LUND UNIVERSITY

LICS - Language for Implementation of Control Systems

Elmqvist, Hilding

1985

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

Link to publication

Citation for published version (APA):
Elmqvist, H. (1985). *LICS - Language for Implementation of Control Systems*. (Research Reports TFRT-3179). Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

LICS

Language for Implementation of Control Systems

Hilding Elmqvist

Department of Automatic Control – Lund Institute of

This work has been carried out under the Program in Information Pro
and Computer Science (Ramprogrammet i Informationsbehandling), wh
initiated and sponsored by the National Swedish Board for Technical De
opment (Styrelsen för Teknisk Utveckling). It has been part of the Ce
Industrial Computer Systems (CID) at Lund Institute of Technology.

LICS

Language for Implementation of Control Systems

Hilding Elmqvist

Department of Automatic Control – Lund Institute of Technology

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
FINAL REPORT

Date of issue
December 1985

Document Number
CODEN: LUTFD2/(TFRT-3179)/1–130/(1985)

Author(s)
Hilding Elmqvist

Supervisor
Karl Johan Åström

Sponsoring organisation
Swedish Board for Technical Development (STU) Contract No 89–3962

Title and subtitle
LICS - Language for Implementation of Control Systems

Abstract

This report describes a language and an interactive environment, called LICS, for implementation of control systems and dynamic modelling. It uses graphics for representation of the structural properties: hierarchical decomposition, interconnection structure, and structure of interfaces. The decomposition and interconnection structures are described by block diagrams. Low level modules are described by equations.

Much effort has been devoted to the design of an integrated language both for analog control and logic control. It resulted in powerful module and interconnection concepts that make it possible to develop libraries of modules for different applications. The user can build systems from such libraries without having to deal with details of their implementation. Documentation can be obtained at different levels of detail.

The system is designed with modern graphical scientific personal computers in mind. Joysticks and mouse are used for continuous scroll, pan and zoom of the picture in real-time. A concept of information zooming is introduced. When, for example, zooming in on a module, it changes to a more detailed representation showing the internal structure.

This work has been a part of the Program in Information Processing and Computer Science (Ramprogrammet i Informationsbehandling) which was initiated and sponsored by the National Swedish Board for Technical Development.

Key words
Computer Aided Control Engineering; Man-Machine Interaction; Computer Graphics; System Description; Control; Simulation; Information zooming

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
130

Recipient's notes

Security classification

Table of Contents

1. Introduction 7
2. Concepts and Requirements 9
3. Controller Language 16
4. Application Examples 36
   - 4.1 Control algorithms 36
   - 4.2 Programmable controllers 42
   - 4.3 Modelling 57
5. Graphical Representation of Program Structures 70
6. Interaction 78
7. Implementation 89
8. Conclusions 97
9. Acknowledgements 99
10. References 100

Appendix 103

1. Syntax diagrams for text in modules 103
2. Syntax for LICS files 110
3. Textual description of TankSystem 112
4. Sorted equations for resource allocation 115
5. PowerSystem in Dymola 118

1. Introduction

The structural properties of a system are very important especially when dealing with large systems. These structures are, however, hard to represent in an easily apprehendable way when a purely textual description is used. The interconnection structure of submodules is, for example, much easier described graphically than textually. The current trend on scientific personal computers indicates that graphical displays will be commonly available. This will revolutionize man-machine interaction. There is, in particular, a great opportunity to invent much more user-friendly problem oriented languages which incorporates graphics naturally.

This report describes a language and an interactive environment, called LICS, for implementation of control systems and dynamic modelling. It uses graphics for representation of the structural properties: hierarchical decomposition, interconnection structure, and structure of interfaces. The decomposition and interconnection structures are described by block diagrams. Low level modules are described by equations.

Much effort has been devoted to the design of an integrated language both for analog control and logic control. It resulted in powerful module and interconnection concepts that make it possible to develop libraries of modules for different applications. The user can build systems from such libraries without having to deal with details of their implementation. Documentation can be obtained at different levels of detail.

The system is designed with modern graphical scientific personal computers in mind. Joysticks and mouse are used for continuous scroll, pan and zoom of the picture in real-time. A concept of information zooming is introduced. When, for example, zooming in on a module, it changes to a more detailed representation showing the internal structure.

The graphical interface is related to the desk-top model developed at Xerox Parc and popularized by Apple's Macintosh. However, the LICS interaction model assumes and exploits more powerful graphical capabilities. Fast generation of raster pictures and double buffering make animation possible. The user does not have to watch in what order graphical objects appear on the screen. Animation has been used to support the user's mental model of the hierarchical space of the control program.

This report is organized as follows. Chapters 2-4 discuss the language: requirements, design and examples. Chapter 5 and 6 are devoted to the man-machine interaction: a general discussion followed by a description of the inter-

action principles used in LICS. The implementation of LICS is briefly described in Chapter 7. The conclusions are given in Chapter 8. Other documents describing the LICS project are Elmqvist (1982, 1983a, 1983b) and Haggård (1982).

2. Concepts and Requirements

Implementation of a control system for an industrial plant is a large effort. A lot of information is collected and generated in the design process. Ideally all information should be easily accessible at all phases of the work, even when the control system is installed. It is important that the control system can be enhanced by plant personel as more knowledge about the process is gained.

Presently, the documentation and the implementation are often separated and there is a problem of inconsistency. For example, the documentation might include block diagrams showing how different modules are connected together. The programs that implement these functions typically require that this structural information is converted to tabular form.

Process control is often handled by standard regulators. However, there is in many circumstances a need for more flexibility. Examples are nonlinear compensators, feedforward, multi-variable and adaptive control. A simple algorithmic language would be a useful complement to the standard regulators. The logic control involves application dependent programming. It is presently done in low level languages. A unification of notation and languages between regulators and programmable controllers is essential.

Information system

The available information about the control system must be carefully structured in order to be useful. The programming aids should encourage this. Graphical representations are particularly useful to describe structures.

The information system designed aims at large industrial plants. A main issue in the development of a control system is to maintain all the information. An information system is needed which includes information about the plant and its control system such as:

- Plant documentation
- Controller requirements
- Controller documentation
- Controller design
- Implementation
- Run time information

The first three issues deal with arbitrarily structured pieces of text and drawings. The controller design might involve running design and simulation

programs. The dialogues and the plotted results could be stored in the information system. The programs in the control computer are described as formal text and figures. The information system should be connected to the control computers in order to distribute code updates and for presentation of run time information.

One example of an information system for process control applications dealing with the first three issues is the EPOS system (Lauber, 1980). Examples of systems that uses graphics for description of control system implementations are Asea Master (see Asea) and Step5 (Siemens, 1984). A standardization committe within IEC is working on a proposal for a programming language for programmable controllers (IEC, 1985).

Decentralized control system

The control system will in general be distributed, having fast communication links. Communication between modules should be expressed in the same way independently of in what processors the modules reside. The programming environment should include an efficient incremental compilation technique to allow small changes to be made quickly. The run time system in each control processor should allow selective updating of compiled modules. There should be a possibility to specify a "transition algorithm" mapping the old state of the module to the new state representations. This is required in order to allow bumpless replacement of algorithms. It should be possible to display run time information in the graphical representations of the control system.

Abstraction, modularization and coupling

A basic problem with software is how to deal with complexity. The reason for this is the human inability to deal with many entities at the same time. The system must therefore be structured in such a way that it can be viewed from different viewpoints. Each viewpoint should have a fairly small number of *related* entities.

Two basic principles are used to obtain such a structure: *abstraction* and *modularization*. "The essence of abstraction is to extract essential properties while omitting inessential details" (Booch, 1983). Levels of abstraction are defined, allowing the system to be viewed from different viewpoints with increasing detail. The first abstraction level for a system might be just its name or icon and an explanation. The next level might contain a description of usage and a third level details about realization.

The amount of information increases at lower abstraction levels. Modularization must be used in order to maintain useful views with a limited number

of related concepts. Modularization means that the information at one abstraction level is decomposed into smaller entities, modules. The selection of module boundaries is guided by our perception of the problem space.

There is almost always interaction (coupling) between modules. In order to be useful, a decomposition must be chosen in a way that the external interaction complexity is small compared to the internal complexity. Furthermore, the entities in a module should be related (cohesion).

Concepts

The most critical issue in language design for a certain application area is to find the basic concepts and their relations. The language should then include constructs that allow the basic concepts to be modelled. There is a process of unification in order that the language will not include a lot of similar constructs. Instead, more general constructs are sought that models several similar concepts of the application area. This makes the language smaller and easier to learn. It might also give additional power.

A discussion of basic concepts involved in implementation of control systems now follows.

Automatic control

The theory for analysis and synthesis of computerized control systems is based on *sampling* and *difference equations*. The basic theory models the situation that the computer repeatedly performs the following actions

- A/D conversion of input signals, i.e. sampling.
- Evaluation of control algorithms.
- D/A conversion of output signals (the outputs are held constant between sampling instants).
- Wait until next sampling instant.

The internal behaviour of the computer can be modelled by difference equations.

$$\begin{aligned} Y(T_I) &= G(X(T_I), U(T_I), P) \\ X(T_{I+1}) &= F(X(T_I), U(T_I), P) \\ X(T_0) &= X_0 \end{aligned} \tag{2.1}$$

where

T_I time if I:th sampling instant
X vector of stored information between sampling instants
U vector of process inputs

Y vector of process outputs
P vector of operator parameters
X_0 initial value for X.

The *control loop* is a basic concept for analysis, synthesis and modularization. It is concerned with the control strategy for some part of the process with respect to certain variables. It involves the *signal flow* from process measurements through different *computational blocks* to actuators of the process. Associated with each control loop are requirements for the control and maybe also a process model. The coupling between different process variables often results in couplings between the control loops. Examples of computational blocks are

PID-regulators
filters
adaptive controllers
feed-forward blocks
sensor compensators.

This kind of blocks can be found in DDC-packages which are based on a fixed set of *block types*. The application programmer can interactively create *block instances* of these types and connect them together. Since such packages often lacks the possibility to enter arbitrarily equations, there are often block types such as

adders
multipliers
min- and max-selectors.

The computational blocks can be described as *discrete dynamical systems* as follows

$$\begin{aligned} y(t_j) &= g(x(t_j), u(t_j), p) \\ x(t_{j+1}) &= f(x(t_j), u(t_j), p) \\ x(t_0) &= x_0 \end{aligned} \tag{2.2}$$

u, y and x are the vectors of *inputs* , *outputs* and *states* of the subsystem. t_j are the sampling instants of the system. The dynamics of different parts of the controlled process may have different demands on the sampling rates for modules. A thorough understanding of what variables constitute the state is essential in order that additional delays are not introduced.

In *linear* control theory, a dynamical system is represented by

$$
\begin{aligned}
x(t_{j+1}) &= A(p)x(t_j) + B(p)u(t_j) \\
y(t_j) &= C(p)x(t_j) + D(p)u(t_j)
\end{aligned}
\tag{2.3}
$$

where A, B, C and D are real matrices.

Controller synthesis based on linear theory naturally involve *matrix equations*. An example is the updating of the covariance matrix of an recursive least squares identification algorithm

$$
PNew = \frac{P - P * f * Transpose(f) * P}{lambda + Transpose(f) * P * f} \frac{1}{lambda} \tag{2.4}
$$

Cascade coupling of regulators is common. Regulator blocks often have special features for performing *balancing* calculations at start-up and to avoid *integrator wind-up* when limitations occur at the actuators. These actions might involve sending information "up-streams" (reverse to the usual signal flow) to an outer regulator for back-calculation of a state variable such as the integral term.

Programmable controllers

The basic concepts for using programmable controllers are Boolean assignment statements, *timers* and *counters*. Variables in the assignments are either inputs or outputs at some IO-addresses or internal variables. Allocation of such internal variables are typically done manually. If an internal variable is referenced before it is assigned to, it represents a state of the controller. The states are thus defined indirectly by the order of the statements.

Timer- and counter-modules have Boolean inputs and outputs but integer states. A programmable controller can be represented by a discrete dynamical system of the form (2.1). The equations are often executed repeatedly "as fast as possible" with the sampling rate dependent on the length of the program.

The boolean functions are in many cases specified using simple assembly like languages. Alternative representations such as *ladder diagrams* and *function diagrams* are also used (see IEC, 1985).

The programs often have a structure which corresponds to a decomposition of the controlled process. Handshaking signals (request, acknowledge, etc.) are used to synchronize program modules. There are therefore connections with bidirectional data flows.

An important property of programmable controllers is their ability to control *sequences* of actions as well as *interlocking actions*. This can be thought of as changing the set of equations that are currently being executed. However,

in many implementations the information about which actions are active is represented by boolean state variables associated with each action. These mode variables then appear in the expressions to make the effect of an equation conditional.

Parallel and alternative sequences are also introduced as well as looping. Chapter 5 contains more examples on the use of programmable controllers.

Modelling

When developing a model for a physical system one uses fundamental laws such as mass balances, energy balances and phenomenological equations. These are either algebraic equations or *ordinary differential equations* which relate certain variables.

It should be noted that the basic concept is general *equations*, not assignment statements. A model can thus be represented by

$$f(t, \dot{x}, x, p) = 0 \tag{2.5}$$

Many integration algorithms require that the derivatives are solved explicitly, i.e.

$$\dot{x} = F(t, x, p) \tag{2.6}$$

The transformation between (2.5) and (2.6) is not trivial. In many cases it can, however, be performed automatically. Elmqvist (1978, 1979b) discusses transformations using graph theoretical methods (Tarjan, 1972) (Wiberg, 1977) and formula manipulation. A structural analysis of the equations is performed. This information can be very helpful during modelling of large systems since it gives information about causalities and algebraic loops.

The use of general equations makes the model documentation better and there is no risk of introducing errors during manual transformation to assignment statements. Furthermore, the equation form is the only natural one when different types of calculations, such as simulation or calculation of operating point, should be performed. Different environments to submodel instances might also lead to different causalities and therefore different transformations.

A model is often decomposed into *submodels* according to the physical decomposition into subsystems. When a subsystem is isolated its boundaries are first determined. To model the interaction of the subsystem with its environment it is necessary to introduce variables which describes what happens at the boundaries. Such variables are called *cut variables*.

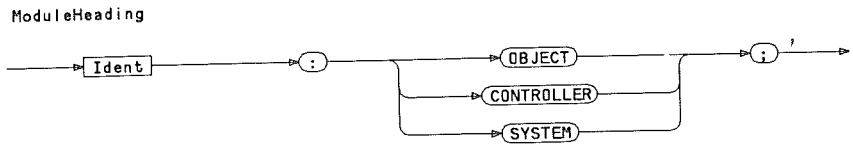
The physical subsystems are often connected together by distinct *connection mechanisms* such as pipes, shafts and electrical wires. With each such connection mechanism there might be several cut variables associated. There is a need for grouping related cut variables together. By such a grouping, the over-all structure of connected submodels becomes much easier to comprehend. Such groups are called *cuts* in the modelling language Dymola (Elmqvist, 1978, 1979a).

There are different kinds of connection semantics depending on what the cut variables represent. Consider, for example, three connected electrical wires. Each wire is represented by a voltage and a current. The constraints at the junctions are that the voltages are equal and the *sum* ot the currents are equal to zero. These two types of cut variables are sometimes called *across variables* and *through variables*.

3. Controller Language

Modules

Hierarchical decomposition is a basic mechanism for dealing with complex systems. LICS has a hierarchical module concept with three kinds of modules: **Object**, **Controller** and **System**. The syntax of the module heading is shown in the following syntax diagram.



Objects correspond to the physical subsystems of the controlled process. The intention is that objects can describe two aspects: the interface to the real world and a dynamical model. The current form of the LICS description of the IO-interface is very simple. It uses standard functions AnalogIn and DigitalIn for input and standard procedures AnalogOut and DigitalOut for output. The description should be extended to allow both the IO-interface and a model to be described at the same time in an object. It should be possible to simulate a new version of a control system together with the object models while an older version actually controls the process.

Controller modules correspond to the programs in the computers controlling the process. Both the objects and the controllers can be hierarchically decomposed. The control system can thus be structured in different layers. The hierarchical structure of objects and controllers can be mixed together using **System** modules.

The language should include a *module type* facility. It would allow a pattern for modules to be declared and several *module instances* of such a type to be created. This facility has not yet been specified but it would be similar to the submodel construct of Dymola.

For *distributed control systems* it is natural to structure the description according to the functional behaviour independent of the particular processor that performs certain calculations. However, the secondary information about assignment of control modules to processors can be added as attributes to

controllers. If such an attribute is not given the module is placed in the same processor as its parent module.

Controllers are executed repeatedly at certain *sampling instants*. They should have a mechanism for specifying such events. In many cases the sampling intervals are constant. In order to make the processor load more even it might be necessary to specify different phases for controller execution. There are cases when a set of controllers should be executed synchronously, but with varying sampling intervals. The interval might, for example, depend on a flow in the process.

Constructs for controller assignment to processors and sampling specification has not been specified in detail.

Connections and interfaces

There are generally interaction between modules. The modularization should, however, be chosen so that such interaction is small.

The ability for the viewer of a system to easily comprehend the interaction structure greatly improves the understanding of the system. The interaction structure can be thought of as a graph with the submodules as nodes and each interaction channel represented by an edge. Block diagrams are, in fact, such graphs.

The interaction mechanisms of LICS has been designed to give several abstraction levels for connections. The first level, the block diagram, shows which modules do interact. The next level is concerned with *how* modules interact, i.e. the variables that are involved. The most detailed level, the effect of the interaction, can be seen in the equations containing interaction variables.

The interaction variables cannot be associated with the connections. The reason is that it should be possible to develop a submodule without knowing in what environment it should be used. This is necessary in order to allow for module types and modules libraries. Alias variables representing the connection variables have to be introduced. This corresponds to formal variables of procedures in general programming languages. The formal parameter list describes the interaction with the calling procedure via its actual parameters. Connection (information flow) between two called procedures is in fact established indirectly by having the same actual parameters appearing in both procedure calls.

A module often has interaction with several other modules. This means that the formal interface variables should be grouped corresponding to the different possible connections. Such groups are called *interfaces*. There are no

declaration of "actual" interface variables. Interfaces in different modules are instead connected together directly.

The graphical representation of modules contains an interface area as shown in Fig. 3.1.

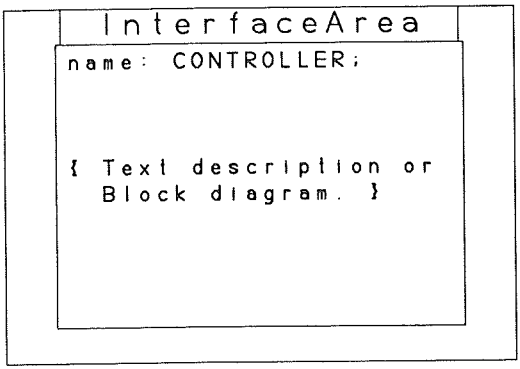
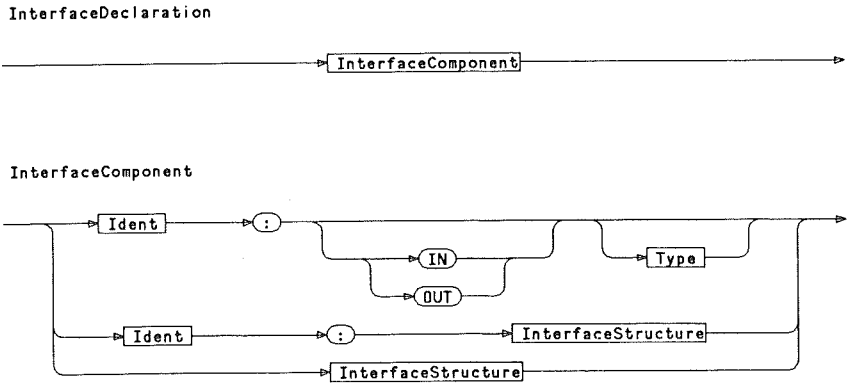


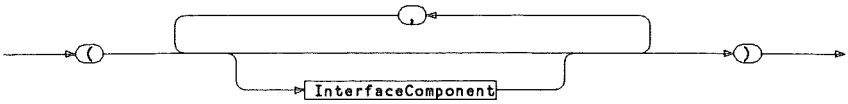
Fig. 3.1. Module layout.

Modules at the lowest hierarchical level has a textual description containing equations and declarations of variables and parameters.

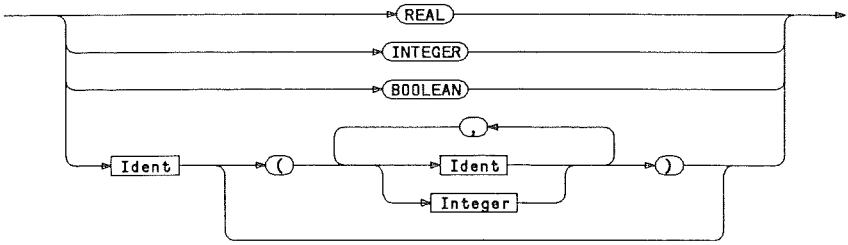
Interfaces may have a hierarchical structure. They are presently declared textually according to the following syntax.



InterfaceStructure



Type



Examples

PIDref: **in real;**
Next: (Enable: **out boolean,** *Reset:* **in boolean);**
Prev: (Enable: **in boolean,** *Reset:* **out boolean);**
InSteam: (P1: **real,** *H1:* **real,** *W1:* **in real);**
OutSteam: (P2: **real,** *H2:* **real,** *W2:* **out real);**

■

The graphical layout of interfaces is done automatically. The present layout does not include **in**, **out** and the type.

If two structured interfaces are connected together, it means that their corresponding components are connected. The number of components must be the same in the two interfaces. If a structured interface has only one component, the component and the interface itself becomes connected. Primitive interface components might also be used to pass through a structured connection to submodules. Any non-reserved type name can currently be used for that purpose. Connection lines terminates externally at interfaces or internally at interface components.

Primitive interface components of controllers have **in** or **out** attributes. This attribute indicates the direction of information flow. Connected primitive

interface components must have the same type. The following primitive variable types are currently allowed: **real**, **integer** and **boolean**.

It should be noted that the direction of information flow may differ for different components in an interface. This solves the problem of requesting back-calculations in a regulator and request-acknowledge signals.

Example

Using the previously defined interfaces Next and Prev results in the layout shown in Fig 3.2. It also indicates where the two interface variables should be defined according to the **in**/**out**-attributes.

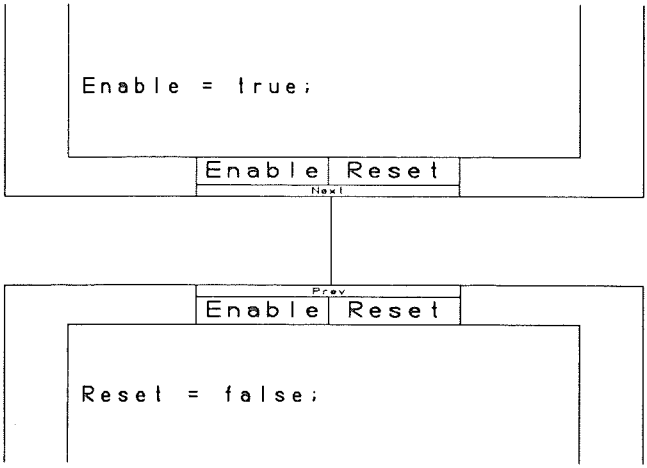


Fig. 3.2. Connected interfaces.

■

The semantics of the **in**/**out**-attributes is different for modules of object type. The **in**/**out**-attribute is omitted in objects if the cut variable is of across type (voltage type). An **in**/**out**-attribute indicates that the variable is of through type (current type) and it also gives a reference direction which will influence the balance equation generated.

Example

Consider the models M1, M2 and M3 in Fig. 3.3. The semantics of the two connections are described by the following equations.

$$M1.P2 = M2.P1$$
$$M1.P2 = M3.P1$$

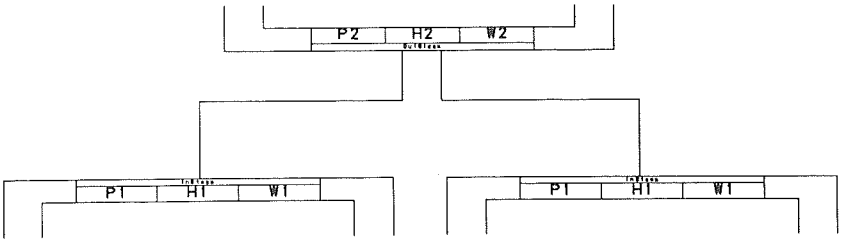


Fig. 3.3. Connected models.

$$M1.H1 = M2.H1$$
$$M1.H1 = M3.H1$$
$$M1.W2 = M2.W1 + M3.W1$$

■

The presented concepts for handling of modules and their connections is influenced by "object oriented programming". The language Smalltalk-80 (Goldberg, 1984) has *objects* as a basic modularization facility. Object *instances* are created from patterns described as *classes*. Objects communicate by sending messages. Objects have associated *methods* that describe how it should act and respond to a certain message. An essential property of the object oriented approach is to group related information together and to avoid global accesses. The package facility of Ada has this property. A package definition describes the procedures that are callable from the outside. Internally it may contain other procedures and variables that are not available from the outside.

Variables

The language has five kinds of variables. Their use is summarized below. *Interface variables* are used to describe the interaction between modules as was discussed previously. *Constants* and *parameters* are declared as shown in the following examples.

const *PI = 3.14159; Area = 5E-4;*
par *Gain = 16.7;*

Parameters are declared as an interface to the operator who could use an operator communication program to change their values on-line. The given parameter value is a default value used at start-up. In connection with module types there should be a construct to modify default parameter values at module instanciation.

State variables and *auxiliary variables* are declared in the following way

state *x1, x2:* **real**; *Activated:* **boolean**;
var *Temp:* **real**; *u:* **vector***(5)*; *p:* **matrix***(10,10)*;

State variables hold information between the sampling instants of controllers (c.f. equation (3.3)) while **var**-variables are defined and used at each instant.

Matrices are supposed to be declared using parametricized types such as **vector**(n) and **matrix**(n, m) for one and two dimensional real matrices.

The scope rules for variables are very simple because of the powerful connection concept. Variables can only be referenced from equations in the module where they are declared. Dot-notation can not be used to reach a variable from the outside, nor can a variable be reached from submodules.

Equations

The behaviour of modules at the lowest level are described by equations in textual form. Experience with the simulation language Simnon (Elmqvist, 1977) shows the convenience of describing dynamical systems by equations instead of assignment statements. The user need not worry about getting the correct evaluation order, since the system sorts the equations automatically based on declarations of state variables. The general form of an equation is

$$\textit{expression} = \textit{expression} \tag{3.1}$$

This form is only allowed in object models. Controllers are specified using the following restricted form

$$\textit{variable} = \textit{expression} \tag{3.2}$$

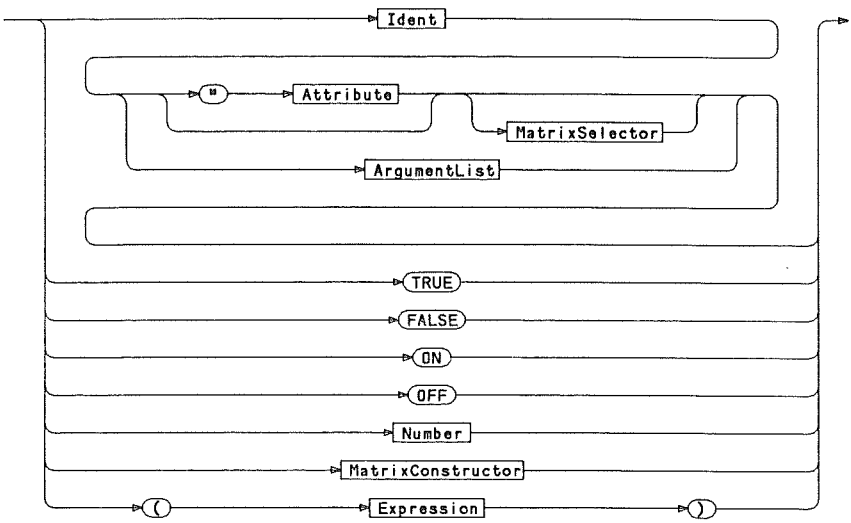
The direction of information flow (causality) is fixed for controllers in accordance with IN/OUT-attributes of the interface components.

Expressions have the usual syntax (see Appendix 1) with arithmetic, relational and boolean operators. The syntax of a primary of an expression is shown in the following syntax diagram.

Attributes are used together with STATE-variables. The current value of a state $x(t_j)$ of a controller is denoted x, whereas the next value $x(t_{j+1})$ is denoted x'**new**. Correspondingly, a continuous state $x(t)$ of a model is denoted x and its time derivatives $\dot{x}(t)$ and $\ddot{x}(t)$ are denoted x'**der** and x'**der2** respectively. The **old**-attribute is discussed later. The literals **on** and **off** are alternative names for **true** and **false**.

Indexing of a matrix is done by a list of integer-valued expressions called a matrix constructor. A *matrix constructor* is introduced to create a matrix from

Primary



expressions corresponding to its elements. A list of expressions is given for each row of the matrix. Block matrices can also be used.

The treatment of matrix expressions are not fully developed. In addition to the matrix operators $+, -, *$ several matrix functions would have to be introduced. The evaluation of functions like *Inverse*(m) and *Transpose*(m) would be delayed as much as possible. This is important since inverses are often pre- or postmultiplied by some matrix. For example, the expression

$$x = Inverse(A) * B$$

would then be evaluated by solving for x in the matrix equation

$$A * x = B$$

A variable in the left hand side of (3.2) is a restricted primary which may only contain an attribute and matrix selector.

Procedures and functions

Predefined functions may be used. The following functions are currently

available

integer $\mapsto$ **integer** : **abs**

real $\mapsto$ **real** : **abs, sqrt, exp, ln, sin, cos, arctan**

real $\mapsto$ **integer** : **round**

integer $\mapsto$ **real** : **AnalogIn**

integer $\times$ **integer** $\mapsto$ **boolean** : **DigitalIn**

The integer argument to AnalogIn is a channel number. The arguments of DigitalIn are group and channel numbers.

Predefined functions provide one way to extend the power of the language. However, functions allow only one output value. The procedure call can be seen as a generalization having a set of input arguments and a set of output arguments.

Two standard procedures are currently defined.

call *AnalogOut(channel, value);*
call *DigitalOut(group, channel, value);*

A convenient mechanism has been provided for the user to incorporate new procedures and functions written in Pascal. This will be illustrated by an example.

Example

Make the following Pascal procedure callable from LICS.

procedure *Proc(in1: real; in2: integer;*
var *out1: boolean;* **var** *out2: real);*

A global enumeration type FuncProcType in package ProcName has to be extended with an element, say Proc1.

Definition of the procedure is done in a procedure DefineUserProcFunc in package UserProc. The following statements are added.

DefineProc(Proc1, 'Proc ', ProcedureType, NoDefType, UndefType);
DefineArg(Proc1, RealType, InArg);
DefineArg(Proc1, IntType, InArg);
DefineArg(Proc1, BoolType, OutArg);
DefineArg(Proc1, RealType, OutArg);

A procedure EvalUserProc(ProcIndex: FuncProcType) in package UserProc is called when the user defined procedure Proc should be executed. It contains a case-statement over ProcIndex. A new alternative thus has to be added.

Proc1: Proc(ActArg[1].RealVal, ActArg[2].IntVal,
ActArg[3].BoolVal, ActArg[4].RealVal);

■

It is important to have a mechanism for extending the computational power. The function/procedure construct presented enables extension without loosing the security and convenience of the equation oriented language. Procedure invocations are sorted in the same way as equations.

Changing behaviour of modules

The equations of a control system are generally not fixed. Start-up of a process may, for example, require other regulators than those used at continuous operation. Programmable controllers are often used to control sequences of actions. The language contains four levels of modifications to the behaviour.

