DYMOLA - A Structured Model Language for Large Continuous Systems

Elmqvist, Hilding

1979

Document Version:
Early version, also known as pre-print

Summer Computer Simulation Conference,
Toronto, Canada, July 1979.

DYMOLA - A STRUCTURED MODEL LANGUAGE FOR LARGE CONTINUOUS SYSTEMS

Hilding Elmqvist

Department of Automatic Control
Lund Institute of Technology
Lund, Sweden

ABSTRACT

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

INTRODUCTION

A common principle for solving large problems is *decomposition* of the problem into a set of smaller subproblems which are either solved directly or decomposed further. The original problem is then solved by combining the subproblem solutions.

This principle is used when modelling large systems. The system is considered as a set of subsystems. This decomposition is often inherent in the physical system. An industrial plant, for example, is in fact designed according to the decomposition principle. The language to describe the model should reflect this and encourage the use of *submodels*. The languages of CSSL-type have a MACRO-concept which corresponds to submodels.

A model must also contain a description of how submodels interact with each other. The introduction of submodels are often done in a way that the interactions between submodels are rather limited. The interaction is often restricted to a set of *connection mechanisms*. Such connection mechanisms often correspond to some physical devices such as shafts, pipes,electrical wires etc.

In the languages of CSSL-type there are no constructs corresponding to such connection mechanisms. The connections are done by means of variables. Each

macro has formal input and output variables. The connection of two submodels is done by having the same variable appear as actual variable in both of the corresponding macro calls. This way of describing the connections between submodels tends to hide the connection structure of the model. One reason is that the details of the connection mechanisms, such as the variables involved, are considered at the same time. A model language ought to have a means of defining abstract connection mechanisms and a means of describing the connection structure in a natural way.

The fundamental way of describing submodels is by equations. Physical laws are formulated as for example mass and energy balances and phenomenological equations. In CSSL such equations must be entered as assignment statements which, after automatic sorting, give an algorithm for assignment of derivatives and auxiliary variables.

This paper describes a model language called DYMOLA (Dynamic-Modelling Language). It is designed to overcome some of the drawbacks with languages of CSSL-type.

Dymola has a hierarchical submodel concept. Abstract connection mechanisms are introduced by means of the concept *cut* which defines the variables associated with each connection. The connection structure of the model is described by a *connection statement*. The model equations are entered in their original form. They need not be converted to assigment statements.

The compiler translates the connection statements to equations. The equations obtained can then be used for different calculations such as simulation or static design calculations. The user specifies which variables are considered known. The equations are automatically sorted and transformed to assignment statements to get an algorithm that assigns the unknown variables. Systems of equations that have to be solved simultaneously are then detected. The assignment statements are obtained by automatic formula manipulation.

The complete definition of the Dymola language can be found in Elmqvist [1].

SUBMODELS

A model can be defined hierarchically according to the following pattern.

```
model <model ident>
  declaration of submodels
  declaration of variables and connection mechanisms
  equations and connection statements
end
```

If several subsystems have the same model, their descriptions do not have to be duplicated. It is possible to define a model type with the same structure as model . Such a model type can then be duplicated with a submodel -statement.

VARIABLES AND EQUATIONS

The following types of variables can be declared in a model: parameter, constant, local, terminal, input and output (see the syntax in the appendix).

The terminal variables decribe the interdependence between a submodel and its environment. The input and output variables are special cases of terminal variables. They are introduced to make it possible to indicate causalities in the model.

Some types of variables can be referenced from the outside of a submodel using a dot-notation.

The equations of the submodels have the form

```
<expression> = <expression>
```

where <expression> is defined as for Algol-60. An equation can thus contain the if-then-else construction. It is also possible to call functions and procedures written in some algorithmic language. Derivatives are denoted by x', x'' or der(x), der2(x) etc.

CUTS AND CONNECTIONS

When a subsystem is considered, its boundaries are first determined. Such boundaries are 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*. An example from rigid body mechanics is the necessity of introducing reaction forces as cut variables when a part of the rigid body is considered.

When connecting submodels, it is natural to view a submodel in the same way as the corresponding subsystem. One then wants to work with, for example, the physical mechanisms that connect the subsystems. Certain variables are associated with each mechanism. Examples of such mechanisms and their associated variables are:

shaft: angle, torque
pipe: flow-rate, pressure, temperature
electrical wire: 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:

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

The notation [ ]*means that what are inside the brackets may be repeated none or more times. See the appendix for the complete syntax notation used.

