Bond Graph Theory and its Application to Computer Aided Modelling

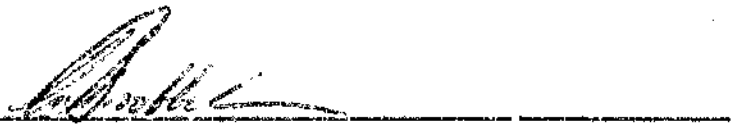
Grant Bruce Grobbelaar

A dissertation submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, 1991

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science in Engineering at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.



Grant Bruce Grobbelaar

30 day of August 19 91

Abstract

The theory required to model a system using a bond graph is presented and a computer aided bond graph modelling program is implemented. The program allows the bond graph to be entered and modified and automatically assigns the causality. Using the bond graph and the equations entered by the user, the program creates an ACSL input file and performs an ACSL simulation. The program is able to deal with systems containing algebraic loops and derivative causality. The LISP source code is included in the dissertation. Numerous bond graph models of systems in different domains are presented and examples of how the program is used to model, simulate and obtain the linear state-space equations of a number of systems are shown. The bond graph is a powerful multidomain model representation, particularly when incorporated in a computer aided modelling program.

Keywords: Bond graph, modelling, simulation, causality, computer aided modelling, multidomain representation.

Acknowledgements

I would like to express my gratitude to my supervisor, Mr A.L. Stevens, for his guidance, assistance and dedication.

Thanks also to my friends and colleagues for their valuable comments and discussions.

I would also like to acknowledge the financial assistance received from the Foundation for Research Development, the University of the Witwatersrand, the Standard Bank of Africa Limited and the A.M. Jacobs Scholarship.

Finally, my sincere thanks to my family for their continual support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to bond graphs

In order to design a controller or simulate a system, a mathematical model of the process is required. A mathematical model is usually written as a set of differential equations relating the outputs to the inputs.

Obtaining the mathematical model is however as much of an art as it is a science. This is so for a number of reasons:

- There is no definite sequence of events to follow.
- A thorough understanding of the process is usually required and the detailed behaviour must be known.
- The modeller has to know which effects must be included in the model because they significantly influence the behaviour of a system and which effects can be ignored, because their influence is negligible.

The bond graph modelling technique attempts to address some of these difficulties.

The bond graph is a multi-domain model representation, which uses symbols, to characterise the behaviour of a system. Consider the bond graph shown in Figure 1.1. This bond graph

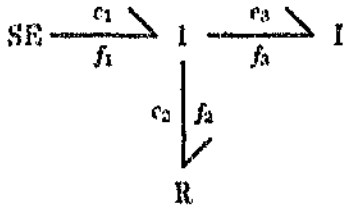


Figure 1.1: A bond graph.

consists of four different elements; SE, R, I and 1, linked together by half arrows, called bonds, representing the *power transfer*. The variables on either side of each bond are termed power variables, because their product is the instantaneous power. That is, the product of e_1 and f_1 is the power leaving the effort source (SE), the product of e_2 and f_2 is the power entering the resistor (R) and so on. If the bond graph were a model of an electrical circuit,

the effort, e would represent a voltage and the flow, f would represent a current, the product of the voltage and the current being the power.

There are nine different bond graph elements, each one having a different type of relation governing its behaviour. For example, in Figure 1.1, the resistor (R), has a relation between e_2 and f_2 and the inertia (I) has a relation between f_3 and $\int e_3\,dt$. Because each bond graph element has a particular relation, the differential equations governing the behaviour of the system are obtained from the bond graph.

Depending on the accuracy of the model required, an aspect, component or subsystem of a plant is represented by a bond graph element. That is, in order to increase the accuracy of the model, each bond graph element has to represent a smaller part of the system. For example, an inductor in an electrical circuit can be represented by a bond graph inertia. If a more accurate model is required, other bond graph elements can be added to the inertia. For example, a bond graph resistor could be included to account for the resistance of the coil. In a similar manner, if a less accurate model is required, one bond graph element can represent a number of circuit elements.