- conditional expressions
- conditional equations
- start/stop of modules
- replacement of equations by implementor

Conditional expressions

The functions f and g of (2.2) may be non-linear and time varying. Small and local changes are conveniently introduced by allowing multiple expressions in the right hand side and associated mutually exclusive predicates for their applicability.

$$variable = \begin{cases} expression_1 & predicate_1 \\ expression_2 & predicate_2 \\ \dots & \\ expression_{n-1} & predicate_{n-1} \\ expression_n & otherwise \end{cases}$$

A notation for this kind of equation is the if-then-else expression of Algol-60.

variable = **if** *condition*$_1$ **then** *expression*$_1$
elsif *condition*$_2$ **then** *expression*$_2$
...
elsif *condition*$_{n-1}$ **then** *expression*$_{n-1}$
else *expression*$_n$ **end if**

The conditions need not be mutually exclusive since the selection of the i:th alternative requires the conditions $1, 2, \dots i-1$ to be false and the i:th condition to be true.

Conditional equations

The next level of changes to the equations is *selection* of one set of *equations* from several sets. It is expressed using an analogue to an **if**-statement.

if $condition_1$ **then**
  $set_of_equations_1$
elsif $condition_2$ **then**
  $set_of_equations_2$
...
elsif $condition_{n-1}$ **then**
  $set_of_equations_{n-1}$
else
  $set_of_equations_n$
end if

If it is required that the sets of variables appearing at the left hand side of each set of equations to be the same then the conditional equations are equivalent to equations with conditional expressions as right hand sides, if the conditions are duplicated for each equation.

Start and stop of modules

A modification to a control scheme such as switching to another regulator could be done by changing connections. One way of doing this is to introduce switch modules which select the regulator output to be used. This does not generally work well because breaking a regulator loop may result in drifting states. It may also be cumbersome to switch back to the regulator. Furthermore, it would be a waste of computing power to continue to evaluate the equations of the disconnected module.

A more useful mechanism would be the ability to start and stop modules. Starting and stopping of a module is an external influence (at least starting). External influences are described in interfaces. A natural start-stop mechanism is thus to introduce two reserved boolean interface variables:

start: in boolean
stop: in boolean

The module would start when the **start** signal becomes true and stop when the **stop** signal becomes true. Alternatively, a reserved interface variable

on: in boolean

could be used to tell when the module should be active.

Starting a module means that its submodules having no **start**-interface component or **start** = **true** is started. Furthermore, a module can only be started if its parent is active. Stopping a module means that all submodules are stopped. The start-stop mechanism has not been implemented.

Sections

The textual descriptions of modules may contain several sections of equations, conditional equations and procedure calls. The basic section has the header

equations

The heading of other sections have the form

section name

The following reserved section names are used

init
start
stop
NewParameters
NewAlgorithm
exception

Use of sections with other names has been made obsolete since general equations are put in the **equations**-section. Therefore, the syntax ought to be changed to omit the **section**-keyword.

The **init**-section only contains assignment of initial values of states. The **start**- and **stop**-sections can be seen as conditional sets of equations. The conditions become true when the module starts or stops respectively. The equations are used to "resynchronize" with the environment at start and to ensure that outputs and states are left at well-defined values at stop.

The **NewParameters**-section is activated when the operator has changed any parameters in the module. It allows back-calculation of state in order to get bumpless parameter changes.

The equations in the **NewAlgorithm**-section are activated when the programmer has done any changes in a module. The module is compiled and transferred to the control computer. There is a special problem when the declarations of state-variables are changed. The reason is that they might contain important information about the status of the process. This information must not be lost. However, it may be necessary to code this information differently in other state-variables. This section may thus contain special

equations for converting state information. Values of state-variables of the old version can be accessed using the attribute **old**.

The **exception**-section is intended to include equations that are activated when arithmetic exceptions occur in the module.

Only the **init**-section has been implemented.

Semantics of controller equations

This section discusses the semantics of controller equations. A discussion of model equations can be found in (Elmqvist, 1978, 1983a).

The LICS language is based on signal flows and equations. It thus has very much in common with *data flow languages* (Ackerman, 1979). Such languages are often based on *definitions*: a variable is defined to be equal to some expression. The *single assignment rule* is obeyed, i.e. there are not multiple definitions of the same name. This is in contrast to *imperative languages* (traditional programming languages) where assignments to a certain variable may occur several times.

Iteration has to be handled in a special way in single assignment languages. The data flow languages VAL, ID, LUCID (Ackerman, 1979) introduces special loop variables. Special notations are used to define initial values and to define new values to be used during the next iteration.

These loop variables correspond to state variables in LICS and iteration is performed over sampling instants.

Data flow computers are composed of many computational nodes. Before a computation "fires" it is tested that all input data are available.

LICS' semantic model assumes that execution is performed on a fairly small number of parallel computational nodes. Many modules will thus be executed on the same sequential computer. It is important to avoid scheduling overhead. The equations are therefore sorted in correct evaluation order at compile time. The ability to sort the equations requires that there are no mutually dependent definitions, i.e. systems of equations that have to be solved simultaneously. Such equations have to be solved by calling a user defined procedure.

Each module will now be studied in order to discuss restrictions in order that its dynamical behaviour is described by equation 2.2. Consider a module only containing simple equations without matrices (no conditional equations). The equations can be converted to form (2.2) by substituting all auxiliary variables appearing in the right hand sides of equations defining $y(t_j)$ and $x(t_{j+1})$. The variables are substituted by the right hand side of their definitions.

Variables of type $y(t_j)$ and $x(t_{j+1})$ appearing in the right hand sides are substituted similarly. This substitution process terminates if there are no mutual dependencies.

Conditional expressions poses no problem since there is only one variable in the left hand side. It just introduces conditional definitions for f and g.

Functions and procedures poses no problem since there are no side effects. Procedures are just considered as vector valued functions.

There is a problem with the matrix selector when used in left hand sides of equations. The single assignment rule is then violated. If only indices with constant values are used, it is possible to use the matrix constructor in the right hand side instead. If variable indices should be allowed there is the potential problem that some matrix elements are never defined while others might be multiply defined. A feasible solution might be to specify a default value such as zero and give an exception if multiple definitions of a certain element occur. All definition equations for a matrix must precede all uses of that matrix.

Conditional equations violates the single assignment rule. If it is required that the same variables are defined in all sets of equations then the conditional equation is equivalent to a set of conditional expressions with the same conditions. It is just a more convenient notation.

However, there are cases when an auxiliary variable is defined and used in some of the sets of equations but not outside the conditional equation. It should then be permissible to omit the definition in the sets that do not use the variable. Furthermore, it may be convenient to omit definition of $x(t_{j+1})$ for some component that does not change, i.e. a default update function f_i, the identity equation, is introduced.

The different sets of equations in an if-equation are sorted separately. The if-equation is then considered as a function with inputs equal to the *union* of the inputs to the sets of equations. The outputs are the *intersection* of the outputs from the sets of equations.

Start and stop of modules can be seen as just changes to the set of sampling instants $\{t_j\}$ by deleting some instants.

We will now consider connecting together several modules. The total system will be of the form (2.1). The different submodules will have different sampling instants $\{t_j\}$. The sampling instants of the total system $\{T_I\}$ will be the merge of sampling instants for each module. Multiple equal instants are deleted.

By introducing a new state variable (z) for each output, it is possible to convert each submodule to the form (2.1) having the common set of sampling

instants. The functions F and G will be time varying, often being trivial.

$$
\begin{aligned}
y(T_I) &= \begin{cases} g(x(T_I), u(T_I), p) & \exists j | T_I = t_j \\ z(T_I) & \text{otherwise.} \end{cases} \\
x(T_{I+1}) &= \begin{cases} f(x(T_I), u(T_I), p) & \exists j | T_I = t_j \\ x(T_I) & \text{otherwise.} \end{cases} \\
z(T_{I+1}) &= \begin{cases} g(x(T_I), u(T_I), p) & \exists j | T_I = t_j \\ z(T_I) & \text{otherwise.} \end{cases} \\
x(T_0) &= x_0 \\
z(T_0) &= z_0
\end{aligned}
\tag{3.3}
$$

The equations (3.3) shows that x and y keep their values between the sampling instants. This can also be viewed as *zero order hold circuits* or memory cells for x and y.

$$
\begin{aligned}
y(t) &= \begin{cases} y(t_j) & t_j \leq t < t_{j+1}, j = 0, 1, \ldots \\ y_0 & t < t_0 \end{cases} \\
x(t) &= \begin{cases} x(t_j) & t_j \leq t < t_{j+1}, j = 1, 2, \ldots \\ x_0 & t < t_1 \end{cases}
\end{aligned}
\tag{3.4}
$$

$y_0 = z_0$ defines the value of y before the first sampling instant t_0 which might be $> T_0$.

Semantics of connections

The semantics of connections will now be studied in more detail. Introduce the following notations.

$D =$ "the set of all interface declarations"
$C =$ "the set of all interface components"
$S =$ "the set of all interface structures"
$P =$ "the set of all interface primitives"
$I =$ "the set of all interfaces"

The following relations are obtained from the syntax definitions.

$$
\begin{aligned}
D &\subseteq C \\
C &= P \cup S \\
I &= D \cup C \cup S \cup P
\end{aligned}
$$

Introduce the binary relation $\longleftrightarrow$ "is connected to". The fact that there exists a connection between two interfaces $a, b \in I$ is written

$$a \longleftrightarrow b$$

There are three different cases of connections.

External - external:

$$d_1 \longleftrightarrow d_2; \qquad d_1, d_2 \in D$$

Internal - internal:

$$p_1 \longleftrightarrow p_2; \qquad p_1, p_2 \in P$$

External - internal (internal - external):

$$d \longleftrightarrow p; \qquad d \in D, \quad p \in P$$

Introduce the relation $\models$ ("has the components"). The fact that a structured interface $s \in S$ has an associated list of interface components

$$(c_1, c_2, \ldots, c_n), c_i \in C$$

is written

$$s \models (c_1, c_2, \ldots, c_n)$$

The semantics of connecting two interfaces s and t is the following

$$\begin{gathered} s \longleftrightarrow t \quad \wedge \\ s \models (c_1, c_2, \ldots, c_{ns}) \quad \wedge \\ t \models (e_1, e_2, \ldots, e_{nt}) \quad \wedge \\ ns \geq 2 \wedge ns = nt \\ \Longrightarrow \\ c_1 \longleftrightarrow e_1 \quad \wedge \quad c_2 \longleftrightarrow e_2 \quad \wedge \quad \ldots \quad c_{ns} \longleftrightarrow e_{nt} \end{gathered} \tag{3.5}$$

where $s, t \in S, c_i, e_i \in C$.

The special case of only one component in a structured interface is described by $(s \in S, c \in C)$

$$s \models (c) \quad \Longrightarrow \quad s \longleftrightarrow c \tag{3.6}$$

In order to interpret a program it must be determined, for each interface, which interfaces it is connected to. This can be done by observing that the relation $\longleftrightarrow$ is an *equivalence relation* (Shannon, 1975) since for all $i, j, k \in I$

$$i \longleftrightarrow i \qquad (\textit{reflexive})$$

$$i \longleftrightarrow j \Rightarrow j \longleftrightarrow i \qquad (\textit{symmetric})$$

$$i \longleftrightarrow j \wedge j \longleftrightarrow k \Rightarrow i \longleftrightarrow k \qquad (\textit{transitive})$$

The problem is then equivalent to finding the equivalence classes of I.

$$M(i) = \{j : i \longleftrightarrow j\}$$

Example

Consider the modules, interfaces and connections in Fig. 3.4.

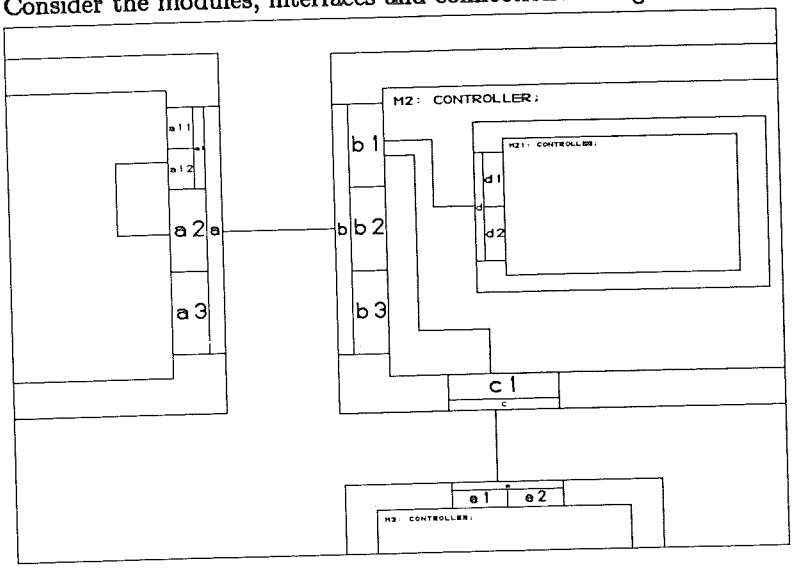


Fig. 3.4. Connections

The sets of interfaces are

$$D = \{a, b, c, d, e\}$$

$$C = \{a_1, a_{11}, a_{12}, a_2, a_3, b_1, b_2, b_3, c_1, d_1, d_2, e_1, e_2\} \quad \cup \quad D$$

$$S = \{a, a_1, b, c, d, e\}$$

$$P = \{a_{11}, a_{12}, a_2, a_3, b_1, b_2, b_3, c_1, d_1, d_2, e_1, e_2\}$$

The following connections are drawn.

$$a \longleftrightarrow b$$

$$a_{12} \longleftrightarrow a_2$$

$$b_1 \longleftrightarrow d$$

$$b_1 \longleftrightarrow c_1$$

$$c \longleftrightarrow e$$

The connection $a \longleftrightarrow b$ together with rule (3.5) gives the additional connections

$$a_1 \longleftrightarrow b_1$$
$$a_2 \longleftrightarrow b_2$$
$$a_3 \longleftrightarrow b_3$$

Rule (3.6) gives

$$c_1 \longleftrightarrow c$$

The connection $b_1 \longleftrightarrow d$ together with $a_1 \longleftrightarrow b_1$ implies $a_1 \longleftrightarrow d$ introducing the connections

$$a_{11} \longleftrightarrow d_1$$
$$a_{12} \longleftrightarrow d_2$$

Correspondingly, $c_1 \longleftrightarrow c$ together with $a_1 \longleftrightarrow b_1 \quad \wedge \quad b_1 \longleftrightarrow c_1 \quad \wedge \quad c \longleftrightarrow e$ implies $a_1 \longleftrightarrow e$ introducing

$$a_{11} \longleftrightarrow e_1$$
$$a_{12} \longleftrightarrow e_2$$

The equivalence classes become

$$\{a, b\}$$
$$\{a_1, b_1, d, c_1, c, e\}$$
$$\{a_{11}, d_1, e_1\}$$
$$\{a_{12}, d_2, e_2, a_2, b_2\}$$
$$\{a_3, b_3\}$$

■

Interconnected modules

The equivalence classes containing primitive interface components will influence the interconnection of the controllers described by (2.2). The primitive interface

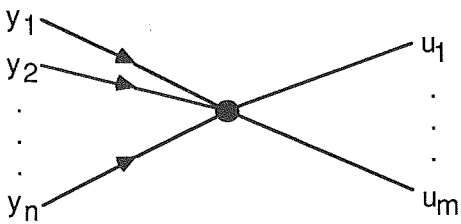


Fig. 3.5. Connection cluster.

components have associated directions, **in** or **out**. An equivalence class can thus be represented as shown in Fig. 3.5.

If there is only one **out**-component (one y variable in Fig. 3.5) the connections give rise to the following equations.

$$\begin{aligned} u_1(T_I) &= y(T_I) \\ u_2(T_I) &= y(T_I) \\ &\textit{etc.} \end{aligned} \tag{3.7}$$

By combining the subcontroller equations (3.3) with (3.7) we get the total system on the form (2.1). Note that all equations must be sorted jointly. There may be direct terms relating inputs and outputs in which case no delay is introduced. The case discussed also takes care of controllers having different sampling rates since hold circuits are introduced at outputs.

When there are several **out**-components in a cluster their definition times must be considered. This kind of connections is used when alternative modules should generate a signal. They should be turned on and off consistently so that exactly one module is on at any time.

The sampled and held outputs $y(T_I)$ of (3.3) can not be used. The outputs $y(t_j)$ of (2.2) must be used and sample and hold introduced at the cluster and not at each module. The cluster outputs are thus

$$\begin{aligned} u_1(T_I) &= \begin{cases} y_l(t_j^l) & \exists l, j | T_I = t_j^l \\ w_1(T_I) & \text{otherwise.} \end{cases} \\ w_1(T_{I+1}) &= w_1(T_I) \\ u_2(T_I) &= \ldots \end{aligned} \tag{3.8}$$

The case when there exists several l, j, i.e. several driving modules is considered an error. The security of the language may be improved further by introducing

a way of specifying whether the otherwise alternative may be used or not, i.e. if a hold circuit should be introduced. This would diagnose, for example, if a module was not started at the right time.

Asynchronous connections

The connection semantics discussed previous is synchronous. This implies a *partial ordering* for the evaluation of the equations.

When the evaluation of equations is performed in several processors, the partial ordering implies that the processors must be synchronized. This synchronization reduces the parallelism because a processor may have to wait for a signal. The sorting of equations on one processor thus need to take the load and execution speed of other processors into account in order to benefit from the fact that the ordering is only partial. In that way the waiting may be reduced by matching the instant a signal is required in one processor to the time when it becomes available in another processor.

Another drawback is the additional overhead when the processors communicate over a network. Very small messages (often one signal) have to be exchanged.

Another problem with the synchronous execution is the mixing of modules with short required response times and modules with long execution times. If there are no connections between these sets of modules their equations could be evaluated by different *concurrent processes*. A real-time kernel with handling of clock interrupts and priorities is needed to schedule the processes (see Elmqvist, et al, 1981).

If there are connections between the sets of modules, a similar synchronization need as for multiple processors occurs. It may be natural to require that the allocation of modules to processors and processes is done in such a way that connections are "loose" in the sense that synchronization is not required.

An *asynchronous connection semantics* is introduced for connections between concurrent processes (in one or several processors). It is important that each module gets consistent information from the other modules. It is not acceptable for a module to deliver one recently updated variable and another related variable which is not yet updated. Buffers are thus introduced at the inputs and outputs of the modules. Inputs from all modules in other processes are sampled and held in the input buffer before evaluation of equations in the process. New values of outputs are temporarily stored in the output buffer. Both the sampling and the delivering of outputs are performed as indivisible operations.

4. Application Examples

This chapter gives examples within the main application areas of LICS.

- Control algorithms
- Logic control including sequences
- Modelling

4.1 Control algorithms

One of the goals of the LICS project was to develop a system with enough expressive power to describe useful regulators and at the same time allow building control systems based on available library modules in a clear and consistent manner. The modularization facility, the two levels of abstraction, the structured connections and the textual algorithms gives these possibilities.

This example deals with a PID-regulator. It could be considered a library module which might be used for level control of a tank. Figure 4.1 shows the overall structure. The operator module is used to simulate operator actions because LICS does not contain any facilities for operator communication.

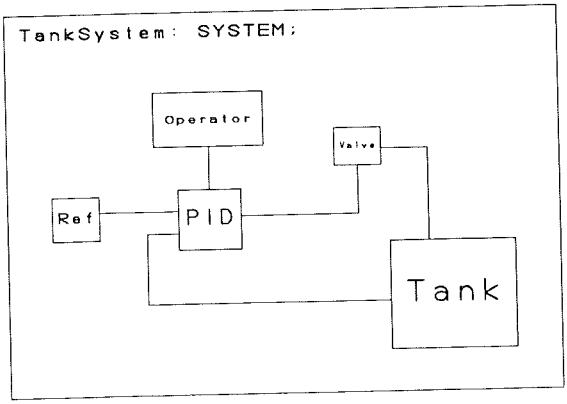


Fig. 4.1. Tank system.

The problems considered for the PID controller are

- limitation of the control signal (integrator wind-up)
- balancing of cascade loops
- bumpless mode changes
- bumpless parameter changes

Limitation of the control signal may occur in the actuator. This leads to integrator wind-up. One way of avoiding that is to back-calculate the integrator

state when limitation occurs in such a way that the corresponding control signal exactly matches the limit.

The problem is interesting because the information flow is opposite to the control signal flow. The actuator would have information about the limits and detect violations of the bounds. This information should be sent back to the regulator. However, if there is a cascade loop with an inner proportional regulator, information about the constraint violation should be passed to the regulator generating the reference value. It should be noted that it is not feasible to have limit parameters in the outer regulator because they would depend on the process measurement to the inner loop. The situation is even more complex if there are other types of blocks such as Summers etc.

Since the equations are sorted, LICS makes it possible to distribute the back-calculations needed to the modules, which have the necessary information. This does not disturb the overall structure because structured connections can be used for the two-ways data flows. The interface for the control signal of PID is thus

C: (y: **out real,** *ySet:* **in boolean,** *yNew:* **in real***)*

where y is the output. If this value is not considered valid by the following module, it sets ySet=true and gives a desired value as yNew. The reference module also has this output interface. Correspondingly, the reference signal of PID (r) is contained in the interface:

S: (r: **in real,** *rSet:* **out boolean,** *rNew:* **out real***)*

The process measurement interface is:

P: (u: **in real***)*

The control signal is calculated as follows.

$$y = \begin{cases} yOld + yI & \text{if Manual} \\ K * (e + Td/DT * (e - eOld)) & \text{elsif PDreg} \\ K * (e + DT/Ti * i + Td/DT * (e - eOld)) & \text{else} \end{cases}$$

The limited output is calculated as

$$yl = \begin{cases} yNew & \text{if ySet} \\ y & \text{else} \end{cases}$$

Back-calculation is performed at parameter changes, output limitation or for tracking of process variable in Manual mode.

BackCalc = *ParSet* **or** *ySet* **or** *Manual*

The back-calculated integrator state (in) for a PID-regulator is given by substituting y with yl and using the updated parameters.

$$yl = Kn * (e + DT/Tin * in + Tdn/DT * (e - eOld))$$

$$\Rightarrow in = (yl/Kn - e - Tdn/DT * (e - eOld))/(DT/Tin)$$

The reference value should be modified for a PD-regulator. The back-calculated control error (e') is thus given by

$$yl = Kn * (e' + Tdn/DT * (e' - eOld))$$

$$\Rightarrow e' = (yl/Kn + (Tdn/DT) * eOld)/(1 + Tdn/DT)$$

The control error is calculated as

$$e = \begin{cases} 0.0 & \text{if Manual} \\ SetPoint - u & \text{elsif Auto} \\ r - u & \text{else} \end{cases}$$

The previous limited output is stored as yOld independently of the controller mode. This allows bumpless transfer to *Manual mode*. The output is incremented by yI in Manual mode.

Auto mode means that the PID-algorithm is active with a stored SetPoint. The SetPoint is set equal to the process variable in Manual mode (process variable tracking). For a PD-regulator in Manual mode, the SetPoint is back-calculated. For cascade mode, the incoming reference value is stored as SetPoint.

In order to attain bumpless transfer from Manual or Auto mode to cascade mode, the SetPoint is sent to the module generating the reference value (balancing of cascade loops, Honeywell, 1976, 1978). This should also be done for a PD-regulator if back-calculation has been performed due to limitation of the output or parameter changes.

rSet = *Manual* **or** *Auto* **or** *BackCalc* **and** *PDreg*

The reference value is generated by a square wave generator, Ref, in the example. The PID-regulator may request a change of its bias by setting PID.rSet (=Ref.ySet).

The PID-regulator, valve and reference generator are shown in Fig. 4.2-4.4. The text is also shown in Appendix 3.

Parameters are increased or decreased as follows.

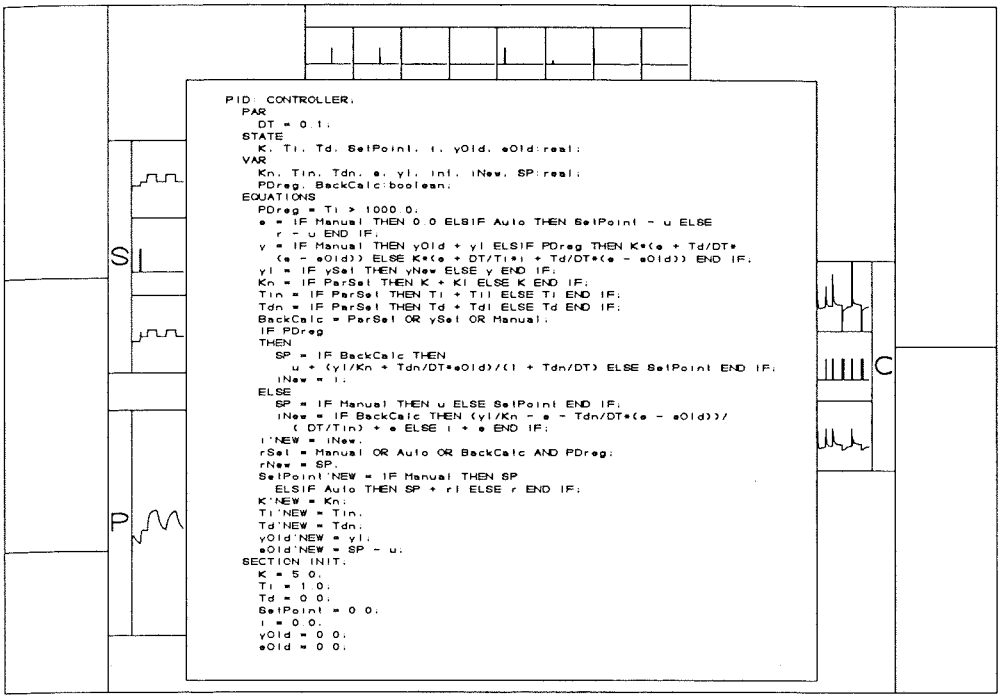


Fig. 4.2. PID controller with run-time data displayed.

$$Kn = \begin{cases} K + KI & \text{if ParSet} \\ K & \text{else} \end{cases}$$

etc.

Figure 4.5 contains the module used for simulating operator requests for mode and parameter changes. The tank model is shown in Fig. 4.6. LICS does not include any integration algorithm for ordinary differential equations. The model is therefore simulated by explicitly updating the tank level using Euler's integration method. Graphs of the interface signals appear at the interfaces during simulation (see Fig. 4.2).

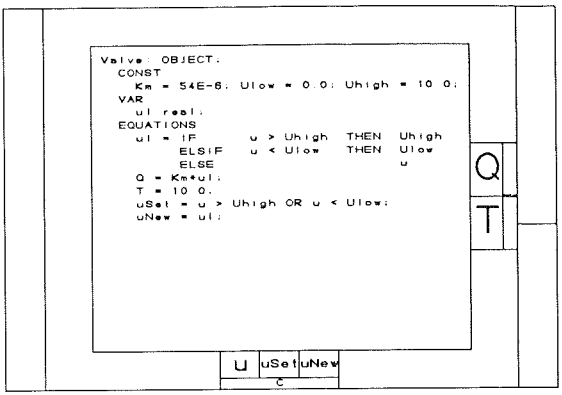


Fig. 4.3. Valve.

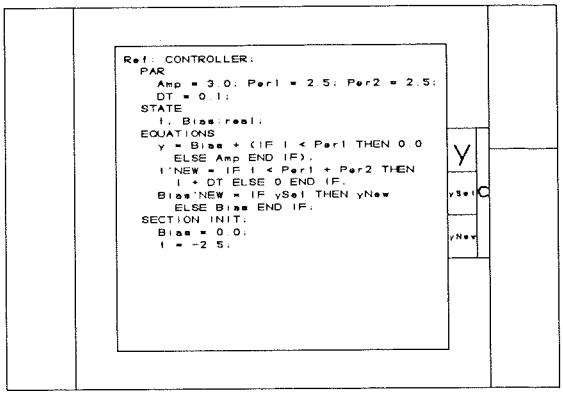


Fig. 4.4. Reference generator.

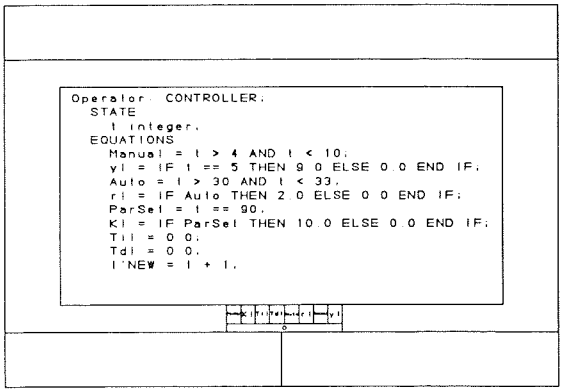


Fig. 4.5. Operator module.

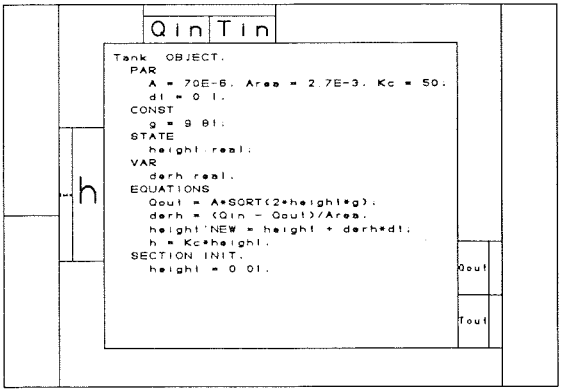


Fig. 4.6. Tank model.

4.2 Programmable controllers

LICS does not contain any predefined modules for logic control. This example shows how such a library can be constructed. Figure 4.7 shows a library module containing modules for logical operations (And1, Or1, Not1), input-output (In1, Out1), timer and counter. Boolean states are represented by a delay element (D). (The names of some modules contain a 1 because LICS does not allow modules to have names equal to keywords used in expressions.)

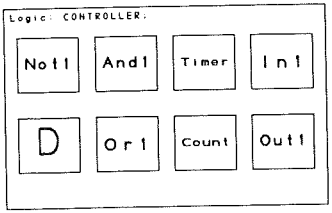


Fig. 4.7. Logic library.

The library is zoomed-in in Fig. 4.8 to show the trivial equations.

Figure 4.9 contains a module with two different implementations of an SR flip-flop, SR1 and SR2. Both modules have the inputs S (set) and R (reset) and the output Q. The flip-flop SR1 is built up by copying modules from the logic library and connecting them together. The implementation of SR2 is described by simply entering the update equation for the state.

$$x'\,\mathbf{new} = S \textbf{ or } x \textbf{ and not } R$$

The example demonstrates the users freedom to choose at what level to use equations. Graphic representation of equations are supported by LICS. The representation can be changed by the Change command. Figure 4.10 shows the SR2 module as a Function diagram. One diagram is generated for each equation.

The SR flip-flops are connected to two input modules. The parameters for digital input channels were chosen so that the LEFT and MIDDLE buttons of the mouse can be used for input. The interfaces show the signal histories during an execution.

The use of many small modules does not necessarily imply more code and slower execution. Figure 4.11 shows the generated intermediate stack-oriented code for the example. It shows how the equations are sorted. Simple, so called peep-hole optimizations, can be used to get rid of unnecessary store and load combinations.