Submodels can be connected by using the cuts and an at -*operator* in a *connection statement.*

```
connect <model ident>:<cut ident>
  {at <model ident>:<cut ident>}*
```

All connection statements are translated to equations. The two lists of variables in the cut declaration is then treated in different ways. The variables in the first list are called *across variables.* When two cuts are connected by the at-operator, equations are generated with equality between corresponding across variables in the two cuts. The variables in the second list in the cut declaration are called *through variables.* When a set of cuts are connected by at-operators, equations which sums corresponding through variables in all cuts to zero are generated.

The introduction of through variables is motivated by the fact that the connection of a set of submodels often implies balance equations of the type:

$$v_1 + v_2 + ... = 0$$

This is for example the case with electrical currents, flows, forces and torques. These equations are often mass or energy balances for an artificial model surrounding the connection joint. The introduction of through variables actually eliminates the need for describing such an artificial submodel. However, if there are more complex relations than equalities and sums equal to zero, such relations can be introduced in a submodel surrounding the connection joint.

Example

Consider the connection of three capacitors as shown in fig 1.

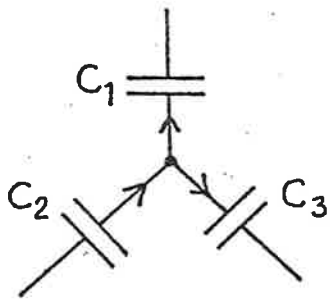

Fig 1. Part of electrical network

A model type capacitor can be described as follows.

```
model type capacitor
  cut wire1(V1 / I)  wire2(V2 / -I)
  local V
  parameter C
  V = Va - Vb
  C*der(V) = I
end
```

The duplication of the model type capacitor and the connection can then be done as shown below.

```
model network
  submodel (capacitor) C1 C2 C3
  . . .
  connect C1:wire1 at C2:wire2 at C3:wire1
  . . .
end
```

The reference direction for currents has, in this example, been chosen as flowing into the components. If a through variable has the opposite direction it is preceeded by a minus sign in the cut declaration. This has been done in the cut wire2.

The following equations are generated from the connect statement:

```
C1.V1 = C2.V2
C2.V2 = C3.V1
C1.I + C3.I = C2.I
```

■

The points where several connection mechanisms are joined are usually called *nodes*. It is sometimes convenient to use nodes in the connection statements. They

can be declared as follows.

```
node <node ident> [(<variable cut>)]
```

It is possible to build hierarchical cuts consisting of other cuts as shown below.

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

All at-operators on cuts and nodes are translated to equations involving the variables in the cuts. An at-operation on two hierarchical cuts are defined as at-operations on the corresponding subcuts. The following example illustrates the rules that govern the generation of equations.

Example

Consider the model structure in fig 2.

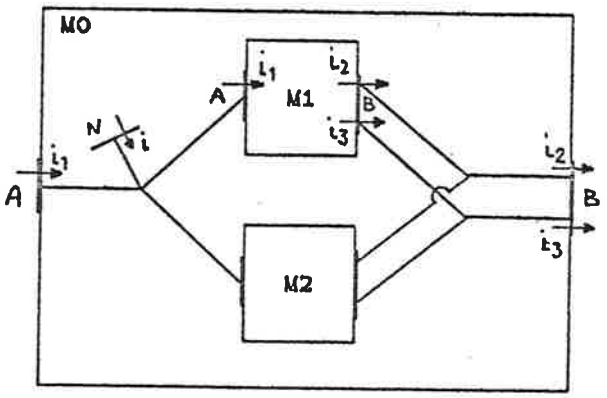

Fig 2. A hierarchical model structure.

The corresponding cut declarations and connection statements are shown below.