If each part of a system is represented by a bond graph element, these elements can be connected together in a logical manner (based on their connections in the system) to form the bond graph model of the system. The bond graph modelling technique therefore follows a logical progression.

Once the bond graph structure has been obtained, an exact equation has to be found to represent each element. This is also not an easy step in general, because the systems studied (particularly process systems), are often very complex and in some cases, the equations are not known at all. The bond graph modeling technique does simplify this step because the model is subdivided into a number of bond graph elements. The modeller can therefore study each part of the system, represented by a bond graph element (for which the form of the equation is known), separately.

Using these equations and the cause-effect relationships, derived from the structure of the bond graph, the differential equations governing the behaviour of the system are systematically obtained. These differential equations can be used to simulate the system.

If the model is not a sufficiently accurate representation of the system, the bond graph can be easily modified, by removing or adding elements. The new differential equations can be obtained and the system resimulated.

The bond graph graph modelling technique is therefore well ordered and allows iterative model development.

1.2 Computer aided modelling using bond graphs

The bond graph modelling technique, is a technique particularly well suited to computer implementation for the following reasons:

- A bond graph consists of a limited number of different bond graph elements connected together by half arrows representing power flow. A bond graph can therefore be clearly displayed on a computer screen and easily entered from the keyboard.
- Since modelling is an iterative process, it may be necessary to change the bond graph structure a number of times before it is an adequate representation of the system.

Because of the inherent ability of a computer to store and retrieve data, a computer implementation is particularly valuable.

- As computers have the ability to store large amounts of data, bond graph models can be constructed by connecting together previously stored bond graph fragments.
- The *causality assignment procedure* is a sequential procedure, governed by a well defined set of rules. Such a procedure is easily implemented in a computer program.
- The equation formulation routine consists mainly of equation substitutions. Software and simulation languages in particular, are written to solve such problems.

The bond graph modelling technique therefore constitutes a powerful computer aided modelling tool.

1.3 Historical development of bond graphs

Bond graphs were invented by Henry Paynter in 1955 [22] and his first paper on the subject appeared in 1961. It was in this paper, that the concepts of causality were first discussed. Two of Paynter's students, Dean Karnopp and Ronald Rosenberg are, however, credited with developing the rules and symbols of the bond graph as it is known today. Their first book on bond graphs appeared in 1968 [4].

The number of bond graph papers published annually increased dramatically in the early 1970's, possibly due to an issue of "The Transactions of ASME, System Dynamics, Measurement and Control" [13] being devoted to bond graphs. It was in this period, that most of the theory was developed.

However, it is the author's opinion that Karnopp and Rosenberg's text book, "System Dynamics, A Unified Approach" [14], published in 1975, caused the bond graph to be recognized as a major modelling tool. The book covers all aspects of bond graph theory and presents numerous bond graph modelling examples.

The majority of papers published after 1975, are concerned with modelling a particular system or a particular type of system using bond graphs (see bibliographies [4,38]). Most of the systems modelled are relatively complex and often involve heat and mass transfer or have parameters that are distributed in space (for example [21,22,31]). During this period, the papers published on bond graph theory were concerned mainly with the junction structure, algebraic loops and derivative causality (for example [23,25]).

Although ENPORT[24], a computer program which simulates a system modelled by a bond graph was written by Rosenberg in 1974, only later did computer aided modelling and simulation using bond graphs become widely accepted [9,32]. This is probably due to the increasing availability and decreasing cost and size of the computers required for this purpose. The field of computer aided modelling and simulation continues to grow with papers constantly being published (for example [10,11]).

Lately there has been an increasing number of papers published describing the mathematical topology of bond graphs. Also, more papers have been published on topics closely related to control theory, for example stability, controllability, observability and eigenvectors of bond graphs (for example [6,37]).

1.4 Motivation for the research

The Control Group of the Department of Electrical Engineering at the University of the Witwatersrand is involved primarily with research into control system design for process plants. This field is diverse and stretches from system modelling through to controller design.

Because process control systems are inherently large and complex, no single technique encompasses all the requirements for control system design. The development of a Computer Aided Process Modelling And Controller Design (CAPMACD) package [29] has therefore been proposed. There are presently a number of people developing and evaluating modules for this package. The bond graph forms one of these modules.