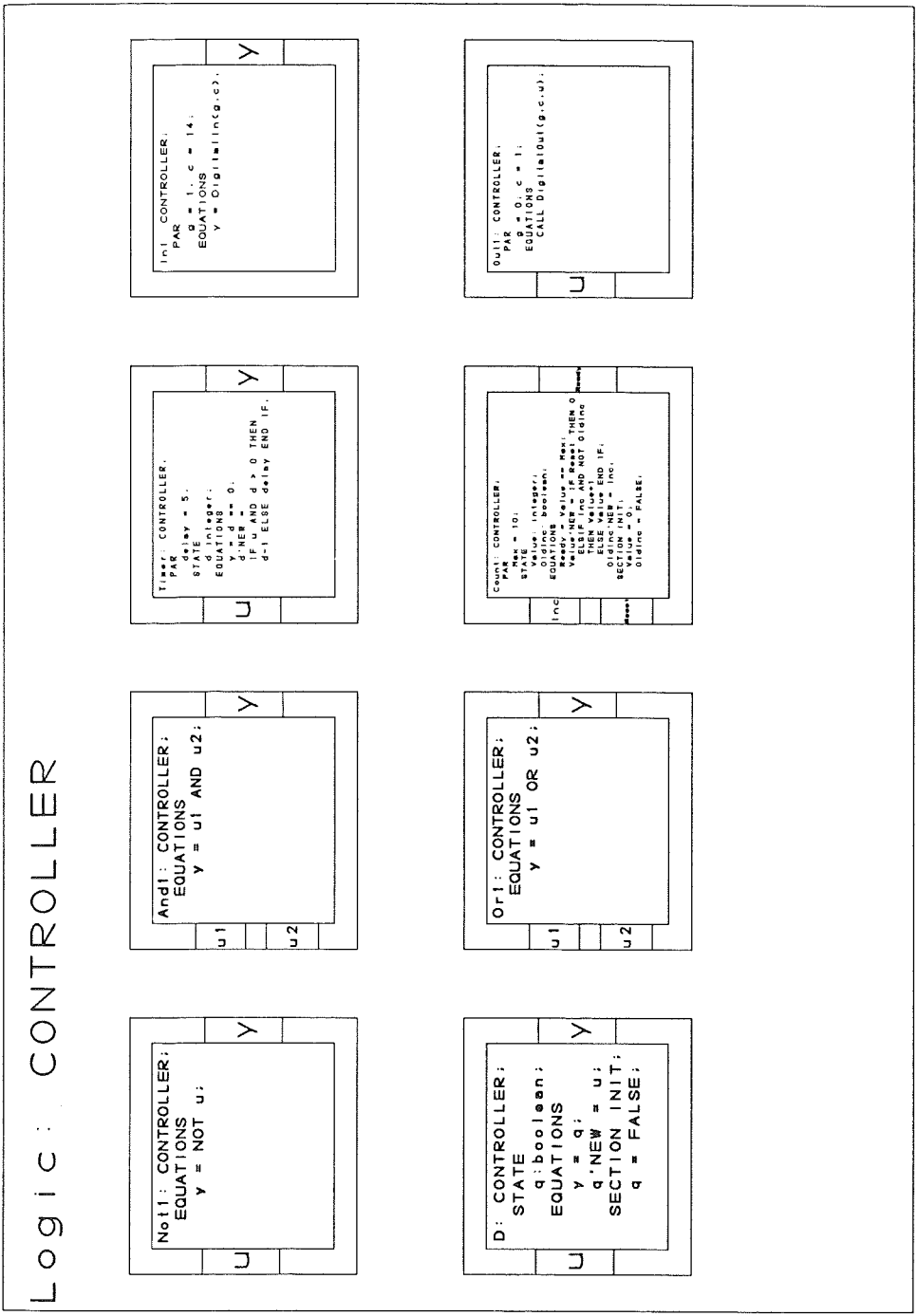


Fig. 4.8. Internals of logic modules.

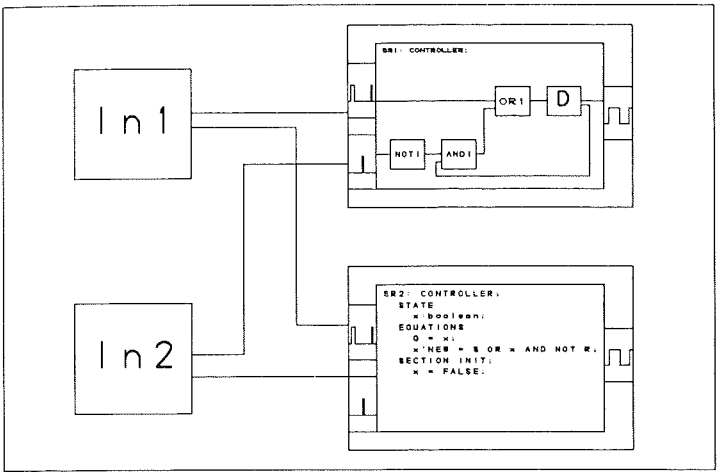


Fig. 4.9. Test of SR flip-flops.

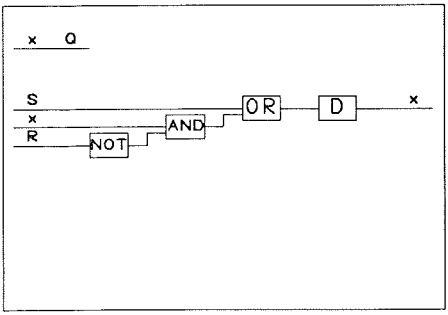


Fig. 4.10. Function diagram for SR flip-flop SR2.

```
INIT:

  { SR1.D:  q = FALSE; }
Lit       Bool       FALSE
Store     Bool                3  { q }

  { SR2:  x = FALSE; }
Lit       Bool       FALSE
Store     Bool                6  { x }

LOOP:

  { In1:  y = DigitalIn(g, c); }
Load      Int                 7  { g }
Load      Int                 8  { c }
DigitalIn Bool
Store     Bool               11  { y }

  { In2:  y = DigitalIn(g, c); }
Load      Int                 9  { g }
Load      Int                10  { c }
DigitalIn Bool
Store     Bool               12  { y }

  { SR1.NOT1:  y = NOT u; }
Load      Bool               12  { u }
Not       Bool
Store     Bool               16  { y }

  { SR1.D:  y = q; }
Load      Bool                3  { q }
Store     Bool               15  { y }

  { SR1.AND1:  y = u1 AND u2; }
Load      Bool               16  { u1 }
Load      Bool               15  { u2 }
And       Bool
Store     Bool               13  { y }

  { SR1.OR1:  y = u1 OR u2; }
Load      Bool               11  { u1 }
Load      Bool               13  { u2 }
Or        Bool
Store     Bool               14  { y }

  { SR1.D:  q'NEW = u; }
Load      Bool               14  { u }
Store     Bool                3  { q }

  { SR2:  Q = x; }
Load      Bool                6  { x }
Store     Bool               17  { Q }

  { SR2:  x'NEW = S OR x AND NOT R; }
Load      Bool               11  { S }
Load      Bool                6  { x }
Load      Bool               12  { R }
Not       Bool
And       Bool
Or        Bool
Store     Bool                6  { x }
```

Fig. 4.11. Code for TestSR.

Sequence control

The ability to describe sequences of actions is an important requirement for a control language. Furthermore, there may be several parallel activities which sometimes need to be synchronized.

Petri Nets (Petersen, 1981) is a notation for such systems. A Petri Net is a directed graph. It has two types of nodes (bipartite graph): places and transitions. The places correspond to the different steps of a sequence of actions. The transitions correspond to the conditions controlling steps to be activated and deactivated. An active step is represented by a token (dot) in the corresponding place.

Figure 4.13 shows a simple generic case of sequence control. It contains *alternative* sequences, *parallel* sequences and *looping*.

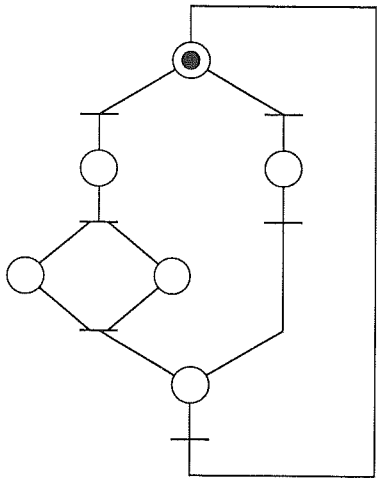


Fig. 4.13. Generic Petri Net.

Alternative sequences start from a place (circle) with several outgoing arcs to the transitions (bars) controlling the selected alternative. Parallel sequences start at a transition.

Petri Nets are used to model concurrent systems of various kinds. A slightly modified representation has been proposed as a standard for describing process control systems (IEC, 1985). Figure 4.14 shows the generic example in this notation.

The initial step is marked with double boxes. There are boolean conditions associated with each transition. A transition may "fire" when its associated

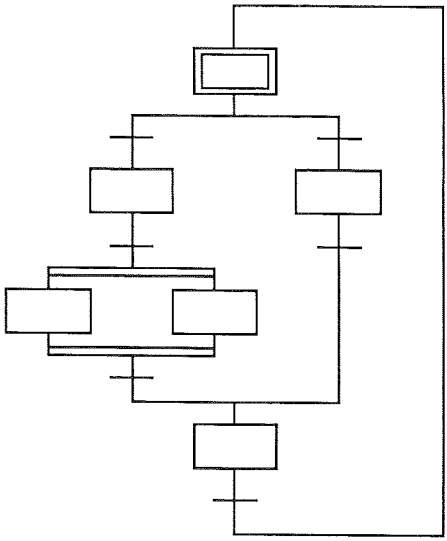


Fig. 4.14. Standard notation for generic sequence example.

condition is true and all immediately preceding steps are active. These steps are then deactivated and the following steps are activated.

Sequence control in LICS

It will now be shown how LICS' interconnected modules can be used for sequence control. Transitions cannot be represented by lines in LICS. They have to be drawn as boxes. The lines between steps and transitions are, however, drawn as ordinary connections.

Figure 4.15 shows a library of sequence modules. Figure 4.16 shows them opened up. Using these sequence modules, the previous generic sequence example can be described as shown in Fig. 4.17 in LICS.

A step is modelled by an SR flip-flop. Its output is used to activate the actions associated with the step. A step is connected to the following transition by a structured connection:

Next: (Enable: **out boolean**, *Reset:* **in boolean***);*

The Enable signal enables the transition. A transition occurs when the Condition AND Enable is true. A Reset signal is then sent to the previous step which resets the flip-flop. An Activate signal for setting of the flip-flop is sent to the following step:

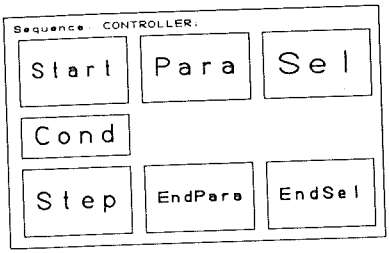


Fig. 4.15. A library of sequence modules.

Activate: **out boolean;**

Parallel and alternative sequences are handled by modules Para - EndPara and Sel - EndSel.

Parallel paths starts with a Para-module which is trivial and, in fact, redundant. It just sends the Activate signal to both the following steps. The EndPara-module needs some logic because there is a resynchronization. The following transition should not be enabled before both parallel sequences have reached EndPara. However, the Reset signal from the transition is just broadcasted to both preceding steps.

The selection module Sel is the most complicated one. The conditions of the transitions following it determine which path should be selected. The problem is to handle the case when both conditions are true at the same time. Only one path may be selected. The Sel module will select the left path in such a case. This is accomplished by conditionally enabling the right transition. It is not enabled if the left transition fires and sends a Reset signal. Either the left or the right transition will send a Reset signal. The signals are OR'ed to form the Reset signal of the preceding step.

The EndSel-module forwards an Activate signal either from the left or the right step which preceds it.

If more than two parallel paths are wanted, several Para modules can be connected after each other. This is, however, not necessary since the Para module is redundant. The Activate signal from a transition could be connected directly to the Activate signals of all the following steps. However, the EndPara, Sel and EndSel modules contain logic and cannot be removed as easily. It turns out that the EndPara and EndSel modules can be removed by changing the semantics of connections.

If the EndPara module was removed, both preceding steps would try to generate the Enable signal of the following transition. However, all the generated

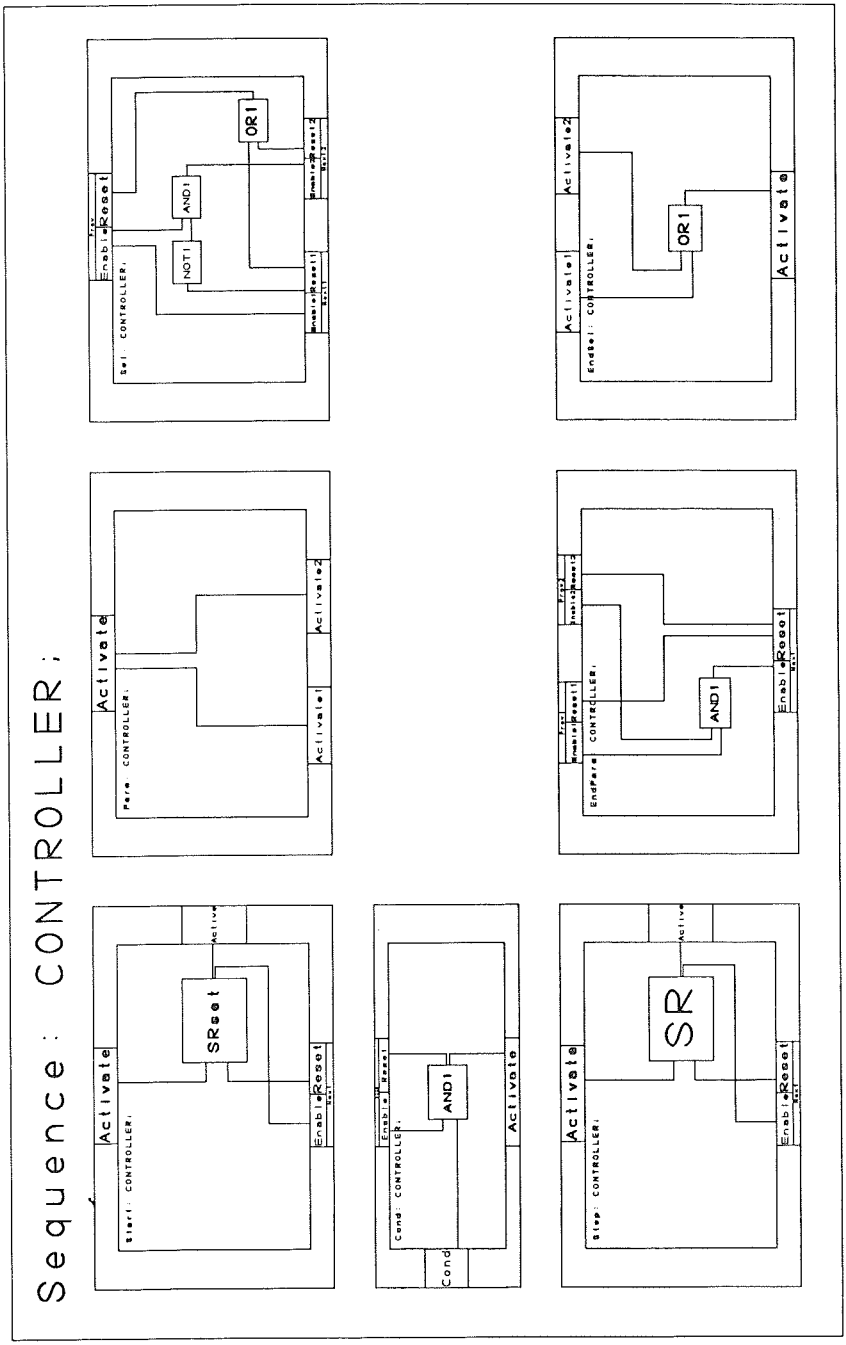


Fig. 4.16. Internals of sequence modules.

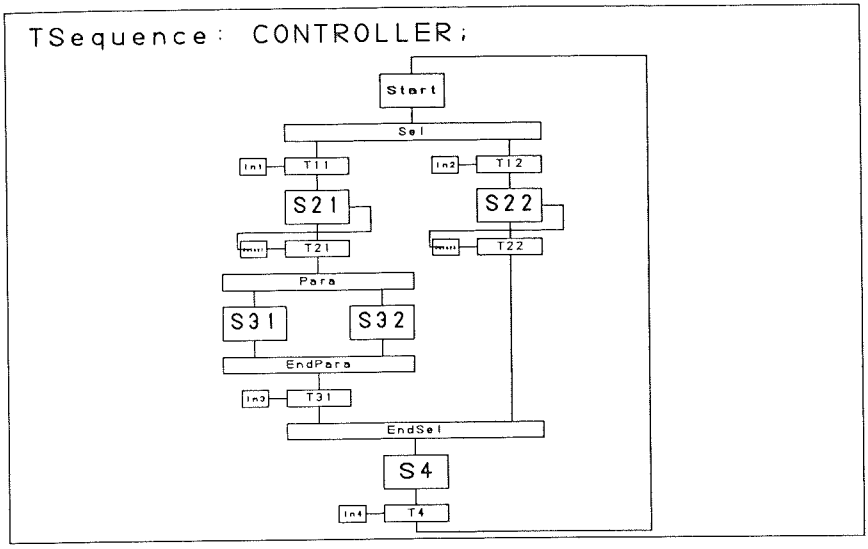


Fig. 4.17. Generic sequence example in LICS.

signals should be AND'ed together to form the Enable signal. Similarly, for the EndSel module, all the Activate signals of the preceding transitions should be OR'ed together to form the Activate signal of the following step. These cases show the need for introducing special OR/AND semantics for connections.

The idea was tested by introducing two new data types: ORboolean and ANDboolean. The interfaces were changed as follows:

Next: (Enable: **out ANDboolean**, *Reset:* **in boolean***);*
Activate: **out ORboolean***;*

It may be convenient to introduce other commutative and associative operators in the same way. The problem is in fact also related to summing incoming and outgoing flows in a dynamical model.

The priority logic of Sel still remains. It implies an ordering of alternatives which is made visible by using distinct interfaces for each alternative. However, the layout of more than two alternatives can be greatly improved by introducing three new selection modules SelL (left), SelM (middle) and SelR (right). A number of SelL, one SelM and a number of SelR can be arranged horizontally to give any number of alternatives. Figure 4.18 shows how these modules are built by combining Sel-modules.

Figure 4.19 shows the generic example using modified sequence modules

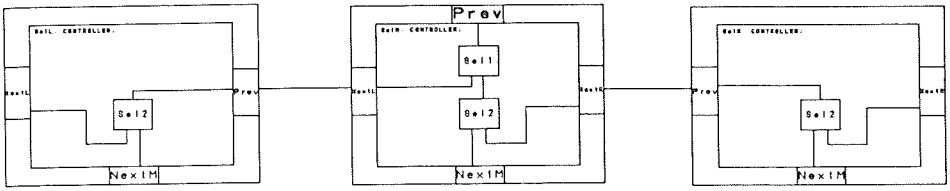


Fig. 4.18. Modules for selection of several alternatives.

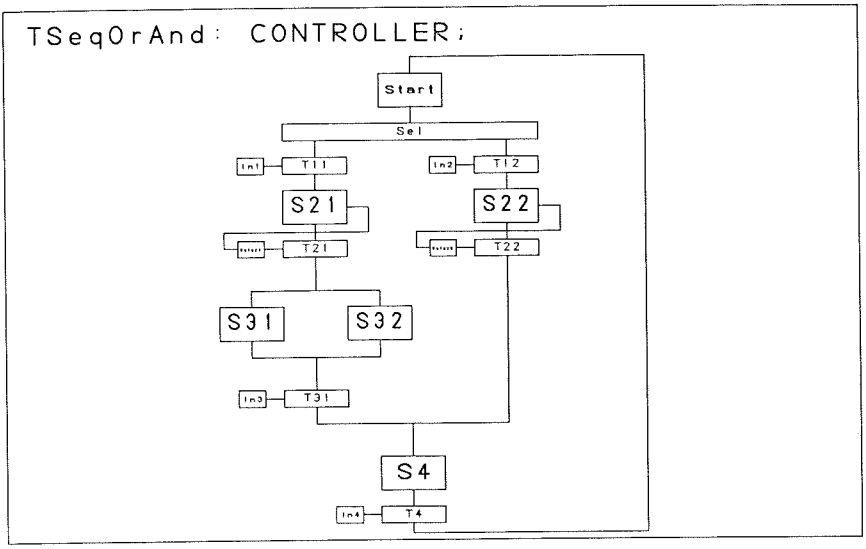


Fig. 4.19. Sequence example with OR/AND semantics.

with OR/AND semantics for connections.

Mutual exclusion

A common problem when dealing with concurrent activities is to give mutually exclusive access to common resources. Figure 4.20 shows the Petri Net for a solution to the mutual exclusion problem for two sequences and one resource. The resource is modelled by a place with an initial token.

This Petri Net is easily converted to LICS notation using the sequence modules with OR/AND semantics. The result is shown in Fig. 4.21.

The EndSel module is used. It cannot be removed in this case. The problem is the following. If T21 fires, an Activate signal should be sent to S31 and

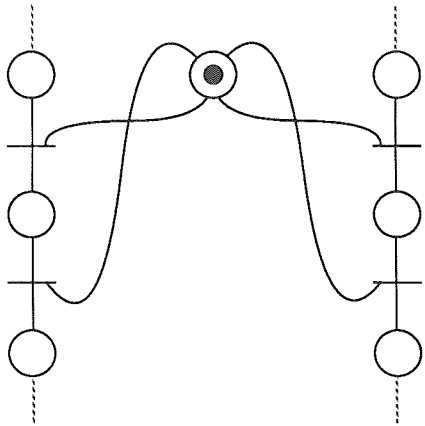


Fig. 4.20. Petri Net for mutual exclusion.

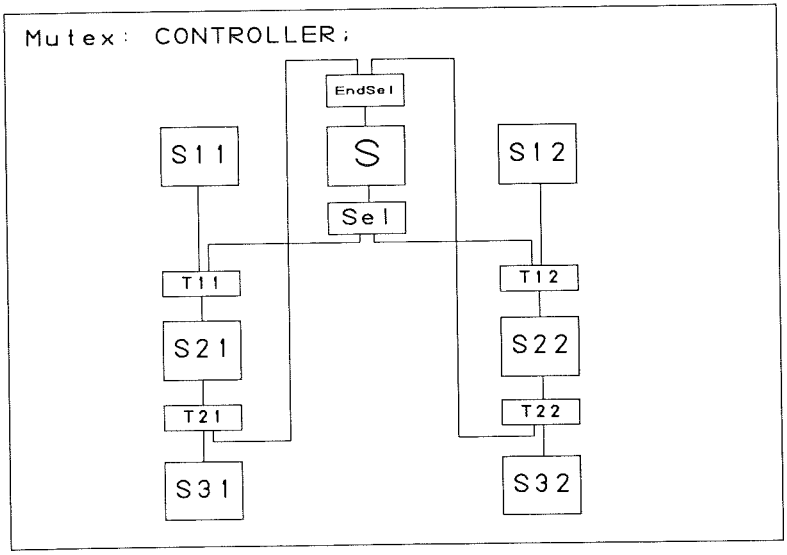


Fig. 4.21. Mutual exclusion using LICS.

S. However, if EndSel is removed, the Activate signal will also be sent to S32. It would propagate from T21 to S and then "backwards" to T22 and S32. The inherent problem is that the current connection semantics does not

consider the direction of signal flows. It just considers the "cluster" of connected interface components. In fact, without EndSel, there would be no difference if the connection S to T22 was replaced by the connection S31 to S32.

Common resources

The previous solution becomes cumbersome if there are more than two sequences competing for a resource. A nice and convenient solution can, however, be obtained by introducing higher level modules.

A natural way to represent a sequence making a request for a resource is to introduce a Request agent, in the form of a module, between two steps. A Release agent is introduced between a transition and a step. The Request and Release agents must be associated with the particular resource. This can be done by connecting the Request agents in a "daisy-chain" ending at the resource, and similarly for the Release agents.

Figure 4.22 shows an example with three sequences and Request and Release agents. The implementation of Resource, Request and Release modules is shown in Fig. 4.23. They are composed of the sequence modules (without OR/AND semantics).

There are unconnected signals to Request and Release of Seq3. A warning about that is given. It is convenient to avoid special versions of the modules for ending the daizy-chains. There should therefore be a way to define default values for unconnected interfaces. Currently, this works because boolean variables are initialized to false. This is the appropriate default value both for the Reset2 of Sel and Activate2 of EndSel. Appendix 4 contains a listing of the sorted equations.

Waiting queue

The resource allocation scheme described previously has one property that may be unsatisfactory. Assume that the resource is reserved and that several new requests occur. When the resource is released, the leftmost request will be granted. It is probably more natural to have a scheme that grants the earliest request first (FIFO). A queue is needed to accomplish this. A straightforward method is to introduce queue numbers. When a request is made, the next available queue number is picked. When a release is performed, all queue numbers are decremented by one and the request having number one is granted.

This scheme can be implemented by just modifying the Resource and Request modules. The left priority rule of Sel in Request is completely replaced. The Sel module does not have information about when the step preceding

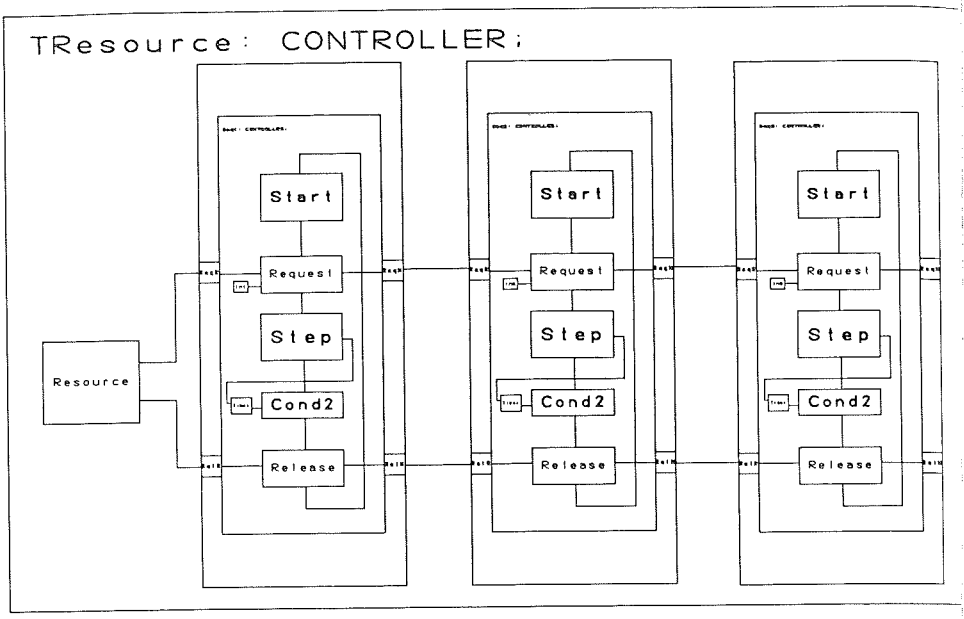


Fig. 4.22. Resource allocation.

the Request module becomes active. This is needed in order to draw a queue number. The interface between EndPara and Sel could be extended to send Enable up to Sel. Furthermore, the condition associated with the request module should be available in EndPara. However, it was chosen to describe the Request module using a textual algorithm.

The first issue is how to keep track of the queue length in order to hand out queue numbers. This is done by making each Request module calculate, and sending to the left, the maximum of his queue number and the value coming in from the right. Note that the modules of not active requests are also involved in this process. Their queue numbers are zero.

The Resource module sends back the received value as the queue length. Each new request implies that the queue length is picked up and stored in the state variable WaitNr. The queue length is incremented by one by waiting modules and sent to the right. This ensures that simultaneous requests do not get the same queue number. For simultaneous requests there will be left priority.

When the resource becomes available (EnableR) the Request agent with WaitNr=1 activates its succeeding step if the condition is still true. All waiting

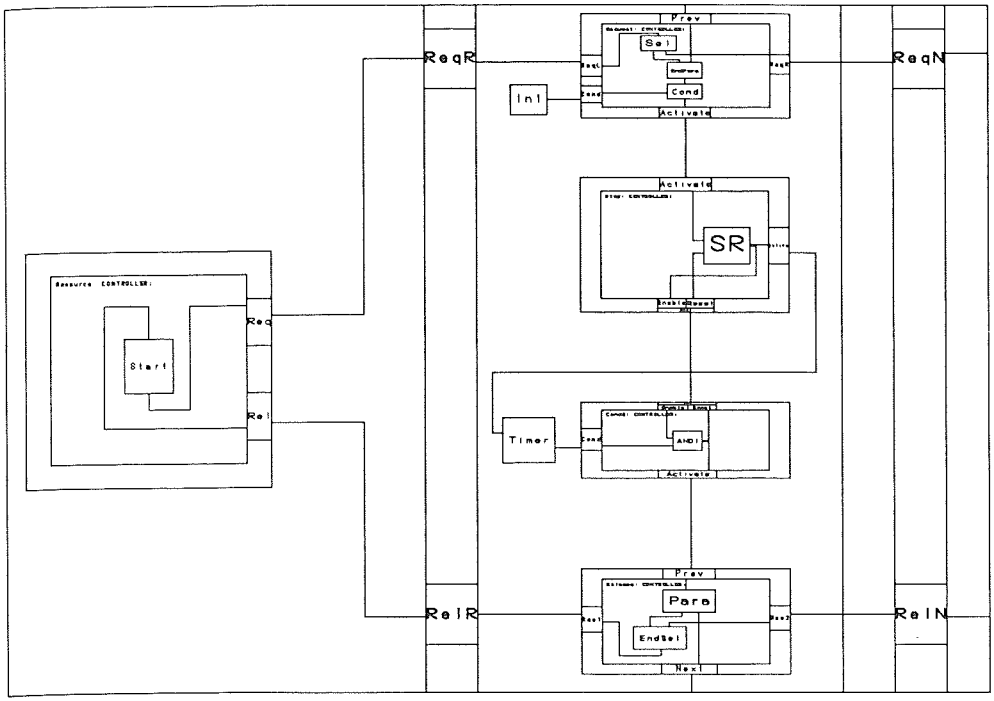


Fig. 4.23. Implementation of resource allocation.

agents decrease their WaitNr. If the condition had become false, the resource step is not reset (ResetR). Another pending request will thus be granted at the next cycle.

Figure 4.24 shows the resulting Request module. The resource module is slightly extended as shown in Fig. 4.25. Appendix 4 contains the sorted equations.

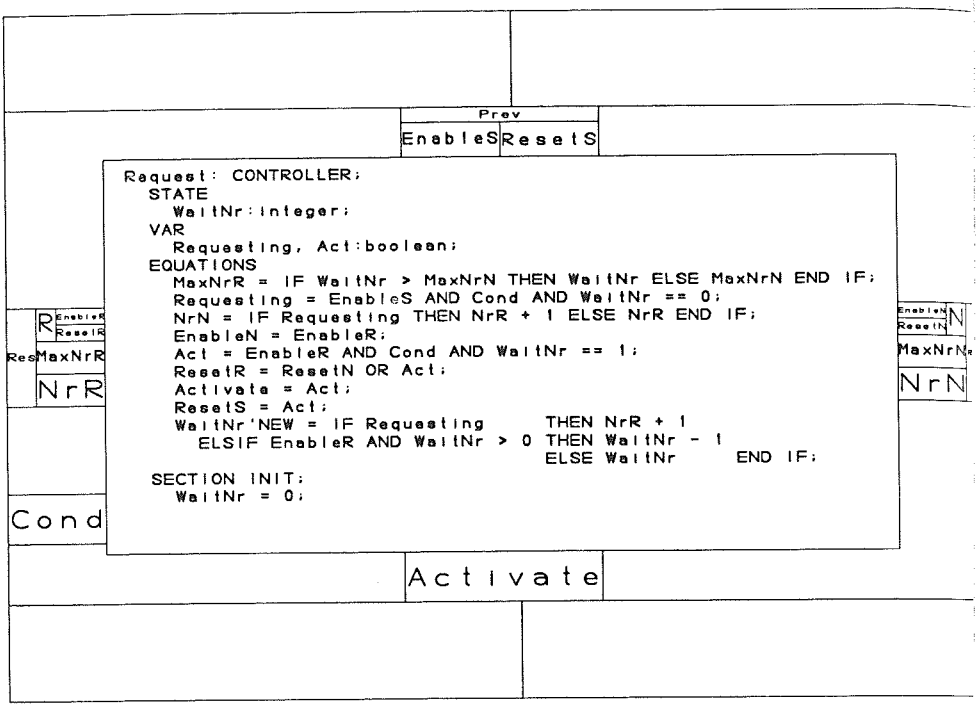


Fig. 4.24. Request with queue.

