the cuts. The colon was chosen because it associates to a vertical bar which often represents a cut.

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

Example 3.3

Consider the electrical circuit in Fig 3.4.

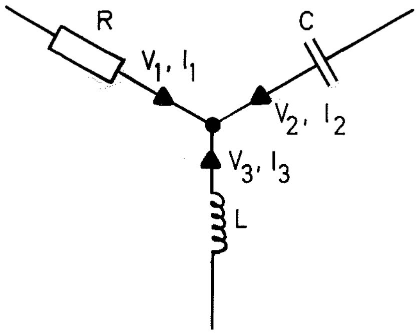


Fig 3.4

The constraints at the connection node are

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

Only the first equation is of the type discussed earlier. A small subsystem with three cuts containing the second equation could of course be introduced to handle the connections. 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

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

would then be equivalent to the following equations

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

[]

Example 3.4

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

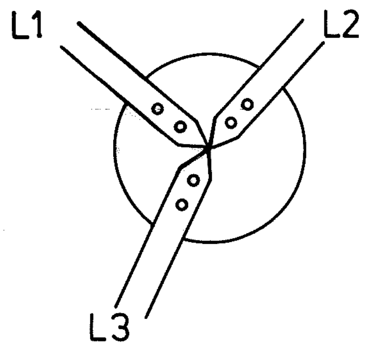


Fig 3.5

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 the following equations.

```
  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 = Ø
  L1.Fy + L2.Fy + L3.Fy = Ø
  L1.Fz + L2.Fz + L3.Fz = Ø

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

[]

The examples show that it is practical to introduce two types of cut variables. The notation across variable is sometimes used in the literature (Koenig, Tokad, Kesavan, 1967) for the variables that are equal in the cuts. The variables that are summed to zero are called through variables.

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

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

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

It is not clear whether there are other types of relations between variables for which it would be practical to have a special notation.

By introducing through variables it is possible to handle 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 preceeded 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 to solve 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.

Nodes

The points where several connection mechanisms are joined are sometimes called nodes. In some cases it is natural to name the nodes and use the names in the connection statements. The standard method for describing electrical networks is to number the nodes and for each component give the numbers of the nodes to which it is connected. One way to declare a node is the following.

node <node identifier>

A submodel can contain equations describing the static or dynamic coupling between its submodels and the environment. There is then a need for an "internal cut" to be connected to the cuts of the submodels. For that reason it is possible to associate variables with a node in the following way.

```
node <node identifier> ( <variable cut> )
```

Hierarchical cuts

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, called hierarchical cuts. Another form of the declaration statement for cuts is thus

```
cut <cut identifier> [ {<cut>}* ]
```

Syntax for cut and node declarations

The complete syntax for declaring cuts and nodes is given below.

```
<cut declaration> ::= [main] cut {<cut identifier>
    [<cut>]}*
  <cut>::=<cut clause> / <cut spec>
  <cut clause> ::= ( <variable cut> ) /
    [ <hierarchical cut> ]
  <variable cut> ::= [ <cut element> ]*
    [ / [ <cut element> ]* ]
  <cut element> ::= <variable> / -<variable> / .
  <hierarchical cut> ::= { <cut> / . }*
  <cut spec> ::= <model spec> [:<cut identifier>] /
    <cut identifier>

<node declaration> ::= node {<node identifier>
    [<node clause>]}*
  <node clause> ::= ( <variable cut> ) /
    [ <hierarchical node> ]
  <hierarchical node> ::= { <node clause> /
    <node identifier> / . }*
```

One cut in each submodel may be declared as main. Some examples of cut and node declarations are given below.

Ex.

```
  cut C1 (v1 v2 / v3 v4)  C2 (v5 . / -v3 .)
  cut C3 [C1 C2]
  main cut C4 [ [ (v6 / v7) (v8 / -v7) ] C1 C2 C3 ]

  node N1 N2
  node N3 (v9 / v1Ø)
  node N4 [N1 N2 N3]
```

[]

<u>Generation of equations</u>

The at-operations on cuts and nodes are translated to equations involving the variables in the cuts.

Equations for the through variables can not be generated until all the at-operations have been processed. The following restriction is made. A cut may only be referenced in the connection statements of one submodel. The reason is that it is then possible to generate equations corresponding to the connection statements of a submodel as soon as the last connection statement of that submodel has been processed. This has some implications. When a submodel is duplicated with the <u>submodel</u> statement it is not necessary to duplicate the connection structure. It will be contained in the equations which are duplicated. It should be possible to obtain a listing of the equations generated. With this restriction the generated equations will be grouped together with the other equations of a submodel.

An at-operation on a variable cut and a hierarchical cut is illegal. The number of across and through variables or subcuts should be the same in all cuts that are connected. An at-operation on two hierarchical cuts are defined as at-operations on corresponding subcuts.

The generated equations should only contain one equality sign and one or more plus signs. If a variable is preceeded by a minus sign in a cut declaration, it is put on the other side of the equal sign. This is important in order not to get cumbersome equations after formula manipulation.

An example illustrates how cuts containing dots are handled.

Example 3.5

The following at operations

```
(v1 v2 / i1 i2) at (v3 . / i3 .) at (v4 v5 / -i4 -i5)
```

gives the following equations

```
v3 = v1
v4 = v3
v5 = v2
i1 + i3 = i4
```

At operations on the following hierarchical cuts

```
[C1 C2] at [C3 .] at [C4 C5]
```

will be replaced by the at operations

```
C1 at C3
C3 at C4
C2 at C5
```

[]

Special attention must be given to the case when a variable cut is used in a connection statement in the same submodel as it is declared. It is easy to see that the through variables of the cuts declared in the same submodel should be negated in order to get consistent equations. The same is true for nodes with an associated variable cut. These facts are illustrated by an example.

Example 3.6

Consider the model structure in Fig 3.6.

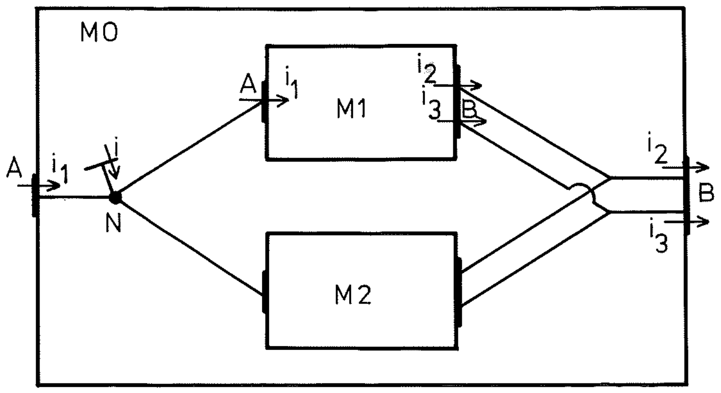


Fig 3.6

The cut declarations and connection statements are shown below.

```
model type M
  cut A (v1 / i1)  B [ (v2 / -i2) (v3 / -i3) ]
  ...
end

model MØ
  submodel (M) M1 M2
  cut A (v1 / i1)  B [ (v2 / -i2) (v3 / -i3) ]
  node N (v / i)

  connect A at N at M1:A at M2:A
  connect B at M1:B at M2:B

end
```

The following equations are generated.

```
v = v1
M1.v1 = v
M2.v1 = M1.v1
M1.i1 + M2.i1 = i1 + i
```

```
M1.v2 = v2
M2.v2 = M1.v2
i2 = M1.i2 + M2.i2
M1.v3 = v3
M2.v3 = M1.v3
i3 = M1.i3 + M2.i3
```

[]

3.6 Model structure

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, become cumbersome to read and they do not contain the structure of the model themselves.

One problem with the description of the model structure is the following. For physical systems the connection structure exists in the three dimensional space. A diagram of the system on a piece of paper is a two dimensional representation. In this case the structure should be represented as a string of characters, which is, in a way, a one dimensional representation.

This section gives an alternative way to describe the structure of a model.

Model graph

The connection structure of a submodel can be considered as a graph. The vertices of the graph are the declared nodes and the cuts in its submodels and in the model itself. The edges of the graph correspond to the connection mechanisms that exists between the nodes and the cuts. Fig 3.7 shows an example of a model graph. An other example can be found in example 7.4 (page 143).

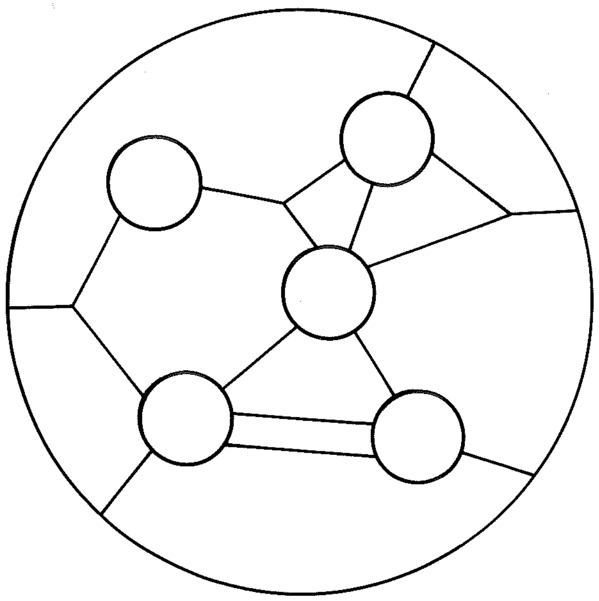


Fig 3.7. A connection structure

The description of such a graph can be done using different principles. One possibility is to concentrate the description on the edges and indicate the vertices each edge is connected to. This can be done in the model langage using the at-operator. The expression C1 at C2 states that there is an edge between the cuts C1 and C2. This type of description does not consider the fact that the cuts are naturally grouped as belonging to the same submodel.

Another possible method is to describe how each submodel is connected. This can be done by giving the vertices to which there are edges from the submodel. When constructing the submodel the cuts are ordered in some way to simplify the connection. The connected vertices are then given in the same order.

This way of describing the connections is the standard method used in analysis programs for electrical networks as e.g. ASTAP (see ASTAP) and TESS (TESS, 1972).

Hierarchical cuts are used for this kind of description in the model language. The ordering of the cuts of the submodel is accomplished by inserting them into a hierarchical cut. The connection is done with the at-operator and a corresponding hierarchical node containing the nodes with edges from the submodel.

Example 3.7

Consider the following submodel.

```
model M
  cut A(v1 v2) B(v3 v4) C(v5 v6)
  main cut D [A B C]
  ...
end
```

This model is connected in a superior model to three nodes N1, N2, N3 using the following statement.

```
connect M at (N1 N2 N3)
```

This connection statement is equivalent to the following statement.

```
connect M:A at N1, M:B at N2, M:C at N3
```

[]

A third philosophy for description of the graph will now be given. It is based on the fact that the connection mechanisms introduces a natural grouping of the submodels.

It is sometimes natural to say that a number of models are connected e.g. in series or in parallel in some respect. The connection mechanisms can belong to different categories, such as electrical or mechanical connections or pipes with different kinds of fluids. It is thus natural to consider several different groupings and relations between the submodels.

Since a model can have several cuts it is important that the description takes the orientation of a submodel into account.

Example 3.8

A verbal description of an industrial plant could contain a sentence like: "Water flows from Pump through Pipe through Valve into Tank". It is then assumed that the outgoing port of the Valve is connected to the inlet of the Tank.

This would be described in the model language as follows.

```
connect (water) Pump to Pipe to Valve to Tank
```

[]

When stating that models are connected after each other or in series in some respect, it is assumed that the models have two sides, between which there exists a direction.

Directions are often inherent in systems. They can e.g. originate from physical observations such as flows through the subsystems and the connection mechanisms. The direction in the model is in fact equal to the direction of the flow when the corresponding variable is positive.

When a system has no inherent directions, the choice of reference directions for variables will impose directions. One example is a resistor in an electrical network. It must be considered as physically symmetric. Its direction will coincide with the current when the current variable is positive.

Another reason to consider directions in a submodel is the perception of the causalities. It is, however, not a well defined concept because one submodel can influence another submodel in one respect but in another respect the influence is in the opposite direction.

As directions are defined in a model it is natural to introduce the concept path. A path exists between cuts at the input and cuts at the output and is declared in the submodels as follows.

path <path identifier> $\underline{<}$ <cut> - <cut> $\underline{>}$

If there are several cuts at the input or at the output they are grouped into a hierarchical cut. Several paths can be declared in a submodel corresponding to different connection mechanisms.

The concept path is also associated with the description of the structure as a text string. There is an inherent direction from left to right in a text string. In the text a model is represented by its identifier. A path in a model thus corresponds to a direction from the left side to the right side of the model identifier.

An alternative way of looking at the introduction of paths is the following. The graph consisting of connection mechanisms, nodes and cuts is normally a set of subgraphs with disjoint vertex sets. By introducing paths, new edges are introduced inside the submodels and a more connected graph is obtained. The graph is then described by selecting a set of subgraphs which are described separately. The division of the graph is arbitrary. The choice is made for convenience. It is guided by natural properties of the system and a desire to have simple descriptions. Such subgraphs can e.g. consist of simple paths, parallel paths, trees and loops.

The union of the vertex sets and edge sets of the subgraphs must be equal to the corresponding sets of the total graph. However, neither the vertex sets nor the edge sets of the submodels need to be pairwise disjoint.

A natural way to describe the subgraphs is to state how the internal edges in the submodels (paths) are joined with edges (connection mechanisms). This description is done with a set of connection operators. The most important is the to-operator.

Example 3.9

Consider the model structure in Fig 3.8.

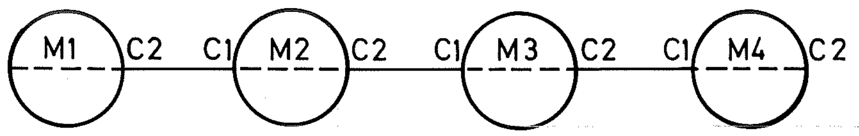


Fig 3.8

This structure can be described with the to-operator as follows.

```
  connect M1 to M2 to M3 to M4
```

If the models M1 - M4 has the following declaration of a path

```
  main path P <C1 - C2>
```

then the connection statement is equivalent to

```
  connect M1:C2 at M2:C1, M2:C2 at M3:C1, M3:C2 at M4:C1
```

[]

Example 3.10

Consider the model structure of Fig 3.7 (page 43) and assume that there are two different types of connection mechanisms and directions as indicated in Fig 3.9.

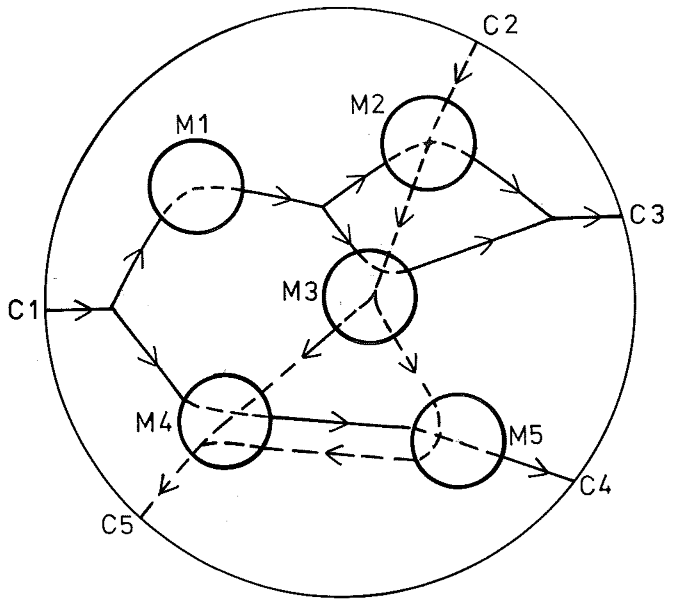


Fig 3.9. A connection structure with directions and paths

If the submodels have paths declared according to the dotted lines inside the submodels then the model structure could be described with the following statements.

<u>connect</u> (path1) C1 <u>branch</u> (M1 <u>to</u> (M2 <u>par</u> M3) <u>to</u> C3
M4 <u>to</u> M5 <u>to</u> C4)

<u>connect</u> (path2) C2 <u>to</u> M2 <u>to</u> M3 <u>to</u> (. M5) <u>to</u> M4 <u>to</u> C5

Note that the second cut of path2 in M3 and the first cut of path2 in M4 is hierarchical.

[]

<u>Syntax for path declaration</u>

The syntax for path declaration is given below.

<path declaration> ::= [<u>main</u>] <u>path</u> {<path identifier>
{<path clause> / <path spec>} }*
<path clause> ::= $\underline{<}$ {<cut>/.} - {<cut>/.} $\underline{>}$

One path in each submodel may be declared as <u>main</u>. Some examples of path declarations are given below.

Ex.

```
  path P1 <C1 - C2>
  path P2 < (v1 / v2) - (v3 / -v2) >
  main path P3 < [C1 C2] - [ (v4 v5) (v6 v7) ] >
```

[]

Connection operand

The connection of the submodels is done with connection statements. The operands in a connection statement are cuts, nodes and paths.

```
  <connection operand> ::= <cut spec> / <path spec> /
    <node identifier>
```

There are several different ways to specify the cuts and paths.

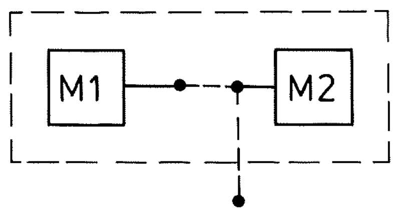
```
  <cut spec> ::= <model spec> [:<cut identifier>] /
    <cut identifier>

  <path spec> ::= <model spec> [..<path identifier>] /
    <path identifier>
```

If no <model spec> is given the named cut or path is assumed to be declared in the submodel where the reference is made. If only a <model spec> is given there are two interpretations. If a <cut identifier> or <path identifier> is given within parentheses after connect in the actual connection statement that identifier is used for reference. If there is no such identifier the operand is the path or cut declared as main.

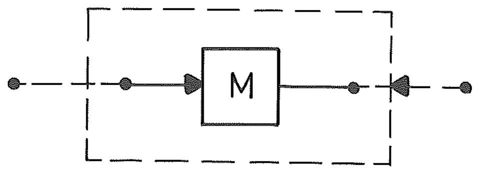
Connection operators

The available connection operators are at, to, from, par, loop, branch, join and reversed. The operators are illustrated in Fig 3.1Ø - 3.17.



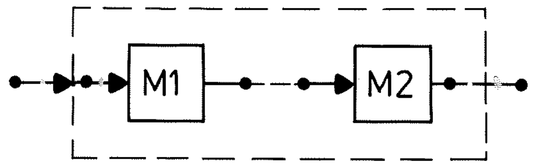
M1 at M2

Fig 3.10



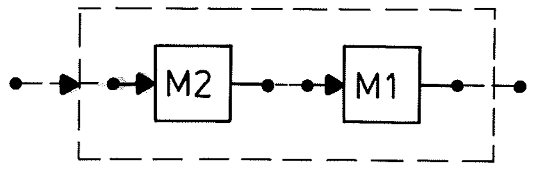
reversed M

Fig 3.11



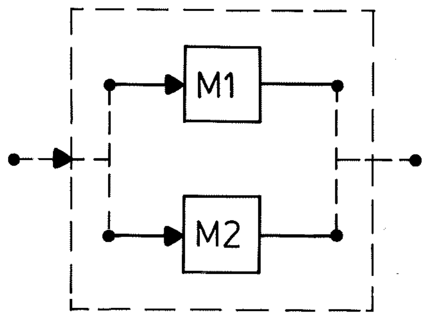
M1 to M2

Fig 3.12



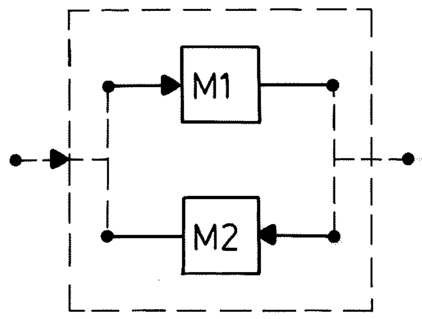
M1 from M2

Fig 3.13



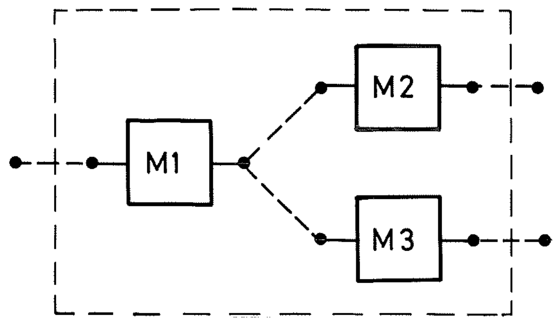
M1 par M2

Fig 3.14



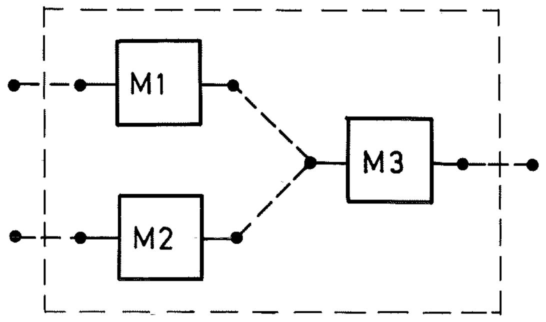
M1 loop M2

Fig 3.15



M1 branch (M2 M3)

Fig 3.16



(M1 M2) join M3

Fig 3.17

A connection expression consists of operands, operators and parentheses. The syntax is given below.

```
  <connection expression> ::= <connection secondary>
    { {at|=|to|-|from|par|//|loop|branch|join}
    <connection secondary> }*
  <connection secondary> ::=
    [reversed|\] <connection primary>
  <connection primary> ::= <connection operand> /
    ( {<connection expression>/.}* )
```

The interpretation of a connection expression is defined using the elementary at-operator. Each operator is translated to a set of at-operations. It also gives a value which is either a cut or a path. The evaluation of the operators is done from left to right if not otherwise indicated by parentheses. One exception is the unary operator reversed which has higher priority than the others. The binary operators have equal priority. However, it might be more natural to give the par and loop operators higher priorities.

Table 3.1 gives the evaluation rules for the operators. The notation C1, C2,... has been used for cuts and nodes and the notation <C1 - C2>,... has been used for paths.

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
5. <C1 - C2> par <C3 - C4>	<C1 - C2>	C1 at C3
		C2 at C4
6. <C1 - C2> loop <C3 - C4>	<C1 - C2>	C1 at C4
		C2 at C3
7. <C1 - C2> branch <[C3 C4 ...] - C5>	<C1 - C5>	C2 at C3
		C2 at C4
		...
8. <C1 - [C2 C3 ...]> join <C4 - C5>	<C1 - C5>	C2 at C4
		C3 at C4
		...
9. (C1)	C1	none
(<C1 - C2>)	<C1 - C2>	none
(C1 C2 ...)	[C1 C2 ...]	none
(<C1 - C2> <C3 - C4> ...)	<[C1 C3 ...] - [C2 C4 ...]>	none

Example 3.11

The structure shown in Fig 3.18 could be expressed as

M1 to (M2 to M3 par M4) to M5

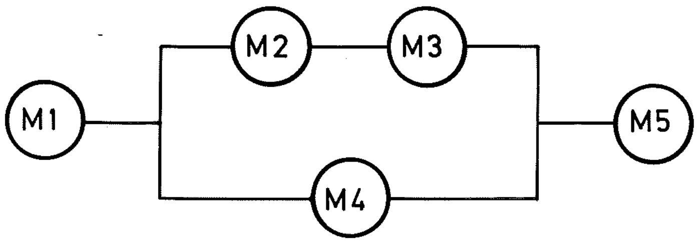


Fig 3.18

Assume that the models M1 - M5 have the following path.

main path P <C1 - C2>

The connection expression could then be written as

```
<M1:C1 - M1:C2> to ( <M2:C1 - M2:C2> to <M3:C1 - M3:C2>
   par <M4:C1 - M4:C2> ) to <M5:C1 - M5:C2>
```

The expression is reduced succesivly generating a set of at-operations. The first expression to evaluate is M2 to M3 because of the parentheses. The resulting at-operation is

```
M2:C2 at M3:C1
```

The result of the operation is the path <M2:C1 - M3:C2> which is inserted into the connection expression.

```
<M1:C1 - M1:C2> to ( <M2:C1 - M3:C2> par <M4:C1 - M4:C2> )
   to <M5:C1 - M5:C2>
```

The next operation to perform is par. It results in two at-operations.

```
M2:C1 at M4:C1
M3:C2 at M4:C2
```

The expression is then reduced to

```
<M1:C1 - M1:C2> to <M2:C1 - M3:C2> to <M5:C1 - M5:C2>
```

The first to-operation gives

```
  M1:C2 at M2:C1
```

The reduced expression is

```
  <M1:C1 - M3:C2> to <M5:C1 - M5:C2>
```

The last operation gives

```
  M3:C2 at M5:C1
```

The result of the connection expression is

```
  <M1:C1 - M5:C2>
```

The effect of the connection expression is five at operations in accordance with Fig. 3.18.

[]

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

at	=
to	-
par	//
reversed	\

Parentheses in connection expression

Parentheses in a connection expression can be used as in arithmetic expressions to indicate priority. However, it is also used to construct hierarchical cuts and to handle parallel paths.

Several connection expressions can be given after each other within parentheses. They are evaluated independently. If the result of all evaluations is a set of cuts then a corresponding hierarchical cut is constructed. If the result is paths then a new path is constructed in which the first cut is hierarchical containing the first cuts of all the paths. The second cut is constructed in the same way. It is not legal to mix cuts and paths. These rules are summarized in rule 9 of table 3.1.

The construction is used e.g. to generate hierarchical nodes when connecting submodels between nodes. Compare example 3.7 (page 42). It is also important together with the branch and join operators. These operators are used to describe subgraphs which are trees.

Connection statement

The syntax for a connection statement is

```
<connection statement> ::=
  connect [(<identifier>)] {<connection expression>}*
```

Each connection expression describes a subgraph of the model graph. The identifier within parentheses is used to specify which cuts and paths of the submodels that are concerned in the connection expressions. Compare connection operand.

Redefinition of cuts and paths

Each level of the hierarchical submodel structure has a connection structure. A connection statement can contain references to cuts and paths declared in the model where the reference is made and in its submodels. It is also possible to reference cuts and paths at lower levels in the submodel hierarchy using the ::-notation. Knowledge about the internal structure of the submodel is then needed. Such references can of course be avoided by declaring a new cut at the outer level. The details of such a cut, e.g. the included variables, are then not interesting. There are thus two ways to redefine a cut at an outer level.

A cut can be declared without <cut clause>.

```
cut <cut identifier>
```

Such a cut can then be used in a connection statement, e.g. as below.

```
connect <cut identifier> at
  <model identifier>:<cut identifier>
```

The same effect can be obtained by the following alternative cut declaration.

```
cut <cut identifier> <cut spec>
```

Paths can also be redefined in a correspong way.

```
path <path identifier> <path spec>
```

An important form of the path declaration is the following.

```
path <path identifier> < <cut spec> - <cut spec> >
```

This is exemplified below.

Example 3.19

Consider the submodel structure of Fig 3.19.

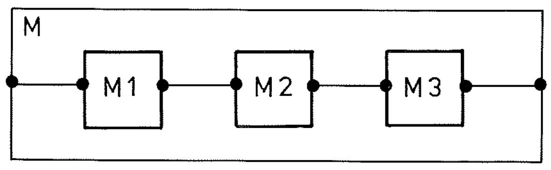


Fig 3.19

Assume that the submodels M1, M2 and M3 have the following declarations.

```
cut C1(...) C2(...)
main path P <C1 - C2>
```

It should be possible to consider M as having a path between the cut C1 of M1 and the cut C2 of M3 when connecting the submodel M. This can be accomplished by the following description of M.