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

model MO
  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
```

MODEL STRUCTURE

The previous sections have shown how the relations between variables in different submodels can be given either by equations using the dot-notation or by using cuts and the at-operator. The at-operator allows models to be connected in arbitrary structures. However, the connection statements do not contain the structure of the model themselves. This section gives an alternative way to describe the structure of a model.

The description of the connections can be made according to different principles. One possibility is to concentrate the description around each *connection mechanism* and indicate which submodels that are connected by it. This principle corresponds to the use of the at-operator.

Another possible method is to concentrate the description around *submodels*. A set of nodes are introduced and for each submodel it is indicated to which nodes it is connected. This is the standard method used in analysis programs for electrical networks. In Dymola this kind of description is done by defining a hierarchical cut containing all the cuts of the submodel. Such a hierarchical cut is then connected, using the at-operator, to a corresponding hierarchical node in a surrounding model.

A third philosophy for description of the connections will now be given. It is based on the fact that the connections introduce a natural *grouping of the submodels*. It is for example often natural to say that a set of submodels are connected in series or in paralell in some respect. It is then assumed that the submodels can be considered as having 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. A direction can also be defined in accordance with the perception of causalities in a system. When a system has no inherent directions, they will be imposed by the choice of reference directions for variables.

As directions are defined in a model it is natural to introduce the concept *path*. A path exists between two cuts and is declared in the following way.

```
path <path ident> < <cut> - <cut> >
```

The direction is defined from the first cut to the second. Several paths can be declared in a submodel corresponding to different kinds of connection mechanisms.

A natural way of describing the connections of a set of submodels is to state how the paths of the submodels are joined by connection mechanisms. This is done with the connection statement using a set of connection operators: at, to, from, par, loop, branch, join and reversed. The complete syntax for the connection statement can be found in the appendix. An example of a connection statement is:

```
connect G to E to R1 to (C par (R2 to R3) ) to G
```

The interpretation of a connection statement 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 is done from left to right if not otherwise stated by parentheses. The only exception is the unary operator reversed which has higher priority than the others.

Table 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.

In order to shorten the connection statements, a reference to a cut or a path can be made by using only the model identifier. If the connection statement contains an identifier within parentheses after connect this identifier is used to specify the cuts and paths. It is also possible to complete the reference by declaring one path and one cut as main in each submodel.

Table 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
C1 to C2	none	C1 at C2
4. <C1 - C2> from <C3 - C4>	<C3 - C2>	C4 at C1
C1 from <C2 - C3>	C2	C3 at C1
<C1 - C2> from C3	C2	C3 at C1
C1 from C2	none	C2 at C1
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

Consider the electrical network in fig 3.

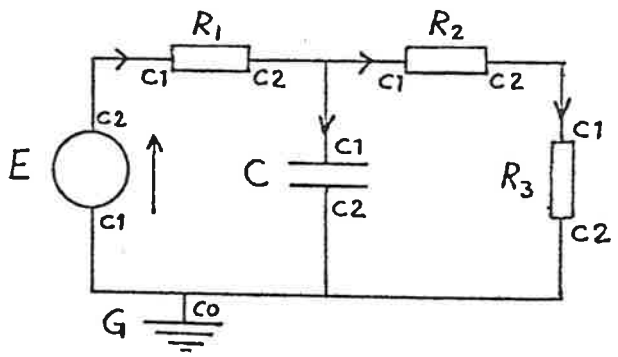

Fig 3. Electrical network

The system can for example be described with the following connection statement.

```
connect G to E to R1 to (C par (R2 to R3) ) to G
```

In order to explain how this statement is evaluated, assume that the submodel G has the cut declaration

```
main cut CO(...)
```

and the other submodels have the path declaration

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

The connection statement could then be represented as