1.5 Objectives of the research

Although the amount of literature published on bond graphs is vast, the modelling technique is still relatively unknown, particularly in the process control field. One of the objectives is therefore to present a concise, coherent body of theory. Using this theory, bond graph models can be created and the implications of these models studied.

The other primary objective was to develop a computer program to automate the bond graph modelling process. This will facilitate rapid model creation, modification and validation. The program will be written so that it can be included in the CAPMACD package.

1.6 Outline of the dissertation

Chapter 2 contains the relevant bond graph theory. Included in this chapter are sections on the definition of a bond graph, the variables used in bond graph theory, the definition of causality, the basic 1-port elements, bond graph fields, the possible causal forms that the 1-port and field elements can be assigned, the Sequential Causality Assignment Procedure and the procedure used to obtain the equations of state.

A number of bond graph models are presented in Chapter 3. This chapter is divided into six sections.

1. An electrical circuit problem is shown in the first section.
2. The second section contains a mechanical translation and mechanical rotation example.
3. The third section is devoted to modelling incompressible fluid flow systems.
4. Heat transfer systems, with no mass flow are discussed in the fourth section.
5. The fifth section is concerned with the theory of modelling compressible thermofluid systems using pseudo bond graphs. Included in this section are a number of pseudo bond graph models of basic process "building blocks".
6. The last section demonstrates, using examples, how these "building blocks" are used to construct larger systems.

Chapter 4 is a discussion of the bond graph modelling program. In this chapter, there is a section devoted to each of the four program modules; entering and modifying, causality assignment, ACSL input file creation and ACSL execution. In addition to these four sections, there are sections on LISP, ACSL, the numbering scheme used in the program and the data structures used. The last section in the chapter lists the limitations of the program.

Chapter 5 illustrates the use of the program, by showing the steps required to model and simulate a parallel connected tank system.

Chapter 6 shows, by means of an example, the importance of a knowledge of the causality prior obtaining the system equations.

The conclusions on bond graph theory and computer aided modelling using the bond graph program are drawn in Chapter 7.

Chapter 8 contains the recommendations for further work and highlights possible inclusions to the bond graph program as well as further research required in bond graph theory.

Following the references are four appendices. The first shows the alterations made by the author to the transition table implementation of the Sequential Causality Assignment Procedure[10]. The second appendix contains a number of application examples, demonstrating the capabilities of the bond graph program. The third appendix is the bond graph program user manual. The last appendix contains the program listings.

Chapter 2

Bond graph theory

2.1 Bond graph generalized variables – efforts, flows, momenta and displacements

When any two dynamical systems are connected together, to form a larger system, power is transferred between them. The representation of this power flow is the basis of bond graph theory. It follows therefore, that the bond graph can be used to represent systems in all domains where power transfer occurs.

The interface between one system and another, facilitating power transfer, is termed a port and a system with one or more ports is termed a multiport. Consider as an example of an engineering multiport, a separately excited direct current (d.c.) motor. The d.c. motor has three external ports, the armature winding, the field winding and the shaft and is therefore termed a 3-port. All three of these ports may be connected to other systems with a resulting power flow between them.

The instantaneous power is usually expressed in terms of the product of two variables, called power variables. For example, the instantaneous power at an electrical port is the voltage multiplied by the current and the instantaneous power at a mechanical rotation port is the torque multiplied by the angular velocity. As the aim is to describe all physical systems using a universal language, the power variables will be termed either *effort e(t)* or *flow f(t)*. Table 2.1 shows effort and flow variables for several domains.

Domain	Effort $e(t)$	Flow $f(t)$
Mechanical translation	Force $F(t)$	Velocity $v(t)$
Mechanical rotation	Torque $\tau(t)$	Angular velocity $\omega(t)$
Hydraulic	Pressure $p(t)$	Volume flow rate $q(t)$
Electric	Voltage $V(t)$	Current $i(t)$
Thermal	Temperature $T(t)$	Entropy flow rate $\dot{S}(t)$
Chemical	Chemical potential $\mu(t)$	Molar flow rate $\dot{n}(t)$

Table 2.1: Effort and flow quantities in several domains.