```
model M
  submodel M1 M2 M3
  main path P <M1:C1 - M3:C2>
  connect M1 to M2 to M3
end
```

[]

3.7 Example

The ideas presented in the previous sections are illustrated by an example. Further examples are given in chapter seven.

Consider the electrical network in Fig 3.2Ø.

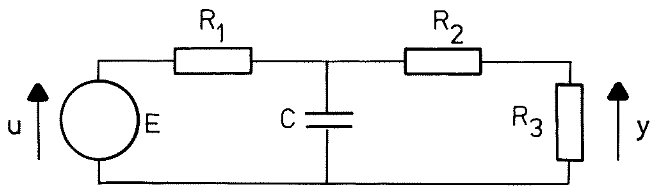


Fig 3.2Ø

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

```
model Network
  input u
  output y
  local $i_1$ $i_2$ $v_c$
  parameter $R_1$ $R_2$ $R_3$ C
  $u=R_1*i_1+v_c$
  $v_c=R_2*i_2+R_3*i_2$
  $y=R_3*i_2$
  $C*v_c'=i_1-i_2$
end
```

Another approach when modelling the network is to develop a library of electrical components which are then connected together.

A library consisting of resistor, capacitor, coil, voltage source, current source and common node is given on page 58. The components are prepared for connections according to the different philosophies discussed. The main cuts are used when connecting the components between nodes and the main paths are used when following paths in the network. The voltage V of a voltage source is a terminal variable. Its value is given using the dot notation. The same holds for the current I of a current source. The cut declarations in the current source contains dots corresponding to the terminal voltages because a current source is independent of the voltages. The description of each network should contain <u>one</u> submodel of type Common. The cut declaration in the submodel Common contains a dot corresponding to the current. The redundant current equation for the network is eliminated in this way (see page 34).

```
{ Library of basic electrical components }

model type resistor
  cut A (Va / I) B (Vb / -I)
  main cut C [A B]
  main path P <A - B>
  local V
  parameter R
  V = Va-Vb
  R*I = V
end

model type capacitor
  cut A (Va / I) B (Vb / -I)
  main cut C [A B]
  main path P <A - B>
  local V
  parameter C
  V = Va-Vb
  C*der(V) = I
end

model type coil
  cut A (Va / I) B (Vb / -I)
  main cut C [A B]
  main path P <A - B>
  local V
  parameter L
  V = Va-Vb
  L*der(I) = V
end

model type voltage
  cut A (Va / I) B (Vb / -I)
  main cut C [A B]
  main path P <A - B>
  terminal V
  V = Vb-Va
end

model type current
  cut A (. / I) B (. / -I)
  main cut C [A B]
  main path P <A - B>
end

model type Common
  main cut C (V / .)
  V = Ø
end
```

This library of components have been used for the following two alternative descriptions of the network in Fig 3.2Ø.

```
model Network
  submodel(resistor) R1 R2 R3
  submodel(capacitor) C
  submodel(voltage) E
  submodel Common
  input u
  output y
  connect Common to E to R1 to (C par (R2 to R3))to Common
  E.V = u
  y = R3.Va
end
```

```
model Network
  submodel(resistor) R1 R2 R3
  submodel(capacitor) C
  submodel(voltage) E
  submodel Common
  node NØ N1 N2 N3
  input u
  output y

  connect Common at NØ,
  E at (NØ N1),
  R1 at (N1 N2),
  C at (N2 NØ),
  R2 at (N2 N3),
  R3 at (N3 NØ)

  E.V = u
  y = R3.Va
end
```

The first description generates the equations listed below. The equations for the second description will not be exactly the same, but they will be equivalent.

Submodel	Equation
R1	V = Va - Vb
	R*I = V
C	V = Va - Vb
	C*der(V) = I
E	V = Vb - Va
Common	V = Ø
R2	V = Va - Vb
	R*I = V
R3	V = Va - Vb
	R*I = V
Network	E.V = u
	y = R3.Va
	R1.Va = E.Vb
	R1.I = E.I
	R3.Va = R2.Vb
	R3.I = R2.I
	R3.Vb = C.Vb
	Common.V = R3.Vb
	E.Va = Common.V
	C.Va = R1.Vb
	R2.Va = C.Va
	C.I + R2.I = R1.I

3.8 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 <u>if</u>-<u>then</u>-<u>else</u> statement or a <u>case</u> 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. It is possible that a transformation of the equations must be done at such a point.

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 3.2Ø

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 dividing the heat exchanger into a number of sections each described by ordinary differential equations (see Fig 3.21).

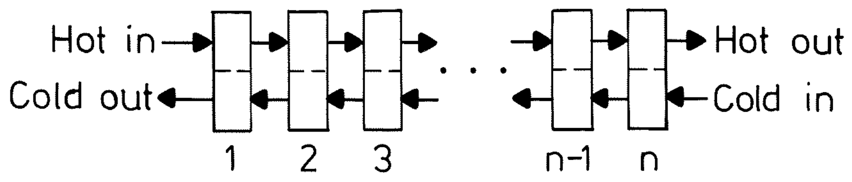


Fig 3.21

The degree of approximation depends of the number of sections. This means that the connection of the sections should be done in such 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
  cut hotin(...) hotout(...) coldin(...) coldout(...)
  path hot <hotin - hotout>
  path cold <coldin - coldout>
  ...
end

model heatexchanger
  structure parameter n
  submodel section[n]
  path hot <section[1]:hotin - section[n]:hotout>
  path cold <section[n]:coldin - section[1]: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
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 (Birthwistle et al, 1973).

Changes in model equations could be handled with structural parameters appearing in if-then-else-, case- and for-statements. 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 fulfilled. 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
  $v_i + v_j + \ldots = v_k + \ldots$
  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 model. A mathematical notation for the models described in the model language is

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

where
- t - time
- x - variables that appear differentiated
- u - inputs
- y - outputs
- p - parameters
- z - other variables

Derivatives of u and y have been eliminated by introducing auxiliary variables. The number of equations is equal to dim x + dim z + dim y for a well posed model.

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 $x_\emptyset(t)$, $z_\emptyset(t)$, $u_\emptyset(t)$ and $y_\emptyset(t)$. Introduce the deviations

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

etc.

Insertion into the model (4.1) gives.

$$f(t,\ x_\emptyset'+Dx',\ x_\emptyset+Dx,\ z_\emptyset+Dz,\ u_\emptyset+Du,\ y_\emptyset+Dy,\ p) = \emptyset$$

Linearization gives

$$f(t,x_\emptyset'(t),x_\emptyset(t),z_\emptyset(t),u_\emptyset(t),y_\emptyset(t),p) +$$
$$\frac{df}{dx'}(.)Dx' + \frac{df}{dx}(.)Dx + \frac{df}{dz}(.)Dz + \frac{df}{du}(.)Du + \frac{df}{dy}(.)Dy = \emptyset$$

Note that partial derivatives are denoted df/dx. The arguments of the Jacobians are the same as for f. If new notations are introduced the linear model can be written

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

If $x=x_\emptyset$, $z=z_\emptyset$, $u=u_\emptyset$ and $y=y_\emptyset$ is a solution to the original model then $F(t)=\emptyset$. In some cases the matrices are constant. The linear model is then:

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

4.3 State equations

The original model can be simulated directly. This will be demonstrated in chapter 5. Many integration methods are developed for systems in state space form:

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

See for e.g. Lambert (1973). Other methods for analysing dynamical systems also use this form. These are reasons for transforming the model to state space form if possible.

If

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

then there are 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 left out.

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

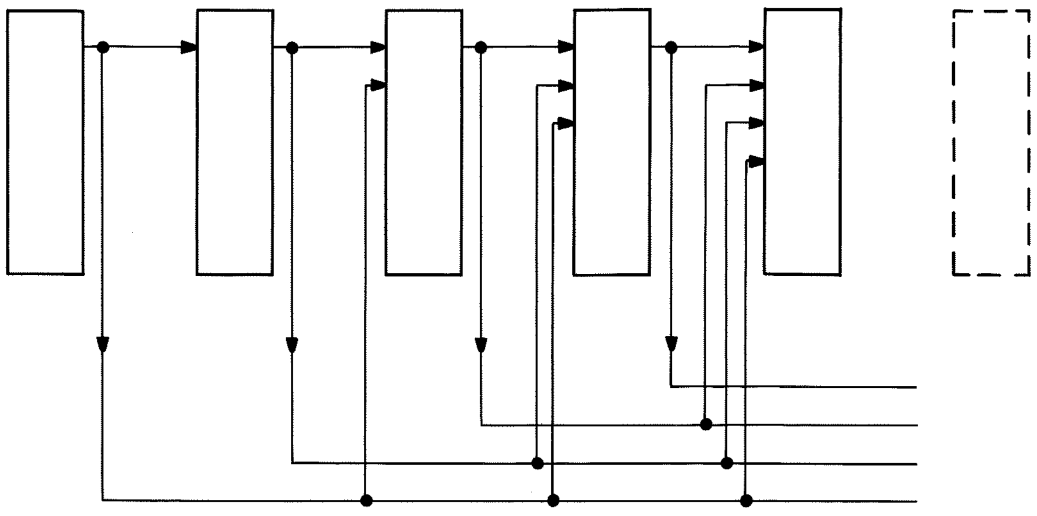


Fig 4.1

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 in Fig 4.1.

Structural controllability and observability

It is difficult to see how different variables can influence 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 type can easily be obtained with a computer and presented to the user.

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

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

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 $t>t_\emptyset$. This situation correspond to connecting different subsystems with known initial values at $t=t_\emptyset+\epsilon$.

Another way of specifying the problem is to give initial values 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 all components of x. The variables x are only characterized by the appearance of their derivatives. It should be possible to give initial values to some of the components of x, z and

y. Introduce the notation x_1, z_1 and y_1 for these components. Some equations should if possible be selected which after transformation gives the initial value for the rest of x, x_2.

$$x_2(t_\emptyset) = g(t_\emptyset, x_1(t_\emptyset), z_1(t_\emptyset), u(t_\emptyset), y_1(t_\emptyset), p)$$

The structure of the equations gives constraints on the selection of x_1, z_1 and y_1. The dimension of x_1 can of course be zero. When x_2 is computed then the 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 $t=t_\emptyset$.

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

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

Find values of the parameters such that the criterion

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

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 express dependence between parameters. The optimization procedure 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 under 'design variable selection'.

The optimization problem can now be formulated as: Select values for 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

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

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.

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

The initial value for this differential equation is

$$\frac{dx}{dp_3}(t_\emptyset) = \frac{dx_\emptyset}{dp_3}$$

The integration of the original equations and the equations for dx/dp_3 is done at the same time.

Static model

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

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

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.

The following example shows how the equations are manipulated in different ways depending on the operation on the model.

Example 4.1

Consider the equations for the network in section 3.7.

$$u = R_1*i_1 + v_c$$
$$v_c = R_2*i_2 + R_3*i_2$$
$$y = R_3*i_2$$
$$C*v_c' = i_1 - i_2$$

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'.

$$i_1 = (u-v_c)/R_1$$
$$i_2 = v_c/(R_2+R_3)$$
$$y = R_3*i_2$$
$$v_c' = (i_1-i_2)/C$$

The derivative of v_c is zero for a static model. The static relationship between u and y is obtained if u and the parameters are assumed known. The following system of equations are obtained.

$$\begin{bmatrix} R_1 & 0 & 1 \\ 0 & R_2+R_3 & -1 \\ 1 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \\ v_c \end{bmatrix} = \begin{bmatrix} u \\ 0 \\ 0 \end{bmatrix}$$

The additional equation is

$$y = R_3*i_2$$

If the system of equations is solved the following static input - output relationship is obtained.

$$y = R_3*u/(R_1+R_2+R_3)$$

For this example it could of course be obtained directly from Fig 3.20.

The model equations can be used for design purposes. It is not possible to determine all parameters from static considerations. Assume e.g. that R_3 is known and that R_1 and R_2 should be determined. The input u, the output y and the operating point v_c are also assumed known. The following equations are obtained.

$i_2 = y/R_3$

$R_2 = (v_c - R_3 * i_2)/i_2$

$i_1 = i_2$

$R_1 = (u - v_c)/i_1$

[]

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

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

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

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

See e.g. Lambert(1973).

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

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

along the trajectory. This condition is not always fulfilled as shown 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

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

This equation can be solved by using Newton's method. Introduce the notation $x_n = x(t_n)$.

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

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

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

The iteration index is m.

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

$$\det\begin{bmatrix}\frac{df}{dx'}\frac{1}{h} + \frac{df}{dx} & \frac{df}{dz} & \frac{df}{dy}\end{bmatrix} \neq \emptyset \tag{5.4}$$

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. It is thus possible to integrate the equations even if they can not be reduced to state space form.

<u>Example 5.1</u>

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

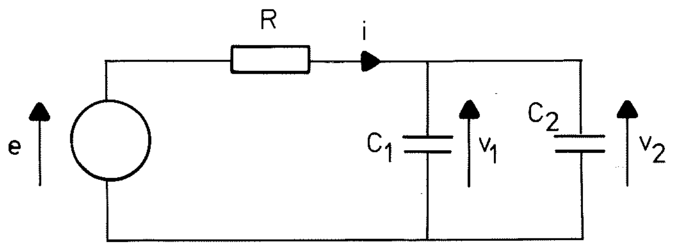


Fig 5.1

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.

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

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

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

$$n = 1,2,...$$

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

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

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

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

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

For $n \geq 2$ 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 implicite 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's method. 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 is 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 important if the equations are 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 correspond to the equations and the variables. Edges must not connect two nodes from the same set. 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.

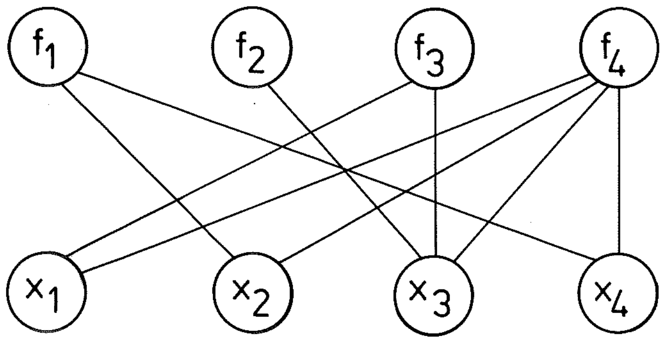


Fig 5.2. A bipartite graph

<u>Output set</u>

In the following sections the system of equations will 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

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

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) (see procedure Assign in the program listing).

Partitioning

Partitioning is used to permute both equations and variables independently in such 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, see Fig 5.3.

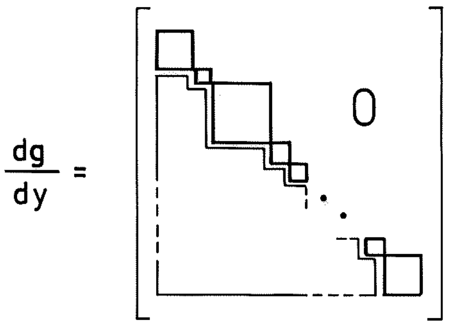


Fig 5.3

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 successively 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's method can be used to solve nonlinear equations. See e.g. Ortega, Rheinboldt (1970).

Algorithms for partitioning can be found in e.g. Steward (1965), Tarjan (1972) (see procedure Strongconnect in the program listing) and Wiberg (1977). Wiberg also gives a comparison between some algorithms. Some of the algorithms first find an output set. The equations and the variables are then permuted by the same permutation matrix.

Partial partitioning

If the system of equations is underdetermined it is sometimes useful to split the equations into two parts: equations that can be solved and equations that can not be solved. The problem can be formulated as follows.

Assume the equations

$$f(x) = 0 \qquad \dim f < \dim x$$

Find permutation matrices P and Q and a partitioning of the equations and the variables such as

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

in a way that $dg_1/dy_2 = 0$ and $\dim g_1 = \dim y_1$ is maximal.

The structure of the Jacobian after permutation is shown in Fig 5.4.

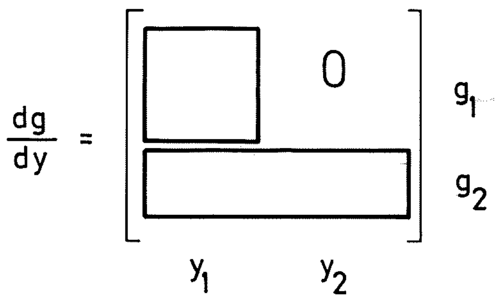


Fig 5.4

The variables y_1 can be solved from g_1.

An algorithm for solving this problem is given below. It is based on adding a set of equations. All of them depend on all variables. The total set of equations are then partitioned.

In order to have as many equations as unknown variables a set of equations I(x) is added.

$$I(x) = \emptyset \qquad \frac{dI_i}{dx_j} \neq \emptyset \; ; \; \forall\, i,j$$

$$\dim I = \dim x - \dim f$$

All the equations are then partitioned by an arbitrary algorithm. The partitioning algorithm fails if it is not possible to assign a variable to some equation. It will thus fail for any other equations that makes the system of equations complete. The problem is thus badly posed if the partitioning algorithm fails.

Since all equations in I(x) depend on all variables it is obvious that they will all be contained in the last system of equations (last strong component of the corresponding graph). The equations in all the other strong components correspond to $g_1(x)$. The equations of g_2 are found in the last strong component. The structure of the division of the equations into g_1 and g_2 is thus correct. The dimension of g_1 is maximal since the equations of g_2 are contained in a minimal system of equations that have to be solved simultaneously. The algorithm thus solves the problem.

An example of the use of the algorithm is given below.

Assume the following system of equations.

$$f(x,y,z) = \emptyset \quad ; \quad \dim f = \dim x$$

The variables y and z are assumed known and x should be solved. Assume, however, that the equations should be solved many times for different y and z and that y is changed more often than z. To make the computations more effective the equations should be split up into two sets.

$$F_1(x_1, z) = \emptyset \qquad \dim F_1 = \dim x_1$$
$$F_2(x_1, x_2, y, z) = \emptyset \qquad \dim F_2 = \dim x_2$$

The equations F_1 then do not have to be evaluated as many times as the eqautions F_2.

The problem is solved by making partial partitioning of the equations that do not depend on y. F_1 then corresponds to g_1. F_2 corresponds to g_2 and the equations depending on y.

Tearing

Tearing is a method to decrease the number of iterated variables when solving systems of equations with iterative technique. Tearing was introduced by Kron (1963).

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 partitioning 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 occur when the dimension of y_2 is high. Algorithms for tearing have 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 does not necessarily give the minimal dimension of y_2 but could be practically 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 make the computations more effective. A good survey of such methods can be found in Tewarson (1973).

5.3 State equations

Consider the basic model

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

The outputs y have been included in z.

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. The equations should then be sorted in a way that it is possible to solve for x' and z when t, x, u and p are known. The partitioning algorithm should thus operate on a function with the following structure.

$$F(x', z) = 0$$

The derivatives of the state variables have to be calculated many times during a simulation. It is thus important to make these computations efficient. A common situation is that many of the variables do not change their value at each evaluation. The computations could thus be made more efficient by grouping the variables according to their dependencies. A natural grouping of the variables (and equations) is the following.

1. Variables which only depend on parameters
2. Variables which only depend on parameters and time
3. Derivatives and variables needed when computing derivatives
4. Communication variables

This grouping could formally be expressed as follows.

$$\begin{aligned}
z_1 &= F_1(p) \\
z_2 &= F_2(t,\ z_1,\ p) \\
\begin{bmatrix} x' \\ z_3 \end{bmatrix} &= F_3(t,\ x,\ z_1,\ z_2,\ p) \\
z_4 &= F_4(t,\ x',\ x,\ z_1,\ z_2,\ z_3,\ p)
\end{aligned}$$

The first group of variables can be calculated once before the iterations start. The second group is evaluated each time the time variable is changed. The third group is computed when the integration routine needs the values of the derivatives. The groups 2 and 3 are considered separately mainly because the implicite integration algorithms need the Jacobian. Numerical evaluation of the Jacobian requires repeated calculation of the derivatives for different values of x. The fourth group of variables are not needed to integrate the differential equations. It should be possible to get them printed and plotted. They need only be computed at points which are accepted by the integration routine.

A method to find these groups of variables and equations is given below.

The first group is found by making partial partitioning of the equations which do not depend on t and x as described on page 87. The second group is found by making partial partitioning of the rest of the equations which do not depend on x.

An output set is then determined for the rest of the equations. The problem is then to find out the equations needed to calculate the derivatives. This is done by backtracking the variables from the equations assigning the derivatives. The variables in such an equation are studied. The equations corresponding to variables which have not been selected previously are added to group 3. This step is repeated for each equation included.

This part of the algorithm could be implemented as a depth first search in the corresponding directed graph. Tarjan's (1972) algorithm Strongconnect for finding the strong components of a directed graph makes a depth first search. By making Strongconnect start its depth first searches at equations assigning derivatives it will find the equations belonging to group 3. The partitioning is also wanted and is found at the same time. The remaining equations belong to group 4.

Differentiation of equations

If the dynamical order of the system is less than the number of variables appearing differentiated 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.

A formal analysis of the properties of this algorithm has not been made. It is conjectured, however, that it will give the state equations if condition (5.4) on page 80 is satisfied structurally.

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

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

to

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

Note that derivatives of u are allowed. 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 in parallel with reapeated partial 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

$$v_1 = F_1(t,x_1,z_1,u,p)$$
$$\cdot$$
$$\cdot$$
$$v_i = F_i(t,x_1,z_1,u,u',\ldots,p,v_1,\ldots,v_{i-1})$$

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 are differentiated. The following equations are thus obtained.

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

Since variables in previous output sets now appear differentiated the corresponding equations have to be differentiated. This step is performed until no more new variables appear differentiated.

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 are 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 variables 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 some examples.

Example 5.2

The model equations for the network in example 5.1 are

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

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

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

and the following system of equations in v_1' and v_2'

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

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

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

Example 5.3

Consider the following model of a pendlum with a moving pivot (Åström, 1976).

$$x' = z$$
$$z' = -\sin x + u \cos x$$
$$y = x$$

The input u is the acceleration of the pivot. x is the angle and z is the angular velocity.

Assume that the inverse model is wanted. The input is then y. The inverse model has no state variables. The differentiation algorithm gives the following sorted equations.

```
x = y          add x' = y'
x' = y'
z = x'         add z' = x'',  x'' = y''
x'' = y''
z' = x''
u = (z' + sin x) / cos x
```

Elimination of x, x', x'', z and z' gives

```
u = (y'' + sin y) / cos y
```

[]

<u>Example 5.4</u>

Consider a superheater for steam. A complete model is given in example 7.4 (page 146). The equations are in fact valid for many other similar processes.

The energy balance is

```
E' = Qin - W*(h - hin)
```

where E = stored energy in the superheater, Qin = incoming heat flow, W = mass flow rate of steam, hin = entalphy of incoming steam and h = entalphy of steam in the superheater.

The stored energy can be expressed as

```
E = V*r*h
```

where V = volume and r = density of steam.

The density is related to the entalphy and pressure according to the Moliere diagram. The pressure is assumed known.

```
r = g(h)
```

The function g could e.g. be implemented using interpolation in tables.

The variables Qin, W, hin and the pressure are considered as inputs to this submodel. There are three alternative ways of chosing state variables: E, h or r.

If E is chosen as state variable the sorted equations are

$$\left.\begin{aligned} E &= V*r*h \\ r &= g(h) \end{aligned}\right\} \quad \text{nonlinear system of equations in r and h}$$

$$E' = Qin - W*(h - hin)$$

If h is chosen as state variable the differentiation algorithm gives.

$$E' = Qin - W*(h - hin)$$
$$r = g(h)$$
$$E = V*r*h$$

$$\left.\begin{aligned} E' &= V*(r'*h + r*h') \\ r' &= g_h(h)*h' \end{aligned}\right\} \quad \text{linear system of equations in r' and h'}$$

The derivative of h is obtained after solving the linear system of equations.

$$h' = E' / (V*(g_h(h)*h + r))$$

If r is chosen as state variable the algorithm gives

$$h = g^{-1}(r)$$
$$E' = Qin - W*(h - hin)$$
$$E = V*r*h$$

$$\left.\begin{aligned} E' &= V*(r'*h + r*h') \\ h' &= g^{-1}{}_r(r)*r' \end{aligned}\right\} \quad \text{linear system of equations in r' and h'}$$

The derivative of r is thus

$$r' = E' / (V*(h + r*g^{-1}{}_{r}(r)))$$

It is difficult to know which choice of state variable is the most effective one for computations. The first choice gave a nonlinear system of equations. The partial derivative g_h is needed for the second choice. The inverse of g and its partial derivative was needed for the third choice.

It is, however, interesting to note that it is very easy to try different alternatives since the manipulations of the equations needed are done automatically by the algorithm. The example also shows a practical problem with design of an algorithm which automatically choses the state variables of a nonlinear model.

[]

5.4 Formula manipulation

Solution of linear equations

It has been stressed earlier that the equations to be solved can often be solved sequentially, one variable at a time. In many cases the equations are also 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. Moreover, it is very interesting to get the manipulated equations written in symbolic form.

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) into $f_\emptyset$ 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-6Ø (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 are 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 5.5

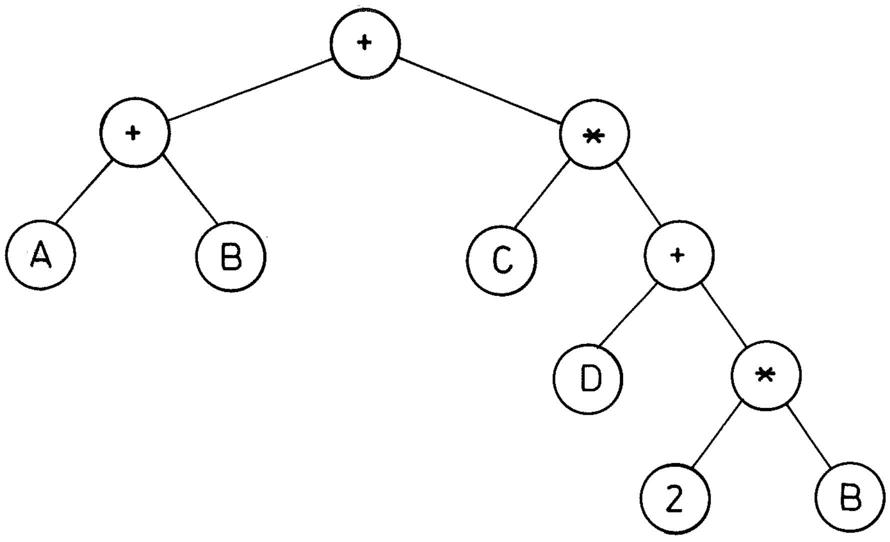


Fig 5.5

Fig. 5.6 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.

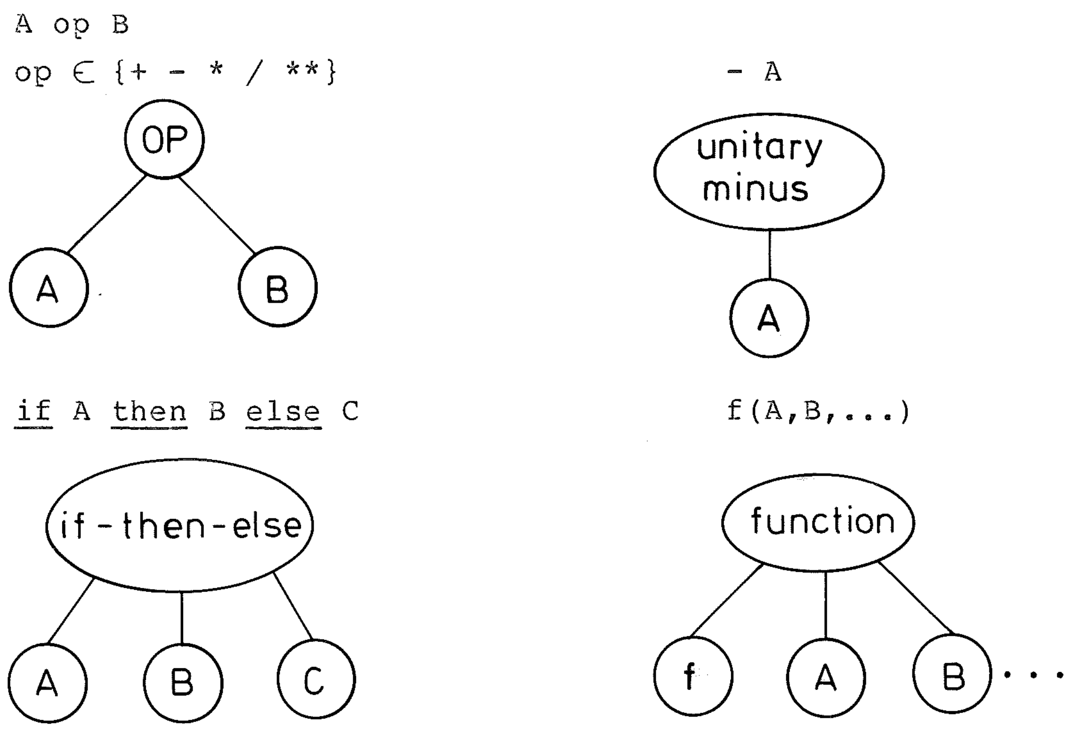


Fig 5.6

Different information can be obtained from the syntax tree by traversing 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 5.7.

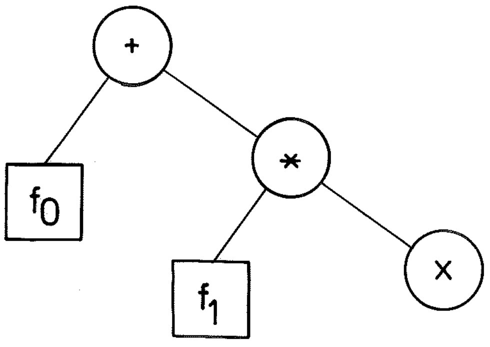


Fig 5.7

The squares indicate the syntax trees for the expressions $f_Ø$ and f_1 which do not contain the unknown variable x.

The transformation is done easily 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 5.8

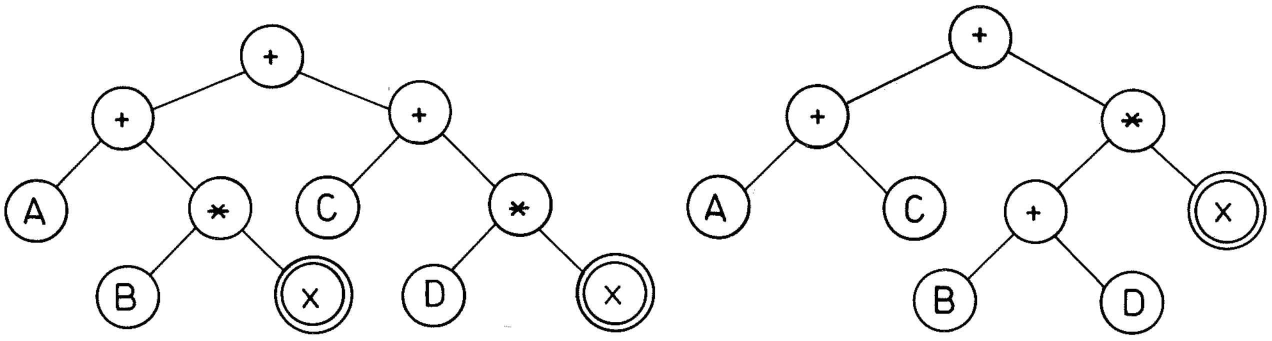


Fig 5.8

The second tree has the desired 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.

Symbolic 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. The numerical solution of problems involving Jacobians may be speeded up if symbolic differentiation is used instead of numerical differentiation.

Symbolic differentiation of an expression is easily done using the syntax tree. A modified syntax tree is built up in the same way as when solving linear equations. The differences are the rules of modification. The rules for symbolic differentiation are given in Table 5.2. Joss (1976) gives a method and a program for differentiating an algorithmically defined function. It accepts an Algol procedure defining the function as input and produces another Algol procedure which evaluates the partial derivatives with respect to specified variables.

Table 5.1 Rules of transformation

1. $(f_\emptyset+f_1x) + (g_\emptyset+g_1x) = (f_\emptyset+g_\emptyset) + (f_1+g_1)x$
2. $(f_\emptyset+f_1x) - (g_\emptyset+g_1x) = (f_\emptyset-g_\emptyset) + (f_1-g_1)x$
3. $(f_\emptyset + f_1x) * (g_\emptyset + \emptyset x) = f_\emptyset g_\emptyset + f_1 g_\emptyset x$
   $(f_\emptyset + \emptyset x) * (g_\emptyset + g_1x) = f_\emptyset g_\emptyset + f_\emptyset g_1 x$
4. $(f_\emptyset + f_1x) / (g_\emptyset + \emptyset x) = f_\emptyset/g_\emptyset + f_1/g_\emptyset x$
5. $(f_\emptyset + \emptyset x) ** (g_\emptyset + \emptyset x) = f_\emptyset ** g_\emptyset + \emptyset x$
6. $F(f_\emptyset + \emptyset x) = F(f_\emptyset) + \emptyset x$
7. <u>if</u> h <u>then</u> $(f_\emptyset + f_1x)$ <u>else</u> $(g_\emptyset + g_1x)$ =
   (<u>if</u> h <u>then</u> $f_\emptyset$ <u>else</u> $g_\emptyset$) +
   (<u>if</u> h <u>then</u> f_1 <u>else</u> g_1)x

<u>Table 5.2</u> 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'
   if h is independent of the differentiation variable
```

```
class SUBMODEL
  procedure Modelbody
  ref(submodel) procedure Newsubmodel

------------------------------

class CUT
  procedure Atop

cut class NODECUT
  procedure Duplicate

cut class HIERARCHICALCUT
  procedure Scanclause
  procedure Duplicate
  procedure Atop

cut class VARIABLECUT
  procedure Scanclause
  procedure Duplicate

cut class CONNECTNODE
  procedure Connect

------------------------------

class VARIABLE
  procedure Duplicate
  procedure Infix

class EQUATIONNODE
  procedure Infix
  procedure Traverse
  procedure Duplicate
  ref(expr) procedure Solve

ref(expr) procedure PRIMARY
                    FACTOR
                    TERM
                    SIMPEXPR
                    EXPRESSION
                    EQUATION

class EXPR
  ref(expr) procedure Add
                      Sub
                      Mult
                      Div
                      Power
                      Equal

ref(expr) procedure MINUS
ref(expr) procedure IFTHENELSE

expr class VARIABLENODE, FUNCTIONOP, NUMBERNODE,
           MINUSOP, IFTHENELSEOP
  procedure Infix
  procedure Traverse
```