```
G:CO to <E:C1 - E:C2> to <R1:C1 - R1:C2> to (<C:C1 - C:C2> par
  (<R2:C1 - R2:C2> to <R3:C1 - R3:C2>) ) to G:CO
```

The steps performed when evaluating this expression are given below.

1. Effect `G:CO at E:C1`
   Result `E:C2 to <R1:C1 - R1:C2> to (<C:C1 - C:C2> par (<R2:C1 - R2:C2> to <R3:C1 - R3:C2>) ) to G:CO`

2. Effect `E:C2 at R1:C1`
   Result `R1:C2 to (<C:C1 - C:C2> par (<R2:C1 - R2:C2> to <R3:C1 - R3:C2>) ) to G:CO`

3. Effect `R2:C2 at R3:C1`
   Result `R1:C2 to (<C:C1 - C:C2> par <R2:C1 - R3:C2>) to G:CO`

4. Effect `C:C1 at R2:C1, C:C2 at R3:C2`
   Result `R1:C2 to <C:C1 - C:C2> to G:CO`

5. Effect `R1:C2 at C:C1`
   Result `C:C2 to G:CO`

6. Effect `C:C2 at G:CO`
   Result `none`

MANIPULATION OF EQUATIONS

The manipulation of the equations is described in Elmqvist [1], [2]. Only a brief survey is given here.

The connection statements are translated to equations by the compiler. This means that the model then consists of only equations. They can formally be

written as

$$f(t, \dot{x}, x, z, p) = 0$$

where t is time, x is a vector of state variables, z is a vector of auxiliary variables and p is a vector of parameters.

In order to use an explicit integration algorithm, $\dot{x}$ and z should be solved for. The model equations often have special characteristics. All variables are not present in each equation. The Jacobian of f with respect to $\dot{x}$ and z thus contains many zero elements, i.e. it is sparse. This means that the solution could be obtained more efficiently by partitioning the system of equations into a set of smaller systems of equations. In fact, many of the systems of equations will be scalar.

The algorithm for partitioning uses only structural information, i.e. whether a variable is present in an equation or not. The first problem is to determine which variable to solve for in each equation. It is called finding an assignment. The next step is to find a partitioning of the equations into minimal systems of equations that must be solved simultaneously and to sort them for correct computational order. This is called finding the strong components of the associated bipartite graph.

The total effect of these two algorithms is finding two permutation matrices operating on the columns and the rows of the Jacobian and making it block triangular.

When this procedure has been performed it is thus known in which order the equations should be solved and which variables to solve for. The blocks correspond to systems of equations that must be solved simultaneously.

The equations are then converted to assignment statements for the unknown variable. This is done only for equations which are linear in this variable. An important observation for model equations is that there is often such a linear dependence in the simulation case.

One important advantage with automatic manipulation of the equations is that the same basic equations could be used for different calculations such as simulation and static design calculations. The algorithms only have to know which variables are known and which are unknown.

The structural analysis is very useful for the modeller. It gives information about causalities and algebraic loops in the model. This can then be compared with the modeller's perception of these properties in the real system.

IMPLEMENTATION

A compiler for the language and the algorithms for the manipulation of the

equations have been implemented using the programming language Simula. The complete program listing can be found in Elmqvist [1]. The input to the program is a model in Dymola and the output is the sorted and grouped equations in symbolic format. The equations which are not included in a system of equations and which are linear in the unknown variable are output as assignment statements. If it is not possible to find an assignment, the program gives guidance about what is incorrect. A list of variables for which no equations could be associated is given as well as a list of equations for which no variables could be associated (redundant equations).

EXAMPLE

The Dymola language is suitable for description of large dynamical models. It has been used for several nontrivial models such as electrical network, mechanical system, electrical power distribution and thermal power plant. These models are listed in [1] together with the output from the program.

Some parts of a thermal power plant are described here. The original model was developed by Lindahl [3]. The structure of the system is shown in fig 4. Several of the submodels are further decomposed. The drum system consists of drum, risers and down-comers. The intermediate and low pressure turbines (IPturb, LPturb) have 4 and 3 turbine sections respectivly. The two preheaters for feed water (preh1, preh2) are built up by preheater sections. Many of the subsystems have identical models with different parameters. There are three submodels of model type superheater (superh1, superh2, superh3) and two attemperators (attemp1, attemp2). There are totally 8 turbine sections and 7 preheater sections.

There are six different kinds of flows in the system (heat transfer, steam, extract steam, feed water, condensate and mechanical power). Each of these corresponds to path declarations and a connection statement. The flows through pipes are characterized by three variables (mass flow, enthalphy and pressure).

The model uses a set of functions (THP, ISEN, ...) to obtain thermodynamical relations by interpolation in the Moliere diagram.

Below is given the model for a turbine section. Comments are given within { }.

```
model type turbsection

{ Variable prefixes:
  W - mass flow, H - entalphy, P - pressure
  N - mechanical power, T - temperature }