The effort and flow variables of mechanical systems are usually defined as in Table 2.1, but in certain texts [33], the effort is defined to be the velocity and the flow is defined to be the

force. This is because the effort variable is an "across variable" (the measurement on the one side of an element is defined relative to the measurement on the other side, for example, the voltage across a resistor) and the flow variable is a "through variable" (the measurement is defined relative to some reference, normally zero, the current flowing through a resistor is an example). The definitions of an effort and a flow in [33] are therefore compatible with these definitions. However, in this dissertation, the efforts and flows will be defined as in Table 2.1, because this is more common.

Since the power can flow in either direction, a sign convention must be established.

As dynamic systems are dealt with, the effort and flow variables and hence the power fluctuate with time. The power $P(t)$ is given by.

$$P(t) = e(t)f(t) \tag{2.1}$$

Two other variables are required to describe dynamical systems, they are the *momentum* $p(t)$ and the *displacement* $q(t)$. These two variables are known as *energy variables* [14].

The momentum is defined as the time integral of effort,

$$p(t) = \int e(t)dt \tag{2.2}$$

and can be written using the definite integral

$$p(t) = p_o + \int_{t_o}^{t} e(t)dt \tag{2.3}$$

where p_o is the momentum at time t_o (the "initial" momentum).

The displacement is defined as the time integral of flow,

$$q(t) = \int f(t)dt \tag{2.4}$$

and as in the case of the momentum, it may be written using a definite integral

$$q(t) = q_o + \int_{t_o}^{t} f(t)dt \tag{2.5}$$

where q_o is the displacement at time t_o (the "initial" displacement).

Table 2.2 shows the momentum and displacement variables in several energy domains.

Domain	Momentum $p(t)$	Displacement $q(t)$
Mechanical translation	Momentum $p(t)$	Displacement $x(t)$
Mechanical rotation	Angular momentum $p_r(t)$	Angle $\theta(t)$
Hydraulic	Pressure momentum $p_p(t)$	Volume $V(t)$
Electric	Flux linkage $\lambda(t)$	Charge $q(t)$
Thermal	-	Entropy $S(t)$
Chemical	-	Molar mass $n(t)$

Table 2.2: Momentum and displacement quantities in several domains.