6. IMPLEMENTATION OF TRANSLATOR

A language translator has been written in the programming language Simula (Birthwistle et al, 1973) on a Univac-11Ø8 computer. A model written in the model language can be entered. The output from the program is the model equations. They are sorted and solved with respect to the unknown variable. It is possible to specify which variables are known and unknown. The program is listed in appendix 3.

When developing a language it is useful to have a translator available for testing ideas and investigating examples.

The idea behind the model language is not only to generate computer instructions for different types of calculations. The computer should also be an aid when preparing the model. The equations are sorted and grouped into systems of equations using a partitioning algorithm. This structural analysis is interesting for the modeller. The sorted equations indicate the causality relations in the model which should be compared with the modeller's perception of the causalities of the system. The systems of equations correspond to algebraic loops. The systems of equations should thus be studied in order to see if the corresponding strong coupling between the submodels is reasonable. It is interesting to test these ideas during practical modelling of large systems. A translator is then needed.

The implementation of a translator gives a test on the language that it can be translated. It is also an aid for sorting out constructions that are difficult to implement.

The description of the semantics is a problem with language definition. The semantics are contained in a translator. By writing a translator in a high level programming language a useful description of the semantics is obtained.

External description of the program

The program is interactive. The input to the program is the model and commands which specify what should be done with the model. The available commands are described below.

```
  partition
```

This command sorts and groups the equations into systems of equations that have to be solved simultaneously. The variable to be solved from each equation is also selected. The partitioning fails if the problem is structurally singular. A printout of unassigned variables and redundant equations are obtained.

```
  print { variables / known / unknown /
          equations / sorted / solved }
```

This command prints variables or equations on the terminal. The list of variables can contain all variables, only known variables or only unknown variables. The list of equations can contain the equations in the order they were generated during compilation or they can be sorted and grouped into systems of equations. If 'solved' is specified the equations are sorted and grouped and the equations which are not included in a system of equations and which are linear in the unknown variable are solved.

The format for printing equations contain two columns, one for model specifications and one for the equations. If the model specification is the same as on the previous line it is omitted.

A system of equations is preceeded and followed by blank lines. All equations included in the system are preceeded by -.

The variable assigned during partitioning is enclosed by [ ] in equations which are not solved.

```
known { <variable spec> }*
unknown { <variable spec> }*
```

These commands specifies variables as known or unknown. All variables which appear differentiated, parameters and inputs are assumed known at compile time. The simulation problem is thus default.

```
do [not] eliminate
```

This command indicates whether trivial equations of type a=b should be eliminated or not. No elimination is default. If elimination should be performed, one of the variables which are equal is selected to substitute the others in the equations.

```
stop
```

This command stops the program.

```
@add <file name>
```

The model is also read from the terminal. The operating system of Univac-11Ø8 allows, however, that the input is temporarily taken from a file. This is done with the add-command. This facility has been used to simplify the program.

Data structure

Equations are the basic structure in the program. In order to manipulate the equations they are stored as a syntax tree (see section 5.4). The nodes in the syntax tree are objects from subclasses to class Expr.

There is sometimes a problem with objects having references to subobjects. If the number of subobjects is not known when the processing of them is started it is not possible to allocate a vector to hold the references. This situation occurs several times in the program. A submodel has submodels, variables, equations, cuts and paths as subobjects. A cut has either cuts or variables as subobjects.

Two different solutions to this problem are used in the program. The first is based on the fact that an arbitrary tree can be transformed to a so-called Knuth binary tree (see Page and Wilson, 1973). Each node in a Knuth binary tree has at most two pointers.

The submodel structure is a tree (see Fig 3.2, page 21). Instead of having a vector of references to submodels in each submodel there are just two pointers.

```
ref(submodel) firstsubmodel
ref(submodel) companionsubmodel
```

The other method is based on use of a global vector for storage of references. An object then has a segment of the vector which references the subobjects. The specification of the segment can be done with two integers, the bias in the vector and the number of subobjects. This method is not suitable for hierarchical structures.

Variables, equations, cuts and paths which are subobjects to submodels are stored using a global vector. A submodel thus has the following attributes.

```
integer ivariables, nvariables
integer iequations, nequations
etc.
```

For variables and equations there are two advantages by using global vectors for storage. The equations are sorted after compilation. The submodel structure is then not of interest. It is then natural to consider the variables and the equations as two sets with references stored in a vector. The other advantage comes from the fact that if there are several submodels of one type then in principle the same equations will appear several times in the set of equations. The only difference is the variable references. They refer to variables in the submodel itself and to all its submodels. Since references to all variables are stored in a vector it is possible to have relative references in the syntax trees. A variable node in a syntax tree contains an integer which is the number of the variable in the current submodel. The bias is fetched from the current submodel and added to give the complete reference. The vector Equations contains pointers to objects from class Equationnode. They have pointers to the syntax trees and to the actual submodels.

The parts of the data structure which have been discussed are shown in Fig 6.1. The solid arrows correspond to reference variables and the dotted arrows correspond to computed references.

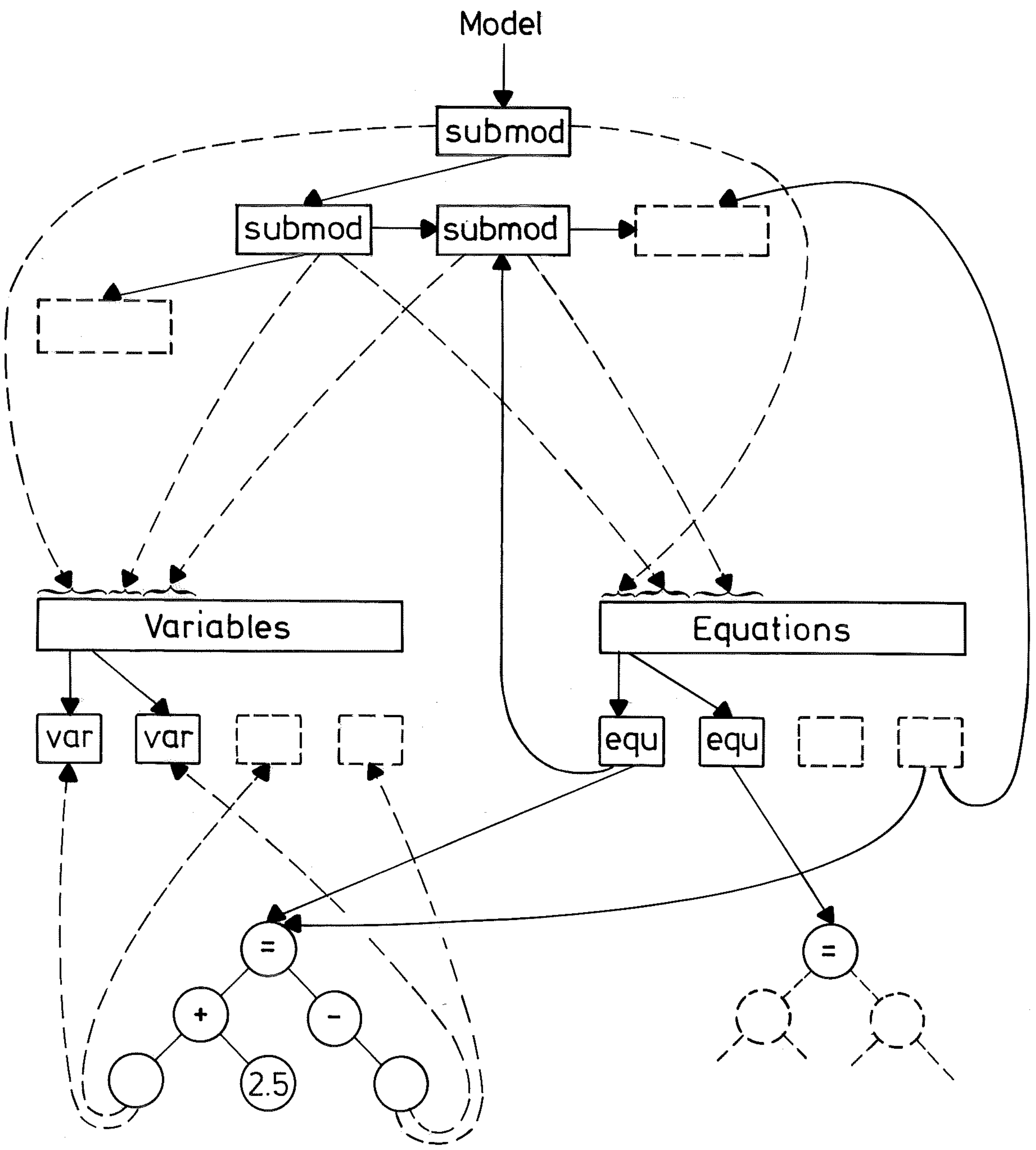


Fig 6.1. Data structure of the translator

Fig 6.1 does not contain all parts of the data structure. Cuts and paths are e.g. not included. References to these objects are stored in two global vectors called Cuts and Paths. The handling of the at operation on cuts is described below.

The equations corresponding to at-operations can not be generated until all the connection statements of a submodel have been processed. Cuts that are connencted are joined in a circular list with a list head. The circular lists are stored in another list using pointers in the list heads.

Example 6.1

Assume that the following at-operations are generated when processing a submodel.

```
A at B
C at D
A at D
```

Fig 6.2 shows how the data structure is changed during the processing of the operations.

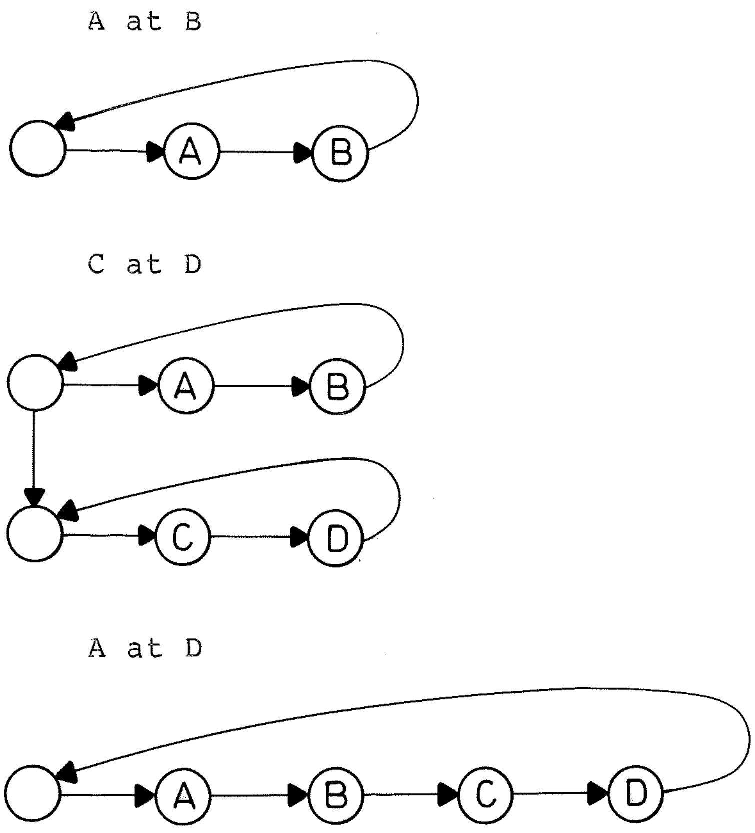


Fig. 6.2

The cuts A and B are linked together with a list head when the expression A at B is processed. Another list containing C and D is generated corresponding to C at D. The list heads are then linked together. When the expression A at D is handled the structure is changed to one circular list.

[]

<u>Program structure</u>

The program is divided into seven parts which handle submodels, cuts, equations, connection statements, auxiliary procedures, partitioning and interaction. The program structure is shown on the two following pages. Only the most important classes and procedures are included. They are listed in the same order as they appear in the complete program listing in appendix 3. If a procedure is an attribute of a class or imbedded in an other procedure the corresponding line is indented two spaces.

```
class SUBMODEL
  procedure Modelbody
  ref(submodel) procedure Newsubmodel

------------------------------

class CUT
  procedure Atop

cut class NODECUT
  procedure Duplicate

cut class HIERARCHICALCUT
  procedure Scanclause
  procedure Duplicate
  procedure Atop

cut class VARIABLECUT
  procedure Scanclause
  procedure Duplicate

cut class CONNECTNODE
  procedure Connect

------------------------------

class VARIABLE
  procedure Duplicate
  procedure Infix

class EQUATIONNODE
  procedure Infix
  procedure Traverse
  procedure Duplicate
  ref(expr) procedure Solve

ref(expr) procedure PRIMARY
                    FACTOR
                    TERM
                    SIMPEXPR
                    EXPRESSION
                    EQUATION

class EXPR
  ref(expr) procedure Add
                      Sub
                      Mult
                      Div
                      Power
                      Equal

ref(expr) procedure MINUS
ref(expr) procedure IFTHENELSE

expr class VARIABLENODE, FUNCTIONOP, NUMBERNODE,
           MINUSOP, IFTHENELSEOP
  procedure Infix
  procedure Traverse
  procedure Linj

expr class BINARYNODE
  procedure Traverse

binarynode class ADDOP, SUBOP, MULTOP,
                 DIVOP, POWEROP
  procedure Infix
  procedure Linj

binarynode class EQUALOP
  procedure Infix
  procedure Add

------------------------------

class PATH

class CONNOP
  ref(connop) procedure Atoper
                        Toop
                        From
                        Par
                        Loop

ref(connop) procedure REVERSED

ref(connop) procedure CONNECTIONOPERAND
                      CONNECTIONPRIMARY
                      CONNECTIONSECONDARY
                      CONNECTIONEXPRESSION
procedure CONNECTIONSTATEMENT

------------------------------

procedure SCAN
boolean procedure SEARCH
procedure ERROR
procedure COMPILE

------------------------------

procedure PARTITION
  boolean procedure Assign
  procedure Strongconnect

class STRONGCOMP
```

Comments to the program

This section contains comments to the most important classes and procedures of the program.

Procedure Modelbody of class Submodel handles the <model body>. It takes care of the submodel part and the declarations and calls the procedures Equation and Connectionstatement for the statement part.

Procedure Newsubmodel duplicates submodels, variables, cuts and equations when the submodel statement is used.

Procedure Atop of class Cut generates circular lists of connected variablecuts and nodes. Procedure Atop of Hierarchicalcut performs at-operations of all the subcuts.

Procedure Connect of class Connectnode generates equations from the circular lists of connected cuts and nodes.

The procedures Primary, Factor, Term, Simpexpr, Expression and Equation implement a recursive descent algorithm to build the syntax tree of an equation.

The nodes in the syntax tree are objects from subclasses of class Expr. The procedures Add, Sub, Mult, Div, Power and Equal of class Expr and the procedures Minus and Ifthenelse generate the nodes of the syntax tree. These procedures contain simplification rules such as $x+0=0$. These rules are needed to get nice expressions during formula manipulation.

Each subclass of Expr has three procedures associated: Infix, Traverse and Linj. They are all declared as virtual in class Expr. This means that a reference variable qualified to class Expr and pointing to a subclass can be used to reference e.g. Infix. The correct version of Infix, i.e. the one declared in the referenced subclass, is then automatically selected. The virtual concept simplifies the programming and makes the program better structured.

The procedures Infix are used to print equations in infix notation, i.e. normal mathematical notation. To generate infix notation, the syntax tree is traversed with a symmetric order traversal (Page and Wilson, 1973, p. 112) which for each node can be expressed as follows.

- traverse the left subtree
- visit the node itself
- traverse the right subtree

When dealing with equations, however, care must be taken about priorities and parentheses. The basic rule is to output a left parenthesis before traversal of the left subtree and a right parenthesis after traversal of the right subtree if the priority of the node (operator) is less then the priority of the superior node. This rule is, however, not enough. The expression a-(b-c) would then be outputted as a-b-c. This problem occurs with the operators **, /, unary -, - and if-then-else. It has been solved by using different priorities for the subtrees.

The procedures Traverse are used during partitioning to obtain the structure of an equation. They make a traversal of the syntax tree and a reference to each variable appearing in the equation is placed in the global vector Equvar.

The procedures Linj implement the transformation rules given in Table 5.1 (page 104) to split up an expression into a constant part and a factor with respect to some variable. This is done for the left part and the right part of an equation in procedure Solve of class Equationnode. The solution of the equation

$$f_{\emptyset}(y) + f_1(y)x = g_{\emptyset}(y) + g_1(y)x$$

with respect to x is

$$x = (g_{\emptyset}(y) - f_{\emptyset}(y))/(f_1(y) - g_1(y))$$

This formula is implemented in procedure Solve by the following statement.

```
  Solve:-x.Equal(gØ.Sub(fØ).Div(f1.Sub(g1)))
```

The procedures Atoper, Toop, From, Par and Loop of class Connop and procedure Reversed implements the rules 1 - 6 of Table 3.1 (page 5Ø)

The procedures Connectionoperand, Connectionprimary, Connectionsecondary and Connectionexpression implement a recursive descent algorithm for translation af a connection expression. The expression is not stored, it is immediately translated to at-operations.

Procedure Partition sorts and groups the equations into systems of equations that have to be solved simultaneously. A list of numbers corresponding to unknown variables is generated for each equation using the procedure Traverse. Procedure Assign (Wiberg, 1977) is called for each equation with this list to find an output set. Procedure Strongconnect (Tarjan, 1972) then finds the systems of equations (or the strong components of the corresponding graph) and stores them in the vector Equsystems.

Comments about the implementation

The objective for the present implementation has not been to produce a program to be widely used. For that reason some simplifications have been made and some features of the language have not been implemented. The error diagnostics are also poor and the translator stops scanning when the first error is found.

The following features are not implemented.
- default, internal and external declarations
- parameter list to submodel
- the operators branch and join
- . in hierarchical cut, path and connection primary
- procedure call
- solution of linear equations
- boolean expressions

- the use of nodes together with hierarchical cuts
- <cut spec> and <path spec> instead of <cut clause> and <path clause> in cut and path declarations
- cuts without <cut clause> or <cut spec>

Derivatives are handled in the following way. If der2(x) is found it is replaced by a new variable der2x. A new variable derx is also generated if it does not already exist. The variables x and derx are assumed as state variables and are thus indicated as known.

7. EXAMPLES

This chapter contains four examples illustrating the use of the model language and the translator.

7.1 Electrical network

Figure 7.1 shows a logical inverter. Assume that it is desired to obtain the response of the inverter when the input is a pulse. A modified Ebers-Moll model has been used tor the transistor. It is shown in Fig 7.2. These models have been adopted from the ASTAP-manual (see ASTAP).

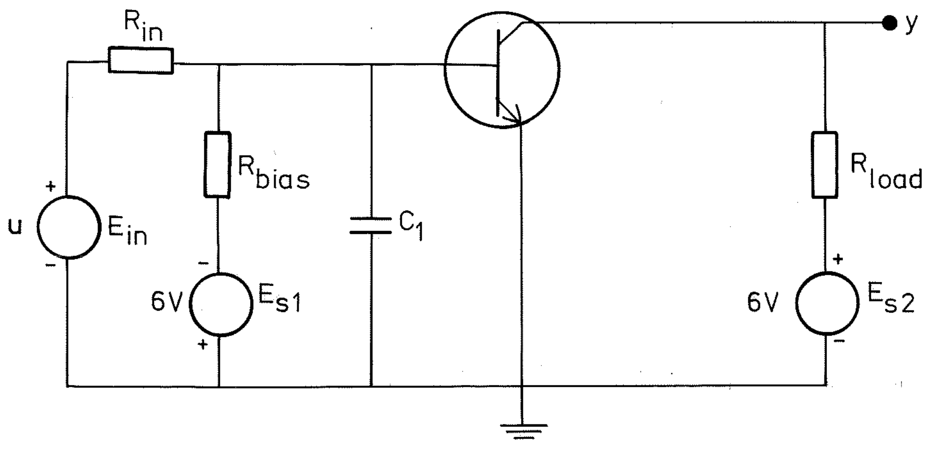


Fig 7.1 Logical inverter

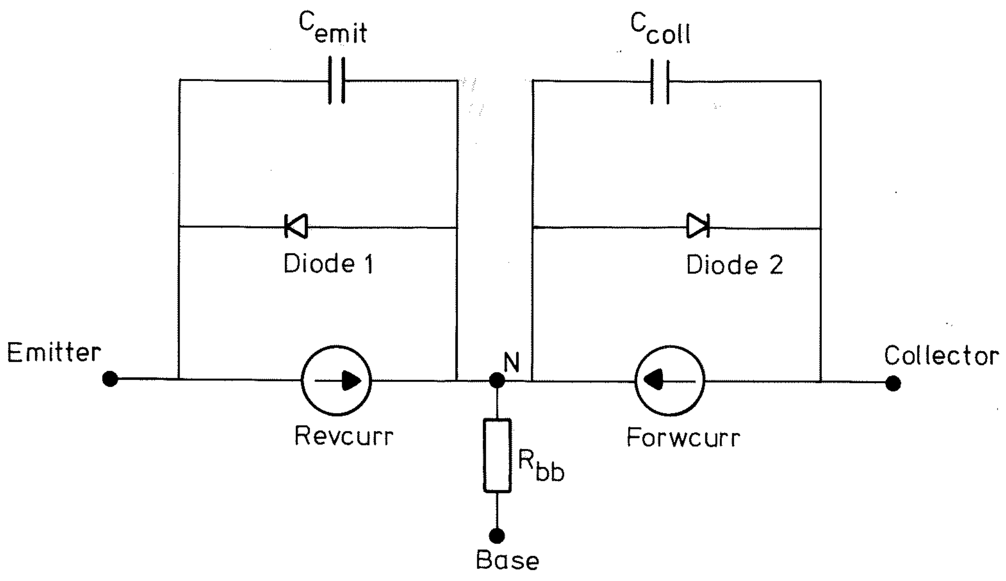


Fig 7.2. Transistor model

The capacitor C1 corresponds to the stray capacitances between base and emitter. The stray capacitances between collector and emitter have been neglected. The reason is that they would be connected in parallel with the series connection of the capacitors Cemit and Ccoll. It is then not possible to obtain the state equations of the network directly. This problem is, however, already discussed in examples 5.1 and 5.2.

The capacitances of Cemit and Ccoll depend on the diode currents and are thus time varying. The model type varcapacitor reflects the fact that the basic equation for a capacitor is

$$\frac{d(CV)}{dt} = I$$

and not

$$C\frac{dV}{dt} = I$$

The model of the inverter is given below. It uses the library of electrical components given in section 3.7 (page 58).