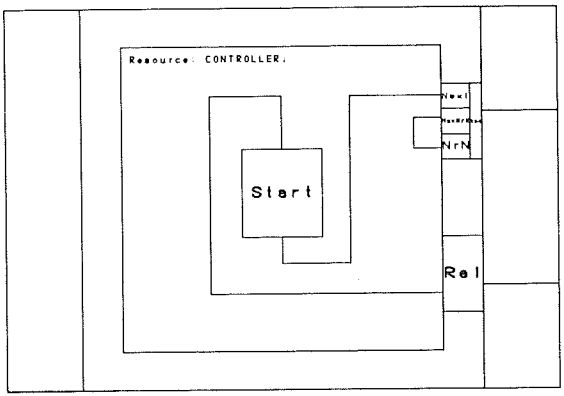


Fig. 4.25. Modified Resource module.

4.3 Modelling

The last application example is devoted to dynamic modelling. Objects are described by algebraic equations and ordinary differential equations (differential algebraic systems). Connections represent physical connections between objects.

The example considered is a thermal power plant. The original model is found in Lindahl (1976). This model is also used as an example for the modelling language Dymola (Elmqvist, 1978). Dymola is a purely textual language. However, it contains hierarchical models, interfaces (cuts) and connections. The LICS implementation contains a command, Dymola, for outputting objects in the Dymola format.

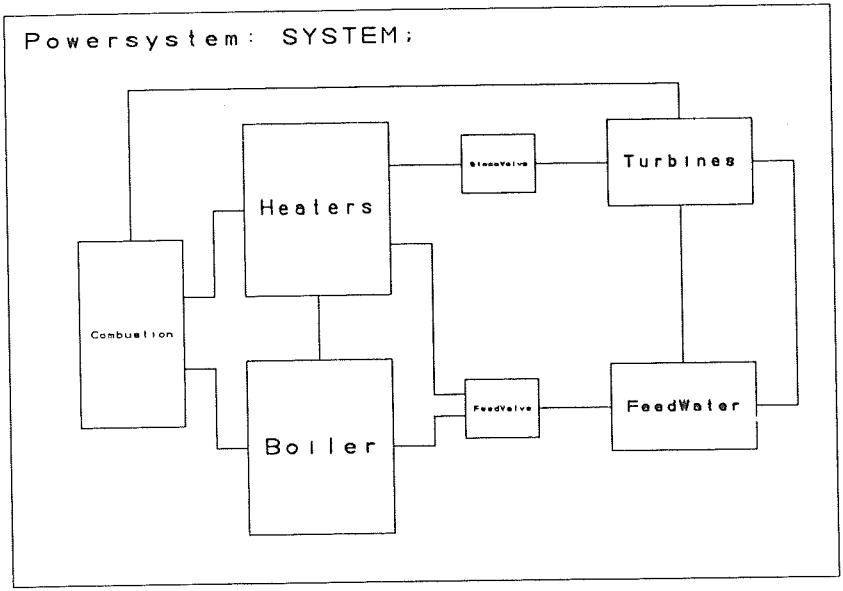
The following pages show the different types of modules. The overall structure in module PowerSystem shows steam connections from Boiler to Heaters to SteamValve to Turbines to FeedWater. Extract steam from the turbines is used to preheat the feedwater. The feedwater goes to the Boiler, but also to the sprayers in the Attemperators. The Combustion chamber delivers energy to the Risers and Economizer of the Boiler as well as to the SuperHeaters in Heaters and to the ReHeater in the Turbine module.

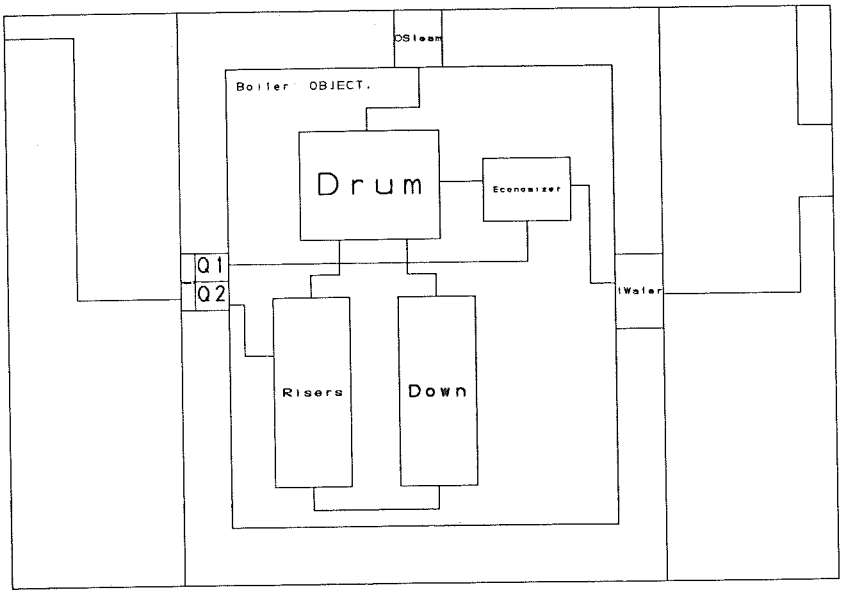
The equations are typically mass- and energy balances. External functions for interpolating in steam tables are also assumed.

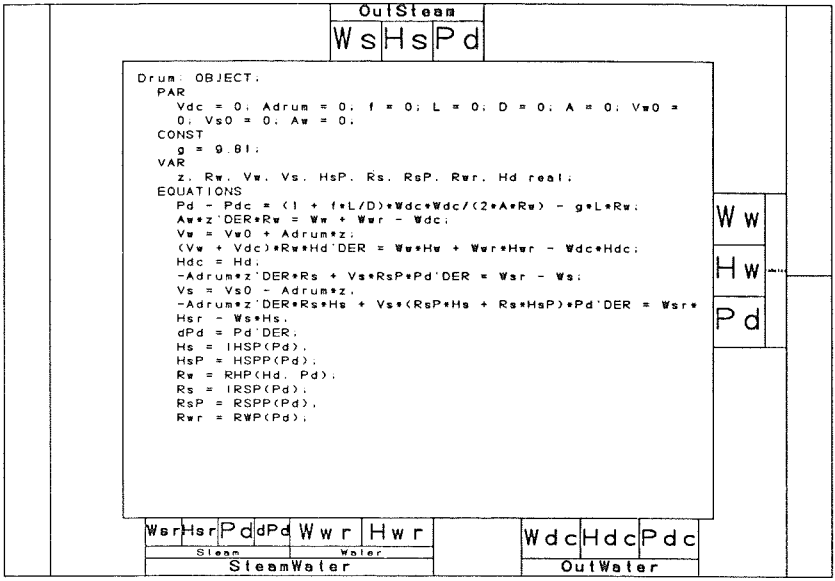
Appendix 5 contains the slightly modified output from the Dymola command. Some changes were made by a text editor mainly due to the different semantics of connections in LICS and Dymola. These differences are not taken care of in the implementation of the Dymola command.

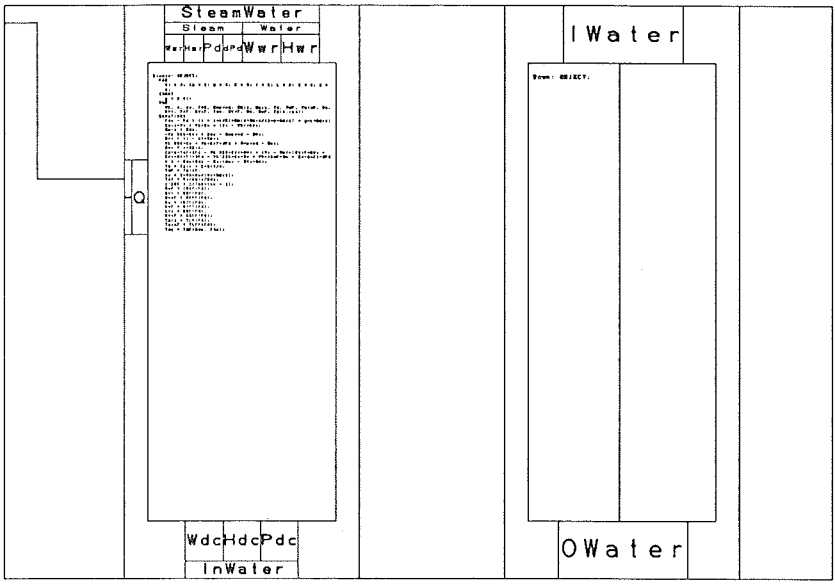
The editor changes are marked in the following way in the Dymola model of Appendix 5.

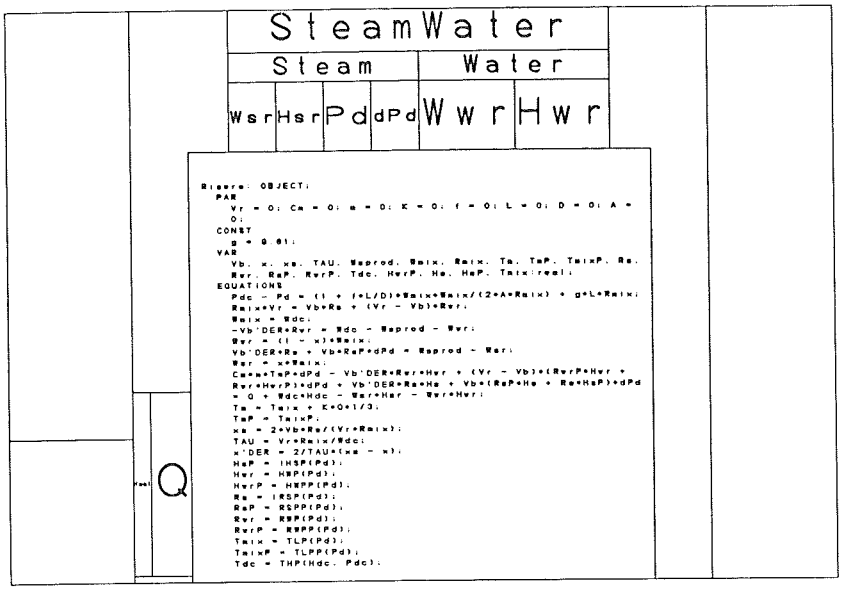
{ *deleted* } *inserted* {}


Powersystem: SYSTEM;
Heaters
SteamValve
Turbines
Combustion
Boiler
FeedValve
FeedWater

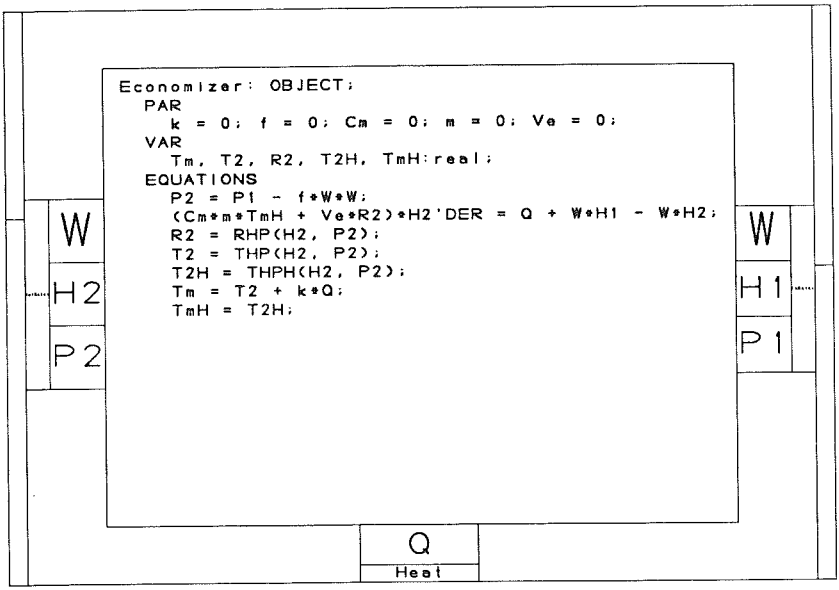

OSteam
Boiler OBJECT.
Drum
Economizer
Q1
Q2
IWater
Risers
Down


OutSteam
Ws Hs Pd
Drum: OBJECT.
PAR
Vdc = 0; Adrum = 0; f = 0; L = 0; D = 0; A = 0; Vw0 =
0; Vs0 = 0; Aw = 0.
CONST
g = 9.81;
VAR
z, Rw, Vw, Vs, HsP, Rs, RsP, Rwr, Hd real;
EQUATIONS
Pd - Pdc = (1 + f*L/D)*Wdc*Wdc/(2*A*Rw) - g*L*Rw;
Aw*z'DER*Rw = Ww + Wwr - Wdc;
Vw = Vw0 + Adrum*z;
(Vw + Vdc)*Rw*Hd'DER = Ww*Hw + Wwr*Hwr - Wdc*Hdc;
Hdc = Hd;
-Adrum*z'DER*Rs + Vs*RsP*Pd'DER = Wsr - Ws;
Vs = Vs0 - Adrum*z;
-Adrum*z'DER*Rs*Hs + Vs*(RsP*Hs + Rs*HsP)*Pd'DER = Wsr*
Hsr - Ws*Hs;
dPd = Pd'DER;
Hs = IHSP(Pd);
HsP = HSPP(Pd);
Rw = RHP(Hd, Pd);
Rs = IRSP(Pd);
RsP = RSPP(Pd);
Rwr = RWP(Pd);
Ww
Hw
Pd
Wsr Hsr Pd dPd Wwr Hwr
Steam Water
SteamWater
Wdc Hdc Pdc
OutWater

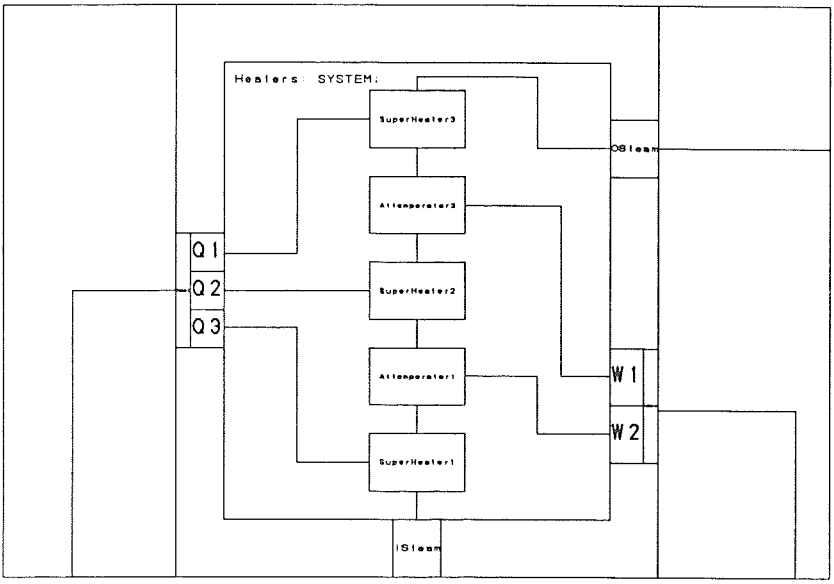

SteamWater
Steam Water
Wsr Hsr Pd dPd Wwr Hwr
Q
Wdc Hdc Pdc
InWater
IWater
Down: OBJECT.
OWater

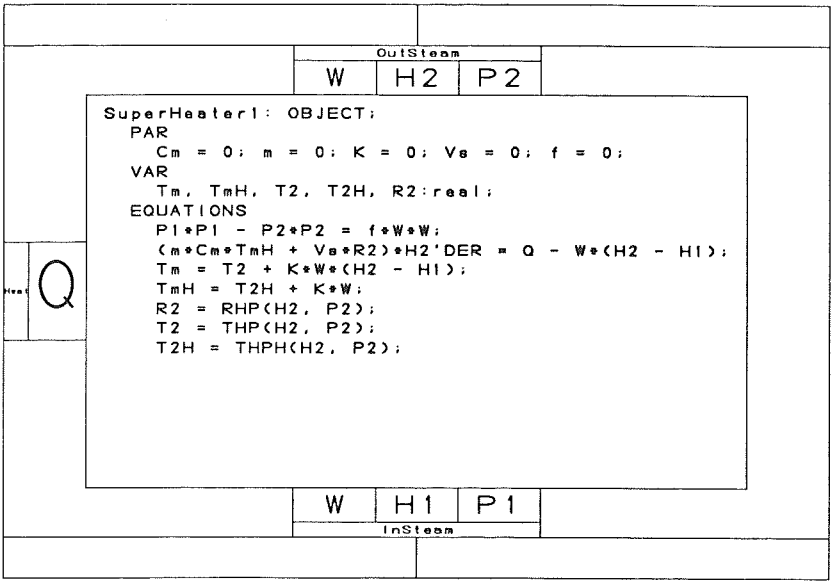


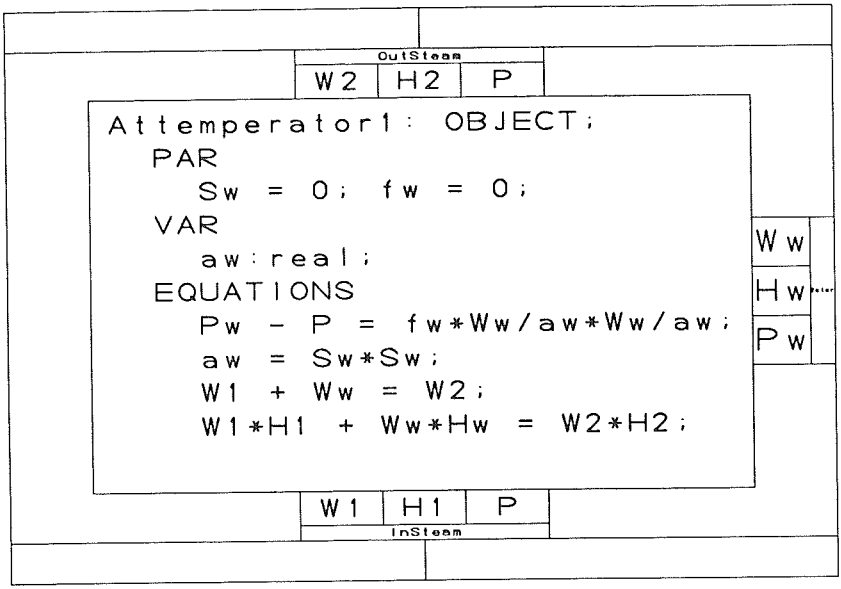

```
Riserre: OBJECT;
  PAR
    Vr = 0; Cm = 0; m = 0; K = 0; f = 0; L = 0; D = 0; A =
    0;
  CONST
    g = 9.81;
  VAR
    Vb, x, xs, TAU, Wsprod, Wmix, Rmix, Tm, TmP, TmixP, Rs,
    Rwr, RsP, RwrP, Tdc, HwrP, Hs, HsP, Tmix:real;
  EQUATIONS
    Pdc - Pd = (1 + f*L/D)*Wmix*Wmix/(2*A*Rmix) + g*L*Rmix;
    Rmix*Vr = Vb*Rs + (Vr - Vb)*Rwr;
    Wmix = Wdc;
    -Vb'DER*Rwr = Wdc - Wsprod - Wwr;
    Wwr = (1 - x)*Wmix;
    Vb'DER*Rs + Vb*RsP*dPd = Wsprod - Wsr;
    Wsr = x*Wmix;
    Cm*m*TmP*dPd - Vb'DER*Rwr*Hwr + (Vr - Vb)*(RwrP*Hwr +
    Rwr*Hwr(P))*dPd + Vb'DER*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;
    x'DER = 2/TAU*(xs - x);
    HsP = IHSP(Pd);
    Hsr = HSP(Pd);
    HwrP = HWPP(Pd);
    Hs = IHSP(Pd);
    RsP = RSPP(Pd);
    Rwr = RWP(Pd);
    RwrP = RWPP(Pd);
    Tmix = TLP(Pd);
    TmixP = TLPP(Pd);
    Tdc = THP(Hdc, Pdc);
```

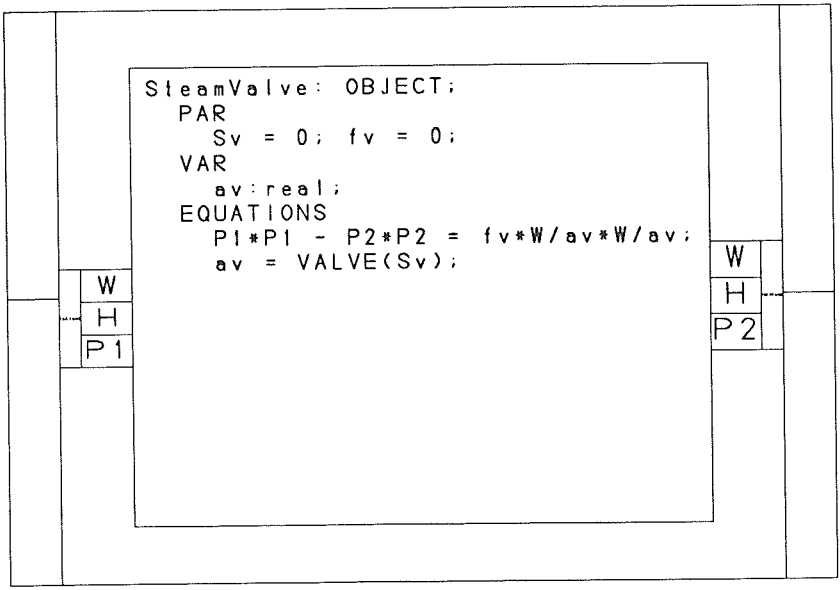


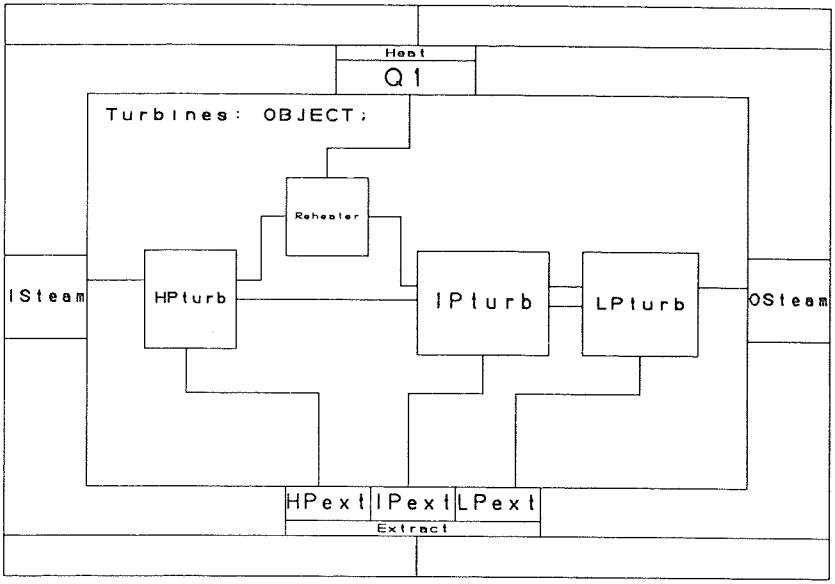

```
Economizer: OBJECT;
  PAR
    k = 0; f = 0; Cm = 0; m = 0; Ve = 0;
  VAR
    Tm, T2, R2, T2H, TmH:real;
  EQUATIONS
    P2 = P1 - f*W*W;
    (Cm*m*TmH + Ve*R2)*H2'DER = Q + W*H1 - W*H2;
    R2 = RHP(H2, P2);
    T2 = THP(H2, P2);
    T2H = THPH(H2, P2);
    Tm = T2 + k*Q;
    TmH = T2H;
```

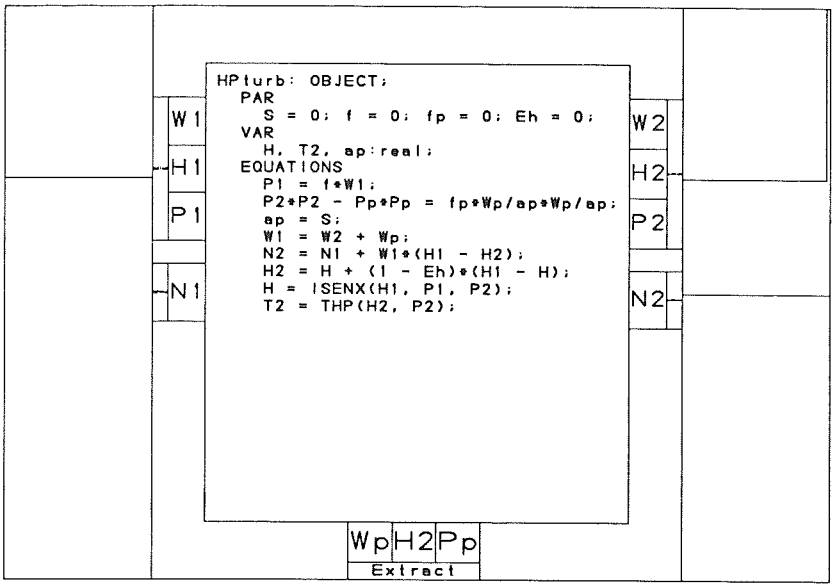

Heaters: SYSTEM;
SuperHeater3
Attemperator3
SuperHeater2
Attemperator1
SuperHeater1
Q1
Q2
Q3
OSteam
W1
W2
ISteam

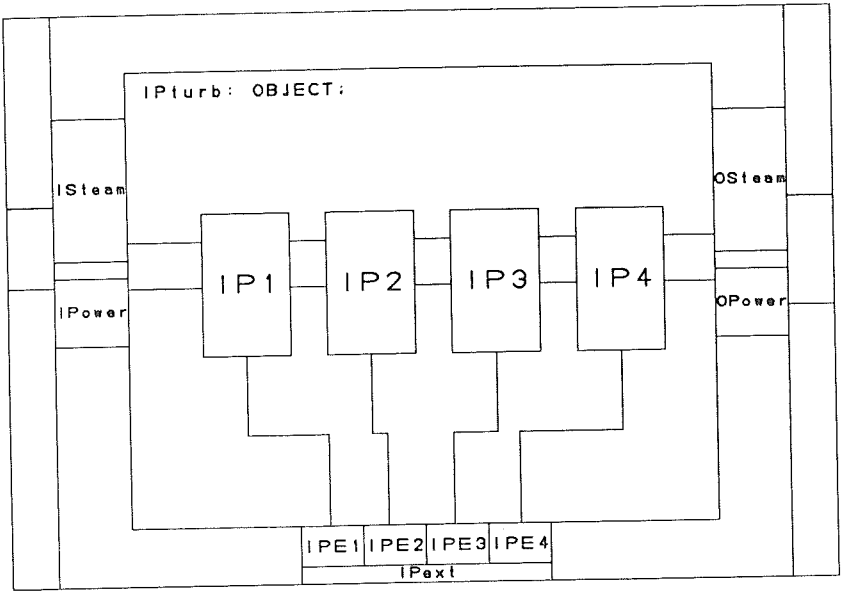

OutSteam
W
H2
P2
SuperHeater1: OBJECT;
PAR
Cm = 0; m = 0; K = 0; Vs = 0; f = 0;
VAR
Tm, TmH, T2, T2H, R2:real;
EQUATIONS
P1*P1 - P2*P2 = f*W*W;
(m*Cm*TmH + Vs*R2)*H2'DER = Q - W*(H2 - H1);
Tm = T2 + K*W*(H2 - H1);
TmH = T2H + K*W;
R2 = RHP(H2, P2);
T2 = THP(H2, P2);
T2H = THPH(H2, P2);
Q
W
H1
P1
InSteam

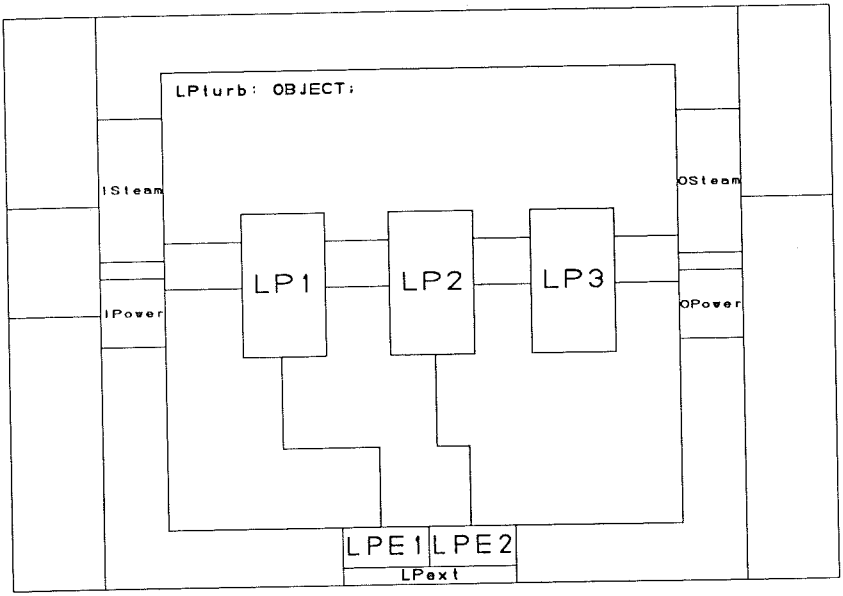

OutSteam
W2
H2
P
Attemperator1: OBJECT;
PAR
Sw = 0; fw = 0;
VAR
aw:real;
EQUATIONS
Pw - P = fw*Ww/aw*Ww/aw;
aw = Sw*Sw;
W1 + Ww = W2;
W1*H1 + Ww*Hw = W2*H2;
Ww
Hw
Pw
W1
H1
P
InSteam

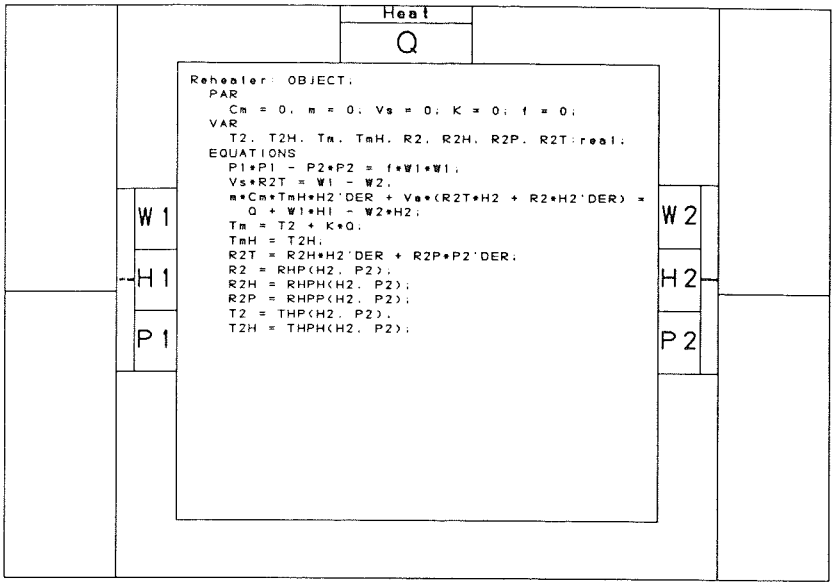

SteamValve: OBJECT;
PAR
Sv = 0; fv = 0;
VAR
av:real;
EQUATIONS
P1*P1 - P2*P2 = fv*W/av*W/av;
av = VALVE(Sv);
W
H
P1
W
H
P2