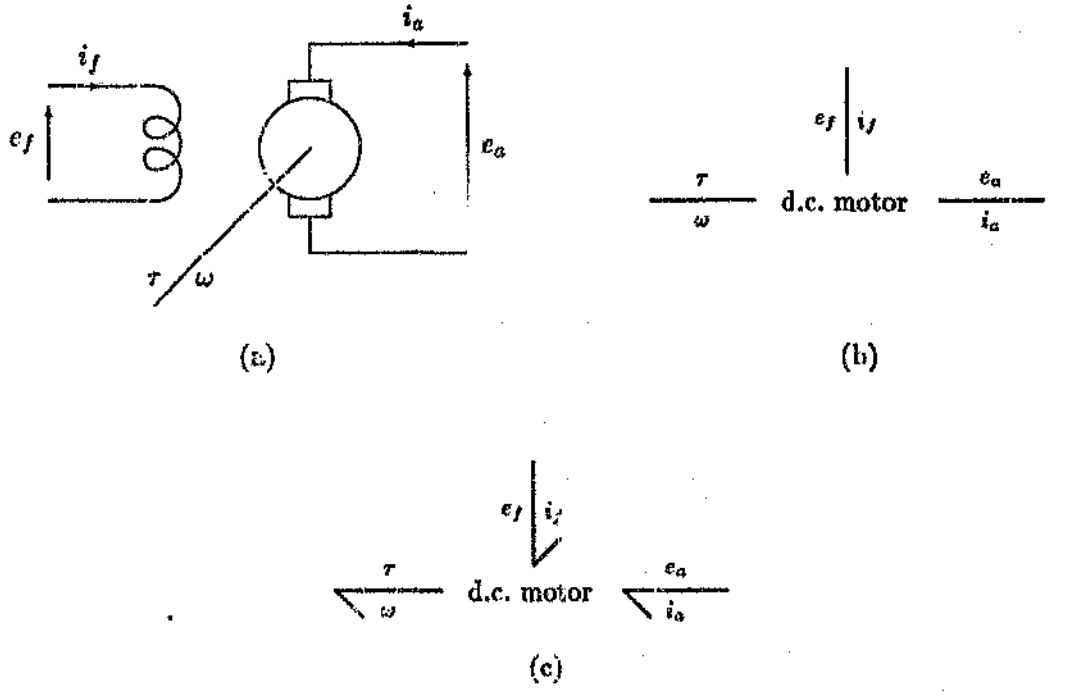


Figure 2.1: A separately excited d.c. motor. (a) Usual diagrammatical representation. (b) Multiport representation. (c) Multiport representation with the sign convention added.

2.2 Bonds and power

Consider the separately excited d.c. motor shown in Figure 2.1a. As mentioned previously, it has three external ports, two electrical and one mechanical. Figure 2.1b is another representation of the motor, where each port is represented by a single line and its name is used to stand for the device. The power variables are written next to the ports for convenience. The power variables are positioned according to the following conventions:

- Efforts are placed either *above* or to the *left* of port lines.
- Flows are placed either *below* or to the *right* of port lines.

In Figure 2.1c, the power sign convention has been added to the port lines. The half arrow on the port lines indicates the direction of power flow when both power variables are positive, or when they are both negative. For example, if ω and τ are positive in the direction shown and the product of ω and τ is positive, the power flows out of the motor, through the shaft.

When two multiports are connected together, the effort variables are constrained to be equal and the flow variables are constrained to be equal. The multiports are then said to have a common *bond*, in analogy to a chemical bond between molecules.

Figure 2.2 shows three multiports, a separately excited d.c. motor, a drive shaft and a hydraulic pump, connected together. The motor and the one end of the shaft have a common angular velocity and torque. The hydraulic pump and the other end of the shaft share a common angular velocity and torque. These angular velocities may or may not be equal depending on whether the shaft is rigid or not. The systems to which the electrical and hydraulic ports are connected are not shown in this example.

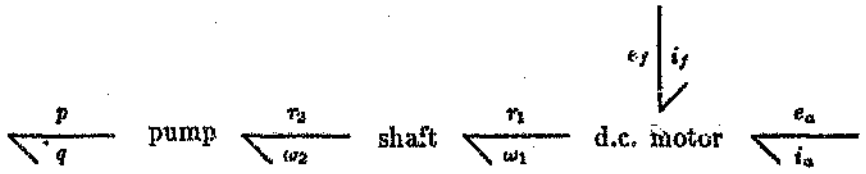


Figure 2.2: A Connected system.

Figure 2.2 is a *bond graph*, that is, it is diagram of components and subsystems linked together by lines representing power flow. When, as in Figure 2.2, the subsystems are represented by words, the diagram is called a *word bond graph*. A word bond graph only depicts the manner in which subsystems are connected.

2.3 Cause and effect relationships

Although an effort and a flow exist at every port, it is not possible to control both variables independently. For example, if a voltage source is connected to the armature port of a d.c. motor, the source impresses a voltage on the motor and the motor responds with an armature current. That is, the voltage is the input to the motor and the current is the output from the motor.

A bond graph uses a *causal stroke* to specify which power variable is the input and which is the output. The causal stroke is a short line perpendicular to the bond drawn at either end, indicating, by convention, the direction of the effort signal (by implication, the flow signal is directed away from the causal stroke).

Figure 2.3 shows the bond graph of a d.c. motor and a voltage source and includes the causal stroke. The stroke indicates that the voltage is the input to the motor and the current is the output from the motor.

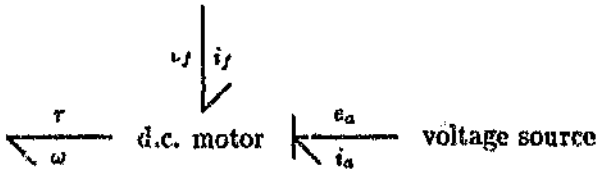


Figure 2.3: A separately excited d.c. motor and voltage source showing the causality at the armature port.