After the model follows the dialogue with the translator program. After compiling the library and the model, the generated equations are printed. The equations are then partitioned and the solved equations printed. Two systems of equations are obtained. They contain six equations each and are due to the dependence between capacitances and diode currents.

```
 model type varcapacitor
  cut A (Va / I) B (Vb / -I)
  main cut C [A B]
  main path P <A - B>
  local V, Q
  terminal C
  V = Va-Vb
  Q = C*V
  der(Q) = I
end

model type diode
  cut A (Va / I) B (Vb / -I)
  main cut C [A B]
  main path diode <A - B>
  parameter IØ, K
  local V
  V = Va-Vb
  I = IØ*(exp(K*V) - 1)
end

model type Transistor
  submodel (diode) Diodel(3.5E-9,28), ->
                   Diode2(7.3E-9,32)
  submodel (current) Revcurr, Forwcurr
  submodel (resistor) Rbb(3Ø)
  submodel (varcapacitor) Cemit, Ccoll

  parameter Al = Ø.47, A2 = Ø.978

  cut Base(Vb / Ib) Emitter(Ve / Ie)
  cut Collector(Vc / Ic)
  main cut Trans [Base, Emitter, Collector]
  path Baseemitter <Base - Emitter>
  path Collectoremitter <Collector - Emitter>
  node N

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

  Cemit.C = 3.ØE-12 + 6.7E-9*Diodel.I
  Ccoll.C = 2.ØE-12 + 18Ø.ØE-9*Diode2.I

  Revcurr.I = Al*Diode2.I
  Forwcurr.I = A2*Diodel.I

end
```

```
model inv   { Logical inverter }

   submodel (Transistor) Tr
   submodel (voltage) Ein, Es1, Es2
   submodel (resistor) Rin(5.6E3) ->
                       Rbias(10E3) ->
                       Rload(1E3)
   submodel (capacitor) C1(3.6E-12)
   submodel Common

   input U
   output Y

   connect Common - Ein - Rin - ( (Rbias - Es1) // C1 // ->
     Tr..Baseemitter ) - Common

   connect Common - Es2 - Rload - Tr..Collectoremitter

   Ein.V = U
   Y = Rload.Vb

   Es1.V = 6
   Es2.V = 6

end
```

```
{ Interaction with the translator }

>@add ellib
>@add inv
>print equations

 Tr::Rbb            V = Va - Vb
                    R*I = V
 C1                 V = Va - Vb
                    C*derV = I
 Ein                V = Vb - Va
 Common             V = Ø
 Tr::Cemit          V = Va - Vb
                    Q = C*V
                    derQ = I
 Tr::Diode1         V = Va - Vb
                    I = IØ*(exp(K*V) - 1)
 Tr::Diode2         V = Va - Vb
                    I = IØ*(exp(K*V) - 1)
 Tr::Ccoll          V = Va - Vb
                    Q = C*V
                    derQ = I
 Tr                 Cemit.C = 3.ØE-12 + 6.7E-9*Diode1.I
                    Ccoll.C = 2.ØE-12 + 18Ø.ØE-9*Diode2.I
                    Revcurr.I = A1*Diode2.I
                    Forwcurr.I = A2*Diode1.I
                    Diode1.Vb = Ve
                    Cemit.Va = Diode1.Vb
                    Cemit.I + Revcurr.I = Ie + Diode1.I
                    Diode2.Vb = Vc
                    Ccoll.Va = Diode2.Vb
                    Ccoll.I + Forwcurr.I = Ic + Diode2.I
                    Rbb.Va = Vb
                    Rbb.I = Ib
                    Diode2.Va = Rbb.Vb
                    Ccoll.Vb = Diode2.Va
                    Diode1.Va = Ccoll.Vb
                    Cemit.Vb = Diode1.Va
                    Diode2.I + Diode1.I = Rbb.I + Ccoll.I +
                      Forwcurr.I + Cemit.I + Revcurr.I
 Es1                V = Vb - Va
 Es2                V = Vb - Va
 Rin                V = Va - Vb
                    R*I = V
 Rbias              V = Va - Vb
                    R*I = V
 Rload              V = Va - Vb
                    R*I = V
 inv                Ein.V = U
                    Y = Rload.Vb
                    Es1.V = 6
                    Es2.V = 6
                    Rin.Va = Ein.Vb
                    Rin.I = Ein.I
                    Es1.Va = Rbias.Vb
                    Es1.I = Rbias.I
                    C1.Vb = Es1.Vb
                    Tr.Ve = C1.Vb
```

```
                         Common.V = Tr.Ve
                         Ein.Va = Common.V
                         Es2.Va = Ein.Va
                         Rbias.Va = Rin.Vb
                         Cl.Va = Rbias.Va
                         Tr.Vb = Cl.Va
                         Rbias.I + Cl.I + Tr.Ib = Rin.I
                         Rload.Va = Es2.Vb
                         Rload.I = Es2.I
                         Tr.Vc = Rload.Vb
                         Tr.Ic = Rload.I

>partition
>print solved

 Common                  V = Ø
 inv                     Tr.Ve = Common.V
                         Cl.Vb = Tr.Ve
 Cl                      Va = V + Vb
 inv                     Tr.Vb = Cl.Va
 Tr                      Rbb.Va = Vb
                         Diodel.Vb = Ve
                         Cemit.Va = Diodel.Vb

-Tr::Diodel              V = [Va] - Vb
-                        I = IØ*(exp(K*[V]) - 1)
-Tr                      Cemit.C = 3.ØE-12 + 6.7E-9*[Diodel.I]
-Tr::Cemit               Q = [C]*V
-                        [V] = Va - Vb
-Tr                      [Cemit.Vb] = Diodel.Va

                         Ccoll.Vb = Diodel.Va
                         Diode2.Va = Ccoll.Vb
                         Rbb.Vb = Diode2.Va
 Tr::Rbb                 V = Va - Vb
                         I = V/R
 inv                     Rbias.Va = Cl.Va
                         Esl.V = 6
                         Esl.Vb = Cl.Vb
 Esl                     Va = Vb - V
 inv                     Rbias.Vb = Esl.Va
 Rbias                   V = Va - Vb
                         I = V/R
 Tr                      Ib = Rbb.I
 inv                     Ein.V = U
                         Ein.Va = Common.V
 Ein                     Vb = V + Va
 inv                     Rin.Va = Ein.Vb
                         Rin.Vb = Rbias.Va
 Rin                     V = Va - Vb
                         I = V/R
 inv                     Cl.I = Rin.I - (Rbias.I + Tr.Ib)
 Cl                      derV = I/C
 inv                     Ein.I = Rin.I

-Tr::Diode2              [I] = IØ*(exp(K*V) - 1)
-                        [V] = Va - Vb
-Tr                      Ccoll.Va = [Diode2.Vb]
```

```
-Tr::Ccoll      V = [Va] - Vb
-               Q = C*[V]
-Tr             [Ccoll.C] = 2.ØE-12 + 18Ø.ØE-9*Diode2.I

                Revcurr.I = A1*Diode2.I
                Forwcurr.I = A2*Diode1.I
 inv            Es2.V = 6
                Es2.Va = Ein.Va
 Es2            Vb = V + Va
 inv            Rload.Va = Es2.Vb
 Tr             Vc = Diode2.Vb
 inv            Rload.Vb = Tr.Vc
 Rload          V = Va - Vb
                I = V/R
 inv            Tr.Ic = Rload.I
 Tr             Ccoll.I = Ic + Diode2.I - Forwcurr.I
                Cemit.I = Diode2.I + Diode1.I - (Rbb.I +
                  Ccoll.I + Forwcurr.I + Revcurr.I)
 Tr::Cemit      derQ = I
 Tr::Ccoll      derQ = I
 Tr             Ie = Cemit.I + Revcurr.I - Diode1.I
 inv            Es1.I = Rbias.I
                Es2.I = Rload.I
                Y = Rload.Vb
```

Simulation

The interactive simulation program SIMNON (Elmqvist, 1975, 1977a) has been used to obtain the time responses.

A Simnon model can be composed of a set of submodels and a connecting system. Each submodel is described by either ordinary differential equations or difference equations in the form of assignment statements. The interaction with the program is normally done via a graphical terminal. The program is controlled by commanads and the resulting time responses are plotted on the terminal.

It is not possible to handle systems of equations directly in Simnon. For that reason the capacitances of Cemit and Ccoll were assumed constant independent of the diode currents.

The network was described by a single continuous system containing variable declarations, the solved equations and parameter statements. The solved equations as outputted by the translator program were completed with respect to variable references. The reference mechanisms :: and . were deleted to obtain legal identifiers.

The dialogue with the Simnon program is shown below. Comments are placed after " in the lines.

```
>SYST SIMINV           " Compile model
>PAR RBBR:2Ø           " Change base resistance
>INIT CCOLLV:6         " Change initial value of state
>STORE INVU INVY ClV CEMITV CCOLLV
>                      " Specify variables to be stored
>SIMU Ø 15ØE-9         " Simulate 15Ø ns
>SPLIT 2 1             " Divide plotting area
>ASHOW INVU INVY       " Scale, draw axes and plot
>TEXT 'Fig 7.3: Logical inverter. Inv.u = 1, Inv.y = 2'
>ASHOW ClV CEMITV CCOLLV
>TEXT 'Fig 7.4: Cl.V = 1, Cemit.V = 2, Ccoll.V = 3'
```

Fig 7.3: Logical inverter. Inv.u = 1, Inv.y = 2

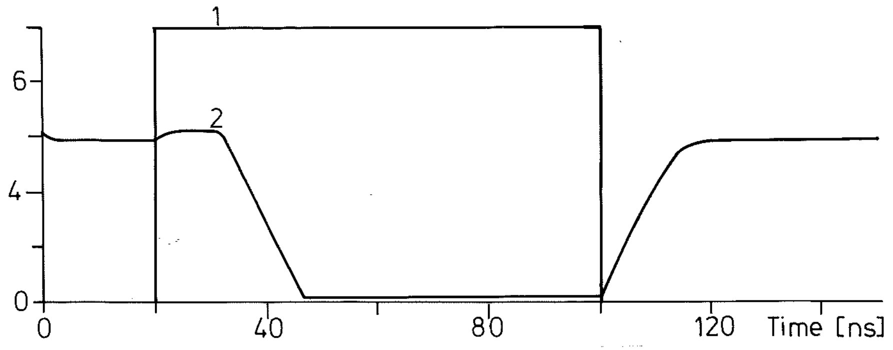
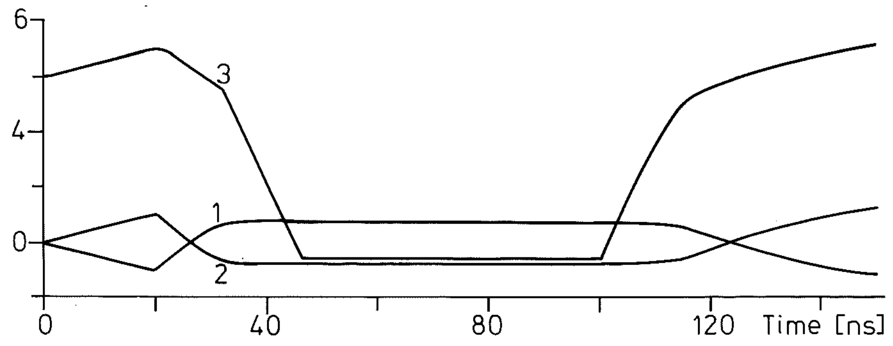


Fig 7.4: Cl.v = 1, Cemit.V = 2, Ccoll.V = 3



7.2 A mechanical model

The formulation of a mechanical model by Newtons equations is naturally done in the model language. The cuts of a mechanical model will contain such variables as coordinates, reaction forces and reaction torques.

It is often not possible to solve for the derivatives in the Newton equations obtained. The equations can then be solved in two ways: by using an integration method of the type discussed in section 5.1 or by using the differentition algorithm of section 5.3 to obtain state equations which are then integrated by an ordinary algorithm.

A mechanical model for a human body is used to illustrate the ideas. A schematic picture of the body is given in Fig 7.5.

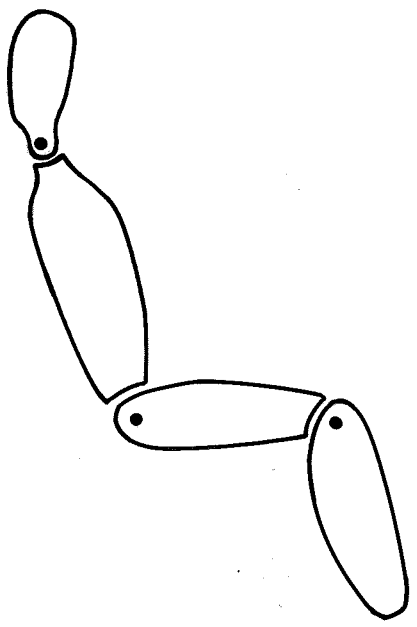


Fig 7.5

The model consists of four rigid bodies: head, trunk, thighs and calves. They are joined at the neck, hips and knees. This type of models of the human body are used to compute stresses and motions of a human during a car crash (see Hornstein, 1976). The model is assumed to be symmetric. No external forces except gravitational force are assumed in the given model. The model is listed on page 133.

If the equations are partitioned the translator will give the following message.

```
Singular problem

Unassigned variables:
head.F2x
head.F2y
neck.derv
trunk.F2x
trunk.F2y
thigh.F2x
thigh.F2y
hip.derv
knee.derv

Redundant equations:
trunk.x1 = neck.x
trunk.y1 = neck.y
trunk.v = neck.v2
thigh.x1 = hip.x
thigh.y1 = hip.y
thigh.v = hip.v2
calf.x1 = knee.x
calf.y1 = knee.y
calf.v = knee.v2
```

The default assumptions made by the program is that all variables appearing differentiated are considered as state variables. All state variables are then assumed known. This explains the redundant equations. On the other hand there are not enough equations to determine the derivatives. Note that there is an arbitrariness about unassigned variables. The selection made depends on the algorithm for finding the output set.

The differentiation algorithm of section 5.3 has not been implemented. It is thus not possible to show the state space representation of the model.

The partitioning algorithm can also be used together with an implicit integration algorithm. It will then make a partitioning of the model into the structure shown in Fig 4.1 (page 7Ø). The partitioning of the equations is done with respect to dynamic loops instead of algebraic loops.

In order to make the program do this partitioning, the der operator is replaced by a arbitrary function, called diff. The dialogue with the program is shown on page 134. The elimination feature has been used. A system of 38 equations was found. It means that the partitioning algorithm has found 1Ø equations to be excluded from the iterations at each time step.

```
{ A mechanical model for the human body }

model type limb

cut A (x1 y1 v / F1x F1y M1)
cut B (x2 y2 v / -F2x -F2y -M2)
main path P <A - B>
local x y N1 N2
parameter J L1 L2 m
constant g=9.81

N1 = (F1x*sin(v) - F1y*cos(v))*L1
N2 = (F2x*sin(v) - F2y*cos(v))*L2
J*der2(v) = N1 + M1 + N2 - M2

m*der2(x) = F1x - F2x
m*der2(y) = F1y - F2y - m*g

x1 = x - L1*cos(v)
y1 = y - L1*sin(v)
x2 = x + L2*cos(v)
y2 = y + L2*sin(v)

end


model type joint

cut A (x y v1 / Fx Fy M)
cut B (x y v2 / -Fx -Fy -M)
main path P <A - B>
parameter A B vØ

v = v1 - v2
M = -A*(v - vØ) - B*sign(der(v))

end


model human

submodel (limb) head trunk thigh calf
submodel (joint) neck hip knee

connect head to neck to trunk to hip to  ->
   thigh to knee to calf

head.F1x = Ø
head.F1y = Ø
head.M1 = Ø
calf.F2x = Ø
calf.F2y = Ø
calf.M2 = Ø

end
```

```
>@add human
>do eliminate
>partition
>print solved

 human    head.F1x = Ø
          head.F1y = Ø
          head.M1 = Ø
          calf.F2x = Ø
          calf.F2y = Ø
          calf.M2 = Ø

-head     m*diff2([x]) = F1x - F2x
-trunk    m*diff2(x) = [head.F2x] - F2x
-         head.x2 = [x] - L1*cos(neck.v2)
-head     [x2] = x + L2*cos(v)
-         J*diff2([v]) = N1 + M1 + N2 - M2
-         [N1] = (F1x*sin(v) - F1y*cos(v))*L1
-         [N2] = (F2x*sin(v) - F2y*cos(v))*L2
-trunk    m*diff2(y) = [head.F2y] - F2y - m*g
-         head.y2 = [y] - L1*sin(neck.v2)
-head     [y2] = y + L2*sin(v)
-         m*diff2([y]) = F1y - F2y - m*g
-trunk    J*diff2([neck.v2]) = N1 + head.M2 + N2 - M2
-         [N1] = (head.F2x*sin(neck.v2) - head.F2y*
            cos(neck.v2))*L1
-neck     [head.M2] = -A*(v - vØ) -B*sign(diff(v))
-         [v] = head.v - v2
-trunk    [N2] = (F2x*sin(neck.v2) - F2y*cos(neck.v2))*L2
-thigh    m*diff2(x) = [trunk.F2x] - F2x
-         trunk.x2 = [x] - L1*cos(hip.v2)
-trunk    [x2] = x + L2*cos(neck.v2)
-thigh    J*diff2([hip.v2]) = N1 + trunk.M2 + N2 - M2
-         [N1] = (trunk.F2x*sin(hip.v2) - trunk.F2y*
            cos(hip.v2))*L1
-         m*diff2(y) = [trunk.F2y] - F2y - m*g
-         trunk.y2 = [y] - L1*sin(hip.v2)
-trunk    [y2] = y + L2*sin(neck.v2)
-calf     m*diff2(y) = [thigh.F2y] - F2y - m*g
-         thigh.y2 = [y] - L1*sin(knee.v2)
-thigh    [y2] = y + L2*sin(hip.v2)
-calf     J*diff2([knee.v2]) = N1 + thigh.M2 + N2 - M2
-         [N1] = (thigh.F2x*sin(knee.v2) - thigh.F2y*
            cos(knee.v2))*L1
-         m*diff2(x) = [thigh.F2x] - F2x
-         thigh.x2 = [x] - L1*cos(knee.v2)
-thigh    [x2] = x + L2*cos(hip.v2)
-knee     [thigh.M2] = -A*(v - vØ) - B*sign(diff(v))
-         [v] = hip.v2 - v2
-calf     [N2] = (F2x*sin(knee.v2) - F2y*cos(knee.v2))*L2
-hip      [trunk.M2] = -A*(v - vØ) - B*sign(diff(v))
-         [v] = neck.v2 - v2
-thigh    [N2] = (F2x*sin(hip.v2) - F2y*cos(hip.v2))*L2

 head     x1 = x - L1*cos(v)
          y1 = y - L1*sin(v)
 calf     x2 = x + L2*cos(knee.v2)
          y2 = y + L2*sin(knee.v2)
```

7.3 Electrical energy transmission

Consider an electrical power system consisting of two synchronous generators, three transmission lines and loads as shown in Fig 7.6 (see Elgerd, 1971).

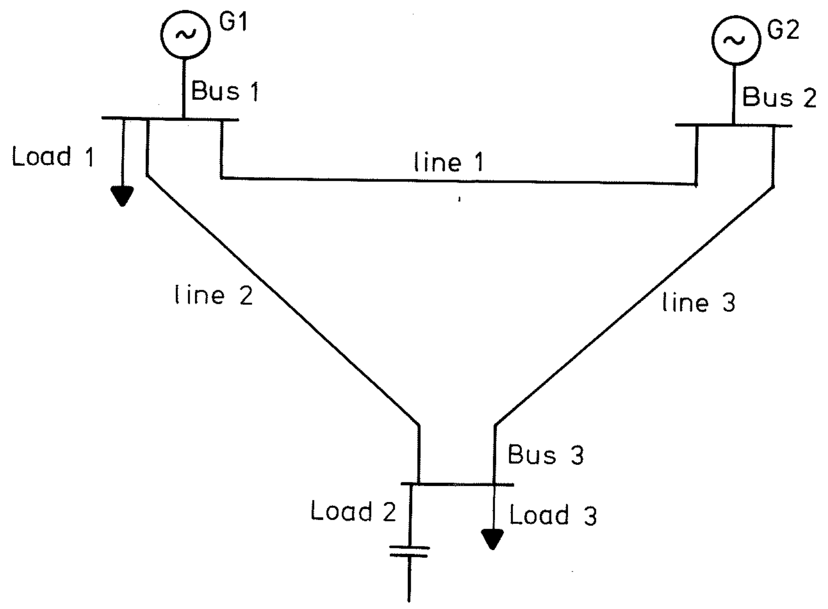


Fig 7.6

The voltages and currents are assumed to be sinousoidal with slowly varying amplitudes and phases. This means that the transmission lines and the loads can be considered as static systems and that the jw-method can be used to calculate the load flow. The system is an example which is naturally described using complex variables. They are split up into their real and imaginary parts in this model. The model is shown on pages 137-138.

The interaction with the translator is shown on pages 139-141. After compiling the model, the equations are partitioned, solved and printed. The partitioning is done for the simulation problem. The model has four states: delt and delt' in the two generators. A system of equations is found. It consists of 22 nontrivial equations and correspond to the load flow calculation.

It is assumed that the initial values of the state variables delt in the two generators are unknown. The parameters of the loads are also unknown. On the other hand, the amplitudes of the bus voltages and and the power flows are assumed to be known in steady state. Bus 1 is selected as reference bus.

The commands "known" and "unknown" are used to specify that the equations should be sorted for computing the initial values. The partitioning is then performed and the solved equations are printed.

For this problem there are six systems of equations. The largest contains 19 nontrivial equations. The others contain only two equations. Three of these small systems of equations are linear. They are easily solved by the computer using formula manipulation.

```
model type Generator

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

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

{ The electrical model for the generator is a voltage source
  behind a reactance:
        E = j*Xd*Ig + V
  The magnitude of E depends on the field current,
  which is assumed constant. 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: }
  der2(delt)*H/(PI*fØ) + der(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 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

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

model power

{ Transmission of electrical energy }

  submodel (Generator) ->
    G1(Xd=Ø.Ø54, H=3Ø, fØ=5Ø, D=Ø) ->
    G2(Xd=Ø.Ø54, H=3ØØ, fØ=5Ø, D=Ø)
  submodel (Line) Line1(Ø.Ø5), Line2(Ø.Ø5), Line3(Ø.Ø5)
  submodel (Load) Load1, Load2(Zx=Ø), Load3

  node Bus1, Bus2, Bus3

  connect G1 at Bus1
  connect G2 at Bus2

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

  connect Load1 at Bus1
  connect Load2 at Bus3
  connect Load3 at Bus3

end
```

```
>{ Interaction with the translator }
>
>@add elpower
>
>{ Simulation problem }
>partition
>print solved

 G2             Ex = E*cos(delt)
                Ey = E*sin(delt)
 G1             Ex = E*cos(delt)
                Ey = E*sin(delt)

-power          [G1.Vx] = Line1.Vx1
-               [Line1.Vx1] = Line2.Vx1
-               [Line2.Vx1] = Load1.Vx
-Load1          [Vx] = Zx*Ix - Zy*Iy
-power          [Load1.Ix] + Line2.Ix + Line1.Ix = G1.Ix
-Line2          Vy1 = [Ix]*XL + Vy2
-power          [Line2.Vy1] = Load1.Vy
-Load1          [Vy] = Zx*Iy + Zy*Ix
-power          [Load1.Iy] + Line2.Iy + Line1.Iy = G1.Iy
-Line2          Vx1 = Vx2 - [Iy]*XL
-power          [Line2.Vx2] = Line3.Vx2
-               [Line3.Vx2] = Load2.Vx
-               [Load2.Vx] = Load3.Vx
-Load3          [Vx] = Zx*Ix - Zy*Iy
-power          [Load3.Ix] + Load2.Ix = Line3.Ix + Line2.Ix
-Load2          Vx = Zx*[Ix] - Zy*Iy
-               Vy = Zx*[Iy] + Zy*Ix
-power          [Load2.Vy] = Load3.Vy
-Load3          [Vy] = Zx*Iy + Zy*Ix
-power          [Load3.Iy] + Load2.Iy = Line3.Iy + Line2.Iy
-               [Line3.Iy] = Line1.Iy + G2.Iy
-Line1          Vx1 = Vx2 - [Iy]*XL
-power          [Line1.Vx2] = Line3.Vx1
-Line3          [Vx1] = Vx2 - Iy*XL
-G2             Ex = Vx - Xd*[Iy]
-power          [G2.Vx] = Line1.Vx2
-               [Line3.Ix] = Line1.Ix + G2.Ix
-Line1          Vy1 = [Ix]*XL + Vy2
-power          [Line1.Vy1] = Line2.Vy1
-               [Line1.Vy2] = Line3.Vy1
-Line3          [Vy1] = Ix*XL + Vy2
-power          [Line3.Vy2] = Load2.Vy
-G2             Ey = Xd*[Ix] + Vy
-power          [G2.Vy] = Line1.Vy2
-G1             Ex = Vx - Xd*[Iy]
-power          [Line2.Vy2] = Line3.Vy2
-G1             Ey = Xd*[Ix] + Vy
-power          [G1.Vy] = Line1.Vy1

 G1             Pg = Ex*Ix + Ey*Iy
                V = sqrt(Vx**2 + Vy**2)
                der2delt = (Pt - Pg - derdelt*D)*PI*f0/H
 Load1          P = Vx*Ix + Vy*Iy
                Q = Vy*Ix - Vx*Iy
                V = sqrt(Vx**2 + Vy**2)
 G2             Pg = Ex*Ix + Ey*Iy
                V = sqrt(Vx**2 + Vy**2)
                der2delt = (Pt - Pg - derdelt*D)*PI*f0/H
 Load2          P = Vx*Ix + Vy*Iy
                Q = Vy*Ix - Vx*Iy
                V = sqrt(Vx**2 + Vy**2)
 Load3          P = Vx*Ix + Vy*Iy
                Q = Vy*Ix - Vx*Iy
                V = sqrt(Vx**2 + Vy**2)


>{ Initial computation }
>
>known G1.Vx G1.Vy G1.derdelt
>known G2.V G2.Pg G2.derdelt
>known Load1.P Load1.Q
>known Load3.P Load3.Q Load3.V
>
>unknown G1.E G1.Pt G1.delt
>unknown G2.E G2.Pt G2.delt
>unknown Load1.Zx Load1.Zy
>unknown Load2.Zy
>unknown Load3.Zx Load3.Zy
>
>partition
>print solved

 power          Line1.Vy1 = G1.Vy
                Line2.Vy1 = Line1.Vy1
                Load1.Vy = Line2.Vy1
                Line1.Vx1 = G1.Vx
                Line2.Vx1 = Line1.Vx1
                Load1.Vx = Line2.Vx1
-Load1          Q = Vy*Ix - Vx*[Iy]
-               P = Vx*[Ix] + Vy*Iy
-Line2          Vx1 = Vx2 - [Iy]*XL
-power          [Line2.Vx2] = Line3.Vx2
-               [Line3.Vx2] = Load2.Vx
-               [Load2.Vx] = Load3.Vx
-Load3          V = sqrt([Vx]**2 + Vy**2)
-               Q = [Vy]*Ix - Vx*Iy
-               P = Vx*[Ix] + Vy*Iy
-power          [Load3.Iy] + Load2.Iy = Line3.Iy + Line2.Iy
-Load2          Vy = Zx*[Iy] + Zy*Ix
-power          [Load2.Vy] = Load3.Vy
-Load2          Vx = Zx*Ix - [Zy]*Iy
-power          Load3.Ix + [Load2.Ix] = Line3.Ix + Line2.Ix
-Line3          Vy1 = [Ix]*XL + Vy2
-power          Line1.Vy2 = [Line3.Vy1]
-               G2.Vy = [Line1.Vy2]
-G2             V = sqrt(Vx**2 + [Vy]**2)
-power          [G2.Vx] = Line1.Vx2
-               [Line1.Vx2] = Line3.Vx1
-Line3          [Vx1] = Vx2 - Iy*XL
-power          [Line3.Iy] = Line1.Iy + G2.Iy
-Line1          Vx1 = Vx2 - [Iy]*XL
-G2             Pg = Ex*Ix + Ey*[Iy]
```

```
-                  [Ex] = Vx - Xd*Iy
-power             Line3.Ix = Line1.Ix + [G2.Ix]
-Line1             Vy1 = [Ix]*XL + Vy2
-G2                [Ey] = Xd*Ix + Vy
-power             [Line3.Vy2] = Load2.Vy
-Line2             Vy1 = [Ix]*XL + Vy2
-power             [Line2.Vy2] = Line3.Vy2

                   G1.Iy = Load1.Iy + Line2.Iy + Line1.Iy
 G1                Ex = Vx - Xd*Iy
 power             G1.Ix = Load1.Ix + Line2.Ix + Line1.Ix
 G1                Ey = Xd*Ix + Vy

-                  Ex = [E]*cos(delt)
-                  Ey = E*sin([delt])

                   Pg = Ex*Ix + Ey*Iy
                   Pt = der2delt*H/(PI*fØ) + derdelt*D + Pg
                   V = sqrt(Vx**2 + Vy**2)
 Load1             V = sqrt(Vx**2 + Vy**2)

-                  Vx = [Zx]*Ix - Zy*Iy
-                  Vy = Zx*Iy + [Zy]*Ix


-G2                Ex = [E]*cos(delt)
-                  Ey = E*sin([delt])

                   Pt = der2delt*H/(PI*fØ) + derdelt*D + Pg
 Load2             P = Vx*Ix + Vy*Iy
                   Q = Vy*Ix - Vx*Iy
                   V = sqrt(Vx**2 + Vy**2)

-Load3             Vx = [Zx]*Ix - Zy*Iy
-                  Vy = Zx*Iy + [Zy]*Ix
```

7.4 A drum boiler - turbine model

A model of a thermal power station has been developed by Lindahl (1976). The model was simulated in SIMNON (Elmqvist, 1975, 1977a). This program requires that the model is specified in state space form. A substantial amount of analysis and trivial rewriting of the model was required to obtain the Simnon description.