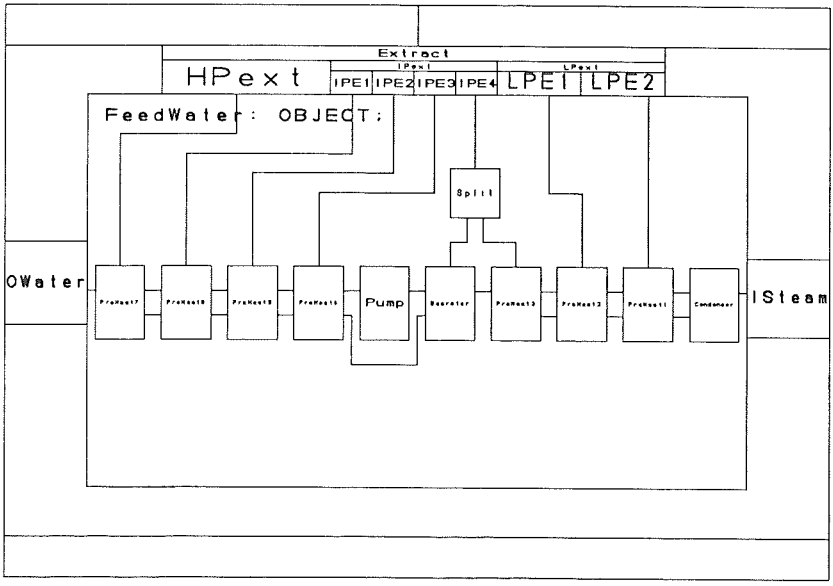

Heat
Q1
Turbines: OBJECT;
Reheater
ISteam
HPturb
IPturb
LPturb
OSteam
HPext
IPext
LPext
Extract

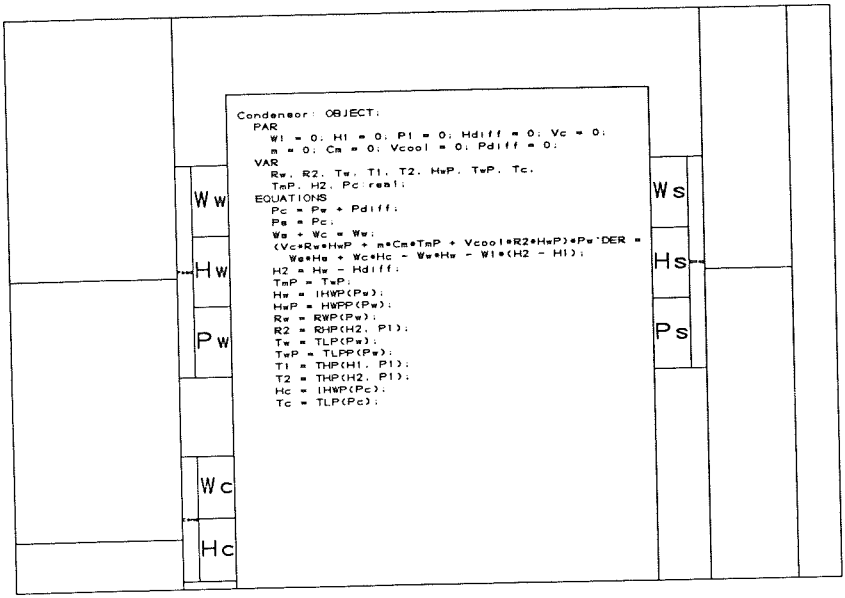

HPturb: OBJECT;
PAR
S = 0; f = 0; fp = 0; Eh = 0;
VAR
H, T2, ap:real;
EQUATIONS
P1 = f*W1;
P2*P2 - Pp*Pp = fp*Wp/ap*Wp/ap;
ap = S;
W1 = W2 + Wp;
N2 = N1 + W1*(H1 - H2);
H2 = H + (1 - Eh)*(H1 - H);
H = ISENX(H1, P1, P2);
T2 = THP(H2, P2);
W1
H1
P1
N1
W2
H2
P2
N2
Wp
H2
Pp
Extract

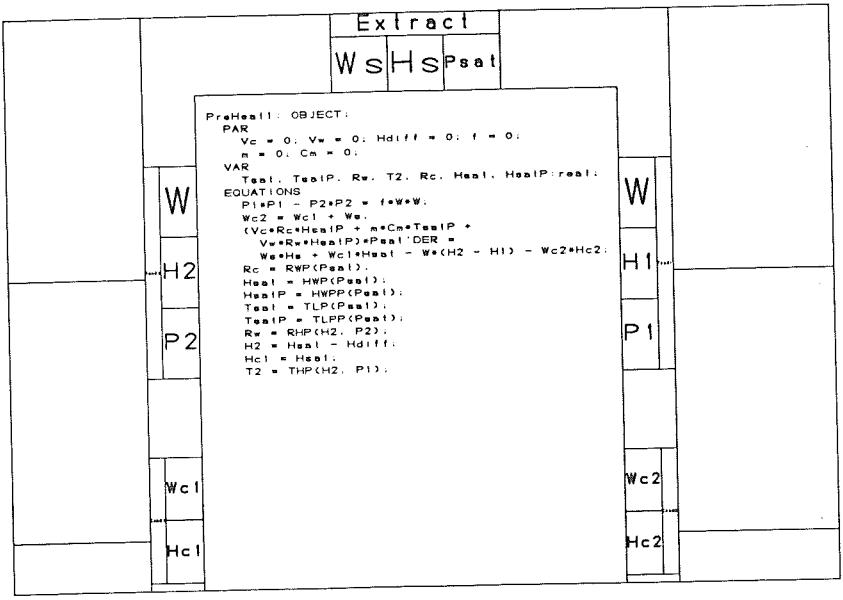

IPturb: OBJECT;
ISteam
IPower
IP1
IP2
IP3
IP4
OSteam
OPower
IPE1
IPE2
IPE3
IPE4
IPext

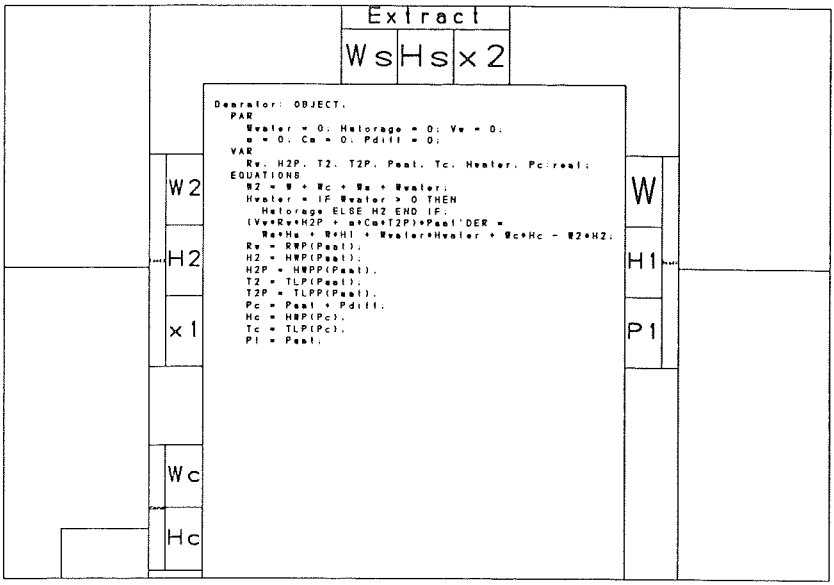

LPturb: OBJECT;
ISteam
IPower
LP1
LP2
LP3
OSteam
OPower
LPE1
LPE2
LPext

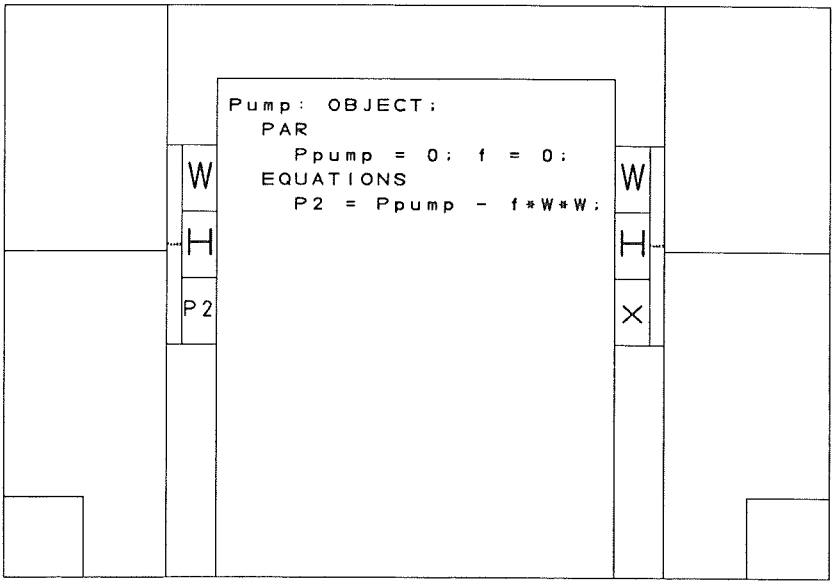

Heat
Q
W1
H1
P1
W2
H2
P2
Reheater: OBJECT;
PAR
Cm = 0; m = 0; Vs = 0; K = 0; f = 0;
VAR
T2, T2H, Tm, TmH, R2, R2H, R2P, R2T:real;
EQUATIONS
P1*P1 - P2*P2 = f*W1*W1;
Vs*R2T = W1 - W2;
m*Cm*TmH*H2'DER + Vs*(R2T*H2 + R2*H2'DER) =
Q + W1*H1 - W2*H2;
Tm = T2 + K*Q;
TmH = T2H;
R2T = R2H*H2'DER + R2P*P2'DER;
R2 = RHP(H2, P2);
R2H = RHPH(H2, P2);
R2P = RHPP(H2, P2);
T2 = THP(H2, P2);
T2H = THPH(H2, P2);

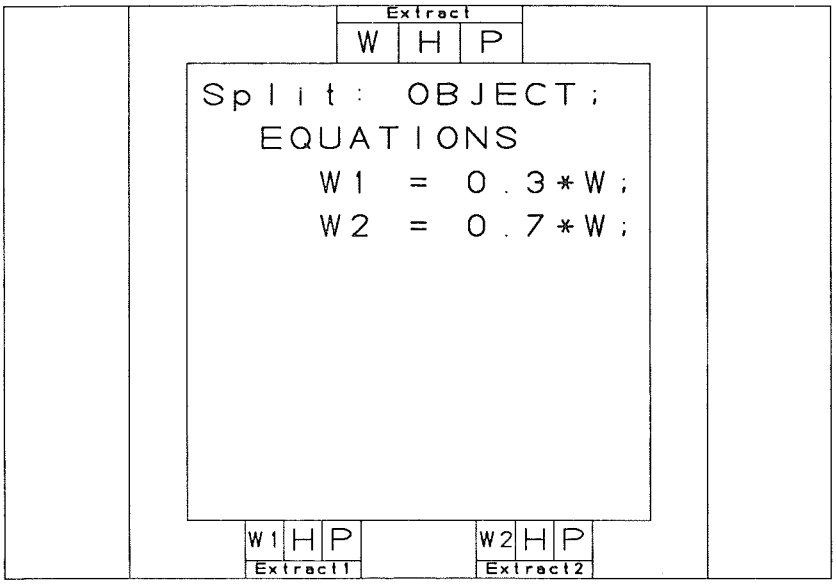

Extract
HPext
IPext
LPext
IPE1
IPE2
IPE3
IPE4
LPE1
LPE2
FeedWater: OBJECT;
Split
OWater
Pump
Deaerator
Condenser
ISteam

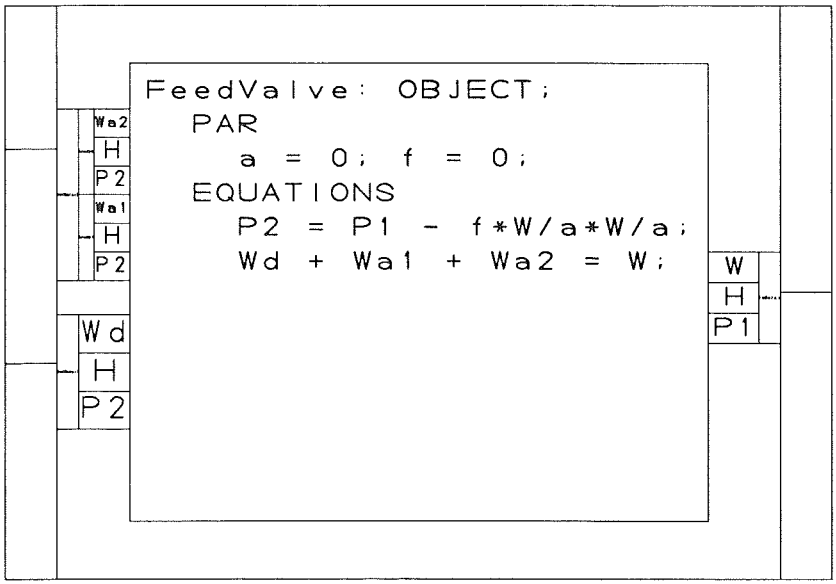

Ww
Hw
Pw
Wc
Hc
Ws
Hs
Ps
Condenser: OBJECT;
PAR
W1 = 0; H1 = 0; P1 = 0; Hdiff = 0; Vc = 0;
m = 0; Cm = 0; Vcool = 0; Pdiff = 0;
VAR
Rw, R2, Tw, T1, T2, HwP, TwP, Tc,
TmP, H2, Pc:real;
EQUATIONS
Pc = Pw + Pdiff;
Ps = Pc;
Ws + Wc = Ww;
(Vc*Rw*HwP + m*Cm*TmP + Vcool*R2*HwP)*Pw'DER =
Ws*Hs + Wc*Hc - Ww*Hw - W1*(H2 - H1);
H2 = Hw - Hdiff;
TmP = TwP;
Hw = IHWP(Pw);
HwP = IHWPP(Pw);
Rw = RWP(Pw);
R2 = RHP(H2, P1);
Tw = TLP(Pw);
TwP = TLPP(Pw);
T1 = THP(H1, P1);
T2 = THP(H2, P1);
Hc = IHWP(Pc);
Tc = TLP(Pc);

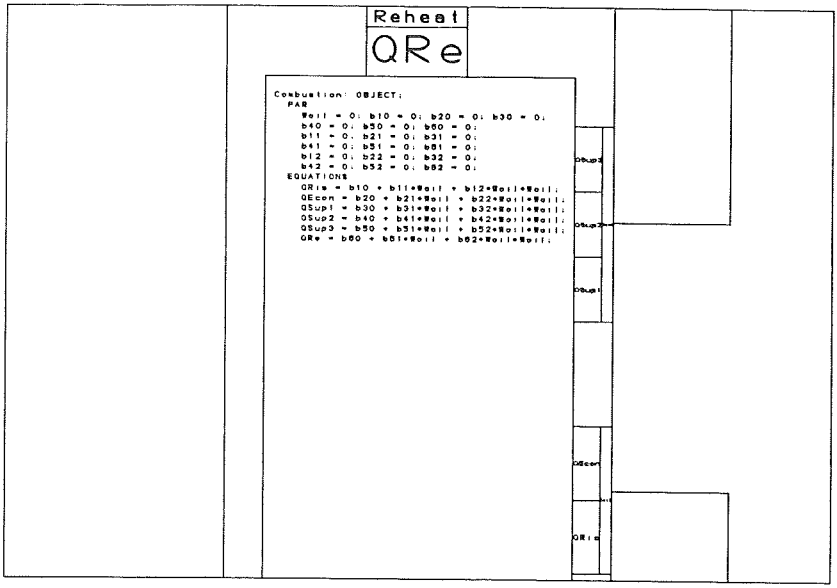

Extract
Ws
Hs
Psat
W
H2
P2
Wc1
Hc1
W
H1
P1
Wc2
Hc2
PreHeat1: OBJECT;
PAR
Vc = 0; Vw = 0; Hdiff = 0; f = 0;
m = 0; Cm = 0;
VAR
Tsat, TsatP, Rw, T2, Rc, Hsat, HsatP:real;
EQUATIONS
P1*P1 - P2*P2 = f*W*W;
Wc2 = Wc1 + Ws;
(Vc*Rc*HsatP + m*Cm*TsatP +
Vw*Rw*HsatP)*Psat'DER =
Ws*Hs + Wc1*Hsat - W*(H2 - H1) - Wc2*Hc2;
Rc = RWP(Psat);
Hsat = HWP(Psat);
HsatP = HWPP(Psat);
Tsat = TLP(Psat);
TsatP = TLPP(Psat);
Rw = RHP(H2, P2);
H2 = Hsat - Hdiff;
Hc1 = Hsat;
T2 = THP(H2, P1);


Extract
Ws
Hs
x2
W2
H2
x1
Wc
Hc
W
H1
P1
Deaerator: OBJECT.
PAR
Wwater = 0; Hstorage = 0; Vw = 0;
m = 0; Cm = 0; Pdiff = 0;
VAR
Rw, H2P, T2, T2P, Psat, Tc, Hwater, Pc:real;
EQUATIONS
WZ = W + Wc + Wm + Wwater;
Hwater = IF Wwater > 0 THEN
Hstorage ELSE H2 END IF;
(Vw*Rw*H2P + m*Cm*T2P)*Psat'DER =
Wm*Hm + W*H1 + Wwater*Hwater + Wc*Hc - WZ*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;


Pump: OBJECT;
PAR
Ppump = 0; f = 0;
EQUATIONS
P2 = Ppump - f*W*W;
W
H
P2
W
H
X


Extract
W H P
Split: OBJECT;
EQUATIONS
W1 = 0.3*W;
W2 = 0.7*W;
W1 H P
Extract1
W2 H P
Extract2


FeedValve: OBJECT;
PAR
a = 0; f = 0;
EQUATIONS
P2 = P1 - f*W/a*W/a;
Wd + Wa1 + Wa2 = W;
Wa2
H
P2
Wa1
H
P2
Wd
H
P2
W
H
P1


Reheat
QRe
Combustion: OBJECT;
PAR
Wall = 0; b10 = 0; b20 = 0; b30 = 0;
b40 = 0; b50 = 0; b60 = 0;
b11 = 0; b21 = 0; b31 = 0;
b41 = 0; b51 = 0; b61 = 0;
b12 = 0; b22 = 0; b32 = 0;
b42 = 0; b52 = 0; b62 = 0;
EQUATIONS
QRis = b10 + b11*Wall + b12*Wall*Wall;
QEcon = b20 + b21*Wall + b22*Wall*Wall;
QSup1 = b30 + b31*Wall + b32*Wall*Wall;
QSup2 = b40 + b41*Wall + b42*Wall*Wall;
QSup3 = b50 + b51*Wall + b52*Wall*Wall;
QRe = b60 + b61*Wall + b62*Wall*Wall;
QSup3
QSup2
QSup1
QEcon
QRis
QRe

5. Graphical Representation of Program Structures

Personal computers

The introduction of powerful personal computers are changing the man-machine interface considerably. Graphics can now be used in all kinds of applications to increase the "communication bandwidth" between the user and the computer.

The decreasing cost of memory has made high resolution raster graphics systems feasible for personal computers. Generation of raster pictures requires, however, a lot of computing power. Powerful micro processors in the personal computers are thus needed. The raster memory is often directly accessed by the micro processor ("bit-mapped display"). High resolution color monitors (such as 1280 × 1024 pixels) are now available at low prices due to the inline-gun technology.

The power of general purpose micro processors are not enough for giving good interactive response on displays having such high resolutions. Special purpose VLSI circuits are designed to offload the micro processor. An example is the Hitachi HD 63484 ACRTC graphics chip (Hitachi, 1984). It scan converts vectors, circles etc. and fills areas. It also performes raster operations (so called BitBlt's) such as copying a rectangular area on the screen from one place to another. Even higher bandwidth to the image memory is obtained by use of distributed controllers for each memory chip (Clark, Hannah, 1980b). The Geometry Engine chips from Silicon Graphics Inc (Clark, 1980a) is another example. It is a pipe-lined architecture mainly for handling coordinate systems, perspective transformations and clipping. It thus performs floating point operations on point coordinates at high speed.

The discussion above indicates how the developments and price reductions on micro processors, graphical processors, raster memory and monitors, i.e. the major components of personal computers, matches. We may thus expect major performance increases in the next generation personal computers.

There will be great possibilities to develop new kinds of man-machine interfaces which will enable computer users to work much more efficiently in their problem solving.

Desk-top model

Early developments on personal computers and new interaction techniques were done at Xerox PARC (Kay, Goldberg, 1977). It resulted in the "screen as

a desk-top"-metaphor. "Windows" may be placed on the screen in the same way as notes, reports, books etc. are placed on one's desk. The windows can contain different types of information such as part a of document being edited, transcript of a dialogue with a program or graphical output. A *window manager* makes it possible to move windows, change their size and make a partially obscured window appear on the top.

The desk-top model is related to the "data as concrete objects"-metaphor. The graphical ability of personal computers makes it possible to develop less abstract man-machine interfaces. Abstract concepts can be made "more concrete" by associating graphical pictures, *icons* with them. However, there must be a facility to manipulate them as well. A "mouse" is often used as a pointing device. It is used to select objects for manipulation, to select commands from menus and to change layout.

These metaphores are being popularized by the Lisa and Macintosh personal computers (Williams, 1983). One example is the representation of tree structured directories. Files have icons that indicate what type of information they contain. Folders correspond to sub-directories. A folder can be inspected by pointing to its icon. A new overlapping window is then created containing icons for files and sub-folders contained in the folder.

Graphical structure space

A problem with the desk-top model is that it has a poor support for structure. The screen will often look like a mess (just like a real desk-top). Reaching a file at a low hierarchical level in the Macintosh folder structure will create many partially overlapping windows.

Abstraction and modularization are two basic principles for dealing with complexity as discussed earlier. Graphical representations for these should therefore be designed. A "graphical structure space" is proposed. The idea is to create the illusion that the user maneuvers in this space in order to find the desired information. A major difference compared to the desk-top model is that the display is used to show consistent views into the structure space instead of flashing windows on the display.

The illusion of movement in the structure space is emphasized by using animation. Double frame buffers are used. A new frame is created invisibly and then switched in. Several pictures are created each second. When the user views a certain piece of information, the screen looks the same independently of how he maneuvered to get there. The animation and the consistent views enables the user to build a *mental model* of the structure space. This makes selection

of information more efficient.

There are several issues involved in creating a graphical structure space and views into it. The following issues will be discussed below.

- limited viewing window
- representation of modularization
- representation of abstraction
- representation of directed graph structures

Limited viewing window

The video monitor gives a fairly narrow window through which the structure is viewed. This makes it hard to maneuver because there is no information about what is outside the window. This situation should be compared with viewing the real world. In this case the rods in our eyes give us a grey-scale diffuse (unfocused) image of the areas around the viewed object.

One way of dealing with this situation is to use two windows: an *overview window* and a *detail window*. The overview window shows the whole picture and an indication (such as a rectangle) of what part is shown in detail. Figure 5.1 illustrates this situation.

The information about what part of a picture that is currently shown can be represented by scroll bars. They can be seen as projections of a view area rectangle onto the horizontal and vertical axes (Fig. 5.2). The scroll bars use little space and the detail window may occupy almost the whole screen. The scroll bars are also convenient mechanisms for maneuvering, c.f. the Macintosh environment.

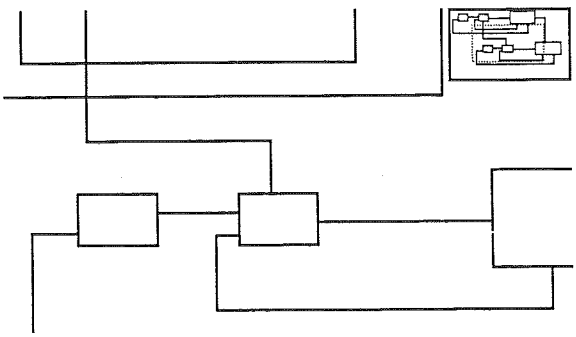


Fig. 5.1. Overview window.

Another way of "getting the whole picture" at the same time as details is to use a *magnifying glass*. A rectangular area around the cursor would be enlarged (Fig. 5.3). The picture could thus be scanned by moving the mouse.

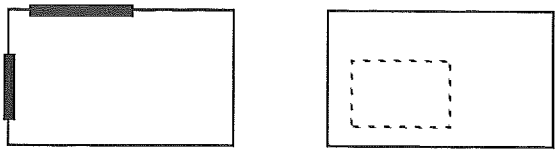


Fig. 5.2. View area rectangle and scroll bars.

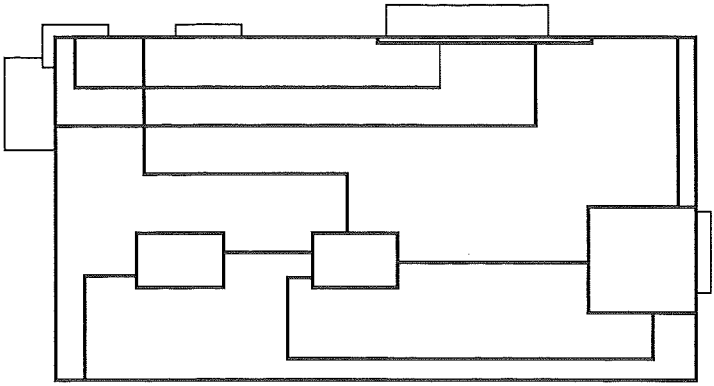


Fig. 5.3. Magnifying glass.

A view similar to our "indirect viewing" could be generated giving both overview and detail. The idea is to compress the parts of the picture outside the area of primary interest, in order to make them fit on the screen. The areas to the left and right of the primary region will thus have reduced scale in the horizontal direction. The top and bottom areas will be reduced in the vertical direction and the areas outside the corners will be reduced in both directions. Figure 5.4 gives an illustration of such a *distorted boundary view*.

It may be natural to show these boundary parts in "dimmed" colors. The distorted picture can be generated as 3×3 subpictures by applying different scaling, translation and clipping.

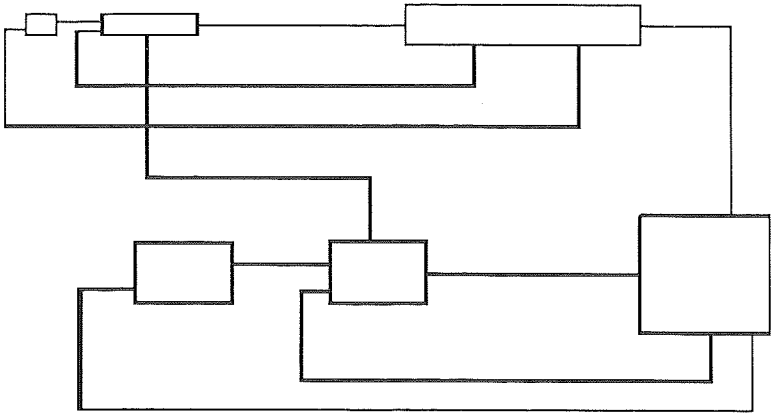


Fig. 5.4. Distorted boundary view.

A powerful property of our human vision system is its ability to select different areas of primary interest very rapidly. This is important when information from different parts of a picture has to be collected repeatedly in order to make decisions. The general maneuvering mechanisms will be too slow in such cases. The different views need to be present on the screen simultaneously in order that rapid selection can be made with eye movements. *Multiple windows*, in addition to the overview window, gives a solution to this problem.

Graphical representation of modularization

A basic property of any system is its decomposition into smaller subsystems. This property is conveniently illustrated by a block diagram. It also shows how the different subsystems are connected together physically or by data flows. Chapter 4 gives many examples of block diagrams.

Graphical representation of abstraction

Levels of abstraction should have graphical representations that correspond to a mental model. Information hiding, i.e. hiding inessential details, is a key function.

A mental model for levels of abstraction might be large screens standing behind each other. To reach information at a more detailed abstraction layer would mean to go behind the screen currently viewed.

A convenient method for maneuvering "behind a screen" is needed. One method is to think of the graphical structure space as three-dimensional. Moving towards a screen would give an enlarged view, c.f. zooming, and actually penetrating the screen would reveal the information on the layer. The method

does not work well when there are many layers to penetrate because to much irrelevant information is presented to the viewer during a selection.

However, it feels natural to penetrate simple abstraction layers having just a name or an icon. They thus appear to open-up, during zoom-in, revealing what is behind.

Another selection mechanism is needed for the case of many levels. One idea is to think about the abstraction layers as register cards instead of screens. Register cards have flaps used for selection (Fig. 5.5). The register card method is used on the personal computer HP-150 (Lemmons, Robertson, 1983). It was tested in a master thesis (Lerup, 1984) concerned with design of a graphical environment for Ada programs. Packages were represented by four abstraction levels: name, glossary of procedures, specification and body.

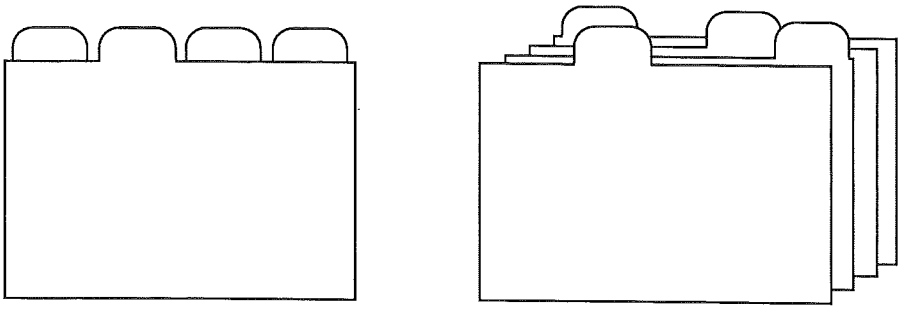


Fig 5.5. Register cards for abstraction (view and model).

Maneuvering in graphical structure space

Objects have a graphical representation with a rectangular shape. The general graphical rule is that objects having the same father are layed out without overlapping each other, inside the rectangle of their father.

Each object has at least two views: a remote view and a complete view. When an object's size on the screen is below a certain limit, it is represented by its remote view. The remote view does not include subrelated objects. It is typically just a name or an icon. (This is a way of stopping the recursive display algorithm from going deeper in the structure. It is particularly important since the structure might in general be circular!)

When the object is zoomed-in it opens-up to show the complete view. Zooming, panning and scrolling are the basic maneuvers for browsing in the structure. A natural way to "approach" an object is to just point at it, thereby requesting an automatic zoom-in.

The graphical presentation discussed is completely general. Any program structure or data structure can be represented. It thus covers both static and dynamic structures.

Moving in cyclic structures is somewhat peculiar. If we just continue to zoom along the object references we eventually come back to the same object. However, in a different environment. We are now in the environment of the preceding object. Different environments exist when several references point to the same object. If we are moving in a double linked circular list, it is possible to start going backwards from any node by just panning over to the backward pointer and zooming-in.

There should be a general mechanism to select another path when zooming back. It could be done by using a pop-up menu containing all the objects having references to the actual object. There would be one for each path from the root object.

The graphical representation chosen has the merits of both showing the "belong to" and "connected to" relations. It may, however, be better to show non-communicating objects, in data structures such as linked lists, trees or graphs, with the objects at the same level and arrows to represent object references. Generally, it would be natural to have this representation for any object and its subrelated objects. The structure would then be generated according to a standard layout. Such a representation could be based on a tree generated by a depth-first search.

It should also be possible to quickly toggle over from a representation of an object to a representation of its object type. Being able to follow a connection would also be helpful. It could be done with a menu in a similar way to selecting another trace back.

Spatial information management

The spatial approach for information management has similarities with how a desk is organized. It is also related to the use of control panels for operator communication. The main advantage is that it is easy to find information. Our computerized approach also handles *hierarchically* structured information by use of *information zooming*. We think, however, it is essential that the information on the screen is changed as smoothly as possible, so that the user doesn't get lost. The information zooming is therefore combined with continuous zooming in real time.

The spatial approach to handling of information has been used in other applications as well, for example in the Spatial Data Management System

(SDMS) at MIT (Donelson, 1978) and in a management system for ships (Herot, Carling, Friedell, 1980). Other related works are the TELL system (Rhyne, Richter, Carlson, 1981) and an integrated system for program visualization (Herot, et al, 1982). Feiner et al (1982) describes an interactive system for structured documents composed of text and graphics.

6. Interaction

This chapter describes how the LICS prototype program is operated.