It must be noted that the power half arrow and the causal stroke are completely independent.

2.4 Basic component models

In this section, the basic set of multiports that are used to model systems will be introduced. There are nine idealized elements in bond graph theory, one dissipative element, two energy storage elements, two source elements and four power conserving elements or junctions. Each element type is governed by a relation having a particular form. The acausal relation governing each element is termed the constitutive relation.

2.4.1 The 1-port resistor

The 1-port resistor is an element where a relation exists between the effort and the flow. The bond graph representation of a 1-port resistor is shown in Figure 2.4a. The power half arrow is shown directed into the resistor implying that the resistor dissipates power.

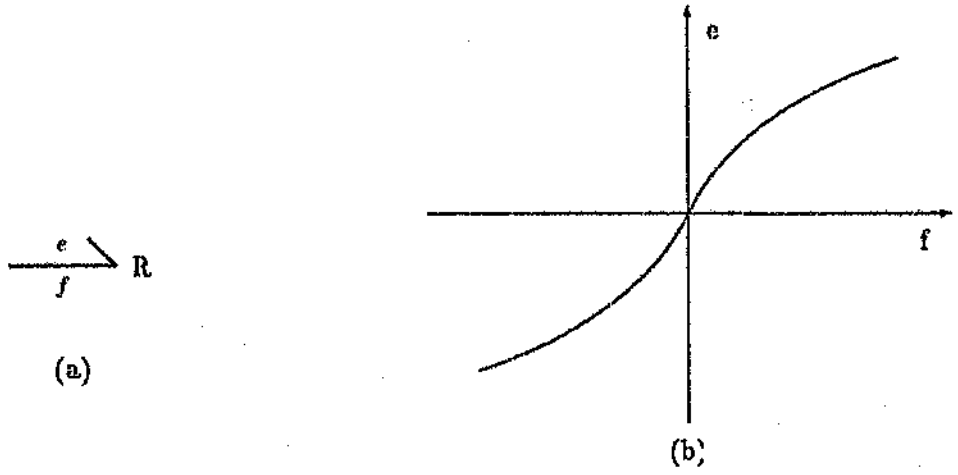


Figure 2.4: The 1-port resistor. (a) The bond graph symbol. (b) A non-linear constitutive relation.

The constitutive relation of a resistor, with the effort as the output, is

$$e = \Phi_R(f)$$

and with the flow as the output is

$$f = \Phi_R^{-1}(e) \tag{2.6}$$

The linear relation of the resistor, with the effort as the output is

$$e = Rf$$

and with the flow as the output is

$$f = \frac{1}{R}e \tag{2.7}$$

where R is the resistance and $\frac{1}{R}$ is the conductance. A non-linear constitutive relation is shown in Figure 2.4b.

Electrical resistors, mechanical dampers or dashpots and the flow through hydraulic lines are usually modelled using resistors. The models of these four systems may in addition to resistors, include other bond graph elements to account for other effects.

If a resistor supplies power, as in the case of irreversible thermodynamics (see Section 3.4), the half arrow sign convention may be changed, to show that the power is leaving the port. But if the sign convention is retained, the constitutive relation must be altered.

2.4.2 The 1-port capacitor

The 1-port capacitor is an element where a relation exists between the effort and the displacement. A capacitor stores and gives up energy without loss. Figure 2.5a is the bond graph

representation of a capacitor. The sign convention is the same as for the resistor, that is, the power flows into the port when the product of the effort and the flow is positive. Although the relation exists between the effort and the displacement, the power variables are written next to the bonds on the bond graph.