cut insteam (W1 H1 P1)  outsteam (W2 H2 P2)
path steam < insteam - outsteam >
```

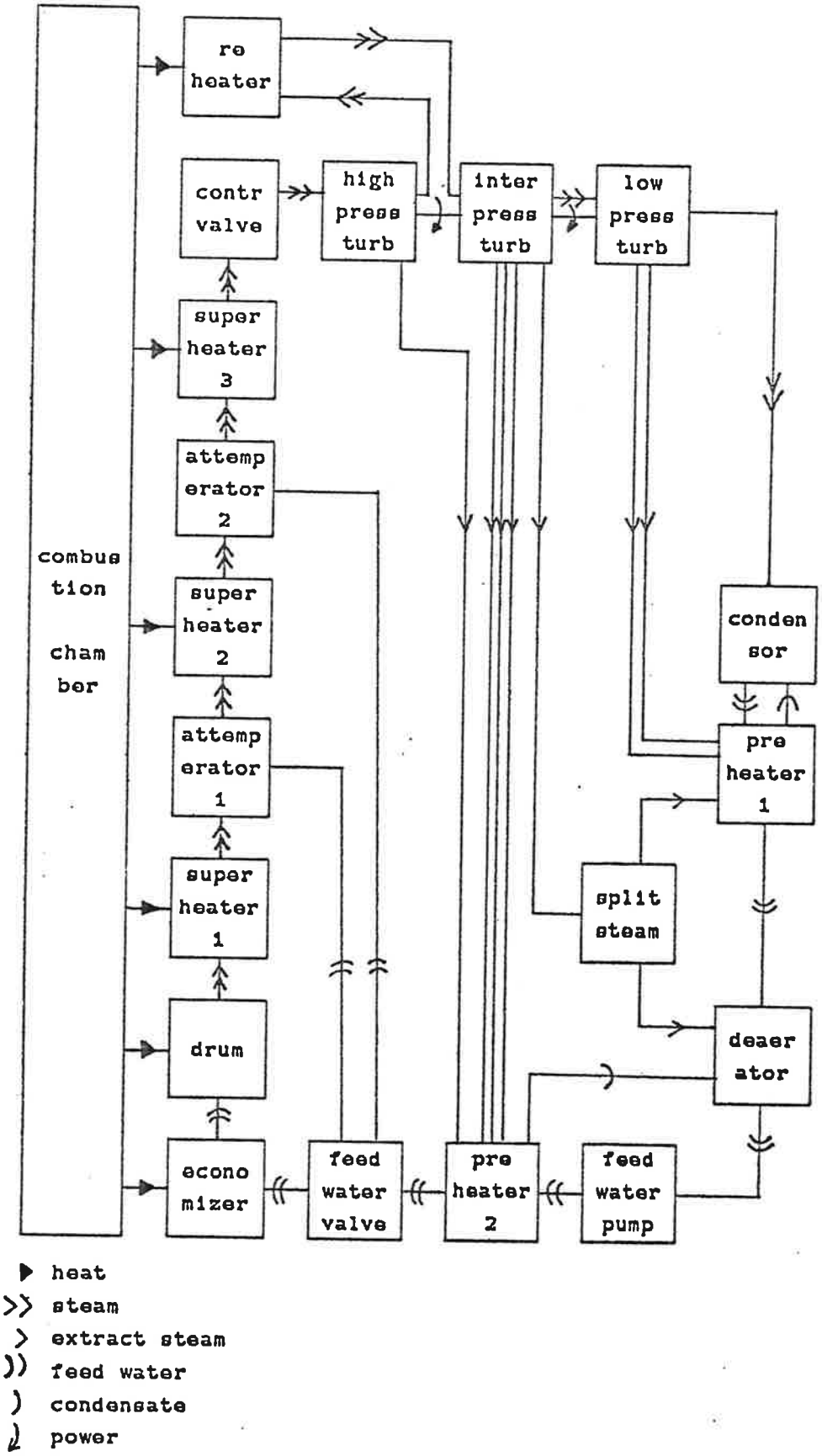

Fig 4. A thermal power plant

```
cut extract (Wp H2 Pp)
cut inpower (N1)  outpower (N2)
path power < inpower - outpower >
input S  {to valve for extractsteam}
local H T2 ap
parameter f fp Eh

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

{mass balance}
W1 = W2 + Wp

{energy balance}
N2 = N1 + W1*(H1-H2)

H2 = H + (1-Eh)*(H1-H)
H = ISEN(H1,P1,P2)
T2 = THP(H2,P2)

end
```

The intermediate pressure turbine is built up by four turbine sections as shown in fig 5.

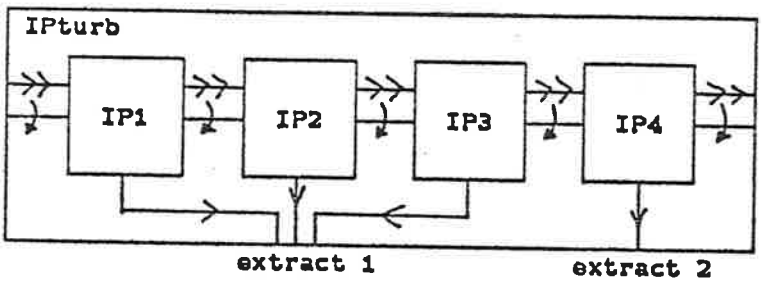

Fig 5. Intermediate pressure turbine.

The description of the IPturb is shown below.