Windows

When LICS is initialized, one window appears on the screen. It will contain the graphical representation of the control system. Several windows of this type can be created by using the command VIEW. The layout of the windows on the screen can be changed by the command LAYOUT (see section Commands). A module also acts as a window to its sub-structure. The inner rectangle is then the window frame. There is thus an hierarchical structure of windows.

The contents of the windows can be zoomed, panned and scrolled by use of joysticks or mouse (see below). Rectangles outline the parts of a window that also appear in other windows (c.f. Fig. 5.1).

Another type of window called "interaction window", appears at the bottom of the screen when the system prompts for keyboard input and to give instructions. Figure 6.1 shows a possible layout on the screen including several windows and the interaction window.

The definition of colors and their use are described in the set-up file LICSCOL. Colors are defined using the HSV-model (hue, saturation, value) (Foley, Van Dam, 1982).

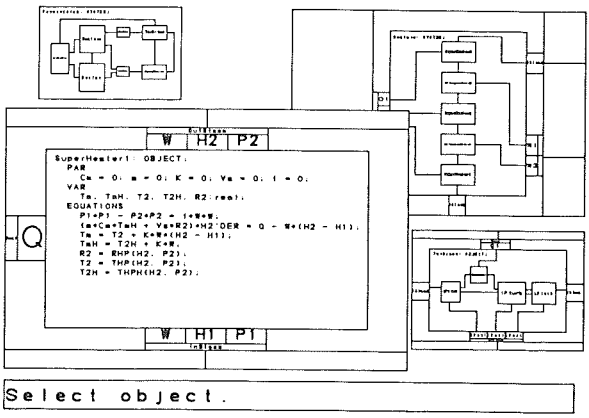


Fig. 6.1. Example of window layout.

Mouse

A mouse is used for pointing "on the screen". A cursor follows the movements of the mouse. The mouse has three buttons (LEFT, MIDDLE, RIGHT) for selection of actions.

MENU SELECTION – LEFT

A pop-up menu is shown when the LEFT button is pressed and released. The desired command is selected by pointing at the corresponding menu entry and pressing and releasing the LEFT button again. The command alternatives are described in Section Commands. If no menu entry or the frame of the menu is selected the menu just disappears. Command actions sometimes require additional use of the mouse which are prompted for in the interaction window. There are three cases.

SELECTION – LEFT
Objects on the screen are selected by pointing at them and pressing and releasing the LEFT button.

RUBBER-BANDING – LEFT
Connections are drawn by rubber-banding (see command CONNECT).

STRETCHING – LEFT, MIDDLE
The layout of objects (position, size) on the screen is done by "Two-button-stretching". The lower left corner and the upper right corner of the object follows the mouse in different ways depending on which of the LEFT and MIDDLE buttons are pressed.

LEFT – lower left corner moves
MIDDLE – upper right corner moves
LEFT and MIDDLE – the whole object moves (both corners)

The cursor points to the lower left corner, the upper right corner or the center of the object respectively. The stretching is finished when both buttons are released.

AUTOMATIC ZOOMING – MIDDLE

An object (module, interface, curve or text) can be zoomed-in by pointing at it and pressing the MIDDLE button. The zooming is done smoothly in several

steps and the final size of the object is chosen as large as possible while still being contained in the "zoom window" (see below).

Text objects are treated specially. Their final size is chosen so that the longest line of the text matches the width of the zoom window. The text line pointed at is scrolled to the center of the zoom window. This allows convenient scrolling within a window.

A connection is considered to belong to the enclosing module. Pointing at a connection is therefore a convenient way of zooming out to a higher hierarchical level.

If the MIDDLE button is pressed without pointing at anything, a zoom-out to 70 % is performed.

Note, that it is possible to point to an object in a window other than the current zoom window. Pointing in an overview window thus allows rapid inspection of different objects.

PANNING - SCROLLING – RIGHT

Pressing the RIGHT button means freezing the cursor in the underlying picture. This allows panning and scrolling. There are two submodes:

ZOOMING – MIDDLE

When the MIDDLE button is pressed the cursor acts as a control for a sliding potentiometer for the zoom factor. Moving the cursor up means zooming-in.

ZOOM WINDOW SELECTION – LEFT

Zooming, scrolling and panning can be performed in one window at a time, the *zoom window*. The zoom window is selected by using the RIGHT button followed by the LEFT button. One of the main windows (the initial one or windows created by the VIEW command) or a module window can be selected as zoom window.

A zoom window is selected by pointing to an object inside a window. The window at the hierarchically lowest level is then chosen. However, if that window is already the current zoom window, the window at the next higher hierarchical level is chosen. This allows going back to zooming in windows at higher levels even if their contents are not shown on the screen.

Joysticks

Two two-axis joysticks are used for zooming, scrolling and panning.

RIGHT JOYSTICK

X-axis: pan (move the contents of the zoom window horizontally)

Y-axis: scroll (move the contents of the zoom window vertically)

LEFT JOYSTICK

X-axis: information zoom (change the size when modules open up)

Y-axis: zoom (zoom the contents of the zoom window)

Keyboard

The keyboard is used mainly for replying to prompts and for editing of text. However, all commands selectable from the menu can be given as control characters from the keyboard. The correspondence between control characters and menu commands is given in Table 6.1.

Module	^M
Copy	^X
Get	^G
Save	^S
Dymola	^P
Interface	^I
Connect	^C
Text	^T
Change	^A
Run	^R
Remove	^D
Display	^Z
View	^V
Layout	^L
HardCopy	^H
Exit	^E

Table 6.1. Control characters for commands.

Several interactive features have been tested, of which some are available as options. These are selected by first giving an option command (^O) followed by a control character according to Table 6.2.

The following features are available via control characters:

^H	This help information.
^P	Pop zoom window.
^V	Toggle view orientation mode.
^T	Toggle text information zooming mode.
^O	Toggle window overlapping mode.
^L	Toggle layer open-up mode.
^F	Toggle fast zoom mode.
^Z	Start pure zoom.
^I	Toggle instant zooming.
^D	Toggle drag mode.
^X	Toggle zoom selection - zooming
^C	Make hardcopy on laser printer

Table 6.2. Option (^O) control characters.

Commands

Commands are chosen from a pop-up menu as explained previously.

Module

Module

The MODULE command creates a module and its graphical representation (two rectangles and the first text line). The first text line is prompted for.

Module:

The answer should have the syntax of ModuleHeading. If no module has been defined before, this will be the main module. It is then layed out to cover the whole window. Otherwise the father module is prompted for.

Select superior module.

The superior module is selected by pointing to its outer or inner frame or to its text object and pressing button LEFT. The next action which is prompted for is

Stretch module.

The layout is done by "Two-button-stretching" as described previously. The stretching is restricted to the area defined by the window of the superior module. The error messages during handling of the module heading are in this implementation given on the text terminal. The interaction could be improved by making the program construct the module heading after first prompting for name and then allowing selection of module type from a pop-up menu.

Copy

Copy

It copies a module and all its submodules, interfaces, connections and the text. The module to be copied is selected by using button LEFT after the prompt:

Select module.

The hierarchical level for the new copy is then requested.

Select superior module.

The name of the copy is then given.

Module name:

The layout is then done in the same way as for the command MODULE.

Stretch module.

Get

Get

It reads a module description from a text file according to the syntax of Appendix 1. The file name is prompted for.

File name:

The hierarchical position and layout is then given in the same way as for MODULE if a system has been defined earlier.

Select superior module.

Stretch module.

Save

Save

The SAVE command stores the current module hierarchy on a file whose name is prompted for.

File name:

The system is stored as text in a format compatible with GET. An example is shown in Appendix 3.

Dymola *Dymola*

The DYMOLA command outputs the current object hierarchy to a file.

File name:

The format is compatible with the input format to the Dymola program (Elmqvist, 1978). Appendix 4 contains an example of this format and section 6.3 discusses restrictions.

Interface *Interface*

The INTERFACE command is used to declare and position an interface. The declaration is currently done textually with the syntax of InterfaceDeclaration.

Interface:

The positioning of the interface starts by selecting the module it should belong to, by pointing to its inner frame and pressing LEFT.

Select module (inner frame).

The layout is then done within one of the four interface areas using a "Two Button Stretch".

Stretch interface.

The implementation does not contain any facility to edit or change the layout of interfaces. However, they can be deleted.

Connect *Connect*

Connections between interfaces are created using the mouse and the LEFT button to input a sequence of line segments.

Draw connection.

A connection belongs to a module, starting and ending either at interface primitives of the module (inner frame) or at interfaces of submodules (outer

frame). The start interface is first selected by pressing LEFT. Break points are given by releasing and pressing LEFT. The last line segment is refreshed while the mouse is moved (rubber-banding) and the interface structure is searched for the destination interface.

Text	*Text*

The TEXT command is used to edit the textual parts of modules. The text is selected using the mouse.

Select text

Edit text

The editing is performed using a subset of a screen oriented editor called TEVE (Andersson et al, 1984). Table 6.3 gives a short description of the editing commands (this help information is available by ^H).

```
   LICS SCREEN EDITOR - CONTROL CHARACTER COMMANDS
Ctrl-R  Remove char under cursor    DEL     Delete char left of cursor
Ctrl-D  Delete line                 Ctrl-W  Undelete one line
Ctrl-L  Insert line/ split line     Ctrl-U  Join two lines
Ctrl-P  Put line in store buffer    Ctrl-G  Get the store buffer
Ctrl-X  Copy the store buffer       Ctrl-N  Insert character on/off
Ctrl-B  Cursor at line beginning    Ctrl-E  Cursor at line end
Ctrl-Z  Exit from screen editor     Ctrl-C  Enter command mode

       COMMAND MODE COMMANDS
EXIT          Exit from screen editor
F  filename   Get text from file
IL            Insert line mode
UCASE         Toggle upper case search sensitivity
L string      Locate string starting after cursor position.
              If no string given, old string is used
```

Table 6.3. Editor commands.

The text cursor is moved using the cursor keys (arrows). Cursor movement by use of the mouse is not implemented. The command Exit or ^Z is used to leave the editor. The text description is then parsed by the compiler. The declarations, equations and sections of the actual module are replaced if the text is syntactically correct. The syntax for the textual part of modules is given in Appendix 1.

Change

Change

The change command is used to change the representation of control algorithms. There are three different representations.

- Textual (default)
- Function diagram
- Ladder diagram

The command changes representation from Textual to Function, from Function to Ladder or from Ladder to Textual. The textual representation is currently the only one allowing modification (text editing). Examples of the different representations are given in Chapter 4.

Run

Run

The Run command executes the system a specified number of times.

Stop time:

A global analysis of connections and data flows is first performed in order to sort the equations into correct computational order. All INIT-sections are then executed once before the iteration starts. The results of the execution are presented as trend curves in the interfaces and in *Displays* for other variables (see command Display).

Remove

Remove

The command Remove allows deletion of modules, interfaces and connections. The object to be deleted is selected using the mouse.

Point at object.

Note. All connections to a certain module or interface should be deleted before the module or interface is deleted. The implementation does not contain any Undelete function and there is no way to delete windows.

Display *Display*

Displays can be created for presentation of the run-time results of any variable. The selection of the interesting variable is done in two steps. First by pointing to the module to which it belongs and then giving the name of the variable.

Select module.

Variable:

The layout of the display is then done by stretching in the usual way.

Stretch object.

View *View*

The View command creates a new window for viewing the control system. See section 6.1. The layout is done by stretching.

Stretch object.

Layout *Layout*

The command Layout is used for changing the position and size of windows and modules. The object of interest is selected by pointing and the new layout is given by stretching.

Select object.

Stretch object.

Modules cannot be moved out of their parent module. Currently, this has to be done by Copy and Remove. There is no facility to change layout of interfaces and connections.

HardCopy

HardCopy

The Hardcopy command causes a hard copy of the current content of the screen (not including the menu) to be produced for a laser printer, a printer or plotter.

Device (Laserprinter / Dotprinter / Plotter):

Answer l, d, or p. The horizontal size is then prompted for. The maximum size for the laser printer is 250 mm giving a raster of approximatively 2500 × 1750 pixels. If the size is greater than 200 mm the figure is tilted 90 degrees on the output.

Horizontal size (mm):

Colors other than black (color code = 0) will appear black on the laser printer and matrix printer, others white. There is a possibility to redefine how colors are used in order that filled areas remain white on the paper. This is done in set-up file LICSCOL. Furthermore, line style (dot pattern) for hardcopy can be specified for each color in file STYLES.

Exit

Exit

The Exit command stops the LICS program.

7. Implementation

Software structure

Pascal is used for the implementation of LICS. The software is highly modular. The module structure is shown in Fig. 7.1. The *Graphics* module is shown opened-up and zoomed-in in Fig. 7.2. Other modules such as *LicsMod*, *Language* etc. also have submodules. The figures are obtained using a Graphical Ada Environment (Lerup, 1984).

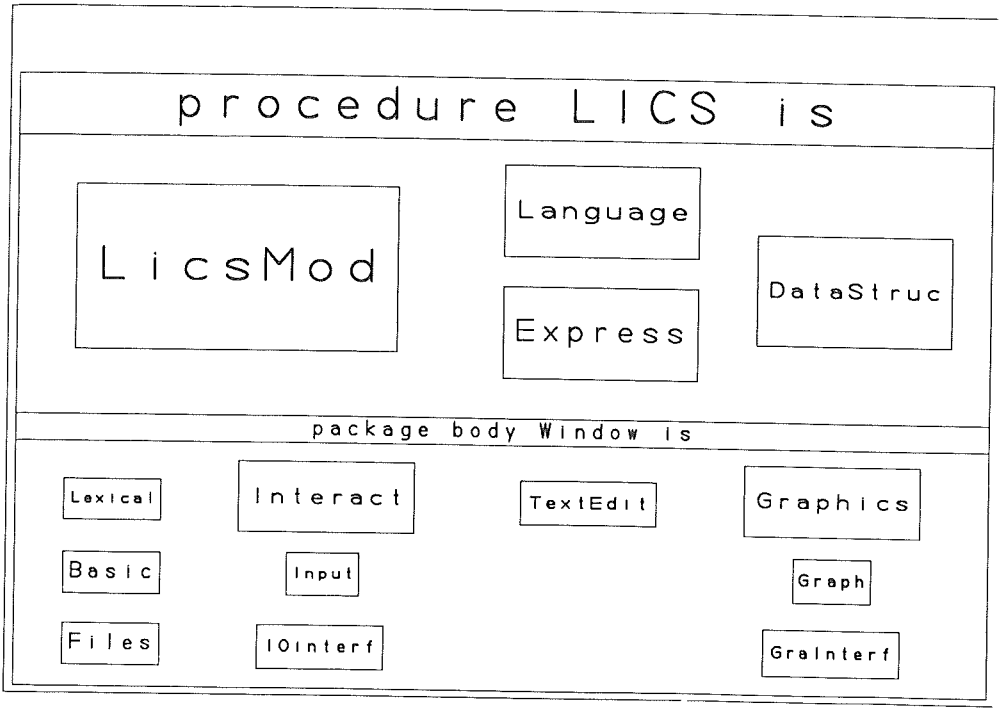


Fig. 7.1. Structure of LICS program.

Modularization implies that *related* types, variables, procedures etc. should be considered as a unit. However, Pascal requires that all global types must be declared together. A preprocessor has been used that produces a standard Pascal program from different module files.

A module consists of several sections of code preceded by headings such as

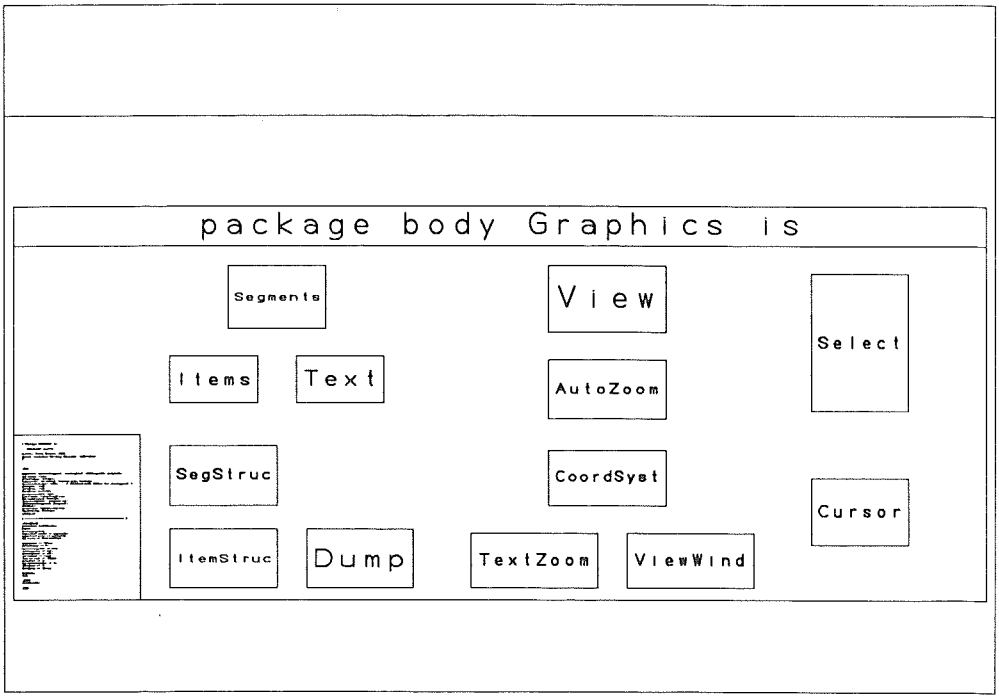


Fig. 7.2. Structure of Graphics package.

```
.PROGRAM
.LABEL
.CONST
.TYPE
.VAR
.FORWARD
.PROCEDURE
.INIT
.MAIN
.END
```

Information hiding is accomplished by dividing each module in two parts: specification and implementation. The specification contains what is accessible from the outside. Procedures are declared as **forward** only giving their parameter list. The bodies of procedures are given in the implementation part. The preprocessor does not enforce these rules, but has nevertheless proven very powerful.

Portability of the code is accomplished by clear module specifications for machine dependent facilities for file handling, graphics and input devices.

Compiler

The data structure is based on the abstract syntax of the language. Examples of nodes in the syntax tree are given in Fig. 7.3. Information which is not syntax related is added as attributes to the tree. A module's position and size in the coordinate system of its parent is given as *xScale, yScale, xTrans* and *yTrans*. References to generated graphical representations are stored as ModSeg, NameSeg etc.

```
type Modul = record
  Name: Ident;
  ModuleType: (Object, Controller, System);
  Decl: Declarations;
  SubModules: List; { Modul }
  Connections: List; { Connection }
  Sections: List; { Section }

  Attr: record
    xScale, yScale, xTrans, yTrans: MasterCoord;
    Text: TextPnt;
    Father: ^Modul;
    ModSeg, NameSeg,
    OuterSeg, InnerSeg, SubSeg,
    DiagramSeg, ladderSeg: SegmentId;
    end;
  end;

Expr = record
  Attr: record
    case tExprType of
      ExprVariable:      (VarRef: NameRef);
      ExprFunction:      (FuncIndex: FuncProcType);
      ExprNumber:        ( );
      ExprConstant:      ( );
      ExprUnary:         ( );
      ExprBinary:        ( );
      ExprIfThenElse:    ( );
      ExprMatrixConstr:  ( );
    end;

  case ExprType: tExprType of
    ExprVariable:      (VarId: Ident; MatrixVar: ExpressList;
                        VaraAtribute: VarAttributeType);
    ExprFunction:      (Func:  Ident; ArgumentList: ExpressList);
    ExprNumber:        (Num: Number);
    ExprConstant:      (Constant: ConstantType);
    ExprUnary:         (Expr: pExpr; UnaryType: tUnaryType);
    ExprBinary:        (Expr1, Expr2: pExpr; BinType: tBinType);
    ExprIfThenElse:    (IfList: List { ExprClause } );
    ExprMatrixConstr:  (ConstrList: List { ExpressList } );
  end;
```

Fig. 7.3. Nodes in the syntax tree.

The submodules of a module are stored in a list. A general list package

for doubly linked, headed lists is used. It has operations such as NewList, Into, First, and SuccElem (c.f. Ekman, Karlsson, 1981).

Variant records are needed in order to make the list package able to handle different types of records. A separate *Node* contains the variants and pointers to the different records. It also contains forward and backwards pointers as shown in Fig. 7.4.

```
type
Element = ^Node;
List = Element;

Node = record
  Kind: (ListKind, ElementKind);
  pre, suc: ^Node;

  case ElementType: ElementTypes of
    vModule:     (dModule:     ^Modul);
    vConnection: (dConnection: ^Connection);
    vSection:    (dSection:    ^Section);
    ...

  end;
```

Fig. 7.4. List node.

The interactive environment of LICS requires an incremental parser. When editing of a text description of a module is finished, only that text should be parsed.

If the text is syntactically correct, the parser builds a subtree for the module. This is combined with the SubModules, Interfaces and Connections of the previous version of the module, since these are specified with the graphical editor.

Parts of the parser are also used for Moduleheading and InterfaceDeclaration when modules and interfaces are created interactively. Furthermore, the parser is used when getting a whole system from a file.

A hand-coded recursive descent parser (Aho, Ullman, 1978) was implemented. Later, a compiler-compiler, called CC (Karlsson, 1982) was used to automatically generate the parser. Figure 7.5 shows some examples of input to the compiler-compiler, including syntax and semantic actions to build the syntax tree.

```
SubModules<eModu: Element>        [[ var eSubMod: Element; ]] =
  'SUBMODULE'
  { Module<eModu^.dModule, eSubMod>
                                  [[ Into(eSubMod, eModu^.dModule^.SubModules); ]]
  } .

SimpleExpression<var exp: pExpr>  [[ var expr1: pExpr; ]] =
  ( Term<exp>
  | '+' Term<exp>
```

```
  | '-' Term<exp>                  [[ exp := UnaryOp(UnaryMinusOp, expr1); ]]
  | 'NOT' Term<exp>                [[ exp := UnaryOp(UnaryNotOp, expr1); ]]
  )
  { ( '+' Term<expr1>              [[ exp := BinaryOp(BinAddOp, exp, expr1); ]]
    | '-' Term<expr1>              [[ exp := BinaryOp(BinSubOp, exp, expr1); ]]
    )
  } .
```

Fig. 7.5. Compiler-compiler input.

The generated parsing procedures may have user defined parameters. Formal and actual parameters are enclosed within <>. Local variable declarations and semantic actions are enclosed in [[ ]]. Figure 7.6. shows the generated parsing procedures for the specifications in Fig. 7.5.

```
procedure SubModules(eModu: Element);
  var eSubMod: Element;
  begin
  if Token = Q38 (* 'SUBMODULE' *) then
    begin
    Token = Lexi(LexString);
    end
  else
    Abort(ErrorMessageGen41);
  while Token = Iden do
    begin
    Module(eModu^.dModule, eSubMod);
    Into(eSubMod, eModu^.dModule^.SubModules);
    end;
  end;

procedure SimpleExpression(Follow: FSet; var exp: pExpr);
  var expr1: pexpr;
  begin
  if Token in FirstSet[13] then
    begin
    Term(FirstSet[14] + Follow, exp);
    end
  else if Token = Q14 (* '+' *) then
    begin
    Token = Lexi(LexString);
    Term(FirstSet[14] + Follow, exp);
    end
  else if Token = Q15 (* '-' *) then
    begin
    Token = Lexi(LexString);
    Term(FirstSet[14] + Follow, expr1);
    exp := UnaryOp(UnaryMinusOp, expr1);
    end
  else if Token = Q48 (* 'NOT' *) then
    begin
    Token = Lexi(LexString);
    Term(FirstSet[14] + Follow, expr1);
    exp := UnaryOp(UnaryNotOp, expr1);
    end
  else
    Abort(ErrorMessageGen63);
  while Token in FirstSet[14] do
    begin
    if Token = Q14 (* '+' *) then
      begin
      Token = Lexi(LexString);
```

```
      Term(FirstSet[14] + Follow, expr1);
      exp := BinaryOp(BinAddOp(exp, expr1);
      end
    else if Token = Q15 (* '-' *) then
      begin
      Token = Lexi(LexString);
      Term(FirstSet[14] + Follow, expr1);
      exp := BinaryOp(BinSubOp, exp, expr1);
      end
    else
      Abort(ErrorMessagegen64);
    end;
  end;
```

Fig. 7.6. Compiler-compiler output.

Semantic processing

The parser checks for syntactic errors and builds the syntax tree. The tree is then traversed several times for semantic processing. Some of the traversals are described below.

Resolve:

It links use of identifiers to their declarations. Expression nodes of variable type have attributes for pointing to the corresponding variable record in the declaration.

MakeConnections:

The equivalence classes of connections are built. It is a list of lists of InterfaceComponents. The lists of connected components are built according to the rules given in Chapter 3. Each component has a pointer to its equivalence list for easy access to the other connected components.

Allocate:

Allocates variable space in an array of values. One entry is reserved for each equivalence class of connections.

SectionAttributes:

Determines which variables are input resp. output for each section.

Schedule:

The sorting of equations into correct computational order is done by a depth-first search (Tarjan, 1972) using the previously determined section attributes.

Interpretation

The sorted sections are interpreted when the Run command is given. The syntax tree for each equation is traversed and calculations corresponding to the node's

operation are performed. A code similar to Pascal's P-code is also generated, but no interpretator has been developed for it.

Graphics

The zooming capability is central to the graphical system. Continuous zooming requires high resolution coordinates. All coordinates are thus stored as real numbers. This also makes it easier to maintain device independence.

The hierarchy of modules is represented by rectangles within rectangles. When one module is moved and scaled, it is essential that all its submodules are also moved and scaled accordingly. This is conveniently accomplished by defining local coordinate systems for each module.

The use of local coordinate systems is common in computer graphics applications (Newman, Sproull, 1979) (Foley, Van Dam, 1982). A translated and scaled coordinate system is established by calling the procedures:

Translate(Tx, Ty)
Scale(Sx, Sy)

Figure 7.7. shows the relations between the old and the new coordinate systems.

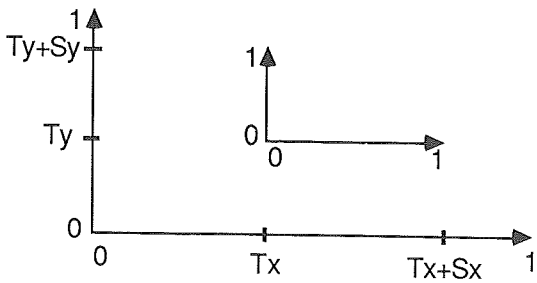


Fig. 7.7. Coordinate transformations.

Drawing is performed using of primitives like: *MoveTo, LineTo, SetColor, CharSize, DrawCharLine.* Character fonts are defined by splines similarly to METAFONT (Knuth, 1979). Bitmaps of different sizes are precalculated and stored in order to get fast output of text.

The total picture of the system is very complex. It is essential to avoid traversing of subpictures not shown on the screen. So called, "box testing" is therefore done during traversal. The information zooming concept assures that very small pictures are not drawn in detail.

A graphical data structure is built up in parallel to the syntax tree. It is based on *segments* similar to those used in graphical systems such as CORE and GKS. A segment consists of a list of graphical primitives such as lines and text strings. In addition, there is a set of attributes:

visible or not
zoom bounds for visibility (for information zoom)
scale, translation (for layout)
zoom, pan, scroll (for maneuvering)
window limits
background color
etc.

The hierarchical structures are handled by allowing segments to have references to other segments.

One segment can be selected for maneuvering. Subsequent use of the joysticks or mouse will imply changing the zoom, pan, scroll attributes of that segment. The segment structure is traversed to refresh the screen and to test for picking. The algorithm for calculating the effect of maneuvering is

Translate(xZoomCenter, yZoomCenter);
Scale(ZoomScale, ZoomScale);
Translate(-xZoomCenter, -yZoomCenter);
Translate(Pan, Scroll);

A package for presenting and editing mathematical formulas of the form (2.4) was developed as a master thesis work (Taube, 1984).

Hardware

The program was developed on a VAX-11/780 running VAX/VMS. A frame buffer with resolution $512 \times 512 \times 8$ was installed directly on the Unibus in order to get the desired updating speed for the graphics (Nielsen, Elmqvist, 1983). The frame buffer was used in double buffer mode for the animation. The LICS program was also ported on an Apollo DN 600 work station.

8. Conclusions

The report describes an effort to design an integrated language and programming environment for process control and modelling. A unification of language constructs for analog control and programmable controllers are sought. Both are based on cyclic execution. Analog controllers are currently based on fix function blocks which are connected together. More flexibility is often needed. On the other hand, programmable controllers are programmed by equations manually translated to assembly-like languages. Modularization constructs and application libraries are often lacking.

The proposed language is non-procedural. It is instead based on equations. This gives more emphasize on data flow. Control engineers have traditionally considered signal flow as a basic concept. Block diagrams have been used for a long time to document signal flows between function blocks.

The language contains a strong module concept allowing clear definition of module interfaces. Connections can be structured. This is very important in order to accomplish information hiding. Structured connections thus match the hierarchical module structure.

The evaluation order of the equations is determined from their dependencies. However, the sorting needs information about dependencies on information from previous sampling instants. *State variables* need thus to be explicitly declared. This is in contrast to how programmable controllers are programmed today. Their equations has to be sorted manually, i.e. the user has to consider which variables carries information between sampling instants. This can be a source of fatal synchronization errors or unnecessary time delays.

Sequence control can be handled by introducing sequence modules as shown in Chapter 4. These examples clearly show the power of the LICS module and interconnection concepts. However, it would be better to introduce special constructs for dealing with control flow and replacements of equations. This would include facilities for conflict resolution, resource allocation etc.

The user has full freedom for layout of modules and connections. This freedom is, however, sometimes a burden. On the other hand, fully automatic layout also has drawbacks. Small structural changes, such as adding a block, may drastically influence where blocks are positioned. The user will in such circumstances have problems to recognize the similarities to the previous version of the documentation. A solution is to give the user facilities for automatic layout of parts of a structure such as blocks connected in series or in parallel.

The graphical interface to the user has been a major concern. The strong

development of personal computers and special purpose harware for graphics give quite new possibilities. Animation for zooming, panning and scrolling has been tried out as a mean to support of the user's mental model of a system.

No rigourous evaluation of the user interface has been done. However, many demonstrations have been made with very positive response, for example at the second IEEE Computer Aided Control System Design Symposium, Boston (Elmqvist, 1983b). LICS was then demonstrated on an Apollo DN 600 work station.

There are still many desirable facilities that may be feasible to develop for use on powerful personal computers.

There was not much effort put on developing the interactive facilities for editing etc. in LICS. It would probably be better to develop an object oriented interaction model, i.e. the object to be operated on is first selected, then the operation. The graphical segment structure should be replaced by storing all graphical information in the application data structure, i.e. in this case the syntax tree. That would give more flexibility for implementing operations on the objects shown.

The report might be seen as a collection of ideas for an integrated language and programming environment for process control. However, many details have to be investigated further. It should be pointed out that the development of man-machine interaction techniques has been hampered by lack of feasible hardware for a long time. Many of these restrictions are now removed by using personal computers with powerful graphics.

9. Acknowledgements

The author first of all would like to thank Professor Karl Johan Åström for his strong support throughout the project. He also suggested many improvements to this report.

Tommy Essebo programmed most of the system. Thanks to him, it was possible to test many new ideas on interaction techniques.

The author is also very thankful to Boris Magnusson and Oskar Permvall, Computer Science Department and to Lars Nielsen for many stimulating discussions.

Many thanks to Hans Thildervist and Poul Kongstad, SattControl Inc., and to Sven Erik Mattsson for valuable comments on the manuscript.

This project has been carried out under the auspices of the Program in Information Processing and Computer Science (Ramprogrammet i Informationsbehandling) which was initiated and sponsored by the National Swedish Board for Technical Development (STU). This work has been part of the Center for Industrial Computer Systems supported under contract 80–3962.