The structure of the system is shown in Fig 7.7 and the model is given on pages 144-152. It should be noted how easy it is to describe the structure of the model using the model language. This is done on page 152. It is also very easy to understand each submodel because of the well defined cuts. There are functions HSP, RHP, etc. which defines the state of steam and water by interpolation in the Moliere diagram.

The sorted, grouped and solved equations are shown on pages 153-161. The elimination feature has been used in order to shorten the list of solved equations. There are 252 nontrivial equations. The partitioning algorithm found 11 systems of equations. Many of them are nonlinear but could easily be solved by hand. There is one system of 17 equations involving steam and feed water flows. In the original model (Lindahl, 1976) a simplification was made in the attemperator models. Two systems of equations were then obtained instead, one for steam flow and one for feed water flow. It was possible to solve these nonlinear systems of equations by hand because the quadratic terms cancelled when the equations were added.

The example shows that the partitioning algorithm is very useful when transforming the model. It also helps to indicate when simplifications are required to avoid too large systems of equations.

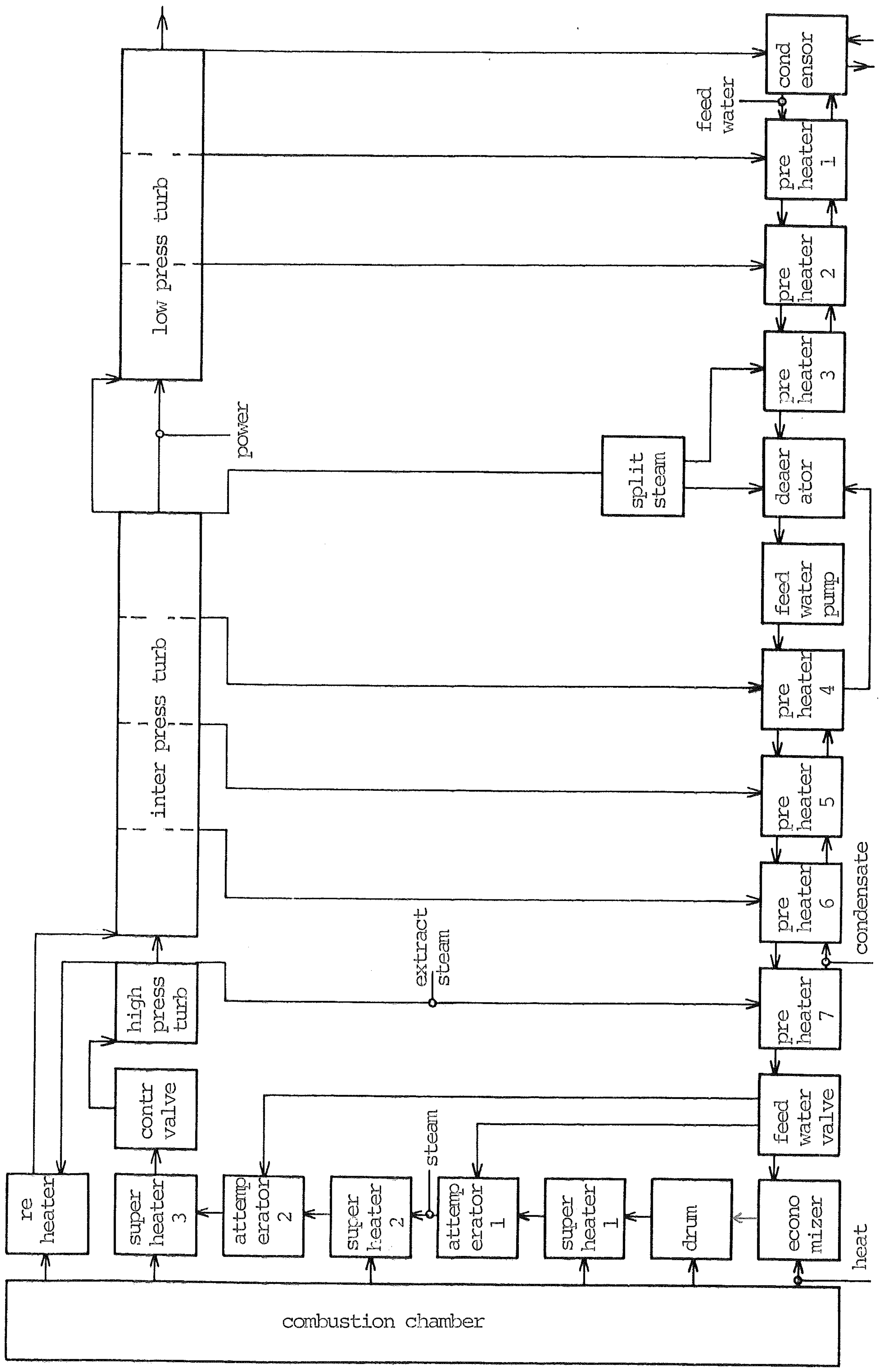

combustion chamber
re heater
super heater 3
attemp erator 2
super heater 2
steam
attemp erator 1
super heater 1
drum
econo mizer
heat
contr valve
high press turb
feed water valve
pre heater 7
extract steam
inter press turb
pre heater 6
condensate
pre heater 5
pre heater 4
feed water pump
deaer ator
split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

Fig. 7.7

```
{ A nonlinear 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 type drum { with downcomers }

cut inwater (Wwr, Hwr, .)
cut outwater (Wdc, Hdc, Pdc)
path water <inwater - outwater>
cut insteam (Wsr, Hsr, Pd, dPd)
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, Vw0, Vs0, Aw
constant g=9.81

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 - Wdc
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

dPd = der(Pd)
Hs = IHSP(Pd)
HsP = HSPP(Pd)
Rw = RHP(Hd,Pd)
Rs = IRSP(Pd)
RsP = RSPP(Pd)
Rwr = RWP(Pd)

end

  {-------------------------------------------------------}
```

```
model type risers

cut inwater (Wdc,Hdc,Pdc)
cut outwater (Wwr, Hwr, .)
path water <inwater - outwater>
cut steam (Wsr, Hsr, Pd, dPd)
cut heat (Q)
local Vb, x, xs, TAU, Wsprod, Wmix, Rmix, Tm, TmP, TmixP
local Rs, Rwr, RsP, RwrP, Tdc, HwrP, Hs, HsP, Tmix
parameter Vr, Cm, m, K, f, L, D, A
constant g=9.81

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

  { 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*dPd = Wsprod -Wsr
Wsr = x*Wmix

  { energy balance }
  { der(Cm*m*Tm + (Vr - Vb)*Rwr*Hwr + Vb*Rs*Hs) = }
Cm*m*TmP*dPd - der(Vb)*Rwr*Hwr +  ->
  (Vr - Vb)*(RwrP*Hwr + Rwr*HwrP)*dPd +  ->
  der(Vb)*Rs*Hs + Vb*(RsP*Hs + Rs*HsP)*dPd =  ->
  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)

HsP =IHSP(Pd)
Hwr = HWP(Pd)
HwrP = HWPP(Pd)
Rs = IRSP(Pd)
RsP = RSPP(Pd)
Rwr = RWP(Pd)
RwrP = RWPP(Pd)
Tmix = TLP(Pd)
TmixP = TLPP(Pd)
Tdc = THP(Hdc, Pdc)

end

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

model type drumsyst

submodel drum
submodel risers
```

```
connect (water) drum loop risers
connect (steam) risers to drum

end

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

model type superheater

cut instream (W, H1, P1)
cut outsteam (W, H2, P2)
path steam < instream - 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 instream (W1, H1, P)
cut outsteam (W2, H2, P)
path steam <instream - outsteam>
cut feedwater (Ww, Hw, Pw)
input Sw
local aw
parameter fw

Pw - P = 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 insteam (W, H, P1)
cut outsteam (W, H, P2)
path steam < insteam - outsteam>
input Sv
local av
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, R2T

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) = }
R2T = 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) outpower (N2)
```

```
path power < inpower - outpower >
input S
local H, T2, ap
parameter f, fp, Eh
{default N1=Ø, Wp=Ø, 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 IPturb

submodel (turbsection) IP1, IP2, IP3, IP4

path steam < IP1:insteam - IP4:outsteam >
cut extractsteam [ IP1:extractsteam IP2:extractsteam  ->
  IP3:extractsteam IP4:extractsteam ]
path power < IP1:inpower - IP4:outpower >

connect (steam) IP1 to IP2 to IP3 to IP4
connect (power) IP1 to IP2 to IP3 to IP4

end

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

model type LPturb

submodel (turbsection) LP1, LP2, LP3

path steam < LP1:insteam - LP3:outsteam >
cut extractsteam [ LP1:extractsteam LP2:extractsteam ]
path power < LP1:inpower - LP3:outpower >

connect (steam) LP1 to LP2 to LP3
connect (power) LP1 to LP2 to LP3

end

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

model type condensor

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

Pc = Pw + Pdiff
Ps = Pc

  { 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 - Wl*(H2-Hl)

H2 = Hw - Hdiff
TmP = TwP

Hw = IHWP(Pw)
HwP = HWPP(Pw)
Rw = RWP(Pw)
R2 = RHP(H2, Pl)
Tw = TLP(Pw)
TwP = TLPP(Pw)
Tl = THP(Hl, Pl)
T2 = THP(H2, Pl)
Hc = IHWP(Pc)
Tc = TLP(Pc)

end

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

model type preheater

path feedwater < (W, Hl, Pl) - (W, H2, P2) >
cut extractsteam ( Ws, Hs, Psat )
path condensate < (Wcl, Hcl, .) - (Wc2, Hc2, .) >
local Tsat, TsatP, Rw, T2, Rc, Hsat, HsatP
parameter Vc, Vw, Hdiff, f, m, Cm

Pl**2 - P2**2 = f*W**2

  { mass balance }
Wc2 = Wcl + 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 + Wcl*Hsat - W*(H2-Hl) - Wc2*Hc2

Rc = RWP(Psat)
Hsat = HWP(Psat)
HsatP = HWPP(Psat)
Tsat = TLP(Psat)
TsatP = TLPP(Psat)
Rw = RHP(H2, P2)
```

```
H2 = Hsat - Hdiff
Hcl = Hsat
T2 = THP(H2, P1)

end

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

model type splitsteam

path extractsteam < (W, H, P) - [ (W1, H, P), (W2, H, P) ] >

W1 = Ø.3*W
W2 = Ø.7*W

end

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

model type dearator

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

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

Hwater = if Wwater { > Ø } 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 combchamber

cut heat [ (Q1), (Q2), (Q3), (Q4), (Q5), (Q6) ]
input Woil
parameter b1Ø, b2Ø, b3Ø, b4Ø, b5Ø, b6Ø
parameter b11, b21, b31, b41, b51, b61
parameter b12, b22, b32, b42, b52, b62

Q1 = b1Ø + b11*Woil + b12*Woil**2
Q2 = b2Ø + b21*Woil + b22*Woil**2
Q3 = b3Ø + b31*Woil + b32*Woil**2
Q4 = b4Ø + b41*Woil + b42*Woil**2
Q5 = b5Ø + b51*Woil + b52*Woil**2
Q6 = b6Ø + b61*Woil + b62*Woil**2

end

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

model type economizer

path feedwater < (W, H1, P1) - (W, H2, P2) >
cut heat (Q)
local Tm, T2, R2, T2H, TmH
parameter k, f, Cm, m, Ve

  { 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

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

model powerstation

submodel drumsyst
submodel (superheater) superh1, superh2, superh3
submodel (attemperator) attemp1, attemp2
submodel reheater
submodel controlvalve
submodel (turbsection) HPturb
submodel IPturb
submodel LPturb
submodel condensor
submodel (preheater) preh1, preh2, preh3, preh4, preh5,
         preh6, preh7
submodel splitsteam
submodel dearator
submodel feedwaterpump
submodel feedwatervalve
submodel combchamber
submodel economizer

connect (heat) combchamber to (economizer,
  drumsyst::risers, superh1, superh2, superh3, reheater)

connect (steam) drumsyst::drum to superh1 to attemp1  ->
  to superh2 to attemp2 to superh3 to  ->
  controlvalve to HPturb to reheater to IPturb  ->
  to LPturb to condensor

connect (extractsteam) HPturb to preh7,
  IPturb to (preh6, preh5, preh4,
    splitsteam to (dearator, preh3) ),
  LPturb to (preh2, preh1)

connect (feedwater) condensor to preh1 to preh2 to  ->
  preh3 to dearator to feedwaterpump to preh4  ->
  to preh5 to preh6 to preh7 to  ->
  feedwatervalve to  ->
  (economizer to drumsyst::drum, attemp1, attemp2)

connect (condensate) preh7 to preh6 to preh5  ->
    to preh4 to dearator,
  preh3 to preh2 to preh1 to condensor

connect (power) HPturb to IPturb to LPturb

HPturb.N1 = Ø
LPturb::LP3.Wp = Ø

end
```

```
>@add powersyst
>do eliminate
>partition
>print solved

 drumsyst::drum     Rw = RHP(Hd,Pd)
 drumsyst::risers   Rs = IRSP(drumsyst::drum.Pd)
                    Rwr = RWP(drumsyst::drum.Pd)
                    Rmix = (Vb*Rs + (Vr - Vb)*Rwr)/Vr

-                   drumsyst::drum.Pdc - drumsyst::drum.Pd = (1 + f*L/D)*[drumsyst::drum.Wdc]**2/
                      (2*a*Rmix) + g*L*Rmix
-drumsyst::drum     Pd - [Pdc] = (1 + f*L/D)*Wdc**2/(2*A*Rw) - g*L*Rw

 drumsyst::risers   drumsyst::drum.Wwr = (1 - x)*drumsyst::drum.Wdc
                    drumsyst::drum.Hwr = HWP(drumsyst::drum.Pd)
 attemp2            aw = Sw**2
 controlvalve       av = VALVE(Sv)
 attemp1            aw = Sw**2

-superh1            drumsyst::drum.Pd**2 - P2**2 = f*[drumsyst::drum.Ws]**2
-attemp1            feedwatervalve.P2 - [superh1.P2] = fw*(feedwatervalve.Wa1/aw)**2
-economizer         drumsyst::drum.Pd = [feedwatervalve.P2] - f*feedwatervalve.Wd**2
-feedwatervalve     [Wd] + Wa1 + Wa2 = dearator.W2
-attemp1            drumsyst::drum.Ws + [feedwatervalve.Wa1] = W2
-attemp2            [attemp1.W2] + feedwatervalve.Wa2 = W2
-                   feedwatervalve.P2 - superh2.P2 = fw*([feedwatervalve.Wa2]/aw)**2
-superh2            superh1.P2**2 - [P2]**2 = f*attemp1.W2**2
-HPturb             controlvalve.P2 = f*[attemp2.W2]
-controlvalve       superh3.P2**2 - [P2]**2 = fv*(attemp2.W2/av)**2
-superh3            superh2.P2**2 - [P2]**2 = f*attemp2.W2**2
-feedwaterpump      P2 = Ppump - f*[dearator.W2]**2
-preh4              [feedwaterpump.P2]**2 - P2**2 = f*dearator.W2**2
-preh5              [preh4.P2]**2 - P2**2 = f*dearator.W2**2
-preh6              [preh5.P2]**2 - P2**2 = f*dearator.W2**2
-preh7              [preh6.P2]**2 - P2**2 = f*dearator.W2**2
```

```
-feedwatervalve    P2 = [preh7.P2] - f*(dearator.W2/a)**2

 drumsyst::drum    Hs = IHSP(Pd)
                   Vw = VwØ + Adrum*z
                   Vs = VsØ - Adrum*z
                   HsP = HSPP(Pd)
                   Rs = IRSP(Pd)
                   RsP = RSPP(Pd)
                   Rwr = RWP(Pd)
                   derz = (feedwatervalve.Wd + Wwr - Wdc)/(Aw*Rw)
                   derHd = (feedwatervalve.Wd*economizer.H2 + Wwr*Hwr - Wdc*Hd)/((Vw + Vdc)*Rw)
 drumsyst::risers  Wsr = x*drumsyst::drum.Wdc
 drumsyst::drum    drumsyst::risers.dPd = (drumsyst::risers.Wsr - Ws + Adrum*derz*Rs)/(Vs*RsP)
                   drumsyst::risers.Hsr = (Vs*(RsP*Hs + Rs*HsP)*drumsyst::risers.dPd -
                     Adrum*derz*Rs*Hs + Ws*Hs)/drumsyst::risers.Wsr
 drumsyst::risers  xs = 2*Vb*Rs/(Vr*Rmix)
                   TAU = Vr*Rmix/drumsyst::drum.Wdc
                   RsP = RSPP(drumsyst::drum.Pd)

-                  derVb*Rs + Vb*RsP*dPd = [Wsprod] - Wsr
-                  -[derVb]*Rwr = drumsyst::drum.Wdc - Wsprod - drumsyst::drum.Wwr

                   Tmix = TLP(drumsyst::drum.Pd)
 combchamber       Q2 = b2Ø + b21*Woil + b22*Woil**2
 drumsyst::risers  Tm = Tmix + K*combchamber.Q2**(1/3)
                   TmixP = TLPP(drumsyst::drum.Pd)
                   RwrP = RWPP(drumsyst::drum.Pd)
                   Tdc = THP(drumsyst::drum.Hd,drumsyst::drum.Pdc)
                   HwrP = HWPP(drumsyst::drum.Pd)
                   HsP = IHSP(drumsyst::drum.Pd)
                   Hs = (combchamber.Q2 + drumsyst::drum.Wdc*drumsyst::drum.Hd - Wsr*Hsr -
                     drumsyst::drum.Wwr*drumsyst::drum.Hwr - (Cm*m*TmixP*dPd -
                     derVb*Rwr*drumsyst::drum.Hwr + (Vr - Vb)*(RwrP*drumsyst::drum.Hwr +
                     Rwr*HwrP)*dPd + Vb*Rs*HsP*dPd))/(derVb*Rs + Vb*RsP*dPd)
                   derx = 2/TAU*(xs - x)
 superhl           T2 = THP(H2,P2)
```

```
                     Tm = T2 + K*drumsyst::drum.Ws*(H2 - drumsyst::drum.Hs)
                     T2H = THPH(H2,P2)
                     TmH = T2H + K*drumsyst::drum.Ws
                     R2 = RHP(H2,P2)
 combchamber         Q3 = b30 + b31*Woil + b32*Woil**2
 superhl             derH2 = (combchamber.Q3 - drumsyst::drum.Ws*(H2 - drumsyst::drum.Hs))/
                       (m*Cm*TmH + Vs*R2)
 preh7               Hsat = HWP(Psat)
                     H2 = Hsat - Hdiff
 attempl             H2 = (drumsyst::drum.Ws*superhl.H2 + feedwatervalve.Wal*preh7.H2)/W2
 IPturb::IPl         reheater.W2 = reheater.P2/f
 reheater            T2 = THP(H2,P2)
                     T2H = THPH(H2,P2)
 combchamber         Q6 = b60 + b61*Woil + b62*Woil**2
 reheater            Tm = T2 + K*combchamber.Q6
                     R2 = RHP(H2,P2)
                     R2H = RHPH(H2,P2)
                     R2P = RHPP(H2,P2)

-HPturb              attemp2.W2 = [W2] + Wp
-                    P2**2 - preh7.Psat**2 = fp*([Wp]/S)**2
-reheater            [HPturb.P2]**2 - P2**2 = f*HPturb.W2**2

                     R2T = (HPturb.W2 - W2)/Vs
 HPturb              H = ISENX(superh3.H2,controlvalve.P2,P2)
                     H2 = H + (1 - Eh)*(superh3.H2 - H)
 reheater            derH2 = (combchamber.Q6 + HPturb.W2*HPturb.H2 - W2*H2 - Vs*R2T*H2)/
                       (m*Cm*T2H + Vs*R2)
                     derP2 = (R2T - R2H*derH2)/R2P

-IPturb::IPl         reheater.W2 = [W2] + Wp
-                    P2**2 - preh6.Psat**2 = fp*([Wp]/S)**2
-IPturb::IP2         [IPturb::IPl.P2] = f*IPturb::IPl.W2

 IPturb::IPl         H = ISENX(reheater.H2,reheater.P2,P2)
                     H2 = H + (1 - Eh)*(reheater.H2 - H)
```

```
powerstation        HPturb.N1 = Ø
HPturb              N2 = N1 + attemp2.W2*(superh3.H2 - H2)
IPturb::IP1         N2 = HPturb.N2 + reheater.W2*(reheater.H2 - H2)
                    T2 = THP(H2,P2)

-IPturb::IP2        IPturb::IP1.W2 = [W2] + Wp
-                   P2**2 - preh5.Psat**2 = fp*([Wp]/S)**2
-IPturb::IP3        [IPturb::IP2.P2] = f*IPturb::IP2.W2

IPturb::IP2         H = ISENX(IPturb::IP1.H2,IPturb::IP1.P2,P2)
                    H2 = H + (1 - Eh)*(IPturb::IP1.H2 - H)
                    N2 = IPturb::IP1.N2 + IPturb::IP1.W2*(IPturb::IP1.H2 - H2)
                    T2 = THP(H2,P2)

-IPturb::IP3        IPturb::IP2.W2 = [W2] + Wp
-                   P2**2 - preh4.Psat**2 = fp*([Wp]/S)**2
-IPturb::IP4        [IPturb::IP3.P2] = f*IPturb::IP3.W2

IPturb::IP3         H = ISENX(IPturb::IP2.H2,IPturb::IP2.P2,P2)
                    H2 = H + (1 - Eh)*(IPturb::IP2.H2 - H)
                    N2 = IPturb::IP2.N2 + IPturb::IP2.W2*(IPturb::IP2.H2 - H2)
                    T2 = THP(H2,P2)

-IPturb::IP4        IPturb::IP3.W2 = [W2] + Wp
-                   P2**2 - preh3.Psat**2 = fp*([Wp]/S)**2
-LPturb::LP1        [IPturb::IP4.P2] = f*IPturb::IP4.W2

IPturb::IP4         H = ISENX(IPturb::IP3.H2,IPturb::IP3.P2,P2)
                    H2 = H + (1 - Eh)*(IPturb::IP3.H2 - H)
                    N2 = IPturb::IP3.N2 + IPturb::IP3.W2*(IPturb::IP3.H2 - H2)
                    T2 = THP(H2,P2)

-LPturb::LP1        IPturb::IP4.W2 = [W2] + Wp
-                   P2**2 - preh2.Psat**2 = fp*([Wp]/S)**2
-LPturb::LP2        [LPturb::LP1.P2] = f*LPturb::LP1.W2
```

```
LPturb::LP1         H = ISENX(IPturb::IP4.H2,IPturb::IP4.P2,P2)
                    H2 = H + (1 - Eh)*(IPturb::IP4.H2 - H)
                    N2 = IPturb::IP4.N2 + IPturb::IP4.W2*(IPturb::IP4.H2 - H2)
                    T2 = THP(H2,P2)

-LPturb::LP2        LPturb::LP1.W2 = [W2] + Wp
-                   P2**2 - preh1.Psat**2 = fp*([Wp]/S)**2
-LPturb::LP3        [LPturb::LP2.P2] = f*LPturb::LP2.W2

 LPturb::LP2        H = ISENX(LPturb::LP1.H2,LPturb::LP1.P2,P2)
                    H2 = H + (1 - Eh)*(LPturb::LP1.H2 - H)
                    N2 = LPturb::LP1.N2 + LPturb::LP1.W2*(LPturb::LP1.H2 - H2)
                    T2 = THP(H2,P2)
 powerstation       LPturb::LP3.Wp = Ø
 LPturb::LP3        W2 = LPturb::LP2.W2 - Wp
 condensor          LPturb::LP3.P2 = Pw + Pdiff
 LPturb::LP3        H = ISENX(LPturb::LP2.H2,LPturb::LP2.P2,P2)
                    H2 = H + (1 - Eh)*(LPturb::LP2.H2 - H)
                    P2**2 - [Pp]**2 = fp*(Wp/S)**2
                    N2 = LPturb::LP2.N2 + LPturb::LP2.W2*(LPturb::LP2.H2 - H2)
                    T2 = THP(H2,P2)

-preh1              condensor.Pw**2 - P2**2 = f*[condensor.Ww]**2
-preh2              [preh1.P2]**2 - P2**2 = f*condensor.Ww**2
-preh3              [preh2.P2]**2 - dearator.Psat**2 = f*condensor.Ww**2

 condensor          Hw = IHWP(Pw)
                    H2 = Hw - Hdiff
                    Rw = RWP(Pw)
                    R2 = RHP(H2,P1)
                    Tw = TLP(Pw)
                    T1 = THP(H1,P1)
                    T2 = THP(H2,P1)
                    HwP = HWPP(Pw)
                    TwP = TLPP(Pw)
                    Tc = TLP(LPturb::LP3.P2)
```

```
                  prehl.Wc2 = Ww - LPturb::LP3.W2
                  prehl.Hc2 = IHWP(LPturb::LP3.P2)
                  derPw = (LPturb::LP3.W2*LPturb::LP3.H2 + prehl.Wc2*prehl.Hc2 - Ww*Hw -
                    Wl*(H2 - Hl))/(Vc*Rw*HwP + m*Cm*TwP + Vcool*R2*HwP)
prehl             preh2.Hc2 = HWP(Psat)
                  H2 = preh2.Hc2 - Hdiff
                  Tsat = TLP(Psat)
                  TsatP = TLPP(Psat)
                  Rw = RHP(H2,P2)
                  T2 = THP(H2,condensor.Pw)
                  Rc = RWP(Psat)
                  HsatP = HWPP(Psat)
                  preh2.Wc2 = Wc2 - LPturb::LP2.Wp
                  derPsat = (LPturb::LP2.Wp*LPturb::LP2.H2 + preh2.Wc2*preh2.Hc2 - condensor.Ww*
                    (H2 - condensor.Hw) - Wc2*Hc2)/(Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)
splitsteam        Wl = Ø.3*IPturb::IP4.Wp
                  W2 = Ø.7*IPturb::IP4.Wp
dearator          H2 = HWP(Psat)
                  Pc = Psat + Pdiff
                  Hwater = if Wwater then Hstorage else H2
                  Rw = RWP(Psat)
                  H2P = HWPP(Psat)
                  T2 = TLP(Psat)
                  T2P = TLPP(Psat)
                  Tc = TLP(Pc)
preh3             Hsat = HWP(Psat)
                  H2 = Hsat - Hdiff
dearator          preh4.Wc2 = W2 - (condensor.Ww + splitsteam.Wl + Wwater)
                  preh4.Hc2 = HWP(Pc)
                  derPsat = (splitsteam.Wl*IPturb::IP4.H2 + condensor.Ww*preh3.H2 + Wwater*Hwater +
                    preh4.Wc2*preh4.Hc2 - W2*H2)/(Vw*Rw*H2P + m*Cm*T2P)
combchamber       Ql = blØ + bll*Woil + bl2*Woil**2
                  Q4 = b4Ø + b41*Woil + b42*Woil**2
                  Q5 = b5Ø + b51*Woil + b52*Woil**2
economizer        T2 = THP(H2,drumsyst::drum.Pd)
                  Tm = T2 + k*combchamber.Ql
                  R2 = RHP(H2,drumsyst::drum.Pd)
                  T2H = THPH(H2,drumsyst::drum.Pd)
                  derH2 = (combchamber.Q1 + feedwatervalve.Wd*preh7.H2 - feedwatervalve.Wd*H2)/
                    (Cm*m*T2H + Ve*R2)
superh2           T2 = THP(H2,P2)
                  Tm = T2 + K*attemp1.W2*(H2 - attemp1.H2)
                  T2H = THPH(H2,P2)
                  TmH = T2H + K*attemp1.W2
                  R2 = RHP(H2,P2)
                  derH2 = (combchamber.Q4 - attemp1.W2*(H2 - attemp1.H2))/(m*Cm*TmH + Vs*R2)
superh3           T2 = THP(H2,P2)
attemp2           H2 = (attemp1.W2*superh2.H2 + feedwatervalve.Wa2*preh7.H2)/W2
superh3           Tm = T2 + K*attemp2.W2*(H2 - attemp2.H2)
                  T2H = THPH(H2,P2)
                  TmH = T2H + K*attemp2.W2
                  R2 = RHP(H2,P2)
                  derH2 = (combchamber.Q5 - attemp2.W2*(H2 - attemp2.H2))/(m*Cm*TmH + Vs*R2)
HPturb            T2 = THP(H2,P2)
preh2             preh3.Hc2 = HWP(Psat)
                  H2 = preh3.Hc2 - Hdiff
                  Tsat = TLP(Psat)
                  TsatP = TLPP(Psat)
                  Rw = RHP(H2,P2)
                  T2 = THP(H2,preh1.P2)
                  Rc = RWP(Psat)
                  HsatP = HWPP(Psat)
                  preh3.Wc2 = Wc2 - LPturb::LP1.Wp
                  derPsat = (LPturb::LP1.Wp*LPturb::LP1.H2 + preh3.Wc2*preh3.Hc2 - condensor.Ww*
                    (H2 - preh1.H2) - Wc2*Hc2)/(Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)
preh3             Wc1 = Wc2 - splitsteam.W2
                  Tsat = TLP(Psat)
                  TsatP = TLPP(Psat)
                  Rw = RHP(H2,dearator.Psat)
                  T2 = THP(H2,preh2.P2)
                  Rc = RWP(Psat)
                  HsatP = HWPP(Psat)
                  derPsat = (splitsteam.W2*IPturb::IP4.H2 + Wcl*Hsat - condensor.Ww*
                    (H2 - preh2.H2) - Wc2*Hc2)/Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)
preh4             preh5.Hc2 = HWP(Psat)
                  H2 = preh5.Hc2 - Hdiff
                  Tsat = TLP(Psat)
                  TsatP = TLPP(Psat)
                  Rw = RHP(H2,P2)
                  T2 = THP(H2,feedwaterpump.P2)
                  Rc = RWP(Psat)
                  HsatP = HWPP(Psat)
                  preh5.Wc2 = Wc2 - IPturb::IP3.Wp
                  derPsat = (IPturb::IP3.Wp*IPturb::IP3.H2 + preh5.Wc2*preh5.Hc2 - dearator.W2*
                    (H2 - dearator.H2) - Wc2*Hc2)/(Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)
preh5             preh6.Hc2 = HWP(Psat)
                  H2 = preh6.Hc2 - Hdiff
                  Tsat = TLP(Psat)
                  TsatP = TLPP(Psat)
                  Rw = RHP(H2,P2)
                  T2 = THP(H2,preh4.P2)
                  Rc = RWP(Psat)
                  HsatP = HWPP(Psat)
                  preh6.Wc2 = Wc2 - IPturb::IP2.Wp
                  derPsat = (IPturb::IP2.Wp*IPturb::IP2.H2 + preh6.Wc2*preh6.Hc2 - dearator.W2*
                    (H2 - preh4.H2) - Wc2*Hc2)/(Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)
preh6             preh7.Hc2 = HWP(Psat)
                  H2 = preh7.Hc2 - Hdiff
                  Tsat = TLP(Psat)
                  TsatP = TLPP(Psat)
                  Rw = RHP(H2,P2)
                  T2 = THP(H2,preh5.P2)
                  Rc = RWP(Psat)
                  HsatP = HWPP(Psat)
                  preh7.Wc2 = Wc2 - IPturb::IP1.Wp
                  derPsat = (IPturb::IP1.Wp*IPturb::IP1.H2 + preh7.Wc2*preh7.Hc2 - dearator.W2*
                    (H2 - preh5.H2) - Wc2*Hc2)/(Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)
preh7             Wcl = Wc2 - HPturb.Wp
```

```
Tsat = TLP(Psat)
TsatP = TLPP(Psat)
Rw = RHP(H2,P2)
T2 = THP(H2,preh6.P2)
Rc = RWP(Psat)
HsatP = HWPP(Psat)
derPsat = (HPturb.Wp*HPturb.H2 + Wc1*Hsat - dearator.W2*(H2 - preh6.H2) -
  Wc2*Hc2)/(Vc*Rc*HsatP  + m*Cm*TsatP + Vw*Rw*HsatP)
```