```
model type IPturb

submodel (turbsection) IP1 IP2 IP3 IP4

path steam <IP1:insteam - IP4:outsteam>
cut extract1 [IP1:extract IP2:extract IP3:extract]
cut extract2 [IP4:extract]
```

```
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
```

The overall description of the power station in accordance with fig 4 is shown below.

```
model powerstation

submodel (superheater) superh1 superh2 superh3
submodel (attemperator) attemp1 attemp2
submodel (turbsecton) HPturb
submodel drumsyst reheater controlvalve
submodel IPturb LPturb
submodel condensor splitsteam dearator
submodel preh1 preh2
submodel feedwaterpump feedwatervalve
submodel combchamber 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 (extract) (HPturb IPturb:extract1) to preh2 ->
        IP:extract2 to splitsteam to (dearator preh1:extract2)  ->
        LP to preh1:extract1

connect (feedwater) condensor to preh1 to dearator to   ->
        feedwaterpump to preh2 to feedwatervalve to   ->
        (economizer to drumsyst::drum attemp1 attemp2)

connect (condensate) preh2 to dearator ->
        preh1 to condensor

connect (power) HPturb to IPturb to LPturb

end
```

The total number of equations for this model is about 400. The equations was sorted for the simulation case. Eleven systems of equations was discovered. The largest contained 17 nontrivial equations.

CONCLUSIONS

The Dymola language contains several new constructs that correspond to the way large dynamical models are developed.

A model can be decomposed into a set of submodels. The interaction of a submodel with its environment is often naturally considered as a set of interactions from different other submodels through distinct connection mechanisms. This corresponds to cut declarations. Interactions might be further decomposed by introducing hierarchical cuts. The basic level of defining an interaction is by associating a set of terminal variables with it.

The connection of submodels is often viewed as a block diagram or a graph. The connection statement makes it possible to conveniently describe such diagrams as text. Since directions are often associated with this kind of diagrams, it is natural to introduce the concept of a path.

Because the basic means of describing models is equations, the models can be written in a form that is independent of what calculations they are used for.

ACKNOWLEDGEMENTS

The author wants to thank Professor Karl Johan Åström for his support and guidance during this work.

This research was supported by the Swedish Institute of Applied Mathematics (ITM).

REFERENCES

[1] Elmqvist, H., A Structured Model Language for Large Continuous Systems, Ph.D. Thesis, TFRT-1015, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden (May 1978).

[2] Elmqvist, H., Manipulation of Continuous Models Based on Equations to Assignment Statements, Proc. IMACS Congress 1979 / Simulation of Systems, Sorrento, Italy (September 1979).

[3] Lindahl, S., A Nonlinear Drum Boiler - Turbine Model, Report TFRT-3132, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden (1976).

APPENDIX

Syntax Notation

The following syntax notation is used: | means or, { } groups terms together, [ ] means that a group of terms is optional, { }* means repetition one or more times, [ ]* means repetition none or more times.

To make the syntax smaller, it has purposly been left incomplete in some respects. It does not contain the definitions of basic items such as <identifier> and <number>. Trivial productions such as <model ident>::=<identifier> are omitted. New line is treated as ;. Continuation of a statement on the next line is indicated by -> at the end of the line.

Syntax For The Dymola Language

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

<model>::=model <model ident>; <model body> end
<model type>::=model type <model type ident>; <model body> end
  <model body>::=<submodel part> <declaration part>
          <statement part>

    <submodel part>::=[<model>; | <model type>; |
          <submodel incorp>;]*
      <submodel incorp>::=submodel [(<model type ident> ) ]
          {<model ident> [(<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>}* |
          external {<variable>}* |
          internal {<variable>}*
```

```
<variable>::=<identifier>

<cut declaration>::=[main] cut {<cut ident> [<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 ident>]|
      <cut ident>
    <model spec>::=<model ident>[::<model ident>]*

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

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

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

<equation>::=<expression> = <expression>
  <variable spec>::=[<model spec>.]<variable>
<procedure call>::={<variable spec>}*=
    <procedure ident> ( {<expression>}* )
<connection statement>::=connect [(<ident>)]
    {<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 ident>
```