10. References

Ackerman W.B. (1979), "Data flow languages", *National Computer Conference.*

Aho A.V., Ullman J.D. (1978), *Principles of Compiler Design*, Addison-Wesley Publ. Comp.

Andersson L., Dagnegård E.(1984), TEVE. TFRT-7279, Dept of Automatic Control, Lund Institute of Technology, Sweden.

Asea, "Asea MasterPiece – PC-Programming", (in Swedish: "Asea MasterPiece – PC-programmering"), CA06-1009/1, Asea Electronics, Sweden.

Booch G. (1983), *Software Engineering with Ada*, The Benjamin/Cummings Publ. Comp.

Clark J.H. (1980a), "A VLSI Geometry Processor for Graphics", *Computer* **13**, 59–68.

Clark J.H., M.R. Hannah (1980b), "Distributed Processing in a High-Performance Smart Image Memory", *LAMBDA*, Fourth Quarter, 1980.

Donelson W. C. (1978), Spatial Management of Information. *Computer Graphics*, **12**, 203–209.

Ekman T., J. Karlsson (1981), *Pascal för dig som tror du kan programmera*, in Swedish, Studentlitteratur.

Elmqvist H. (1977), SIMNON – An Interactive Simulation Program for Nonlinear Systems, *Proc. Simulation '77*, Montreux, Switzerland.

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

Elmqvist H. (1979a), DYMOLA – A Structured Model Language for Large Continuous Systems. *Proc. Summer Computer Simulation Conference*, Toronto, Canada.

Elmqvist H. (1979b), Manipulation of Continuous Models Based on Equations to Assignment Statements. *Proc Simulation of Systems '79*, IMACS Congress 1979, Sorrento. North Holland Publ. Comp.

Elmqvist H., Mattsson S.E., Olsson G. (1981), Computers in Control Systems – Real Time Programming, (in Swedish: "Datorer i Reglersystem – Realtidsprogrammering"), Compendium, Dept of Automatic Control, Lund Institute of Technology, Sweden.

Elmqvist H. (1982), A Graphical Approach for Documentation and Implementation of Control Systems. *Proc. 3rd IFAC/IFIP Symposium on Software for Computer Control*, Madrid, Spain.

Elmqvist H. (1983a), Combining Graphical Descriptions and Equations for Dynamical Modelling, *Proc. Fourth IASTED International Symposium on Modelling and Simulation*, Lugano, Switzerland.

Elmqvist H. (1983b), A Graphical System for Modelling and Implementation of Control Systems. *Proc. Second IEEE Computer Aided Control System Design Symposium*, Boston.

Feiner S., Nagy S., Van Dam A. (1982), An Experimental System for Creating and Presenting Interactive Graphical Documents. *ACM Trans. on Graphics*, Vol. 1, No. 1, 59-77.

Foley J.D., Van Dam A. (1982), *Fundamentals of Interactive Computer Graphics*, Addison–Wesley Publ. Comp.

Goldberg A. (1984), *Smalltalk-80. The interactive programming environment*, Addison-Wesley Publ. Comp.

Haggård L. (1982), "Graphical Compiler Creates Control Program", (in Swedish: "Grafisk kompilator gör reglerprogram"), *Industriell Datateknik*, nr 12.

Herot C.F., R. Carling, M. Friedell, D. Kramlich (1980), A Prototype Spatial Data Management System. *Computer Graphics*, **14**, 64–70.

Herot C.F., G.P. Brown, R.T. Carling, M. Friedell, D. Kramlich, R.M. Baecker (1982), An Integrated Environment for Program Visualization. *Proc. Automated Tools for Information System Design and Development*, New Orleans.

Hitachi (1984), "Hitachi HD 63484 ACRTC Advanced CRT Controller – User's Manual", Hitachi.

Honeywell (1976), "Algorithm TDC 2000", Technical Data, Reference CB-10-03, Honeywell Inc., USA.

Honeywell (1978), "Controller configuration TDC 2000", Technical Data, Reference CB-10-04, Honeywell Inc., USA.

IEC (1985), "Working Draft Standard for Programmable Controllers, Part 3: Programming Languages", IEC, SC65A/WG6/TF3.

Karlsson P. (1982), A Translator Writing System, TRITA-TDE-8202, Dept of Applied Electronics, Royal Institute of Technology, Stockholm, Sweden.

Knuth D.E. (1979), *TEX and METAFONT – New Directions in Typesetting*, Digital Press.

Knuth D.E. (1984), *The TEXbook*, Addison–Wesley Publ. Comp.

Lauber R. (1980), Design Methods for Computer Controlled Real-time Automation Systems. *Proc. 6th IFAC/IFIP Conf. on Digital Computer Applications to Process Control*, Pergamon Press.

Lemmons P., Robertson B. (1983), "The HP 150", *Byte*, October 1983, pp. 36–56.

Lerup P. (1984), "A Graphical Tool for Program Development in Ada", (in Swedish: "Ett grafiskt hjälpmedel för programutveckling i Ada"), Master Thesis, TFRT-5312, Department of Automatic Control, Lund Institute of Technology, Sweden.

Lindahl S. (1976), A Nonlinear Drum Boiler–Turbine Model, TFRT-3132, Dept of Automatic Control, Lund Institute of Technology, Sweden.

Newman W.M., R.F. Sproull (1979), *Principles of Interactive Computer Graphics*, Second Edition, McGraw Hill Book Company.

Nielsen L., H. Elmqvist (1983), "An Image Laboratory", TFRT-7261, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Petersen J.L. (1981), *Petri Net Theory and the Modelling of Systems*, Prentice-Hall.

Rhyne J.R., U.M. Richter, E.D. Carlson (1981), TELL: A General Graphical Editor for Design and Documentation Charts. San Jose Research Lab, IBM Corp, 5600 Cottle Road, San Jose, CA 95193.

Siemens (1984), "Grafisches Programmiren von Ablaufsteuerungen mit dem Programmiergerät 675", Siemens Energietechnik, 6. Jahrgang, Heft 2, März/April 1984, 68-71.

Shannon L. (1975), *Introduction to Abstract Algebra*, McGraw-Hill.

Taube M. (1984), Graphical Presentation and Editing of Mathematical Expressions and Ladder Diagrams, (in Swedish: "Grafisk presentation och editering av matematiska uttryck och reläschema"), Master Thesis, TFRT-5317, Dept of Automatic Control, Lund Institute of Technology, Sweden.

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

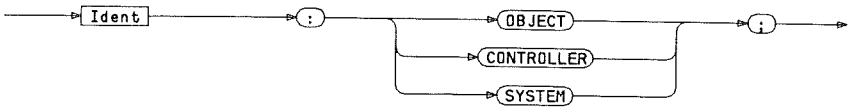
Wiberg T. (1977), Permutation of an Unsymmetric Matrix to Block Triangular Form, Ph.D. Thesis, Department of Information Processing, University of Umeå, Umeå, Sweden.

Williams G. (1983), "The LISA Computer System. Apple designs a new kind of machine", *Byte*, vol. 8, February.

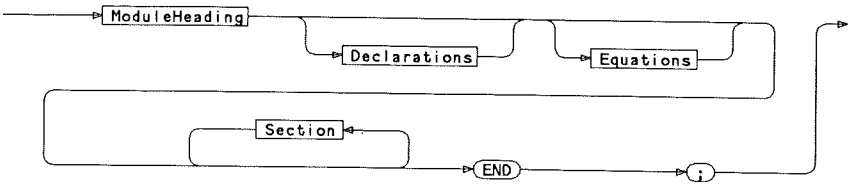
Appendix

1. Syntax diagrams for text in modules

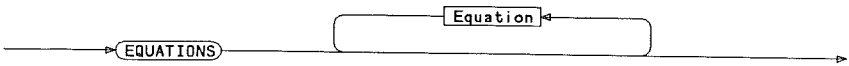
ModuleHeading



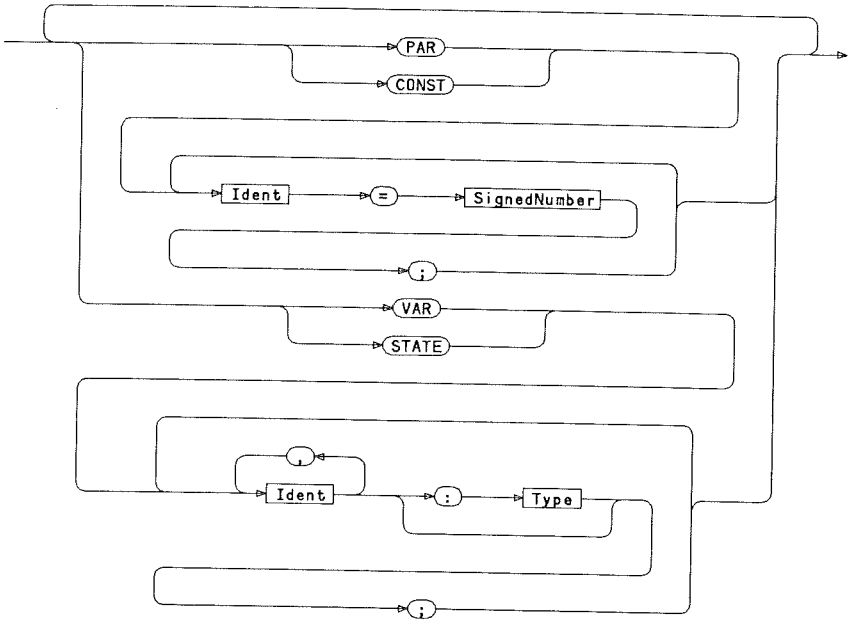
Module



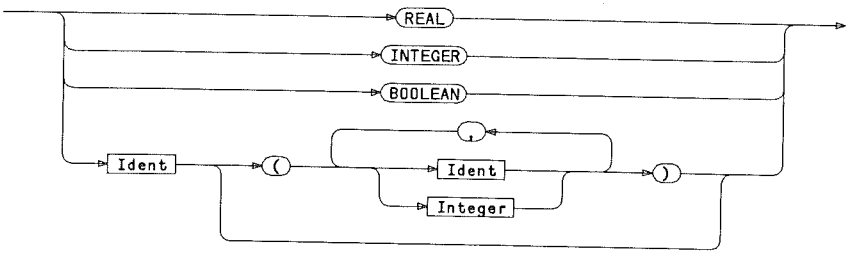
Equations



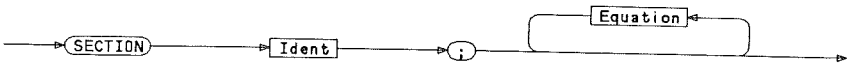
Declarations



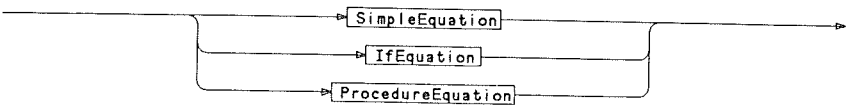
Type



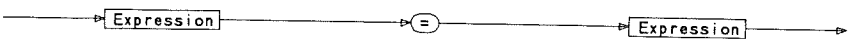
Section



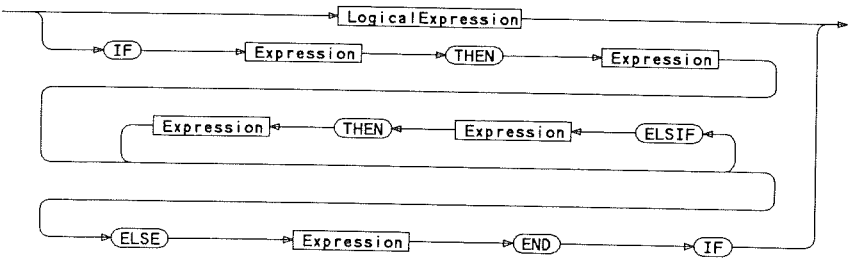
Equation



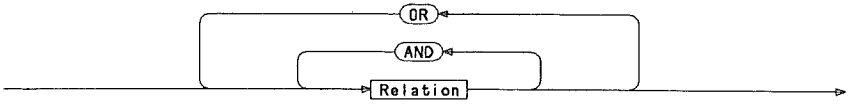
SimpleEquation



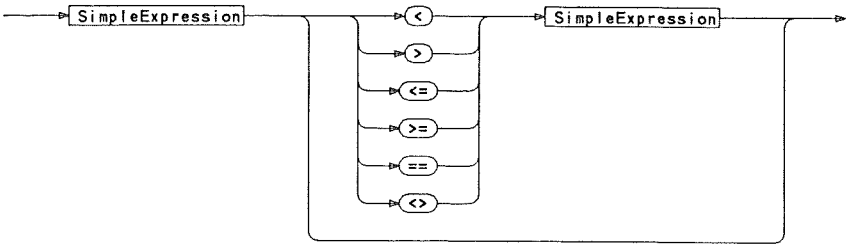
Expression



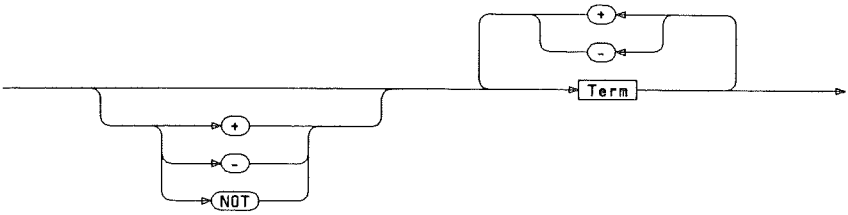
LogicalExpression



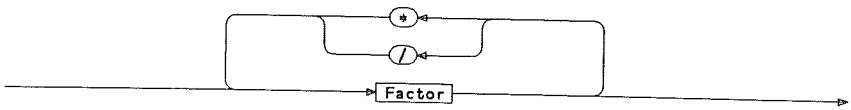
Relation



SimpleExpression

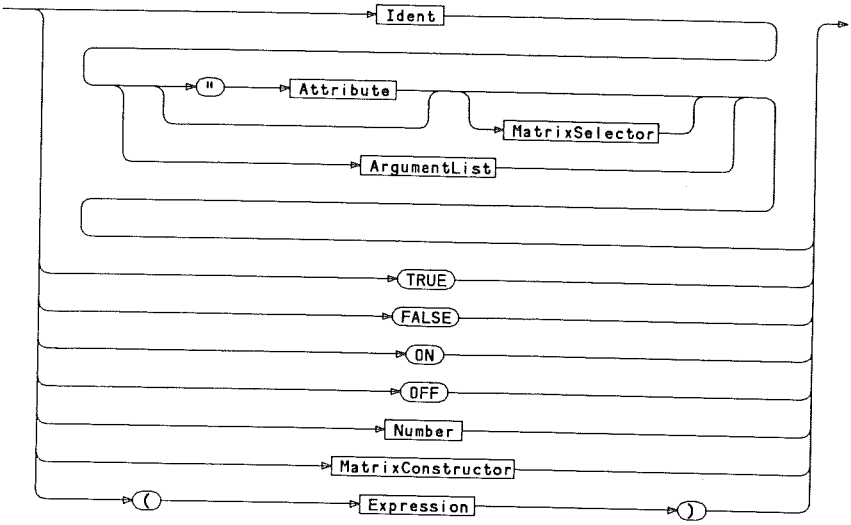


Term

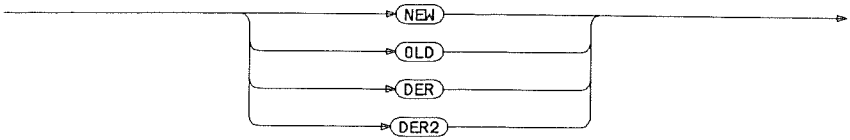


Factor

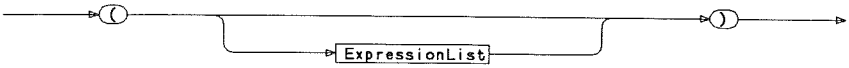
Primary



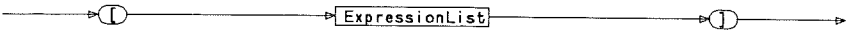
Attribute



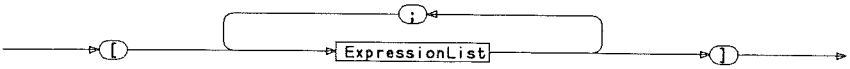
ArgumentList



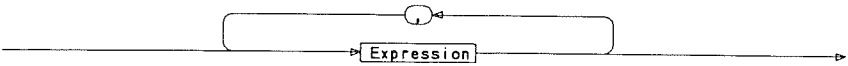
MatrixSelector



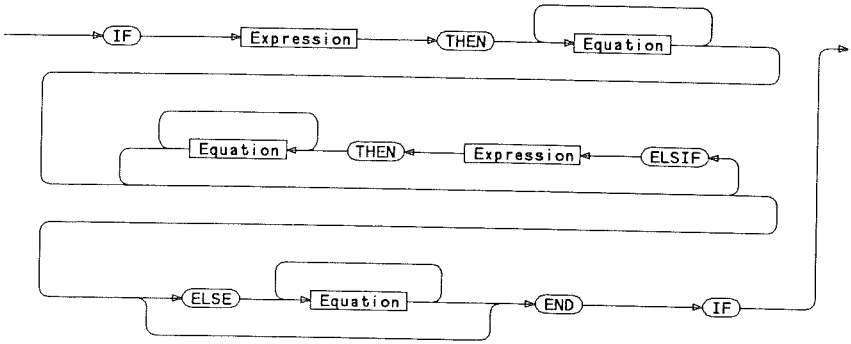
MatrixConstructor



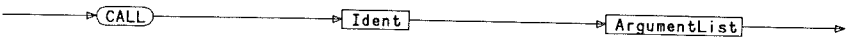
ExpressionList



IfEquation



ProcedureEquation



2. Syntax for LICS files

Syntax notation:

[ ] optional

{ } repeat one or more times

{ $ } repeat - exit, $\{a\$b\}$ equivalent to $a[\{ba\}]$

```
Module = Ident ':' ( Object / Controller / System ) .

Object = 'OBJECT' Graphics ';' Body 'END' ';' .

Controller = 'CONTROLLER' Graphics ';' Body 'END' ';' .

System = 'SYSTEM' Graphics ';' Body 'END' ';' .

Graphics = [ 'AT' '(' Number ',' Number ')'
             'SIZE' '(' Number ',' Number ')' ] .

Body = [ Declarations ] [ Submodules ] [ Connections ]
  [ Equations ] { $ Section } .

Declarations = { Interface / Par / State / Var / Const } .

Interface = 'INTERFACE' { InterfaceDeclaration } .

Par = 'PAR' { ConstantDeclaration } .

State = 'STATE' { VariableDeclaration } .

Var = 'VAR' { VariableDeclaration } .

Const = 'CONST' { ConstantDeclaration } .

InterfaceDeclaration = InterfaceComponent Graphics ';' .

InterfaceComponent = Ident ':' ( InterfacePrimitive / InterfaceStructure ) /
                     InterfaceStructure .

InterfaceStructure = '(' { [ InterfaceComponent ] $ ',' } ')' .

InterfacePrimitive = [ 'IN' / 'OUT' ] [ Type ] .

VariableDeclaration = { Ident $ ',' } [ ':' Type ] ';' .

ConstantDeclaration = Ident '=' SignedNumber ';' .

Type = Ident [ '(' { ( Ident / Integer ) $ ',' } ')' ] .

Submodules = 'SUBMODULE' { Module } .

Connections = 'CONNECT' { [ Ident '.' ] Ident 'TO'
                          [ Ident '.' ] Ident Route ';' } .

Route = [ 'VIA' { '(' Number ',' Number ')' } ] .

Equations = 'EQUATIONS' { $ Equation } .

Section = 'SECTION' Ident ';' { $ Equation } .

Equation = ( SimpleEquation / IfEquation / ProcedureEquation ) ';' .
```

```
SimpleEquation = Expression '=' Expression .

Expression = LogicalExpression /
    ( 'IF' Expression 'THEN' Expression
    { $ 'ELSIF' Expression 'THEN' Expression }
    'ELSE' Expression 'END' 'IF' ) .

LogicalExpression = { { Relation $ 'AND' } $ 'OR' } .

Relation = SimpleExpression
          [ ( '<' / '>' / '<=' / '>=' / '==' / '<>' )
          SimpleExpression ] .

SimpleExpression = [ '+' / '-' / 'NOT' ] { Term $ ( '+' / '-' ) } .

Term = { Factor $ ( '*' / '/' ) }.

Factor = Primary [ '**' Primary ] .

Primary = ( Ident ( [ '"' Attribute ] [ MatrixSelector ]  ) /
           ArgumentList ) / 'TRUE' / 'FALSE' / 'ON' / 'OFF' /
           Number / MatrixConstructor / '(' Expression ')' .

Attribute = 'NEW' / 'OLD' / 'DER' / 'DER2' .

ArgumentList = '(' [ ExpressionList ] ')' .

MatrixSelector = '[' ExpressionList ']' .

MatrixConstructor = '[' { ExpressionList $ ';' } ']' .

ExpressionList = { Expression $ ',' } .

IfEquation = 'IF' Expression 'THEN' { Equation }
    { $ 'ELSIF' Expression 'THEN' { Equation } }
    [ 'ELSE' { Equation } ] 'END' 'IF' .

ProcedureEquation = 'CALL' Ident ArgumentList .


Ident = 'Letter' { $ ( 'Letter' / 'Digit' ) } .
Integer = { 'Digit' } .
Number = Integer [ '.' Integer ] [ 'E' [ '+' / '-' ] Integer ] .
SignedNumber = [ '+' / '-' ] Number .
```

3. Textual description of TankSyst

```
TankSystem: SYSTEM
  AT(-0.125, -0.125) SIZE (1.25, 1.25);
  SUBMODULE
    PID: CONTROLLER
      AT(0.32, 0.46) SIZE (8E-2, 0.11);
      INTERFACE
        C: (y: OUT real, ySet: IN boolean, yNew: IN real)
          AT(0.9, 0.38) SIZE (0.1, 0.28);
        S: (r: IN real, rSet: OUT boolean, rNew: OUT real)
          AT(0, 0.51) SIZE (0.1, 0.31);
        P: (u: IN real)
          AT(0, 0.16) SIZE (0.1, 0.3);
        O: (ParSet: IN boolean, KI: IN real, TiI: IN real, TdI: IN
        real, Auto: IN boolean, rI: IN real, Manual: IN boolean, yI:
        IN real)
          AT(0.25, 0.9) SIZE (0.49, 0.1);

      PAR
        DT = 0.1;
      STATE
        K, Ti, Td, SetPoint, i, yOld, eOld:real;
      VAR
        Kn, Tin, Tdn, e, yl, int, iNew, SP:real;
        PDreg, BackCalc:boolean;
      EQUATIONS
        PDreg = Ti > 1000.0;
        e = IF Manual THEN 0.0 ELSIF Auto THEN SetPoint - u ELSE
          r - u END IF;
        y = IF Manual THEN yOld + yI ELSIF PDreg THEN K*(e + Td/DT*
          (e - eOld)) ELSE K*(e + DT/Ti*i + Td/DT*(e - eOld)) END IF;
        yl = IF ySet THEN yNew ELSE y END IF;
        Kn = IF ParSet THEN K + KI ELSE K END IF;
        Tin = IF ParSet THEN Ti + TiI ELSE Ti END IF;
        Tdn = IF ParSet THEN Td + TdI ELSE Td END IF;
        BackCalc = ParSet OR ySet OR Manual;
        IF PDreg
        THEN
          SP = IF BackCalc THEN
            u + (yl/Kn + Tdn/DT*eOld)/(1 + Tdn/DT) ELSE SetPoint END IF;
          iNew = i;
        ELSE
          SP = IF Manual THEN u ELSE SetPoint END IF;
          iNew = IF BackCalc THEN (yl/Kn - e - Tdn/DT*(e - eOld))/
            ( DT/Tin) + e ELSE i + e END IF;
        END IF;
        i'NEW = iNew;
        rSet = Manual OR Auto OR BackCalc AND PDreg;
        rNew = SP;
        SetPoint'NEW = IF Manual THEN SP
          ELSIF Auto THEN SP + rI ELSE r END IF;
        K'NEW = Kn;
        Ti'NEW = Tin;
        Td'NEW = Tdn;
        yOld'NEW = yl;
        eOld'NEW = SP - u;
      SECTION INIT;
        K = 5.0;
        Ti = 1.0;
        Td = 0.0;
        SetPoint = 0.0;
        i = 0.0;
        yOld = 0.0;
        eOld = 0.0;
```

```
  END;

Tank: OBJECT
  AT(0.59, 0.27) SIZE (0.16, 0.2);
  INTERFACE
    Inlet: (Qin: IN real, Tin: IN real)
      AT(0.19, 0.9) SIZE (0.3, 0.1);
    Outlet: (Qout: OUT real, Tout: OUT real)
      AT(0.9, 0.1) SIZE (0.1, 0.28);
    Meas: (h: OUT real)
      AT(0, 0.33) SIZE (0.1, 0.35);

  PAR
    A = 70E-6; Area = 2.7E-3; Kc = 50;
    dt = 0.1;
  CONST
    g = 9.81;
  STATE
    height:real;
  VAR
    derh:real;
  EQUATIONS
    Qout = A*SQRT(2*height*g);
    derh = (Qin - Qout)/Area;
    height'NEW = height + derh*dt;
    h = Kc*height;
  SECTION INIT;
    height = 0.01;
  END;

Valve: OBJECT
  AT(0.52, 0.61) SIZE (6E-2, 7E-2);
  INTERFACE
    C: (u: IN real, uSet: OUT boolean, uNew: OUT real)
      AT(0.37, 0) SIZE (0.25, 0.1);
    OutLet: (Q: OUT real, T: OUT real)
      AT(0.9, 0.36) SIZE (0.1, 0.28);

  CONST
    Km = 54E-6; Ulow = 0.0; Uhigh = 10.0;
  VAR
    ul:real;
  EQUATIONS
    ul = IF      u > Uhigh  THEN  Uhigh
         ELSIF   u < Ulow   THEN  Ulow
         ELSE                     u
         END IF;
    Q = Km*ul;
    T = 10.0;
    uSet = u > Uhigh OR u < Ulow;
    uNew = ul;
  END;

Operator: CONTROLLER
  AT(0.29, 0.65) SIZE (0.14, 0.1);
  INTERFACE
    O: (ParSet: OUT boolean, KI: OUT real, TiI: OUT real, TdI:
    OUT real, Auto: OUT boolean, rI: OUT real, Manual: OUT boolean,
    yI: OUT real)
      AT(0.4, 0) SIZE (0.21, 0.1);

  STATE
    t:integer;
  EQUATIONS
    Manual = t > 4 AND t < 10;
    yI = IF t == 5 THEN 9.0 ELSE 0.0 END IF;
```

```
      Auto = t > 30 AND t < 33;
      rI = IF Auto THEN 2.0 ELSE 0.0 END IF;
      ParSet = t == 90;
      KI = IF ParSet THEN 10.0 ELSE 0.0 END IF;
      TiI = 0.0;
      TdI = 0.0;
      t'NEW = t + 1;
    END;

  Ref: CONTROLLER
    AT(0.16, 0.48) SIZE (6E-2, 8E-2);
    INTERFACE
      C: (y: OUT real, ySet: IN boolean, yNew: IN real)
        AT(0.9, 0.34) SIZE (0.1, 0.34);

    PAR
      Amp = 3.0; Per1 = 2.5; Per2 = 2.5;
      DT = 0.1;
    STATE
      t, Bias:real;
    EQUATIONS
      y = Bias + (IF t < Per1 THEN 0.0
        ELSE Amp END IF);
      t'NEW = IF t < Per1 + Per2 THEN
        t + DT ELSE 0 END IF;
      Bias'NEW = IF ySet THEN yNew
        ELSE Bias END IF;
    SECTION INIT;
      Bias = 0.0;
      t = -2.5;
    END;

CONNECT
  Ref.C TO PID.S
    VIA (0.22, 0.53) (0.32, 0.53) ;
  Operator.O TO PID.O
    VIA (0.36, 0.65) (0.36, 0.57) ;
  Valve.C TO PID.C
    VIA (0.55, 0.61) (0.55, 0.52) (0.4, 0.52) ;
  Tank.Inlet TO Valve.OutLet
    VIA (0.64, 0.47) (0.64, 0.64) (0.58, 0.64) ;
  PID.P TO Tank.Meas
    VIA (0.32, 0.49) (0.28, 0.49) (0.28, 0.36) (0.59, 0.36) ;
END;
```

4. Sorted equations for resource allocation

Request without waiting queue

```
INIT:
Resource.Start.SRset:  x = TRUE;
Seq1.Start.SRset:  x = TRUE;
Seq1.Step.SR:  x = FALSE;
Seq2.Start.SRset:  x = TRUE;
Seq2.Step.SR:  x = FALSE;
Seq3.Start.SRset:  x = TRUE;
Seq3.Step.SR:  x = FALSE;

LOOP:
Resource.Start.SRset:  Q = x;
Seq3.Step.SR:  Q = x;
Seq3.Timer:  y = d == 0;
Seq3.Cond2.AND1:  y = u1 AND u2;
Seq3.Release.EndSel.OR1:  y = u1 OR u2;
Seq2.Step.SR:  Q = x;
Seq2.Timer:  y = d == 0;
Seq2.Cond2.AND1:  y = u1 AND u2;
Seq2.Release.EndSel.OR1:  y = u1 OR u2;
Seq1.Step.SR:  Q = x;
Seq1.Timer:  y = d == 0;
Seq1.Cond2.AND1:  y = u1 AND u2;
Seq1.Release.EndSel.OR1:  y = u1 OR u2;
Seq1.Start.SRset:  Q = x;
Seq1.Request.EndPara.AND1:  y = u1 AND u2;
Seq1.In1:  y = DigitalIn(g, c);
Seq1.Request.Cond.AND1:  y = u1 AND u2;
Seq2.Start.SRset:  Q = x;
Seq1.Request.Sel.NOT1:  y = NOT u;
Seq1.Request.Sel.AND1:  y = u1 AND u2;
Seq2.Request.EndPara.AND1:  y = u1 AND u2;
Seq2.In2:  y = DigitalIn(g, c);
Seq2.Request.Cond.AND1:  y = u1 AND u2;
Seq3.Start.SRset:  Q = x;
Seq2.Request.Sel.NOT1:  y = NOT u;
Seq2.Request.Sel.AND1:  y = u1 AND u2;
Seq3.Request.EndPara.AND1:  y = u1 AND u2;
Seq3.In3:  y = DigitalIn(g, c);
Seq3.Request.Cond.AND1:  y = u1 AND u2;
Seq3.Request.Sel.OR1:  y = u1 OR u2;
Seq2.Request.Sel.OR1:  y = u1 OR u2;
Seq1.Request.Sel.OR1:  y = u1 OR u2;
Resource.Start.SRset:  x'NEW = S OR x AND NOT R;
Seq1.Start.SRset:  x'NEW = S OR x AND NOT R;
Seq1.Step.SR:  x'NEW = S OR x AND NOT R;
Seq1.Timer:  d'NEW = IF u AND d > 0 THEN d - 1 ELSE delay END IF;
Seq2.Start.SRset:  x'NEW = S OR x AND NOT R;
Seq2.Step.SR:  x'NEW = S OR x AND NOT R;
Seq2.Timer:  d'NEW = IF u AND d > 0 THEN d - 1 ELSE delay END IF;
Seq3.Start.SRset:  x'NEW = S OR x AND NOT R;
Seq3.Request.Sel.NOT1:  y = NOT u;
Seq3.Request.Sel.AND1:  y = u1 AND u2;
Seq3.Step.SR:  x'NEW = S OR x AND NOT R;
Seq3.Timer:  d'NEW = IF u AND d > 0 THEN d - 1 ELSE delay END IF;
```