REFERENCES

Aarna O. (1976): Continuous Systems Modeling. TRITA - REG - 76Ø1, Department of Automatic Control, The Royal Institute of Technology, Stockholm, Sweden.

ASTAP: Advanced Statistical Analysis Program (ASTAP) - General Information Manual. Program No. 5796-PBH, IBM Corporation.

Aulin B. (1969): Decomposition of systems of high order. TFRT-5Ø46, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden (in swedish).

Birthwistle G.M., Dahl O.J., Myhrhaug B., Nygaard K. (1973): Simula Begin. Studentlitteratur, Lund

van den Bosch P.P.J., Bruijn P.M. (1977): The Dedicated Digital Computer as a Teaching Tool in Control Engineering; Interactive Instruction and Design. Proc. Trends in Automatic Control Education, Barcelona.

Brayton R.K., Gustavson F.G., Hachtel G.D (1972): A New Efficient Algorithm for Solving Differential-algebraic Systems using Implicit Backward Differentiation Formulas. Proc. of the IEEE, vol. 6Ø, No. 1, January 1972, pp 98-1Ø8.

Brennan R.D., Silberberg M.Y. (1968): The System/36Ø Continuous System Modeling Program. Simulation, Dec 1968, 3Ø1-3Ø8.

Brown R.L., 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 618Ø1.

Christensen J., (197Ø): The Structuring of Process Optimization. AIChE Journal, vol. 16, No. 2, March 197Ø, pp 177-184.

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

Cellier F.E., Blitz A.E. (1976): GASP-V: A Universal Simulation Package. Simulation of Systems, L. Dekker, editor. North-Holland Publishing Company.

Desoer C.A., Kuh E.H. (1969): Basic Circuit Theory. McGraw-Hill Book Company.

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-3Ø91, Department of Automatic Control, Lund Institute of Technology, Sweden.

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

Elmqvist H. (1977b): A New Model Language for Continuous Systems. LUTFD2/(TFRT-7132), Department of Automatic Control, Lund Institute of Technology, Sweden.

Fahrland D.A. (197Ø): Combined Discrete Event Continuous Systems Simulation. Simulation, February 197Ø, 61-72.

Fick G. (1975): SPODE - Simulation Package for Systems Modelled by Ordinary, First Order Differential Equations - User's Manual. Report C 2ØØ84-E5(E4). Swedish National Defense Research Institute.

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.

Golden D.G., Schoeffler J.D. (1973): GSL - A Combined Continuous and Discrete Simulation Language. Simulation, Jan. 1973, 1-8.

Gries D. (1971): Compiler Construction for Digital Computers. John Wiley & 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, 1Ø1-113.

Hay J.L., Sangster M.J. (1975): ISIS - An Interactive Simulation Language. Summer Computer Simulation Conference '75, 138-143.

Hornstein G. (1976): Simulation of Occupant Safety During Car Crashes. Simulation of Systems, L. Dekker, editor, North-Holland Publishing Company.

Joss J. (1976): Algorithmisches Differenzieren. Ph.D. Dissertation, Diss. ETH 5757, Eidgenossischen Technischen Hochschule, Zurich.

Koenig H.E., Tokad Y., Kesavan H. (1967): Analysis of Discrete Physical Systems. McGraw-Hill.

Korn G.A., Wait J.V. (1978): Digital Continuous-System Simulation. Prentice-Hall, Inc., New Jersey.

Kron G. (1963): Diakoptics - The piecewise Solution of Large-Scale Systems. MacDonald & Co., London.

Lambert J.D. (1973): Computational Methods in Ordinary Differential Equations. John Wiley & Sons.

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

Lee W., 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, 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.

Ord-Smith R.J., Stephenson J. (1975): Computer Simulation of Continuous Systems. Cambridge University Press.

Ortega J.M., Rheinboldt W.C. (1970): Iterative Solution of Nonlinear Equations in Several Variables. Academic Press.

Page E.S., Wilson L.B. (1973): Information Representation and Manipulation in a Computer. Cambridge University Press.

Pernebo L. (1977): Numerical Algorithms for Polynomial Matrix Systems. Report, Department of Automatic Control, Lund Institute of Technology, Sweden, (to appear).

Pritsker A.A.B., Hurst N.R. (1973): GASP-IV: A Combined Continuous - Discrete Fortran-based Simulation Language. Simulation, Sept 1973, 65-70.

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

Sim R.J.W. (1975): CADSIM - User's Guide and Reference Manual. Publ. No. 75/23, Dept. of Computing and Control, Imperial College, London.

Stadtherr M., Gifford W., 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.

Strauss J.C., editor (1967): The SCi Continuous System Simulation Language. Simulation, Dec 1967, 281-303.

Tarjan R.E. (1972): Depth First Search and Linear Graph Algorithms. SIAM J. Comp., 1, 146-160.

TESS (1972): TESS - Nonlinear Circuit and System Analysis Program - User Information Manual. Publ. No. 86615900, Control Data Corporation.

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

Volgin V., Samoilov Y., 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.

Åström K.J. (1976): Reglerteori (Control Theory). Almqvist & Wiksell, Stockholm (in swedish).

Ören T.I. (1977): Software for Simulation of Combined Continuous and Discrete Systems: A state-of-the-art Review. Simulation, Febr 1977, 33-45.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my adviser Professor Karl Johan Åström for his encouraging support and guidance. His comments on the manuscript has also been invaluable.

I also want to thank Leif Andersson and Sven-Erik Mattsson for their comments on the manuscript and Prof. Torsten Bohlin, Francois Cellier, Sture Lindahl and Jesper Oppelstrup for their comments on the report which preceeded this thesis.

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

This thesis has been prepared using a PDP-11 computer. The text was edited on a text screen using a page oriented editor written in Pascal. The lay out was produced by the RUNOFF program together with a program to handle half line spacing. 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.

/	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 repetition one or more times
[ ]*	denotes repetition none or more times

If any of the symbols / { } [ ] < > should be considered as terminal symbols it is underlined.

It should be noted that the syntax is not complete in some respects. It does not contain the definition of basic items like <identifier> and <number>. Trivial production rules such as <model identifier> ::= <identifier> are omitted. The syntax for <expression> is also omitted. It can e.g. be found in Naur (1962). The only exception is that <variable> is replaced by <variable spec>.

It is possible to use a comma as separator between variables in a list even if this is not included in the syntax. Comments are written within the parantheses { }. New line is treated as ;. Continuation of a statement on the next line is indicated by -> at the end of the line. Continuation is also assumed if a line ends with comma.

2. Syntax for model language

```
<model specification>::=[ <model type>; ]* <model>;

<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>; ]*

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

        <variable> ::= <identifier>

      <cut declaration> ::= [main] cut {<cut identifier>
          [<cut>]}*
        <cut>::=<cut clause> / <cut spec>
    <cut clause> ::= ( <variable cut> ) /
      [ <hierarchical cut> ]
    <variable cut> ::= [ <cut element> ]*
      [ / [ <cut element> ]* ]
    <cut element> ::= <variable> / -<variable> / .
    <hierarchical cut> ::= { <cut> / . }*
    <cut spec> ::= <model spec> [:<cut identifier>] /
      <cut identifier>
    <model spec> ::=
      <model identifier> [::<model identifier>]*

  <node declaration> ::= node {<node identifier>
      [<node clause>]}*
    <node clause> ::= ( <variable cut> ) /
      [ <hierarchical node> ]
    <hierarchical node> ::= { <node clause> /
      <node identifier> / . }*

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

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

  <equation> ::= <expression> = <expression>
    <variable spec> ::= [<model spec>.] <variable>

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

  <connection statement> ::=
      connect [(<identifier>)]
        { <connection expression> }*
    <connection expression> ::= <connection secondary>
      { { at/=/to/-/from/par///loop/
      branch/join } <connection secondary> }*
```

```
<connection secondary> ::=
  [reversed/\] <connection primary>
<connection primary> ::= <connection operand> /
  ( {<connection expression>/.}* )
<connection operand> ::= <cut spec> / <path spec> /
  <node identifier>
```