Request with waiting queue

```
INIT:
Resource.Start.SRset:  x = TRUE;
Seq1.Start.SRset:  x = TRUE;
Seq1.Request:  WaitNr = 0;
Seq1.Step.SR:  x = FALSE;
Seq2.Start.SRset:  x = TRUE;
Seq2.Request:  WaitNr = 0;
Seq2.Step.SR:  x = FALSE;
Seq3.Start.SRset:  x = TRUE;
Seq3.Request:  WaitNr = 0;
Seq3.Step.SR:  x = FALSE;

LOOP:
Resource.Start.SRset:  Q = x;
Seq3.Step.SR:  Q = x;
Seq3.Timer:  y = d == 0;
Seq3.Cond2.AND1:  y = u1 AND u2;
Seq3.Release.EndSel.OR1:  y = u1 OR u2;
Seq2.Step.SR:  Q = x;
Seq2.Timer:  y = d == 0;
Seq2.Cond2.AND1:  y = u1 AND u2;
Seq2.Release.EndSel.OR1:  y = u1 OR u2;
Seq1.Step.SR:  Q = x;
Seq1.Timer:  y = d == 0;
Seq1.Cond2.AND1:  y = u1 AND u2;
Seq1.Release.EndSel.OR1:  y = u1 OR u2;
Seq1.Request:  EnableN = EnableR;
Seq2.Request:  EnableN = EnableR;
Seq3.In3:  y = DigitalIn(g, c);
Seq3.Request:  Act = EnableR AND Cond AND WaitNr == 1;
Seq3.Request:  ResetR = ResetN OR Act;
Seq2.In2:  y = DigitalIn(g, c);
Seq2.Request:  Act = EnableR AND Cond AND WaitNr == 1;
Seq2.Request:  ResetR = ResetN OR Act;
Seq1.In1:  y = DigitalIn(g, c);
Seq1.Request:  Act = EnableR AND Cond AND WaitNr == 1;
Seq1.Request:  ResetR = ResetN OR Act;
Resource.Start.SRset:  x'NEW = S OR x AND NOT R;
Seq1.Start.SRset:  Q = x;
Seq1.Request:  ResetS = Act;
Seq1.Start.SRset:  x'NEW = S OR x AND NOT R;
Seq3.Request:  MaxNrR = IF WaitNr > MaxNrN THEN WaitNr ELSE MaxNrN
END IF;
Seq2.Request:  MaxNrR = IF WaitNr > MaxNrN THEN WaitNr ELSE MaxNrN
END IF;
Seq1.Request:  MaxNrR = IF WaitNr > MaxNrN THEN WaitNr ELSE MaxNrN
END IF;
Seq1.Request:  Requesting = EnableS AND Cond AND WaitNr == 0;
Seq1.Request:  NrN = IF Requesting THEN NrR + 1 ELSE NrR END IF;
Seq1.Request:  Activate = Act;
Seq1.Request:  WaitNr'NEW = IF Requesting THEN NrR + 1 ELSIF EnableR
AND WaitNr > 0 THEN WaitNr - 1 ELSE WaitNr END IF;
Seq1.Step.SR:  x'NEW = S OR x AND NOT R;
Seq1.Timer:  d'NEW = IF u AND d > 0 THEN d - 1 ELSE delay END IF;
Seq2.Start.SRset:  Q = x;
Seq2.Request:  ResetS = Act;
Seq2.Start.SRset:  x'NEW = S OR x AND NOT R;
Seq2.Request:  Requesting = EnableS AND Cond AND WaitNr == 0;
Seq2.Request:  NrN = IF Requesting THEN NrR + 1 ELSE NrR END IF;
Seq2.Request:  Activate = Act;
Seq2.Request:  WaitNr'NEW = IF Requesting THEN NrR + 1 ELSIF EnableR
AND WaitNr > 0 THEN WaitNr - 1 ELSE WaitNr END IF;
Seq2.Step.SR:  x'NEW = S OR x AND NOT R;
Seq2.Timer:  d'NEW = IF u AND d > 0 THEN d - 1 ELSE delay END IF;
```

```
Seq3.Start.SRset:  Q = x;
Seq3.Request:  ResetS = Act;
Seq3.Start.SRset:  x'NEW = S OR x AND NOT R;
Seq3.Request:  Requesting = EnableS AND Cond AND WaitNr == 0;
Seq3.Request:  NrN = IF Requesting THEN NrR + 1 ELSE NrR END IF;
Seq3.Request:  EnableN = EnableR;
Seq3.Request:  Activate = Act;
Seq3.Request:  WaitNr'NEW = IF Requesting THEN NrR + 1 ELSIF EnableR
AND WaitNr > 0 THEN WaitNr - 1 ELSE WaitNr END IF;
Seq3.Step.SR:  x'NEW = S OR x AND NOT R;
Seq3.Timer:  d'NEW = IF u AND d > 0 THEN d - 1 ELSE delay END IF;
```

5. Power system in Dymola

```
model Powersystem;

  model Heaters;

    model SuperHeater1;

      cut InSteam (W, H1, P1)
      cut OutSteam (W, H2, P2)
      cut Heat (Q)
      parameter Cm=0 m=0 K=0 Vs=0 f=0
      local Tm TmH T2 T2H R2
        P1*P1 - P2*P2 = f*W*W;
        (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 Attemperator1;

      cut InSteam (W1, H1, P)
      cut OutSteam (W2, H2, P)
      cut Water (Ww, Hw, Pw)
      parameter Sw=0 fw=0
      local aw
        Pw - P = fw*Ww/aw*Ww/aw;
        aw = Sw*Sw;
        W1 + Ww = W2;
        W1*H1 + Ww*Hw = W2*H2;
      end;

    model SuperHeater2;

      cut InSteam (W, H1, P1)
      cut OutSteam (W, H2, P2)
      cut Heat (Q)
      parameter Cm=0 m=0 K=0 Vs=0 f=0
      local Tm TmH T2 T2H R2
        P1*P1 - P2*P2 = f*W*W;
        (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 Attemperator2;

      cut InSteam (W1, H1, P)
      cut OutSteam (W2, H2, P)
      cut Water (Ww, Hw, Pw)
      parameter Sw=0 fw=0
      local aw
        Pw - P = fw*Ww/aw*Ww/aw;
        aw = Sw*Sw;
        W1 + Ww = W2;
        W1*H1 + Ww*Hw = W2*H2;
      end;

    model SuperHeater3;
```

```
      cut InSteam (W, H1, P1)
      cut OutSteam (W, H2, P2)
      cut Heat (Q)
      parameter Cm=0 m=0 K=0 Vs=0 f=0
      local Tm TmH T2 T2H R2
        P1*P1 - P2*P2 = f*W*W;
        (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;

    cut ISteam {} (Ps1, Hs1, Ws1) {}
    cut OSteam {} (Ps2, Hs2, Ws2) {}
{
    cut Heat (Q1, Q2, Q3)
}
cut Q1 (q1)  Q2 (q2)  Q3 (q3)  Heat [Q1, Q2, Q3] {}
{
    cut InWater (W1, W2)
}
cut W1 (Pw1, Hw1, Ww1)  W2 (Pw2, Hw2, Ww2)  InWater [W1, W2] {}
    connect SuperHeater1:OutSteam at Attemperator1:InSteam;
    connect Attemperator1:OutSteam at SuperHeater2:InSteam;
    connect SuperHeater2:OutSteam at Attemperator2:InSteam;
    connect Attemperator2:OutSteam at SuperHeater3:InSteam;
    connect ISteam at SuperHeater1:InSteam;
    connect SuperHeater3:OutSteam at OSteam;
    connect Q1 at SuperHeater1:Heat;
    connect Q2 at SuperHeater2:Heat;
    connect Q3 at SuperHeater3:Heat;
    connect W1 at Attemperator1:Water;
    connect W2 at Attemperator2:Water;
    end;

  model Boiler;

    model Economizer;

      cut InWater (W, H1, P1)
      cut OutWater (W, H2, P2)
      cut Heat (Q)
      parameter k=0 f=0 Cm=0 m=0 Ve=0
      local Tm T2 R2 T2H TmH
        P2 = P1 - f*W*W;
        (Cm*m*TmH + Ve*R2)*der(H2) = Q + W*H1 - W*H2;
        R2 = RHP(H2, P2);
        T2 = THP(H2, P2);
        T2H = THPH(H2, P2);
        Tm = T2 + k*Q;
        TmH = T2H;
      end;

    model Drum;

      cut OutSteam (Ws, Hs, Pd)
      cut SteamWater {(Steam} [ {} (Wsr, Hsr, Pd, dPd), (Wwr, Hwr) {)} ] {}
      cut OutWater (Wdc, Hdc, Pdc)
      cut InWater (Ww, Hw, Pd)
      parameter Vdc=0 Adrum=0 f=0 L=0 D=0 A=0 Vw0=0 Vs0=0 Aw=0
      constant g=9.81
      local z Rw Vw Vs HsP Rs RsP Rwr Hd
        Pd - Pdc = (1 + f*L/D)*Wdc*Wdc/(2*A*Rw) - g*L*Rw;
```

```
      Aw*der(z)*Rw = Ww + Wwr - Wdc;
      Vw = VwO + Adrum*z;
      (Vw + Vdc)*Rw*der(Hd) = Ww*Hw + Wwr*Hwr - Wdc*Hdc;
      Hdc = Hd;
      -Adrum*der(z)*Rs + Vs*RsP*der(Pd) = Wsr - Ws;
      Vs = VsO - Adrum*z;
      -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 Risers;

    cut SteamWater {(Steam} [ {} (Wsr, Hsr, Pd, dPd), (Wwr, Hwr) {)} ] {}
    cut InWater (Wdc, Hdc, Pdc)
    cut Heat (Q)
    parameter Vr=0 Cm=0 m=0 K=0 f=0 L=0 D=0 A=0
    constant g=9.81
    local Vb x xs TAU Wsprod Wmix Rmix Tm TmP TmixP Rs Rwr RsP RwrP   ->
      Tdc HwrP Hs HsP Tmix
      Pdc - Pd = (1 + f*L/D)*Wmix*Wmix/(2*A*Rmix) + g*L*Rmix;
      Rmix*Vr = Vb*Rs + (Vr - Vb)*Rwr;
      Wmix = Wdc;
      -der(Vb)*Rwr = Wdc - Wsprod - Wwr;
      Wwr = (1 - x)*Wmix;
      der(Vb)*Rs + Vb*RsP*dPd = Wsprod - Wsr;
      Wsr = x*Wmix;
      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 Down;

    cut IWater {} (Pw1, Hw1, Ww1) {}
    cut OWater {} (Pw2, Hw2, Ww2) {}
    connect IWater at OWater;
    end;

  cut OSteam {} (Ps2, Hs2, Ws2) {}
  cut IWater {} (Pw1, Hw1, Ww1) {}
{
  cut Heat (Q1, Q2)
}
cut Q1 (q1)  Q2 (q2)  Heat [Q1, Q2] {}
```

```
    connect Down:OWater at Risers:InWater;
    connect Risers:SteamWater at Drum:SteamWater;
    connect Down:IWater at Drum:OutWater;
    connect Drum:OutSteam at OSteam;
    connect Drum:InWater at Economizer:OutWater;
    connect Economizer:InWater at IWater;
    connect Q1 at Economizer:Heat;
    connect Q2 at Risers:Heat;
    end;

  model Combustion;

{
    cut Boil (QEcon, QRis)
}
cut Boil [ (QEcon), (QRis) ] {}
{
    cut Heat (QSup3, QSup2, QSup1)
}
cut Heat [ (QSup3), (QSup2), (QSup1) ] {}
{
    cut Reheat (QRe)
}
cut Reheat [(QRe)] {}
    parameter Woil=0 b10=0 b20=0 b30=0 b40=0 b50=0 b60=0 b11=0 b21=  ->
      0 b31=0 b41=0 b51=0 b61=0 b12=0 b22=0 b32=0 b42=0 b52=0 b62=  ->
      0
      QRis = b10 + b11*Woil + b12*Woil*Woil;
      QEcon = b20 + b21*Woil + b22*Woil*Woil;
      QSup1 = b30 + b31*Woil + b32*Woil*Woil;
      QSup2 = b40 + b41*Woil + b42*Woil*Woil;
      QSup3 = b50 + b51*Woil + b52*Woil*Woil;
      QRe = b60 + b61*Woil + b62*Woil*Woil;
    end;

  model Turbines;

    model HPturb;

      cut InSteam (W1, H1, P1)
      cut InPower (N1)
      cut OutSteam (W2, H2, P2)
      cut OutPower (N2)
      cut Extract (Wp, H2, Pp)
      parameter S=0 f=0 fp=0 Eh=0
      local H T2 ap
        P1 = f*W1;
        P2*P2 - Pp*Pp = fp*Wp/ap*Wp/ap;
        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 Reheater;

      cut Heat (Q)
      cut InSteam (W1, H1, P1)
      cut OutSteam (W2, H2, P2)
      parameter Cm=0 m=0 Vs=0 K=0 f=0
      local T2 T2H Tm TmH R2 R2H R2P R2T
        P1*P1 - P2*P2 = f*W1*W1;
        Vs*R2T = W1 - W2;
        m*Cm*TmH*der(H2) + Vs*(R2T*H2 + R2*der(H2)) = Q + W1*H1 -   ->
```

```
    W2*H2;
    Tm = T2 + K*Q;
    TmH = T2H;
    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 IPturb;

  model IP1;

    cut InSteam (W1, H1, P1)
    cut InPower (N1)
    cut OutSteam (W2, H2, P2)
    cut OutPower (N2)
    cut Extract (Wp, H2, Pp)
    parameter S=0 f=0 fp=0 Eh=0
    local H T2 ap
      P1 = f*W1;
      P2*P2 - Pp*Pp = fp*Wp/ap*Wp/ap;
      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 IP2;

    cut InSteam (W1, H1, P1)
    cut InPower (N1)
    cut OutSteam (W2, H2, P2)
    cut OutPower (N2)
    cut Extract (Wp, H2, Pp)
    parameter S=0 f=0 fp=0 Eh=0
    local H T2 ap
      P1 = f*W1;
      P2*P2 - Pp*Pp = fp*Wp/ap*Wp/ap;
      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 IP3;

    cut InSteam (W1, H1, P1)
    cut InPower (N1)
    cut OutSteam (W2, H2, P2)
    cut OutPower (N2)
    cut Extract (Wp, H2, Pp)
    parameter S=0 f=0 fp=0 Eh=0
    local H T2 ap
      P1 = f*W1;
      P2*P2 - Pp*Pp = fp*Wp/ap*Wp/ap;
      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 IP4;

      cut InSteam (W1, H1, P1)
      cut InPower (N1)
      cut OutSteam (W2, H2, P2)
      cut OutPower (N2)
      cut Extract (Wp, H2, Pp)
      parameter S=0 f=0 fp=0 Eh=0
      local H T2 ap
        P1 = f*W1;
        P2*P2 - Pp*Pp = fp*Wp/ap*Wp/ap;
        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;

    cut ISteam {} (Ps1, Hs1, Ws1) {}
    cut IPower {} (N1) {}
    cut OSteam {} (Ps2, Hs2, Ws2) {}
    cut OPower {} (N2) {}
{
    cut IPext (IPE1, IPE2, IPE3, IPE4)
}
cut IPE1 (Wei1, Hei1, Pei1)
cut IPE2 (Wei2, Hei2, Pei2)
cut IPE3 (Wei3, Hei3, Pei3)
cut IPE4 (Wei4, Hei4, Pei4)
cut IPext [IPE1, IPE2, IPE3, IPE4]
{}
    connect IP1:InPower at IPower;
    connect IP1:InSteam at ISteam;
    connect IP1:OutSteam at IP2:InSteam;
    connect IP1:OutPower at IP2:InPower;
    connect IP3:InSteam at IP2:OutSteam;
    connect IP3:InPower at IP2:OutPower;
    connect IP4:InPower at IP3:OutPower;
    connect IP4:InSteam at IP3:OutSteam;
    connect IP4:OutSteam at OSteam;
    connect IP4:OutPower at OPower;
    connect IP1:Extract at IPE1;
    connect IP2:Extract at IPE2;
    connect IP4:Extract at IPE4;
    connect IP3:Extract at IPE3;
    end;

  model LPturb;

    model LP1;

      cut InSteam (W1, H1, P1)
      cut InPower (N1)
      cut OutSteam (W2, H2, P2)
      cut OutPower (N2)
      cut Extract (Wp, H2, Pp)
      parameter S=0 f=0 fp=0 Eh=0
      local H T2 ap
        P1 = f*W1;
        P2*P2 - Pp*Pp = fp*Wp/ap*Wp/ap;
        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 LP2;

      cut InSteam (W1, H1, P1)
      cut InPower (N1)
      cut OutSteam (W2, H2, P2)
      cut OutPower (N2)
      cut Extract (Wp, H2, Pp)
      parameter S=0 f=0 fp=0 Eh=0
      local H T2 ap
        P1 = f*W1;
        P2*P2 - Pp*Pp = fp*Wp/ap*Wp/ap;
        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 LP3;

      cut InSteam (W1, H1, P1)
      cut InPower (N1)
      cut OutSteam (W2, H2, P2)
      cut OutPower (N2)
      cut Extract (Wp, H2, Pp)
      parameter S=0 f=0 fp=0 Eh=0
      local H T2 ap
        P1 = f*W1;
        P2*P2 - Pp*Pp = fp*Wp/ap*Wp/ap;
        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;

    cut ISteam {} (Ps1, Hs1, Ws1) {}
    cut IPower {} (N1) {}
    cut OSteam {} (Ps2, Hs2, Ws2) {}
    cut OPower {} (N2) {}
{
    cut LPext (LPE1, LPE2)
}
cut LPE1 (Wel1, Hel1, Pel1)
cut LPE2 (Wel2, Hel2, Pel2)
cut LPext [LPE1, LPE2]
{}
    connect LP1:InPower at IPower;
    connect LP1:InSteam at ISteam;
    connect LP1:OutPower at LP2:InPower;
    connect LP3:InPower at LP2:OutPower;
    connect LP2:InSteam at LP1:OutSteam;
    connect LP3:InSteam at LP2:OutSteam;
    connect LP3:OutSteam at OSteam;
    connect LP3:OutPower at OPower;
    connect LP1:Extract at LPE1;
    connect LP2:Extract at LPE2;
```

```
      end;

    cut ISteam {} (Ps1, Hs1, Ws1) {}
    cut OSteam {} (Ps2, Hs2, Ws2) {}
{
    cut Heat (Q1)
}
cut Q1 (q)  Heat [Q1] {}
{
    cut Extract (HPext, IPext, LPext)
}
cut HPext (Weh, Heh, Peh)
cut IPE1 (Wei1, Hei1, Pei1)
cut IPE2 (Wei2, Hei2, Pei2)
cut IPE3 (Wei3, Hei3, Pei3)
cut IPE4 (Wei4, Hei4, Pei4)
cut IPext [IPE1, IPE2, IPE3, IPE4]
cut LPE1 (Wel1, Hel1, Pel1)
cut LPE2 (Wel2, Hel2, Pel2)
cut LPext [LPE1, LPE2]
cut Extract [HPext, IPext, LPext]
{}
    connect HPturb:InSteam at ISteam;
    connect HPturb:OutSteam at Reheater:InSteam;
    connect Reheater:Heat at Q1;
    connect IPturb:ISteam at Reheater:OutSteam;
    connect HPturb:OutPower at IPturb:IPower;
    connect IPturb:OSteam at LPturb:ISteam;
    connect IPturb:OPower at LPturb:IPower;
    connect HPturb:Extract at HPext;
    connect IPturb:IPext at IPext;
    connect LPturb:LPext at LPext;
    connect LPturb:OSteam at OSteam;
    end;

  model FeedWater;

    model Condensor;

      cut InSteam (Ws, Hs, Ps)
      cut Feed (Ww, Hw, Pw)
      cut Cond (Wc, Hc)
      parameter W1=0 H1=0 P1=0 Hdiff=0 Vc=0 m=0 Cm=0 Vcool=0 Pdiff=  ->
        0
      local Rw R2 Tw T1 T2 HwP TwP Tc TmP H2 Pc
        Pc = Pw + Pdiff;
        Ps = Pc;
        Ws + Wc = Ww;
        (Vc*Rw*HwP + m*Cm*TmP + Vcool*R2*HwP)*der(Pw) = Ws*Hs + Wc*  ->
        Hc - Ww*Hw - W1*(H2 - H1);
        H2 = Hw - Hdiff;
        TmP = TwP;
        Hw = IHWP(Pw);
        HwP = HWPP(Pw);
        Rw = RWP(Pw);
        R2 = RHP(H2, P1);
        Tw = TLP(Pw);
        TwP = TLPP(Pw);
        T1 = THP(H1, P1);
        T2 = THP(H2, P1);
        Hc = IHWP(Pc);
        Tc = TLP(Pc);
      end;

    model PreHeat1;
```

```
  cut Feed1 (W, H1, P1)
  cut Cond1 (Wc1, Hc1)
  cut Feed2 (W, H2, P2)
  cut Cond2 (Wc2, Hc2)
  cut Extract (Ws, Hs, Psat)
  parameter Vc=0 Vw=0 Hdiff=0 f=0 m=0 Cm=0
  local Tsat TsatP Rw T2 Rc Hsat HsatP
    P1*P1 - P2*P2 = f*W*W;
    Wc2 = Wc1 + Ws;
    (Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)*der(Psat) = Ws*Hs +   ->
    Wc1*Hsat - W*(H2 - H1) - Wc2*Hc2;
    Rc = RWP(Psat);
    Hsat = HWP(Psat);
    HsatP = HWPP(Psat);
    Tsat = TLP(Psat);
    TsatP = TLPP(Psat);
    Rw = RHP(H2, P2);
    H2 = Hsat - Hdiff;
    Hc1 = Hsat;
    T2 = THP(H2, P1);
  end;

model PreHeat2;

  cut Feed1 (W, H1, P1)
  cut Cond1 (Wc1, Hc1)
  cut Feed2 (W, H2, P2)
  cut Cond2 (Wc2, Hc2)
  cut Extract (Ws, Hs, Psat)
  parameter Vc=0 Vw=0 Hdiff=0 f=0 m=0 Cm=0
  local Tsat TsatP Rw T2 Rc Hsat HsatP
    P1*P1 - P2*P2 = f*W*W;
    Wc2 = Wc1 + Ws;
    (Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)*der(Psat) = Ws*Hs +   ->
    Wc1*Hsat - W*(H2 - H1) - Wc2*Hc2;
    Rc = RWP(Psat);
    Hsat = HWP(Psat);
    HsatP = HWPP(Psat);
    Tsat = TLP(Psat);
    TsatP = TLPP(Psat);
    Rw = RHP(H2, P2);
    H2 = Hsat - Hdiff;
    Hc1 = Hsat;
    T2 = THP(H2, P1);
  end;

model PreHeat3;

  cut Feed1 (W, H1, P1)
  cut Cond1 (Wc1, Hc1)
  cut Feed2 (W, H2, P2)
  cut Cond2 (Wc2, Hc2)
  cut Extract (Ws, Hs, Psat)
  parameter Vc=0 Vw=0 Hdiff=0 f=0 m=0 Cm=0
  local Tsat TsatP Rw T2 Rc Hsat HsatP
    P1*P1 - P2*P2 = f*W*W;
    Wc2 = Wc1 + Ws;
    (Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)*der(Psat) = Ws*Hs +   ->
    Wc1*Hsat - W*(H2 - H1) - Wc2*Hc2;
    Rc = RWP(Psat);
    Hsat = HWP(Psat);
    HsatP = HWPP(Psat);
    Tsat = TLP(Psat);
    TsatP = TLPP(Psat);
    Rw = RHP(H2, P2);
    H2 = Hsat - Hdiff;
```

```
    Hc1 = Hsat;
    T2 = THP(H2, P1);
  end;

model PreHeat4;

  cut Feed1 (W, H1, P1)
  cut Cond1 (Wc1, Hc1)
  cut Feed2 (W, H2, P2)
  cut Cond2 (Wc2, Hc2)
  cut Extract (Ws, Hs, Psat)
  parameter Vc=0 Vw=0 Hdiff=0 f=0 m=0 Cm=0
  local Tsat TsatP Rw T2 Rc Hsat HsatP
    P1*P1 - P2*P2 = f*W*W;
    Wc2 = Wc1 + Ws;
    (Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)*der(Psat) = Ws*Hs +   ->
    Wc1*Hsat - W*(H2 - H1) - Wc2*Hc2;
    Rc = RWP(Psat);
    Hsat = HWP(Psat);
    HsatP = HWPP(Psat);
    Tsat = TLP(Psat);
    TsatP = TLPP(Psat);
    Rw = RHP(H2, P2);
    H2 = Hsat - Hdiff;
    Hc1 = Hsat;
    T2 = THP(H2, P1);
  end;

model PreHeat5;

  cut Feed1 (W, H1, P1)
  cut Cond1 (Wc1, Hc1)
  cut Feed2 (W, H2, P2)
  cut Cond2 (Wc2, Hc2)
  cut Extract (Ws, Hs, Psat)
  parameter Vc=0 Vw=0 Hdiff=0 f=0 m=0 Cm=0
  local Tsat TsatP Rw T2 Rc Hsat HsatP
    P1*P1 - P2*P2 = f*W*W;
    Wc2 = Wc1 + Ws;
    (Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)*der(Psat) = Ws*Hs +   ->
    Wc1*Hsat - W*(H2 - H1) - Wc2*Hc2;
    Rc = RWP(Psat);
    Hsat = HWP(Psat);
    HsatP = HWPP(Psat);
    Tsat = TLP(Psat);
    TsatP = TLPP(Psat);
    Rw = RHP(H2, P2);
    H2 = Hsat - Hdiff;
    Hc1 = Hsat;
    T2 = THP(H2, P1);
  end;

model PreHeat6;

  cut Feed1 (W, H1, P1)
  cut Cond1 (Wc1, Hc1)
  cut Feed2 (W, H2, P2)
  cut Cond2 (Wc2, Hc2)
  cut Extract (Ws, Hs, Psat)
  parameter Vc=0 Vw=0 Hdiff=0 f=0 m=0 Cm=0
  local Tsat TsatP Rw T2 Rc Hsat HsatP
    P1*P1 - P2*P2 = f*W*W;
    Wc2 = Wc1 + Ws;
    (Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)*der(Psat) = Ws*Hs +   ->
    Wc1*Hsat - W*(H2 - H1) - Wc2*Hc2;
    Rc = RWP(Psat);
```

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

    model PreHeat7;

      cut Feed1 (W, H1, P1)
      cut Cond1 (Wc1, Hc1)
      cut Feed2 (W, H2, P2)
      cut Cond2 (Wc2, Hc2)
      cut Extract (Ws, Hs, Psat)
      parameter Vc=0 Vw=0 Hdiff=0 f=0 m=0 Cm=0
      local Tsat TsatP Rw T2 Rc Hsat HsatP
        P1*P1 - P2*P2 = f*W*W;
        Wc2 = Wc1 + Ws;
        (Vc*Rc*HsatP + m*Cm*TsatP + Vw*Rw*HsatP)*der(Psat) = Ws*Hs +   ->
        Wc1*Hsat - W*(H2 - H1) - Wc2*Hc2;
        Rc = RWP(Psat);
        Hsat = HWP(Psat);
        HsatP = HWPP(Psat);
        Tsat = TLP(Psat);
        TsatP = TLPP(Psat);
        Rw = RHP(H2, P2);
        H2 = Hsat - Hdiff;
        Hc1 = Hsat;
        T2 = THP(H2, P1);
      end;

    model Split;

      cut Extract (W, H, P)
      cut Extract1 (W1, H, P)
      cut Extract2 (W2, H, P)
        W1 = 0.3*W;
        W2 = 0.7*W;
      end;

    model Dearator;

      cut Feed1 (W, H1, P1)
      cut Feed2 (W2, H2, x1)
      cut Cond (Wc, Hc)
      cut Extract (Ws, Hs, x2)
      parameter Wwater=0 Hstorage=0 Vw=0 m=0 Cm=0 Pdiff=0
      local Rw H2P T2 T2P Psat Tc Hwater Pc
        W2 = W + Wc + Ws + Wwater;
{
        Hwater = IF Wwater > 0 THEN Hstorage ELSE H2 END IF;
}
Hwater = H2;
{}
        (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 Pump;

      cut Feed1 (W, H, x)
      cut Feed2 (W, H, P2)
      parameter Ppump=0 f=0
        P2 = Ppump - f*W*W;
      end;

    cut ISteam {} (Ps1, Hs1, Ws1) {}
    cut OWater {} (Pw2, Hw2, Ww2) {}
{
    cut Extract (HPext, IPext (IPE1, IPE2, IPE3, IPE4), LPext (LPE1,   ->
    LPE2))
}
cut HPext (Weh, Heh, Peh)
cut IPE1 (Wei1, Hei1, Pei1)
cut IPE2 (Wei2, Hei2, Pei2)
cut IPE3 (Wei3, Hei3, Pei3)
cut IPE4 (Wei4, Hei4, Pei4)
cut IPext [IPE1, IPE2, IPE3, IPE4]
cut LPE1 (Wel1, Hel1, Pel1)
cut LPE2 (Wel2, Hel2, Pel2)
cut LPext [LPE1, LPE2]
cut Extract [HPext, IPext, LPext]
{}
    connect ISteam at Condensor:InSteam;
    connect Condensor:Feed at PreHeat1:Feed1;
    connect PreHeat1:Feed2 at PreHeat2:Feed1;
    connect PreHeat2:Feed2 at PreHeat3:Feed1;
    connect PreHeat3:Feed2 at Dearator:Feed1;
    connect Dearator:Feed2 at Pump:Feed1;
    connect Pump:Feed2 at PreHeat4:Feed1;
    connect PreHeat4:Feed2 at PreHeat5:Feed1;
    connect PreHeat5:Feed2 at PreHeat6:Feed1;
    connect PreHeat6:Feed2 at PreHeat7:Feed1;
    connect PreHeat7:Feed2 at OWater;
    connect PreHeat7:Cond2 at PreHeat6:Cond1;
    connect PreHeat6:Cond2 at PreHeat5:Cond1;
    connect PreHeat5:Cond2 at PreHeat4:Cond1;
    connect PreHeat4:Cond2 at Dearator:Cond;
    connect PreHeat3:Cond2 at PreHeat2:Cond1;
    connect PreHeat2:Cond2 at PreHeat1:Cond1;
    connect PreHeat1:Cond2 at Condensor:Cond;
    connect Split:Extract at IPE4;
    connect Split:Extract1 at Dearator:Extract;
    connect Split:Extract2 at PreHeat3:Extract;
    connect PreHeat4:Extract at IPE3;
    connect PreHeat5:Extract at IPE2;
    connect PreHeat6:Extract at IPE1;
    connect PreHeat7:Extract at HPext;
    connect PreHeat2:Extract at LPE1;
    connect PreHeat1:Extract at LPE2;
    end;

  model SteamValve;

    cut InSteam (W, H, P1)
    cut OutSteam (W, H, P2)
    parameter Sv=0 fv=0
    local av
      P1*P1 - P2*P2 = fv*W/av*W/av;
      av = VALVE(Sv);
```

```
    end;

  model FeedValve;

    cut InWater (W, H, P1)
    cut OutWater2 {(Atemp2} [ {} (Wa2, H, P2), (Wa1, H, P2) {)} ] {}
    cut OutWater1 (Wd, H, P2)
    parameter a=0 f=0
      P2 = P1 - f*W/a*W/a;
      Wd + Wa1 + Wa2 = W;
    end;

  connect Heaters:ISteam at Boiler:OSteam;
  connect Heaters:InWater at FeedValve:OutWater2;
  connect Boiler:IWater at FeedValve:OutWater1;
  connect Combustion:Heat at Heaters:Heat;
  connect Combustion:Boil at Boiler:Heat;
  connect SteamValve:InSteam at Heaters:OSteam;
  connect SteamValve:OutSteam at Turbines:ISteam;
  connect Combustion:Reheat at Turbines:Heat;
  connect Turbines:Extract at FeedWater:Extract;
  connect Turbines:OSteam at FeedWater:ISteam;
  connect FeedValve:InWater at FeedWater:OWater;
{}
Turbines::HPturb.N1 = 0;
Turbines::LPturb::LP3.Wp = 0;
FeedWater::Dearator.x1 = 0;
{}
  end;
```