3. Listing of the translator program

```
begin

comment  DYMOLA TRANSLATOR

Author:  Hilding Elmqvist
Date:    January, 1978
--------------------------

Global variables and class Submodel.
-----------------------------------;



ref(submodel) array Modeltypes(1:25);
ref(variable) array Variables(1:700);
ref(cut) array Cuts(1:200);
ref(path) array Paths(1:30);
ref(equationnode) array Equations(1:500);
integer Nummodeltypes, Numvariables, Numcuts, Numpaths,
  Numequations;

ref(strongcomp) array Equsystems(1:500);
integer Numequsystems;
ref(variable) array Equvar(1:40);
integer Numequvar;

ref(submodel) Model,Currentsubmodel,Lastsubmodel;
ref(connectnode) Firstconnectnode, Prevconnectnode;
text Nextitem; integer Nexttype; real Nextnumb;
integer Idtype, Deltype, Numbtype;
text Connectionident;
boolean Eliminate;
ref(variable) Markvariable;
ref(expr) zero, one, two;
boolean nonlinear;
integer i;

comment ----------------------------------------------------------;

class SUBMODEL(modeltypeident);
  text modeltypeident;
  begin
  text modelident;
  integer submodelnr;
  ref(submodel) modeltype, superiormodel;
  ref(submodel) firstsubmodel, companionmodel;
  integer nsubmodels;
  integer ivariables, nvariables;
  integer icuts, ncuts;
  integer maincut;
  integer ipaths, npaths;
  integer mainpath;
  integer iequations, nequations;
  boolean occupied;
  comment  The variables ivariables, icuts, ipaths and
    iequations are biases for this submodel in the vectors
    Variables, Cuts, Paths and Equations. ;
```

```
procedure modelbody;
comment <model body> ::=
  <submodel part> <declaration part> <statement part> ;

  begin
  text modelid, modtypeid;
  boolean incorporation, declaration, statements,
    type, main;
  text vartype; ref(variable) var;
  text ident; integer clausenr;
  ref(submodel) submod, modtype, mod;
  ref(submodel) nextsub, prevsub;
  ref(connectnode) nextconnectnode;
  ref(path) newpath;

  comment  <submodel part> ::=
    [<model type> / <model> / <submodel incorporation>]* ;

  incorporation:=true;
  while incorporation do
    if Nextitem="model" then
      begin
      comment  <model type> / <model> ;
      Scan;
      if Nextitem="type" then begin Scan; type:=true; end
        else type:=false;
      if Nexttype=/Idtype then
        Error("Missing identifier");
      modelid:-copy(Nextitem);
      Scan;
      if Nextitem =/ ";" then Error("Missing ;");
      Scan;
      submod:-new submodel(modelid);

      submod.modelbody;

      Scan;
      if Nextitem=/";" then Error("Syntax error");
      Scan;
      if type then
        begin
        Nummodeltypes:=Nummodeltypes+1;
        Modeltypes(Nummodeltypes):-submod;
        end
      else
        begin
        nsubmodels:=nsubmodels+1;
        nextsub:-submod.newsubmodel(this submodel,modelid,
          nsubmodels);
        if firstsubmodel==none then firstsubmodel:-nextsub
          else prevsub.companionmodel:-nextsub;
        prevsub:-nextsub;
        end
      end
    else
    if Nextitem="submodel" then
      begin
    comment  <submodel incorporation> ;
    Scan;
    if Nextitem="(" then
      begin
      Scan;
      if Nexttype=/Idtype then
         Error("Missing modeltype identifier");
      modtypeid:-copy(Nextitem);
      Scan;
      if Nextitem=/")" then Error("Missing )");
      Scan;
      if not search(i,Nummodeltypes,
        Modeltypes(i).modeltypeident=modtypeid) then
          Error("Not existent model type");
      modtype:-Modeltypes(i);
      end
    else
      modtype:-none;
    while Nexttype=Idtype do
      begin
      if modtype==none then
        begin
        if not search(i,Nummodeltypes,
          Modeltypes(i).modeltypeident=Nextitem) then
            Error("Not existent model type");
        mod:-Modeltypes(i);
        end
      else
        mod:-modtype;
      nsubmodels:=nsubmodels+1;
      nextsub:-mod.newsubmodel(this submodel,Nextitem,
        nsubmodels);
      if firstsubmodel==none then firstsubmodel:-nextsub
        else prevsub.companionmodel:-nextsub;
      prevsub:-nextsub;
      Scan;
      comment  Skip <parameter list> ;
      if Nextitem = "(" then
        begin
        while Nextitem =/ ")" do Scan;
        Scan;
        end;
      if Nextitem="," then Scan;
      end;
      Scan;
    end
  else
    incorporation:=false;
  comment  End of <submodel part> ;

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

Currentsubmodel:-this submodel;
ivariables:=Numvariables;
icuts:=Numcuts;
ipaths:=Numpaths;
```

```
declaration:=true;
while declaration do
  if Nextitem="parameter" or Nextitem="constant" or
    Nextitem="terminal" or Nextitem="input" or
    Nextitem="output"  or Nextitem="local" then
    begin
    comment  <variable declaration> ;
    vartype:-copy(Nextitem);
    Scan;
    while Nexttype=Idtype do
      begin
      var:-new variable(this submodel,Nextitem);
      nvariables:=nvariables+1;
      Scan;
      if Nextitem="=" then
        begin
        Scan;
        if Nexttype=/Numbtype then
          Error("Missing number");
        Scan;
        end;
      if vartype="parameter" or vartype="constant"
        or vartype="input" then
        var.known:=true;
      if Nextitem="," then Scan;
      end;
    if Nextitem=/";" then Error("; expected");
    Scan;
    end
  else

  if Nextitem = "cut" or Nextitem = "node"
    or Nextitem = "path" or Nextitem = "main" then
  begin
  if Nextitem = "main" then begin Scan; main:=true; end;
  if Nextitem = "cut" or Nextitem = "node" then
    begin
    comment  <cut declaration> / <node declaration> ;
    Scan;
    while Nextitem =/ ";" do
      begin
      if Nexttype =/ Idtype then
        Error("Missing cut or node identifier");
      ident:-copy(Nextitem);
      Scan;
      if Nextitem = "[" or Nextitem = "(" then
         begin
         clausenr:=cutclause;
         Cuts(icuts+clausenr).identifier:-ident;
         end
       else
         begin
         new nodecut.identifier:-ident;
         end;
      ncuts:=Numcuts-icuts;
      if main then
        begin
        maincut:=Numcuts-icuts;
```

```
          main:=false;
          end;
        if Nextitem = "," then Scan;
        end;
      Scan;
      end
    else
    if Nextitem = "path" then
      begin
      comment <path declaration> ;
      Scan;
      while Nextitem =/ ";" do
        begin
        if Nexttype =/ Idtype then
          Error("Missing path identifier");
        newpath:-new path(Nextitem);
        Scan;
        if Nextitem =/ "<" then Error("Missing <");
        Scan;
        scancut(newpath.cutnr1,newpath.modspec1);
        if Nextitem =/ "-" then Error("Missing -");
        Scan;
        scancut(newpath.cutnr2,newpath.modspec2);
        if Nextitem =/ ">" then Error("Missing >");
        Scan;

        if main then
          begin
          mainpath:=Numpaths-ipaths;
          main:=false;
          end;
        if Nextitem = "," then Scan;
        end;
      npaths:=Numpaths-ipaths;
      ncuts:=Numcuts-icuts;
      Scan;
      end;
    end
  else declaration:=false;
  comment  End of <declaration part> ;

  comment  <statement part> ::=
    [ <equation> / <connetion statement> ]* ;

  iequations:=Numequations;
  Firstconnectnode:-none;
  Currentsubmodel:-this submodel;
  statements:=true;
  while statements do
    if Nextitem = "connect" then
      begin
      connectionstatement;
      end
    else
    if Nextitem =/ "end" then
      begin
      new equationnode(Equation,this submodel);
      if Nextitem=/ ";" then
```

```
          Error("Syntax error: ; expected");
        Scan;
        end
      else
        statements:=false;
    comment  End of <statement part> ;

    comment  Generate equations corresponding to the
      connections statements in this submodel ;

    nextconnectnode:-Firstconnectnode;
    while nextconnectnode =/= none do
      begin
      if nextconnectnode.connected=/=none then
        nextconnectnode.connect;
      nextconnectnode:-nextconnectnode.next;
      end;
    for i:=1 step 1 until Numcuts do
      Cuts(i).connected:-none;

    nvariables:=Numvariables-ivariables;
    nequations:=Numequations-iequations;
    end - modelbody - ;



  ref(submodel) procedure newsubmodel(supmod,modid,submodnr);
    value modid; ref(submodel) supmod; text modid;
    integer submodnr;
    begin
    ref(submodel) mtype, newsub, next;
    ref(submodel) newsubsub, prevsub;
    integer i;
    if not occupied then
      begin
      occupied:=true;
      newsubmodel:-this submodel;
      modelident:-modid;
      modeltype:-this submodel;
      superiormodel:-supmod;
      submodelnr:=submodnr;
      end
    else
      begin
      mtype:-this submodel;
      newsub:-new submodel(modeltypeident);
      inspect newsub do
        begin
        nsubmodels:=mtype.nsubmodels;
        next:-mtype.firstsubmodel;
        for i:=1 step 1 until nsubmodels do
          begin
          newsubsub:-next.modeltype.newsubmodel(newsub,
            next.modelident,i);
          if firstsubmodel==none then firstsubmodel:-newsubsub
            else prevsub.companionmodel:-newsubsub;
          prevsub:-newsubsub;
          next:-next.companionmodel;
          end;
        superiormodel:-supmod;
        modelident:-modid;
        modeltype:-mtype;
        submodelnr:=submodnr;

        duplicate;

        icuts:=Numcuts;
        ncuts:=mtype.ncuts;
        maincut:=mtype.maincut;
        for i:=1 step 1 until ncuts do
          Cuts(mtype.icuts+i).duplicate(this submodel);

        ipaths:=mtype.ipaths;
        npaths:=mtype.npaths;
        mainpath:=mtype.mainpath;

        end;
      newsubmodel:-newsub;
      end;
    end - newsubmodel - ;


  procedure duplicate;
    begin
    integer i;

    if this submodel =/= modeltype then
      begin
      ivariables:=Numvariables;
      nvariables:=modeltype.nvariables;
      for i:=1 step 1 until nvariables do
        Variables(modeltype.ivariables+i).
          duplicate(this submodel);

      iequations:=Numequations;
      nequations:=modeltype.nequations;
      for i:=1 step 1 until nequations do
        Equations(modeltype.iequations+i).
          duplicate(this submodel);

      end;
    end - duplicate - ;


boolean procedure searchsubmodel(ident,submod,submodnr);

comment  Returns true if ident is the model identifier of
  a submodel of the current model. It also returns a
  reference to that submodel and its submodel number. ;

  name submod, submodnr; value ident;
  text ident; ref(submodel) submod; integer submodnr;
  begin
  boolean found;
  submodnr:=0;
  found:=false;
```

```
   while submodnr<nsubmodels and not found do
     begin
     submodnr:=submodnr+1;
     submod:-if submodnr=1 then firstsubmodel else
       submod.companionmodel;
     found:=submod.modelident = ident;
     end;
   searchsubmodel:=found;
   end - searchsubmodel - ;

 ref(modelspec) procedure scanmodelspec;

 comment  Scans for <model spec> and generates a list of
   modelspec-objects describing the specification.
   <model spec>::=<model identifier>[::<model identifier>]* ;

   begin
   ref(submodel) submod;
   integer submodnr;
   ref(modelspec) modspec;
   if not searchsubmodel(Nextitem,submod,submodnr) then
     begin
     scanmodelspec:-none;
     end
   else
     begin
     Scan;
     if Nextitem = "::" then
       begin
       Scan;
       modspec:-submod.scanmodelspec;
       if modspec == none then
         Error("Not incorporated submodel");
       end
     else
       modspec:-none;
     scanmodelspec:-new modelspec(submodnr,modspec);
     end;
   end - scanmodelspec - ;


 procedure infix(subm); ref(submodel) subm;

 comment  Generates a <model spec> for this submodel relative
   to submodel subm and print it. ;

   begin
   if superiormodel =/= none then
     begin
     if superiormodel.superiormodel =/= none and
       superiormodel =/= subm then
       begin
       superiormodel.infix(subm);
       outtext("::");
       end;
     end;
     outtext(modelident);
   end - infix - ;
```

```
  end - SUBMODEL -;

comment --------------------------------------------------;

class MODELSPEC(submodnr,modspec);
  integer submodnr; ref(modelspec) modspec;
  begin

  ref(submodel) procedure specmod(currentmod);
  comment  Evaluates the model specification relative to
    submodel currentmod. ;
    ref(submodel) currentmod;
    begin
    integer i; ref(submodel) submod;
    submod:-currentmod.firstsubmodel;
    for i:=2 step 1 until submodnr do
      submod:-submod.companionmodel;

    specmod:-if modspec == none then submod
      else modspec.specmod(submod);
    end;

  end - MODELSPEC - ;

comment --------------------------------------------------;

ref(modelspec) procedure SPECIFIED(submod);
comment  Generates a list of modelspec-objects for submod
  relative to Currentsubmodel. ;
  ref(submodel) submod;
  begin
  ref(submodel) subm; ref(modelspec) spec;
  subm:-submod;
  while subm =/= Currentsubmodel do
    begin
    spec:-new modelspec(subm.submodelnr,spec);
    subm:-subm.superiormodel;
    end;
  specified:-spec;
  end - SPECIFIED - ;
```

```
comment class CUT
-----------------;



class CUT;
  virtual: procedure duplicate, atop;
  begin
  text identifier;
  ref(submodel) submod;
  ref(cut) connected;

  procedure atop(c2); ref(cut) c2;
    comment  Links variablecuts and nodecuts that are
      connected by the at operator to a circular list
      together with a head, connectnode. When all connection
      statements of a submodel have been translated,
      equations are generated from these lists by the
      procedure connect of connectnode. ;
    begin
    ref(cut) lastcut1, lastcut2, connode;
    integer nacross2, nthrough2;
    if c2 is hierarchicalcut then Error("Incompatible cuts");
    if connected == none then
      begin
      connected:-new connectnode;
      connected qua connectnode.connected:-this cut;
      if this cut is variablecut then
        begin
        connected qua connectnode.nacrossvar:=
          this cut qua variablecut.nacrossvar;
        connected qua connectnode.nthroughvar:=
          this cut qua variablecut.nthroughvar;
        end;
      end;
    lastcut1:-connected;
    while not lastcut1.connected is connectnode do
      lastcut1:-lastcut1.connected;
    lastcut2:-c2.connected;
    if lastcut2 =/= none then
      while not lastcut2.connected is connectnode do
        lastcut2:-lastcut2.connected;
    connode:-lastcut1.connected;
    if c2.connected == none then
      begin
      lastcut1.connected:-c2;
      c2.connected:-connode;
      inspect c2 when variablecut do
        begin
        nacross2:=nacrossvar;
        nthrough2:=nthroughvar;
        end;
      end
    else
      begin
      inspect lastcut2.connected qua connectnode do
        begin
```

```
          nacross2:=nacrossvar;
          nthrough2:=nthroughvar;
          end;
        lastcut1.connected:-lastcut2.connected.connected;
        lastcut2.connected.connected:-none;
        lastcut2.connected:-connode;
        end;
      inspect connode qua connectnode do
        begin
        if nacrossvar = Ø and nthroughvar = Ø then
          begin
          nacrossvar:=nacross2;
          nthroughvar:=nthrough2;
          end
        else
        if nacross2 =/ Ø or nthrough2 =/Ø then
          begin
          if nacrossvar =/ nacross2 or nthroughvar =/ nthrough2
            then Error
            ("Not equal number of across or through variables");
          end;
        end;
      end - atop - ;

    Numcuts:=Numcuts+1;
    Cuts(Numcuts):-this cut;
    submod:-Currentsubmodel;
    end - CUT - ;

  comment ------------------------------------------------------;

  cut class NODECUT;
    begin

    procedure duplicate(subm); ref(submodel) subm;
      begin
      ref(nodecut) ncut;
      ncut:-new nodecut;
      ncut.identifier:-identifier;
      ncut.submod:-subm;
      end;

    end - NODECUT - ;

  comment ------------------------------------------------------;

  cut class HIERARCHICALCUT;
    begin
    ref(subcut) firstcut;
    integer nsubcuts;

    procedure scanclause;
      begin
      ref(subcutnr) nextsubcut, prevsubcut;
      integer cutnr; ref(modelspec) modsp;

      Scan;
      while Nextitem=/ "]" do
      begin

      scancut(cutnr,modsp);
      if modsp == none then
        begin
        nextsubcut:-new subcutnr;
        nextsubcut.cutnr:=cutnr;
        end
      else
        begin
        nextsubcut:-new subcutspec(modsp);
        nextsubcut.cutnr:=cutnr;
        end;

      nsubcuts:=nsubcuts+1;
      if firstcut==none then firstcut:-nextsubcut
        else prevsubcut.nextcut:-nextsubcut;
      prevsubcut:-nextsubcut;
      if Nextitem = "," then Scan;
      end;
    Scan;
    end;

  procedure duplicate(subm); ref(submodel) subm;
    begin
    ref(hierarchicalcut) hcut;
    hcut:-new hierarchicalcut;
    hcut.identifier:-identifier;
    hcut.submod:-subm;
    hcut.firstcut:-firstcut;
    hcut.nsubcuts:=nsubcuts;
    end;

  procedure atop(c2); ref(cut) c2;
    comment Replaces an at operation on two hierarchical cuts
      by at operations on all their corresponding subcuts. ;
    begin
    ref(hierarchicalcut) h2;
    ref(subcut) subcut1, subcut2;
    if not c2 is hierarchicalcut then
      Error("Not compatible cuts");
    h2:-c2;
    if nsubcuts =/ h2.nsubcuts then
      Error("Not equal number of subcuts");
    subcut1:-firstcut;
    subcut2:-h2.firstcut;
    for i:=1 step 1 until nsubcuts do
      begin
      subcut1.cutref(submod).atop(subcut2.cutref(c2.submod));
      subcut1:-subcut1.nextcut;
      subcut2:-subcut2.nextcut;
      end;
    end;

  end - HIERARCHICALCUT - ;

comment ---------------------------------------------------;
```

```
cut class VARIABLECUT;
  begin
  ref(cutvariable) firstacrossvar, firstthroughvar;
  integer nacrossvar, nthroughvar;

  procedure scanclause;
    begin

    ref(cutvariable) procedure variablelist(nvar);
      name nvar; integer nvar;
      begin
      ref(cutvariable) firstcutvar, nextcutvar, prevcutvar;
      integer i; boolean minus;
      nvar:=0;

      while Nextitem =/ "/" and Nextitem =/ ")" do
        begin
        nextcutvar:-new cutvariable;
        if Nextitem = "." then begin end
        else
          begin
          if Nextitem = "-" then
            begin
            minus:=true;
            Scan;
            end;
          if Nexttype=/Idtype then
            Error("Missing variable identifer");
          if not search(i,Currentsubmodel.nvariables,
            Variables(Currentsubmodel.ivariables+i).
            identifier=Nextitem)
            then
            begin
            new variable(Currentsubmodel,Nextitem);
            Currentsubmodel.nvariables:=Currentsubmodel.
              nvariables+1;
            i:=Numvariables-Currentsubmodel.ivariables;
            end;
          nextcutvar.var:=if not minus then i else -i;
          end;
        if firstcutvar == none then firstcutvar:-nextcutvar
          else prevcutvar.nextvar:-nextcutvar;
        prevcutvar:-nextcutvar;
        nvar:=nvar+1;
        Scan;
        if Nextitem = "," then Scan;
        end;
      variablelist:-firstcutvar;
      end;

    Scan;
    firstacrossvar:-variablelist(nacrossvar);
    if Nextitem = "/" then
      begin
      Scan;
      firstthroughvar:-variablelist(nthroughvar);
      end;
      if Nextitem =/ ")" then Error("Missing )");
      Scan;
    end;

  procedure duplicate(subm); ref(submodel) subm;
    begin
    ref(variablecut) vcut;
    vcut:-new variablecut;
    vcut.identifier:-identifier;
    vcut.submod:-subm;
    vcut.firstacrossvar:-firstacrossvar;
    vcut.firstthroughvar:-firstthroughvar;
    vcut.nacrossvar:=nacrossvar;
    vcut.nthroughvar:=nthroughvar;
    end;

  end - VARIABLECUT - ;

comment --------------------------------------------------------;

cut class CONNECTNODE;
  begin
  ref(connectnode) next;
  integer nacrossvar, nthroughvar;

  procedure connect;
    begin
    ref(cut) c;
    ref(expr) array lastvar(Ø:nacrossvar),
      lastequ(Ø:nthroughvar);
    integer i;
    ref(cutvariable) cutvar; ref(expr) varnode;

    for i:=1 step 1 until nthroughvar do
      lastequ(i):-zero.Equal(zero);
    c:-connected;
    while c is nodecut do c:-c.connected;
    while not c is connectnode do
      begin
      cutvar:-c qua variablecut.firstacrossvar;
      for i:=1 step 1 until nacrossvar do
        if cutvar.var =/ Ø then
          begin
          varnode:-if c.submod == Currentsubmodel
            then new variablenode(abs(cutvar.var))
            else new variablespec(abs(cutvar.var),specified
              (c.submod));
          if cutvar.var<Ø then varnode:-Minus(varnode);
          if lastvar(i) =/= none then
            new equationnode(varnode.equal(lastvar(i)),
              Currentsubmodel);
          lastvar(i):-varnode;
          cutvar:-cutvar.nextvar;
          end;

      cutvar:-c qua variablecut.firstthroughvar;
      for i:=1 step 1 until nthroughvar do
        if cutvar.var =/ Ø and lastequ(i) =/= none then
          begin
          varnode:-if c.submod == Currentsubmodel
            then new variablenode(abs(cutvar.var))
            else new variablespec(abs(cutvar.var),specified
              (c.submod));
          if cutvar.var<Ø then varnode:-Minus(varnode);
          if c.submod ==Currentsubmodel then varnode:-
            Minus(varnode);
          lastequ(i).Add(varnode);
          cutvar:-cutvar.nextvar;
          end
        else
          lastequ(i):-none;

      c:-c.connected;
      while c is nodecut do c:-c.connected;
      end;

    for i:=1 step 1 until nthroughvar do
     if lastequ(i) =/= none then
       new equationnode(lastequ(i),Currentsubmodel);
    end - connect - ;

  if Firstconnectnode == none then
    Firstconnectnode:-this connectnode
  else Prevconnectnode.next:-this connectnode;
  Prevconnectnode:-this connectnode;

  Numcuts:=Numcuts-1;
  end - CONNECTNODE - ;

comment ------------------------------------------------------;


class SUBCUT; virtual: ref(cut) procedure cutref;
  begin
  ref(subcut) nextcut;
  end;

subcut class SUBCUTNR;
  begin
  integer cutnr;

  ref(cut) procedure cutref(submod); ref(submodel) submod;
    begin
    ref(submodel) subm;
    subm:-if this subcut is subcutspec then
      this subcut qua subcutspec.modspec.specmod(submod)
      else submod;
    cutref:-Cuts(subm.icuts+cutnr);
    end;

  end - SUBCUTNR - ;

subcutnr class SUBCUTSPEC(modspec);
    ref(modelspec) modspec;;

subcut class SUBCUTREF(c); ref(cut) c;
```

```
    begin
    ref(cut) procedure cutref(submod); ref(submodel) submod;
      cutref:-c;
    end;

  class CUTVARIABLE;
    begin
    ref(cutvariable) nextvar;
    integer var;
    end;

  comment ---------------------------------------------------;

  integer procedure CUTCLAUSE;
    begin
    cutclause:=Numcuts-Currentsubmodel.icuts+1;
    if Nextitem = "[" then
      new hierarchicalcut.scanclause
    else
      new variablecut.scanclause;
    end;

  comment ---------------------------------------------------;

  procedure SCANCUT(cutnr,modspec);
    name cutnr, modspec;
    integer cutnr; ref(modelspec) modspec;
    begin
    ref(submodel) subm;

    if Nextitem = "[" or Nextitem = "(" then
      cutnr:=cutclause
    else
      begin
      if Nexttype =/ Idtype then
        Error("Missing cutclause or cut spec");
      modspec:-Currentsubmodel.scanmodelspec;
      if modspec =/= none then
        begin
        subm:-modspec.specmod(Currentsubmodel);
        if Nextitem = ":" then
          begin
          Scan;
          if not search(cutnr,subm.ncuts,
            Cuts(subm.icuts+cutnr).identifier = Nextitem)
            then Error("Not declared cut");
          Scan;
          end
        else
        if subm.maincut =/ Ø then
          cutnr:=subm.maincut
        else
          Error("No main cut declared");
        end
      else
        begin
        if not search(cutnr,Currentsubmodel.ncuts,
          Cuts(Currentsubmodel.icuts+cutnr).identifier=Nextitem)
      then Error("Not declared cut");
    Scan;
    end;
  end;
end - SCANCUT - ;
```

```
comment  class VARIABLE, class EQUATIONNODE,
  procedure EQUATION and class EXPR
-------------------------------------------;



class VARIABLE(submod,identifier);
  ref(submodel) submod;
  text identifier;
  begin
  boolean known;
  ref(variable) alias;
  integer equnr;

  procedure duplicate(subm); ref(submodel) subm;
    begin
    ref(variable) v;
    v:-new variable(subm,notext);
    v.identifier:-identifier;
    v.known:=known;
    end;

  procedure infix;
    begin
    if this variable == Markvariable then outtext("[");
    if submod =/= Currentsubmodel then
      begin
      submod.infix(Currentsubmodel);
      outtext(".");
      end;
    outtext(identifier);
    if this variable == Markvariable then outtext("]");
    end;

  ref(variable) procedure eliminated;
    eliminated:-if not Eliminate or alias == none
      or alias == this variable then this variable
      else alias.eliminated;

  boolean procedure determined;
    determined:=known or (Eliminate and alias=/=this variable
      and alias =/= none);

  if identifier =/= notext then
    begin
    identifier:-if search(i,Numvariables,Variables(i).
      identifier=identifier) then
      Variables(i).identifier else copy(identifier);
    end;
  Numvariables:=Numvariables+1;
  Variables(Numvariables):-this variable;
  end - VARIABLE - ;

comment ---------------------------------------------------------;

class EQUATIONNODE(express, submod); ref(expr) express;
  ref(submodel) submod;
```

```
begin
integer varnr;  comment Index for assigned variable ;

procedure infix(string); text string;
  if not (Eliminate and trivial) then
    begin
    outtext(string);
    if submod =/= Lastsubmodel then
      begin
      submod.infix(none);
      Lastsubmodel:-submod;
      end;
    setpos(max(20,pos+2));
    Currentsubmodel:-submod;
    Markvariable:-if varnr=0 then none else Variables(varnr);
    express.infix;
    outimage;
    end;

boolean procedure trivial;
  trivial:=express qua Equalop.expr1 in variablenode
    and express qua Equalop.expr2 in variablenode;

procedure elimin;
  begin
  ref(variable) var1, var2;
  if trivial then
    begin
    Currentsubmodel:-submod;
    var1:-express qua Equalop.expr1 qua variablenode.varib;
    var2:-express qua Equalop.expr2 qua variablenode.varib;
    if not var1.determined and not var2.determined then
      var1.alias:-var2.alias:-var2
    else
    if not var1.determined then var1.alias:-var2
    else
    if not var2.determined then var2.alias:-var1
    else
    if var1==var2 then
      begin
      outtext("Redundant equation:"); outimage;
      Eliminate:=false;
      infix(notext);
      Eliminate:=true;
      end
    else
      var1.alias:-var2;

    end;
  end;

procedure traverse;
  begin
  Currentsubmodel:-submod;
  Numequvar:=0;
  express.traverse;
  end;
```

```
   procedure duplicate(subm); ref(submodel) subm;
     new equationnode(express,subm);

   ref(expr) procedure Solve;
     begin
     ref(expr) x, f0, f1, g0, g1;
     Currentsubmodel:-submod;
     express qua equalop.expr1.Linj(Variables(varnr),f0,f1);
     express qua equalop.expr2.Linj(Variables(varnr),g0,g1);
     x:-new variableref(0,Variables(varnr));
     Solve:-x.Equal(g0.Sub(f0).Div(f1.Sub(g1)));
     end;

   Numequations:=Numequations+1;
   Equations(Numequations):-this equationnode;
   end - EQUATIONNODE -;

comment ----------------------------------------------------;


ref(expr) procedure PRIMARY;
   comment <primary> ::= (<expression>) / <variable spec> /
     <function identifier>(<expression>[,<expression>]*) /
     {der/der2}(<variable>) / <variable>{'/''} / <number> ;
   begin

   integer procedure dervar(varid,prefix);
     text varid, prefix;
     begin
     text derid;
     if not search(i,Currentsubmodel.nvariables,
       Variables(Currentsubmodel.ivariables+i).identifier
       =Nextitem) then Error("Not declared variable");
     Variables(Currentsubmodel.ivariables+i).known:=true;
     derid:-blanks(prefix.length+varid.length);
     derid.sub(1,prefix.length):=prefix;
     derid.sub(prefix.length+1,varid.length):=varid;
     if search(i,Currentsubmodel.nvariables,
       Variables(Currentsubmodel.ivariables+i).identifier=derid)
       then dervar:=i
     else
       begin
       new variable(Currentsubmodel,derid);
       dervar:=Numvariables-Currentsubmodel.ivariables;
       end;
     if prefix = "der2" then
       begin
       derid:-blanks(3+varid.length);
       derid.sub(1,3):="der";
       derid.sub(4,varid.length):=varid;
       if search(i,Currentsubmodel.nvariables,
         Variables(Currentsubmodel.ivariables+i).identifier=derid)
         then Variables(Currentsubmodel.ivariables+i).known:=true
       else
         begin
         new variable(Currentsubmodel,derid);
         Variables(Numvariables).known:=true;
```

```
      end;
    end;
  Currentsubmodel.nvariables:=Numvariables-Currentsubmodel.
    ivariables;
  end;

ref(modelspec) modspec; ref(submodel) subm;
text ident; integer narg;
ref(argument) firstargument, nextargument, prevargument;
text der; integer ider;

if Nextitem= "(" then
  begin
  Scan;
  Primary:-Expression;
  if Nextitem=/ ")" then Error("Syntax error: ) expected");
  Scan;
  end

else
if Nexttype = Idtype then
  begin
  modspec:-Currentsubmodel.scanmodelspec;
  if modspec =/= none then
    begin
    if Nextitem =/ "." then Error("Missing .");
    Scan;
    if Nexttype =/ Idtype then Error("Missing variable");
    subm:-modspec.specmod(Currentsubmodel).modeltype;
    if not search(i,subm.nvariables,
      Variables(subm.ivariables+i).identifier=Nextitem) then
      Error("Not declared variable");
    Primary:-new variablespec(i,modspec);
    Scan;
    end
  else
  if search(i,Currentsubmodel.nvariables,
    Variables(Currentsubmodel.ivariables+i).identifier
    =Nextitem) then
    begin
    Scan;
    while Nextitem = "'" do begin ider:=ider+1; Scan; end;
    if ider>Ø then
      begin
      der:-copy("der ");
      der.sub(4,1).putint(ider);
      i:=dervar(ident,der);
      end;
    Primary:-new variablenode(i);
    end
  else
    begin
    ident:-copy(Nextitem);
    Scan;
    if Nextitem =/ "(" then Error("Not declared variable");
    if ident.sub(1,3) =  "der" then
      begin
      Scan;
```

```
        i:=dervar(Nextitem,ident);
        Scan;
        if Nextitem =/ ")" then Error("Missing )");
        Primary:-new variablenode(i);
        Scan;
        end
      else
       begin
        Firstargument:-none;
        while Nextitem =/ ")" do
          begin
          if narg>Ø and Nextitem =/ "," then
            Error("Syntax error: , expected");
          Scan;
          nextargument:-new argument(expression);
          narg:=narg+1;
          if firstargument == none then
            firstargument:-nextargument
            else prevargument.nextarg:-nextargument;
          prevargument:-nextargument;
          end;
        Primary:-new functionop(ident,firstargument,narg);
        Scan;
        end;
      end;
    end

  else
  if Nexttype=Numbtype then
    begin
    Primary:-if Nextnumb=Ø then zero else if Nextnumb=1 then one
      else if Nextnumb=2 then two else
      new numbernode(Nextitem, Nextnumb);
    Scan;
    end
  else
    Error("Missing primary");
  end - PRIMARY - ;

comment -----------------------------------------------------------;

ref(expr) procedure Factor;
  begin
  ref(expr) express;

  express:-Primary;
  while Nextitem = "**" do
    begin
    Scan;
    express:- express.Power(Primary);
    end;
  Factor:-express;
  end;

comment -----------------------------------------------------------;

ref(expr) procedure Term;
  begin
  ref(expr) express;

  express:-Factor;
  while Nextitem = "*" or Nextitem = "/" do
    if Nextitem = "*" then
      begin
      Scan;
      express:-express.Mult(Factor)
      end
    else
      begin
      Scan;
      express:-express.Div(Factor);
      end;
  Term:-express;
  end;

comment ---------------------------------------------------;

ref(expr) procedure Simpexpr;
  begin
  ref(expr) express;

  if Nextitem = "+" then
    begin
    Scan;
    express:- Term;
    end
  else
  if Nextitem = "-" then
    begin
    Scan;
    express:- Minus(Term);
    end
  else
    begin
    express:-Term;
    end;

  while Nextitem = "+" or Nextitem = "-" do
    if Nextitem = "+" then
      begin
      Scan;
      express:-express.Add(Term)
      end
    else
      begin
      Scan;
      express:-express.Sub(Term);
      end;
  Simpexpr:-express;
  end;

comment ---------------------------------------------------;

ref(expr) procedure expression;
  begin
  ref(expr) expr1, expr2, expr3;
  if Nextitem = "if" then
  begin
  Scan;
  expr1:-Expression;
  if Nextitem=/"then" then Error("Syntax error: then expected");
  Scan;
  expr2:-Simpexpr;
  if Nextitem=/"else" then Error("Syntax error: else expected");
  Scan;
  expr3:-Expression;
  Expression:- ifthenelse(expr1,expr2,expr3);
  end

  else
    Expression:-Simpexpr;
  end;

comment ---------------------------------------------------;

ref(expr) procedure EQUATION;
  begin
  ref(expr) express;

  express:- Expression;
  if Nextitem =/ "=" then Error("Syntax error: = expected");
  Scan;
  Equation:-express.Equal(Expression);
  end;

comment ---------------------------------------------------;


class EXPR;
  virtual: procedure infix, traverse, Linj;
  ref(expr) procedure Add, Deriv;
  begin
  comment  The procedures Add, Sub, Mult, Div, Power and Equal
      are used to build the syntax tree of an expression. ;
  ref(expr) procedure Add(y); ref(expr) y;
    comment  Simplification rules:
      x+Ø=x, Ø+y=y, x+(-y)=x-y, -x+y=y-x ;
    Add:-if y==zero then this expr else
         if this expr==zero then y else
         if y is Minusop then this expr.Sub(y qua Minusop.
           express) else
         if this expr is Minusop then
           y.Sub(this expr qua Minusop.express) else
           new addop(this expr,y);

  ref(expr) procedure Sub(y); ref(expr) y;
    comment  Simplification rules:
      x-Ø=x, Ø-y=-y, x-(-y)=x+y ;
    Sub:-if y==zero then this expr else
         if this expr==zero then Minus(y) else
         if y is Minusop then this expr.Add(y qua Minusop.
           express) else
```

```
          new Subop(this expr,y);

    ref(expr) procedure Mult(y); ref(expr) y;
      comment  Simplification rules:
        x*Ø=Ø, Ø*y=Ø, x*1=x, 1*y=y, x*(-y)=-x*y, (-x)*y=-x*y ;
      Mult:-if y==zero or this expr==zero then zero else
            if y==one then this expr else
            if this expr==one then y else
            if y is Minusop then Minus(this expr.Mult
              (y qua Minusop.express)) else
            if this expr is Minusop then Minus
              (this expr qua Minusop.express.Mult(y)) else
            new Multop(this expr,y);

    ref(expr) procedure Div(y); ref(expr) y;
      comment  Simplification rules: Ø/y=Ø, x/1=x,
        (-x)/(-y)=x/y, (x-y)/(-z)=(y-x)/z, x/(y/z)=x*z/y ;
      Div:-if this expr==zero then zero else
           if y==one then this expr else
           if y is Minusop and this expr is Minusop then
             this expr qua Minusop.express.Div(y qua Minusop.
               express) else
           if y is Minusop and this expr is Subop then
             this expr qua Subop.expr2.Sub(this expr qua Subop.
             expr1).Div(y qua Minusop.express) else
           if y is Divop then this expr.Mult(y qua Divop.expr2).
             Div(y qua Divop.expr1) else
           new Divop(this expr,y);

    ref(expr) procedure Power(y); ref(expr) y;
      comment  Simplification rules:
        x**Ø=1, x**1=x ;
      Power:-if y==zero then one else
             if y==one then this expr else
             new Powerop(this expr,y);

    ref(expr) procedure Equal(y); ref(expr) y;
      equal:-new equalop(this expr,y);

    procedure leftpar(pri,priority); integer pri,priority;
      if pri<priority then outtext("(");

    procedure rightpar(pri,priority); integer pri,priority;
      if pri<priority then outtext(")");

    end - EXPR -;

  comment ---------------------------------------------------;

  ref(expr) procedure MINUS(x); ref(expr) x;
    comment  Simplification rules:
      -Ø=Ø, -(-x)=x, -(x-y)=y-x ;
    Minus:-if x==zero then zero else
           if x is Minusop then x qua Minusop.express else
           if x is Subop then x qua Subop.expr2.
             Sub(x qua Subop.expr1) else new Minusop(x);

  comment ---------------------------------------------------;
```

```
ref(expr) procedure IFTHENELSE(x,y,z);
  comment  Simplification rule:
    if x then y else y = y ;
  ref(expr) x, y, z;
  ifthenelse:-if y==z then y else
              new ifthenelseop(x, y, z);

comment --------------------------------------------------;

expr class VARIABLENODE(var); integer var;
  begin

  ref(variable) procedure varib;
    comment  Gives a reference to the variable relative to
      Currentsubmodel;
    begin
    ref(submodel) subm;
    if this variablenode is variableref then
      varib:-this variablenode qua variableref.v
    else
      begin
      subm:-if this variablenode is variablespec then
        this variablenode qua variablespec.modspec.specmod(
        Currentsubmodel) else Currentsubmodel;
      varib:-Variables(subm.ivariables+var).eliminated;
      end;
    end;

  procedure infix(pri); integer pri;
    varib.infix;

  procedure traverse;
    begin
    Numequvar:=Numequvar+1;
    Equvar(Numequvar):-varib;
    end;

  procedure Linj(x,h0,h1); name h0,h1; ref(variable) x;
    ref(expr) h0, h1;
    if varib == x then
      begin h0:-zero; h1:-one; end
    else
      begin h0:-this variablenode; h1:-zero; end;

  ref(expr) procedure Deriv(x); ref(variable) x;
    Deriv:-if varib == x then one else zero;

  end - VARIABLENODE - ;

variablenode class VARIABLESPEC(modspec);
  ref(modelspec) modspec;;

variablenode class VARIABLEREF(v); ref(variable) v;;

comment --------------------------------------------------;

expr class FUNCTIONOP(func,firstarg,nargs);
```

```
  text func; ref(argument) firstarg; integer nargs;
  begin

  procedure infix(pri); integer pri;
    begin
    integer i; ref(argument) arg;
    outtext(func);
    outtext("(");
    arg:-firstarg;
    for i:=1 step 1 until nargs do
      begin
      if i>1 then outtext(",");
      arg.express.infix(20);
      arg:-arg.nextarg;
      end;
    outtext(")");
    end;

  procedure traverse;
    begin
    integer i; ref(argument) arg;
    arg:-firstarg;
    for i:=1 step 1 until nargs do
      begin
      arg.express.traverse;
      arg:-arg.nextarg;
      end;
    end;

  procedure Linj(x,h0,h1); name h0,h1; ref(variable) x;
    ref(expr) h0, h1;
    begin
    integer i; ref(argument) arg; ref(expr) f0, f1;
    arg:-firstarg;
    for i:=1 step 1 until nargs do
      begin
      arg.express.Linj(x,f0,f1);
      if f1 =/= zero then nonlinear:=true;
      arg:-arg.nextarg;
      end;
    if not nonlinear then
      begin
      h0:-this functionop;
      h1:-zero;
      end
    else
      begin
      h0:-zero;
      h1:-zero;
      end;
    end;

  end - FUNCTIONOP - ;


class argument(express); ref(expr) express;
  begin
  ref(argument) nextarg;
  end;

comment -----------------------------------------------------;

expr class NUMBERNODE(id,val); value id; text id; real val;
  begin
  procedure infix(pri); integer pri;
    outtext(id);

  procedure traverse; ;

  procedure Linj(x,h0,hl); name h0,hl;
    ref(variable) x; ref(expr) h0,hl;
    begin
    h0:-this numbernode; hl:-zero;
    end;

  ref(expr) procedure Deriv(x); ref(variable) x;
    Deriv:-zero;

  end - NUMBERNODE - ;

comment -----------------------------------------------------;

expr class MINUSOP(express); ref(expr) express;
  begin
  integer priority,pr;

  procedure infix(pri); integer pri;
    begin
    leftpar(pri,6);
    outtext("-");
    express.infix(5);
    rightpar(pri,6);
    end;

  procedure traverse;
    express.traverse;

  procedure Linj(x,h0,hl); name h0,hl; ref(variable) x;
    ref(expr) h0, hl;
    begin
    ref(expr) f0,fl;
    express.Linj(x,f0,fl);
    h0:-Minus(f0);
    hl:-Minus(fl);
    end;

  ref(expr) procedure Deriv(x); ref(variable) x;
    Deriv:-Minus(express.Deriv(x));

  end - MINUSOP - ;

comment -----------------------------------------------------;

expr class IFTHENELSEOP(exprl, expr2, expr3);
  ref(expr) exprl, expr2, expr3;
  begin
  procedure infix(pri); integer pri;
    begin
    leftpar(pri,1Ø);
    outtext("if ");
    expr1.infix(1Ø);
    outtext(" then ");
    expr2.infix(9);
    outtext(" else ");
    expr3.infix(1Ø);
    rightpar(pri,1Ø);
    end;

  procedure traverse;
    begin
    expr1.traverse;
    expr2.traverse;
    expr3.traverse;
    end;

  procedure Linj(x,hØ,h1); name hØ,h1; ref(variable) x;
    ref(expr) hØ, h1;
    begin
    ref(expr) fØ,f1,gØ,g1,bØ,b1;
    expr1.Linj(x,bØ,b1);
    expr2.Linj(x,fØ,f1);
    expr3.Linj(x,gØ,g1);
    if b1 == zero then
      begin
      hØ:-ifthenelse(bØ,fØ,gØ);
      h1:-ifthenelse(bØ,f1,g1);
      end
    else
      begin
      nonlinear:=true;
      hØ:-zero;
      h1:-zero;
      end;
    end;

  ref(expr) procedure Deriv(x); ref(variable) x;
    Deriv:-ifthenelse(expr1,expr2.Deriv(x),expr3.Deriv(x));

  end - IFTHENELSEOP - ;

comment ---------------------------------------------------;


expr class BINARYNODE(expr1, expr2); ref(expr) expr1, expr2;
  begin

  procedure traverse;
    begin
    expr1.traverse;
    expr2.traverse;
    end;
  procedure infixl(pri,priority1,op,priority2);
    integer pri,priority1,priority2; text op;
    begin
    leftpar(pri,priority1);
    expr1.infix(priority1);
    outtext(op);
    expr2.infix(priority2);
    rightpar(pri,priority1);
    end;

  end - BINARYNODE - ;

comment --------------------------------------------------------;

binarynode class ADDOP;
  begin

  procedure infix(pri); integer pri;
    infixl(pri,8," + ",8);

  procedure Linj(x,h0,h1); name h0,h1; ref(variable) x;
    ref(expr) h0, h1;
    begin
    ref(expr) f0,f1,g0,g1;

    expr1.Linj(x,f0,f1);
    expr2.Linj(x,g0,g1);
    h0:-f0.Add(g0);
    h1:-f1.Add(g1);
    end;

  ref(expr) procedure Deriv(x); ref(variable) x;
    Deriv:-expr1.Deriv(x).Add(expr2.Deriv(x));

  end - ADDOP - ;

binarynode class SUBOP;
  begin

  procedure infix(pri); integer pri;
    infixl(pri,8," - ",7);

  procedure Linj(x,h0,h1); name h0,h1; ref(variable) x;
    ref(expr) h0, h1;
    begin
    ref(expr) f0,f1,g0,g1;

    expr1.Linj(x,f0,f1);
    expr2.Linj(x,g0,g1);
    h0:-f0.Sub(g0);
    h1:-f1.Sub(g1);
    end;

  ref(expr) procedure Deriv(x); ref(variable) x;
    deriv:-expr1.Deriv(x).sub(expr2.Deriv(x));

  end - SUBOP - ;
```

```
binarynode class MULTOP;
  begin

  procedure infix(pri); integer pri;
    infixl(pri,4,"*",4);

  procedure Linj(x,hØ,hl); name hØ,hl; ref(variable) x;
    ref(expr) hØ, hl;
    begin
    ref(expr) fØ,fl,gØ,gl;
    exprl.Linj(x,fØ,fl);
    expr2.Linj(x,gØ,gl);
    if fl == zero then
      begin
      hØ:-fØ.Mult(gØ);
      hl:-fØ.Mult(gl);
      end
    else
    if gl == zero then
      begin
      hØ:-fØ.Mult(gØ);
      hl:-fl.Mult(gØ);
      end
    else
      begin
      nonlinear:=true;
      hØ:-zero;
      hl:-zero;
      end;
    end;

  ref(expr) procedure Deriv(x); ref(variable) x;
    Deriv:-exprl.Deriv(x).Mult(expr2).Add(exprl.
      Mult(expr2.Deriv(x)));

  end - MULTOP - ;

binarynode class DIVOP;
  begin

  procedure infix(pri); integer pri;
    infixl(pri,4,"/",3);

  procedure Linj(x,hØ,hl); name hØ,hl; ref(variable) x;
    ref(expr) hØ, hl;
    begin
    ref(expr) fØ,fl,gØ,gl;
    exprl.Linj(x,fØ,fl);
    expr2.Linj(x,gØ,gl);
    if gl == zero then
      begin
      hØ:-fØ.Div(gØ);
      hl:-fl.Div(gØ);
      end
    else
      begin
      nonlinear:=true;
      hØ:-zero;
```

```
        hl:-zero;
        end;
      end;

    ref(expr) procedure Deriv(x); ref(variable) x;
      Deriv:-exprl.Deriv(x).Div(expr2).Sub(exprl.Mult(expr2.
        Deriv(x)).Div(expr2.Power(two)));

    end - DIVOP - ;

  binarynode class POWEROP;
    begin

    procedure infix(pri); integer pri;
      infixl(pri,2,"**",1);

    procedure Linj(x,hØ,hl); name hØ,hl; ref(variable) x;
      ref(expr) hØ, hl;
      begin
      ref(expr) fØ,fl,gØ,gl;
      exprl.Linj(x,fØ,fl);
      expr2.Linj(x,gØ,gl);
      if fl == zero and gl == zero then
        begin
        hØ:-fØ.Power(gØ);
        hl:-zero;
        end
      else
        begin
        nonlinear:=true;
        hØ:-zero;
        hl:-zero;
        end;
      end;

    ref(expr) procedure Deriv(x); ref(variable) x;
      Deriv:-if expr2 == two then two.Mult(exprl) else
        exprl.Power(expr2).Mult(exprl.Deriv(x).Mult(expr2).
        Div(exprl));
    end - POWEROP - ;

  binarynode class EQUALOP;
    begin

    procedure infix;
      infixl(14,12," = ",12);

    ref(expr) procedure Add(x); ref(expr) x;
      begin
      if not x is minusop then
        exprl:-exprl.Add(x)
      else
        expr2:-expr2.Add(x qua minusop.express);
      Add:-this expr;
      end;

    end - EQUALOP - ;
```

```
comment class PATH, class CONNOP,
  procedure CONNECTIONSTATEMENT.
---------------------------------;


class PATH(identifier); value identifier; text identifier;
  begin
  integer cutnr1, cutnr2;
  ref(modelspec) modspec1, modspec2;

  ref(cut) procedure pathcut1(submod); ref(submodel) submod;
    begin
    ref(submodel) subm;
    subm:-if modspec1 == none then submod
      else modspec1.specmod(submod);
    pathcut1:-Cuts(subm.icuts+cutnr1);
    end;

  ref(cut) procedure pathcut2(submod); ref(submodel) submod;
    begin
    ref(submodel) subm;
    subm:-if modspec2 == none then submod
      else modspec2.specmod(submod);
    pathcut2:-Cuts(subm.icuts+cutnr2);
    end;


  Numpaths:=Numpaths+1;
  Paths(Numpaths):-this path;
  end - PATH - ;

comment ---------------------------------------------------;

class CONNOP;
  begin
  ref(cut) cutspec, pathcut1, pathcut2;

  ref(connop) procedure modconnop(c,c1,c2); ref(cut) c,c1,c2;
    begin
    cutspec:-c;
    pathcut1:-c1;
    pathcut2:-c2;
    modconnop:-this connop;
    end;

  ref(connop) procedure atoper(op2);
    ref(connop) op2;
    if cutspec =/= none and op2.cutspec =/= none then
      begin
      cutspec.atop(op2.cutspec);
      atoper:-modconnop(op2.cutspec,none,none);
      end
    else
      Error("The at operator only operates on cuts");

  ref(connop) procedure toop(op2);
```

```
      ref(connop) op2;
      if pathcutl =/= none and op2.pathcut2 =/= none then
        begin
        pathcut2.atop(op2.pathcutl);
        toop:-modconnop(none,pathcutl,op2.pathcut2);
        end
      else
      if pathcutl == none and op2.pathcutl =/= none then
        begin
        cutspec.atop(op2.pathcutl);
        toop:-modconnop(op2.pathcut2,none,none);
        end
      else
      if pathcutl =/= none and op2.pathcutl == none then
       begin
        pathcut2.atop(op2.cutspec);
        toop:-modconnop(pathcutl,none,none);
        end
      else
        begin
        cutspec.atop(op2.cutspec);
        toop:-modconnop(none,none,none);
        end;

    ref(connop) procedure from(op2);
      ref(connop) op2;
      if pathcutl =/= none and op2.pathcutl =/= none then
        begin
        pathcutl.atop(op2.pathcut2);
        from:-modconnop(none,op2.pathcutl,pathcut2);
        end
      else
      if pathcutl == none and op2.pathcutl =/= none then
        begin
        cutspec.atop(op2.pathcut2);
        from:-modconnop(op2.pathcutl,none,none);
        end
      else
      if pathcutl =/= none and op2.pathcutl == none then
        begin
        pathcutl.atop(op2.cutspec);
        from:-modconnop(pathcut2,none,none);
        end
      else
        begin
        cutspec.atop(op2.cutspec);
        from:-modconnop(none,none,none);
        end;

    ref(connop) procedure par(op2);
      ref(connop) op2;
      if pathcutl =/= none and op2.pathcutl =/= none then
        begin
        pathcutl.atop(op2.pathcutl);
        pathcut2.atop(op2.pathcut2);
        par:-modconnop(none,pathcutl,pathcut2);
        end
      else
      Error("The par operators only operates on paths");

  ref(connop) procedure loop(op2);
    ref(connop) op2;
    if pathcutl =/= none and op2.pathcutl =/= none then
      begin
      pathcutl.atop(op2.pathcut2);
      pathcut2.atop(op2.pathcutl);
      loop:-modconnop(none,pathcutl,pathcut2);
      end
    else
       Error("The loop operator only operates on paths");

  end - CONNOP - ;

comment ---------------------------------------------------;

ref(connop) procedure reversed(op);
  ref(connop) op;
  if op.pathcutl =/= none then
    reversed:-op.modconnop(none,op.pathcut2,op.pathcutl)
  else
    Error("The reversed operator only operates on a path");

comment ---------------------------------------------------;



ref(connop) procedure CONNECTIONOPERAND;
  begin
  integer i;
  ref(submodel) subm; ref(modelspec) modspec;
  ref(connop) op; ref(path) p;

  op:-new connop;
  subm:-Currentsubmodel;
  modspec:-Currentsubmodel.scanmodelspec;

  if modspec =/= none then
    begin
    subm:-modspec.specmod(Currentsubmodel);
    if Nextitem = ":" then
      begin
      Scan;
      if not search(i,subm.ncuts,Cuts(subm.icuts+i).identifier
        = Nextitem) then Error("Not declared cut");
      Scan;
      op.cutspec:-Cuts(subm.icuts+i);
      end
    else
    if Nextitem = ".." then
      begin
      Scan;
      if not search(i,subm.npaths,Paths(subm.ipaths+i).
        identifier = Nextitem ) then Error("Not declared path");
      Scan;
      p:-Paths(subm.ipaths+i);
      op.pathcutl:-p.pathcutl(subm);
```

```
        op.pathcut2:-p.pathcut2(subm);
        end
      else
      if Connectionident =/ notext then
        begin
        if search(i,subm.ncuts,Cuts(subm.icuts+i).identifier =
          Connectionident) then
          op.cutspec:-Cuts(subm.icuts+i);
        if search(i,subm.npaths,Paths(subm.ipaths+i).identifier =
          Connectionident) then
          begin
          p:-Paths(subm.ipaths+i);
          op.pathcut1:-p.pathcut1(subm);
          op.pathcut2:-p.pathcut2(subm);
          end;
        if op.cutspec == none and op.pathcut1 == none then
          Error("Not found cut or path");
        end
      else
        begin
        if subm.maincut=/ Ø then
          op.cutspec:-Cuts(subm.icuts+subm.maincut);
        if subm.mainpath =/ Ø then
          begin
          p:-Paths(subm.ipaths+subm.mainpath);
          op.pathcut1:-Cuts(subm.icuts + p.cutnr1);
          op.pathcut2:-Cuts(subm.icuts + p.cutnr2);
          end;
        if op.cutspec == none and op.pathcut1 == none then
          Error("No main cut or main path declared");
        end;
      end
    else
    if Nexttype = Idtype then
      begin
      if not search(i,subm.ncuts,Cuts(subm.icuts+i).
        identifier = Nextitem) then
        Error("Not declared cut or node");
      Scan;
      op.cutspec:-Cuts(subm.icuts+i);
      end
    else
      Error("Missing connection operand");
    connectionoperand:-op;
    end - CONNECTIONOPERAND - ;

  comment ----------------------------------------------------------;

  ref(connop) procedure CONNECTIONPRIMARY;
    begin

    procedure includecut(h,c,prevsubcut,notcomplete);
      name prevsubcut, notcomplete;
      ref(hierarchicalcut) h; ref(cut) c;
      ref(subcutref) prevsubcut; boolean notcomplete;
      begin
      ref(subcutref) nextsubcut;
    nextsubcut:-new subcutref(c);
    if c == none then notcomplete:=true;
    h.nsubcuts:=h.nsubcuts+1;
    if h.firstcut == none then
      h.firstcut:-nextsubcut
    else
      prevsubcut.nextcut:-nextsubcut;
    prevsubcut:-nextsubcut;
    end;

  if Nextitem =/ "(" then
    connectionprimary:-connectionoperand
  else
    begin
    ref(connop) op;
    Scan;
    op:-connectionexpression;
    if Nextitem = ")" then
      begin
      Scan;
      connectionprimary:-op;
      end
    else
      begin
      ref(hierarchicalcut) h, h1, h2;
      ref(subcutref) prevsubcut, prevsubcut1, prevsubcut2;
      boolean first, nocut, nopath;
      h:-new hierarchicalcut;
      h1:-new hierarchicalcut;
      h2:-new hierarchicalcut;
      Numcuts:=Numcuts-3;

      first:=true;
      while Nextitem =/ ")" do
        begin
        if not first then op:-connectionexpression;
        includecut(h,op.cutspec,prevsubcut,nocut);
        includecut(h1,op.pathcut1,prevsubcut1,nopath);
        includecut(h2,op.pathcut2,prevsubcut2,nopath);

        first:=false;
        if Nextitem = "," then Scan;
        end;
      if nocut then h:-none;
      if nopath then begin h1:-none; h2:-none; end;
      connectionprimary:-op.modconnop(h,h1,h2);
      Scan;
      end;
    end;
  end - CONNECTIONPRIMARY - ;

comment ---------------------------------------------------------;

ref(connop) procedure CONNECTIONSECONDARY;
  if Nextitem = "reversed" or Nextitem = "\" then
    begin
    Scan;
    connectionsecondary:-reversed(connectionprimary);
    end
  else
    connectionsecondary:-connectionprimary;

comment ---------------------------------------------------;

ref(connop) procedure CONNECTIONEXPRESSION;
  begin
  ref(connop) operand;
  boolean continue;

  operand:-connectionsecondary;
  continue:=true;

  while continue do
    if Nextitem = "at" or Nextitem = "=" then
      begin
      Scan;
      operand:-operand.atoper(connectionsecondary);
      end
    else
    if Nextitem = "to" or Nextitem = "-" then
      begin
      Scan;
      operand:-operand.toop(connectionsecondary);
      end
    else
    if Nextitem = "from" then
      begin
      Scan;
      operand:-operand.from(connectionsecondary);
      end
    else
    if Nextitem = "par" or Nextitem = "//" then
      begin
      Scan;
      operand:-operand.par(connectionsecondary);
      end
    else
    if Nextitem = "loop" then
      begin
      Scan;
      operand:-operand.loop(connectionsecondary);
      end
    else
      continue:=false;

  connectionexpression:-operand;
  end - CONNECTIONEXPRESSION - ;

comment ---------------------------------------------------;

procedure CONNECTIONSTATEMENT;
  begin
  Scan;
  if Nextitem = "(" then
    begin
    Scan;
  if Nexttype =/ Idtype then
    Error("Missing cut or path identifier");
  Connectionident:-copy(Nextitem);
  Scan;
  if Nextitem =/ ")" then Error("Missing )");
  Scan;
  end
else
  Connectionident:-notext;

while Nextitem =/ ";" do
  begin
  connectionexpression;
  if Nextitem = "," then Scan;
  end;
Scan;
end - CONNECTIONSTATEMENT - ;
```

```
comment  procedure SCAN, SEARCH, ERROR and COMPILE.
--------------------------------------------------;



procedure SCAN;
comment  Gets next item: identifier, number or delimiter.
  Skips blank lines and comments. Handles the continuation
  symbol -> and the symbols ** .. :: // . The next item is
  put in Nextitem, its type in Nexttype and the value of
  a number in Nextnumb.  ;

  begin
  integer ipos; character ch; boolean skip;
  inspect sysin do
    begin
    skip:=true;
    while skip do
      if not more then
        begin
        if Nextitem=";" or Nextitem==notext or Nextitem=","
          then inimage else skip:=false;
        end
      else
        begin
        ipos:=pos;
        ch:=inchar;
        if ch = ' ' then
        else
        if ch = ';' then
          begin
          if Nextitem = ";" or Nextitem == notext then inimage
          else begin skip:=false; setpos(ipos); end;
          end
        else
        if ch = '{' then
          begin
          ch:=' ';
          while ch =/ '}' do
            begin
            while not more do inimage;
            ch:=inchar;
            end;
          end
        else
        if ch = '-' and more then
          begin
          ch:=inchar;
          while more do if inchar = ' ' then else ch:=' ';
          if ch = '>' then inimage
          else
            begin skip:=false; setpos(ipos); end;
          end

        else
          begin skip:=false; setpos(ipos); end;
        end;
      if not more then
        begin
        Nexttype:=Deltype;
        Nextitem:-copy(";");
        end
      else
        begin
        ipos:=pos;
        ch:=inchar;

        if Digit(ch) then
          begin
          Nexttype:=Numbtype;
          setpos(ipos);
          Nextnumb:=inreal;
          end
        else
        if Letter(ch) then
          begin
          Nexttype:=Idtype;
          while letter(ch) or digit(ch) do
            ch:=if more then inchar else ' ';
          setpos(pos-1);
          end
        else
          begin
          Nexttype:=Deltype;
          if (ch='*' or ch='.' or ch=':' or ch='/') and more
            then
            begin
            if ch=/ inchar then setpos(pos-1);
            end;
          end;
        Nextitem:-image.sub(ipos,pos-ipos);
        end;
      end;
  end - SCAN - ;

comment ---------------------------------------------------;

boolean procedure SEARCH(i,max,cond);
comment  Used to search tables for certain attributes.
  Note that i and cond are called by name. ;
  name i, cond; integer i,max; boolean cond;
  begin
  i:=1;
  while i<=max and not cond do i:=i+1;
  search:=i<=max;
  end - SEARCH - ;

comment ---------------------------------------------------;

procedure ERROR(message); value message; text message;
comment  Outputs error messages together with the current
  input line. Nextitem is underlined. ;
  begin
  integer i;
  outtext(message); outimage;
```

```
   outtext(sysin.image); outimage;
   for i:=1 step 1 until sysin.pos-1-Nextitem.length do
     outchar(' ');
   for i:=1 step 1 until Nextitem.length do outchar('=');
   outimage;
   while Nextitem =/ "end" do Scan; Scan;
   go to command;
   end - ERROR - ;

 comment ---------------------------------------------------;

 procedure COMPILE;
   comment  <model specification> ::= [<model type>]* <model> ;
   begin
   boolean type; text modelid;
   ref(submodel) submod;
   for i:=1 step 1 until Nummodeltypes do Modeltypes(i):-none;
   for i:=1 step 1 until Numvariables do Variables(i):-none;
   for i:=1 step 1 until Numcuts do Cuts(i):-none;
   for i:=1 step 1 until Numpaths do Paths(i):-none;
   for i:=1 step 1 until Numequations do Equations(i):-none;
   Nummodeltypes:=0;
   Numvariables:=0;
   Numcuts:=0;
   Numpaths:=0;
   Numequations:=0;

   type:=true;
   while type do
     if Nextitem = "model" then
       begin
       Scan;
       if Nextitem = "type" then Scan else type:=false;
       if Nexttype =/ Idtype then Error("Missing identifier");
       modelid:-copy(Nextitem);
       Scan;
       if Nextitem =/ ";" then Error("Missing ;");
       Scan;
       submod:-new submodel(modelid);
       submod.modelbody;
       Scan;
       if Nextitem =/ ";" then Error("Missing ;");
       if type then
         begin
         Nummodeltypes:=Nummodeltypes+1;
         Modeltypes(Nummodeltypes):-submod;
         Scan;
         end;
       end
     else
       Error("Missing model");

   Model:-submod.newsubmodel(none,modelid,1);
   for i:=1 step 1 until Numcuts do Cuts(i):-none;
   for i:=1 step 1 until Numpaths do Paths(i):-none;
   Firstconnectnode:-none;
   Prevconnectnode:-none;
   end - COMPILE - ;
```

```
comment  procedure PARTITION
----------------------------;


procedure PARTITION;

  comment Sorts and groups the equations into minimal systems of
    equations that can be solved after each other. The vector
    Equsystems contains references to strongcomp-objects.
    Each such object contains references to all equations in
    a system of equations. ;

begin

class node;
  begin
  integer lowlink, number;
  ref(list) adj, visited;
  end;

class list(n); integer n;
  begin
  integer array vert(Ø:n);
  integer cheapcount;
  integer equnr;
  end;

Boolean procedure Assign(v); ref(node) v;
  comment Associates one of the undetermined variables to
    an equation.
    Reference: T. Wiberg: Permutation of an Unsymmetric Matrix
    to Block triangular form, Dissertation, Department of
    Information Processing, University of Umea, Umea, Sweden.;

  begin
  ref(node) w;
  integer k,max;

    while v.adj.cheapcount < v.adj.n do
      begin
      v.adj.cheapcount:=v.adj.cheapcount+1;
      w:-nodes(v.adj.vert(v.adj.cheapcount));
      if w.adj == none then
        begin
        Assign:=true;
        w.adj:-v.adj;
        go to return;
        end;
      end;
  v.visited:-zero.adj;
  max:=v.adj.n;
  for k:=1 step 1 until max do
    begin
    w:-nodes(v.adj.vert(k));
    if w.visited =/= zero.adj then
      begin
```

```
          if Assign(w) then
            begin
            w.adj:-v.adj;
            Assign:=true;
            go to return;
            end;
          end;
      end;
      Assign:=false;
  return:
    end;

  procedure Strongconnect(v); ref(node) v;

    comment Finds minimal systems of equations that have to be
      solved simultaneously.
      Reference: R.E. Tarjan: Depth first search and linear graph
      algorithms, SIAM J. Comp, 1, 1972, pp. 146 - 160. ;

    begin
    ref(node) w;
    integer k,max,ncomp;

    v.lowlink:=v.number:=nextnode:=nextnode + 1;

    stackpoint:=stackpoint+1;
    stack(stackpoint):-v;

    max:=v.adj.n;
    for k:=1 step 1 until max do
      begin
      w:-nodes(v.adj.vert(k));
      if w.number=0 then
        begin
        Strongconnect(w);
        v.lowlink:=min(v.lowlink,w.lowlink);
        end
      else
        if w.number < v.number then
            v.lowlink:=min(v.lowlink,w.number);
      end;

    if v.lowlink=v.number then
      begin
      ncomp:=0;
      while v=/=stack(stackpoint+1) do
        begin
        stack(stackpoint).number:=n+1;
        stackpoint:=stackpoint-1;
        ncomp:=ncomp+1;
        end;
      Numequsystems:=Numequsystems+1;
      Equsystems(Numequsystems):- new strongcomp(ncomp);
      for k:=1 step 1 until ncomp do
        Equsystems(Numequsystems).equ(k):-Equations
          (stack(stackpoint+k).adj.equnr);
      end;
  end;


Boolean singular, heading;
integer n,nextnode,stackpoint;
integer i, j, k;
integer nunknown, nnontrivial;
ref(node) array stack(1:200);
ref(node) zero;
ref(node) array nodes(1:500);
ref(list) adjlist;
integer equnr;
ref(variable) var;

  if Eliminate then
    begin
    for i:=1 step 1 until Numvariables do
      Variables(i).alias:-none;
    for i:=1 step 1 until Numequations do Equations(i).elimin;
    end;

  nunknown:=0;
  for i:=1 step 1 until Numvariables do
    if not Variables(i).determined then
      begin
      nunknown:=nunknown+1;
      variables(i).equnr:=nunknown;
      end;

    if Eliminate then
      begin
      nnontrivial:=0;
      for i:=1 step 1 until Numequations do
        if not Equations(i).trivial then
          nnontrivial:=nnontrivial+1;
      end
    else
      nnontrivial:=Numequations;

  if nnontrivial =/ nunknown then
    begin
    outtext("The number of equations is ");
    outint(Numequations,4); outimage;
    outtext("The number of unknown variables is ");
    outint(nunknown,4); outimage;
    end;

  n:=max(nnontrivial,nunknown);
  zero:-new node;
  for i:=1 step 1 until n do nodes(i):-new node;

  comment  For all equations, generate a list of undetermined
    variables and assign one of them. ;

  nnontrivial:=0;
  for i:=1 step 1 until Numequations do
    begin
```

```
      nnontrivial:=nnontrivial+1;
      if not (Eliminate and Equations(i).trivial) then
        begin
        Equations(i).traverse;
        nunknown:=0;
        for j:=1 step 1 until Numequvar do
          if not Equvar(j).known then nunknown:=nunknown+1;
        adjlist:-new list(nunknown);
        inspect adjlist do
          begin
          n:=0;
          for j:=1 step 1 until Numequvar do
            if not Equvar(j).known then
              begin
              if not search(k,n,vert(k)=Equvar(j).equnr) then
                begin
                n:=n+1;
                vert(n):=Equvar(j).equnr;
                end;
              end;
          end;
        adjlist.equnr:=nnontrivial;
        Equations(i).varnr:=0;

        zero.adj:-adjlist;
        if not Assign(zero) then singular:=true;
        end;
      end;

    if singular then
      begin outtext("Singular problem"); outimage; end;

    comment  Store the result of the assignment as a coupling
      between equation nodes and variable nodes. ;

    heading:=true;
    for i:=1 step 1 until Numvariables do
      if not Variables(i).determined then
        begin
        var:-Variables(i);
        adjlist:-nodes(var.equnr).adj;
        if adjlist =/= none then
          begin
          equnr:=adjlist.equnr;
          var.equnr:=equnr;
          Equations(equnr).varnr:=i;
          end
        else
          begin
          nodes(var.equnr).number:=n+1;
          if heading then
            begin
            outimage;
            outtext("Unassigned variables:"); outimage;
            heading:=false;
            end;
          var.infix; outimage;
          end;
      end;

  heading:=true;
  Lastsubmodel:-none;
  for i:=1 step 1 until Numequations do
    if Equations(i).varnr=Ø
      and not (Eliminate and Equations(i).trivial) then
      begin
      if heading then
        begin
        outimage;
        outtext("Redundant equations:"); outimage;
        heading:=false;
        end;
      Equations(i).infix(notext);
      end;

 comment  Find the partitioning of the equations into smaller
  systems of equations. ;

  Numequsystems:=Ø;
  nextnode:=Ø;
  stackpoint:=Ø;
  for i:=1 step 1 until n do
    if nodes(i).number=Ø then
      Strongconnect(nodes(i));
end;

comment ----------------------------------------------------;

class STRONGCOMP(n); integer n;
  begin
  ref(equationnode) array equ(1:n);
  end;
```

```
comment INTERACTION
-------------------;



boolean stop, known, no;
integer j;
ref(equationnode) equ, solvedequ;
ref(expr) solution;
ref(modelspec) modspec;
ref(submodel) subm;

Idtype:=1; Deltype:=2; Numbtype:=3;

zero:-new numbernode("0",0);
one:-new numbernode("1",1);
two:-new numbernode("2",2);
solvedequ:-new equationnode(none,none);
Numequations:=0;

stop:=false;
command:
while not stop do
  begin
  inimage; Nextitem:-notext;
  Scan;
  Currentsubmodel:-Model;
  Lastsubmodel:-none;
  if Nextitem = "model" then
    compile

  else
  if Nextitem = "print" then
    begin
    Scan;
    if Nextitem = "variables" then
      begin
      outimage;
      for i:=1 step 1 until Numvariables do
        begin Variables(i).infix; outimage; end;
      outimage;
      end
    else
    if Nextitem = "equations" then
      begin
      outimage;
      for i:=1 step 1 until Numequations do
        Equations(i).infix(" ");
      outimage;
      end
    else
    if Nextitem = "known" or Nextitem = "unknown" then
      begin
      known:=Nextitem = "known";
      outimage;
      for i:=1 step 1 until Numvariables do
        if (known and Variables(i).known) or
```

```
          (not known and not Variables(i).known) then
            begin Variables(i).infix; outimage; end;
        outimage;
        end
      else
      if Nextitem = "sorted" then
        begin
        outimage;
        for i:=1 step 1 until Numequsystems do
          if Equsystems(i).n = 1 then
            Equsystems(i).equ(1).infix(" ")
          else
            begin
            outimage;
            for j:=1 step 1 until Equsystems(i).n do
              Equsystems(i).equ(j).infix("-");
            outimage;
            end;
        outimage;
        end
      else
      if Nextitem="solved" then
        begin
        outimage;
        for i:=1 step 1 until Numequsystems do
          if Equsystems(i).n = 1 then
            begin
            equ:-Equsystems(i).equ(1);
            if not Eliminate or not equ.trivial then
              begin
              nonlinear:=false;
              solution:-equ.solve;
              if not nonlinear then
                begin
                solvedequ.express:-solution;
                solvedequ.submod:-equ.submod;
                equ:-solvedequ;
                end;
              end;
            equ.infix(" ");
            end
          else
            begin
            outimage;
            for j:=1 step 1 until Equsystems(i).n do
              Equsystems(i).equ(j).infix("-");
            outimage;
            end;
        outimage;
        end
      else
        begin
        outtext("Bad argument"); outimage;
        end;
      end

    else
    if Nextitem = "known" or Nextitem = "unknown" then
```

```
      begin
      known:=Nextitem = "known";
      Scan;
      while Nextitem =/ ";" do
        begin
        modspec:-Model.scanmodelspec;
        if modspec =/= none then
          begin
          if Nextitem =/ "." then
            begin outtext("Missing ."); outimage;
              go to command; end;
          Scan;
          subm:-modspec.specmod(Model);
          end
        else
          subm:-Model;
        if not search(i,subm.nvariables,
          Variables(subm.ivariables+i).identifier=Nextitem) then
          begin outtext("Not declared variable"); outimage;
          go to command; end;
        Variables(subm.ivariables+i).known:=known;
        Scan;
        end;
      end
    else
    if Nextitem = "do" then
      begin
      Scan;
      no:=Nextitem = "not";
      if no then Scan;
      if Nextitem = "eliminate" then
        Eliminate:=not no
      else
        begin outtext("Bad argument"); outimage; end;
      end
    else
    if Nextitem = "partition" then
      Partition
    else

    if Nextitem = "stop" then
      stop:=true
    else
      begin
      outtext("Invalid command"); outimage;
      end;
    end;

  end
```

INDEX REGISTER