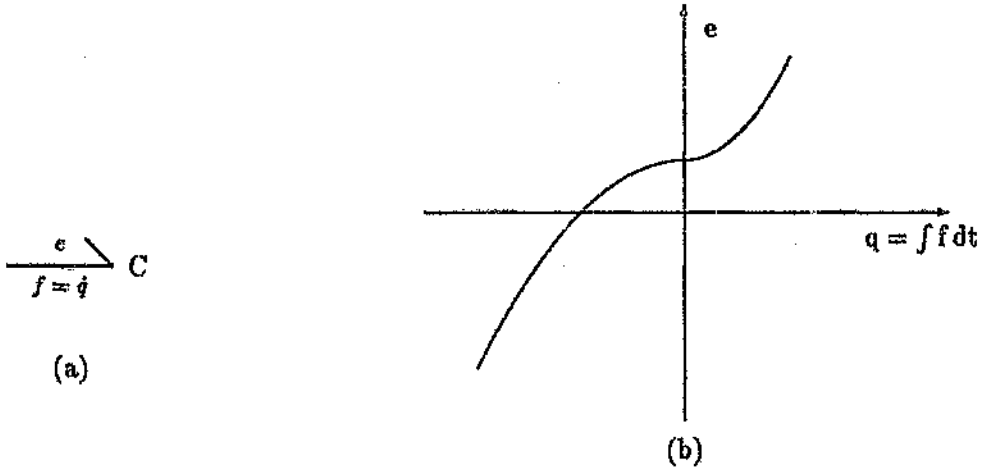


Figure 2.5: The 1-port capacitor. (a) The bond graph symbol. (b) A non-linear constitutive relation.

The constitutive relation of a capacitor, with the displacement as a function of the effort is

$$q = \Phi_C(e)$$

and with the effort as a function of the displacement is

$$e = \Phi_C^{-1}(q) \tag{2.8}$$

The linear relation, with the displacement as the output is

$$q = Ce$$

and with the effort as the output is

$$e = \frac{1}{C}q \tag{2.9}$$

where C is the capacitance. Figure 2.5b shows a non-linear constitutive relation for a capacitor.

Electrical capacitors, springs, torsion bars and tanks are modelled using capacitors.

2.4.3 The 1-port inertia

A 1-port inertia is an element where a relation exists between the flow and the momentum. As with a capacitor, the inertia also conserves energy. Figure 2.6a is the bond graph representation of an inertia. Again the sign convention is the same as for the resistor. Although the relation exists between the flow and the momentum, the power variables are written next to the bonds in the bond graph.

The constitutive relation for an inertia, with the momentum as the output is

$$p = \Phi_I(f)$$

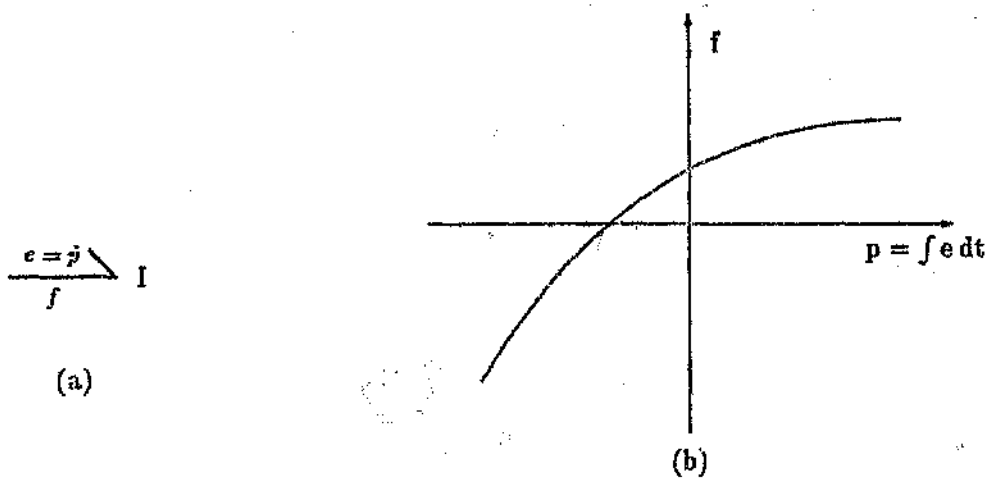


Figure 2.6: The 1-port inertia. (a) The bond graph symbol. (b) A non-linear constitutive relation.

and with the flow as the output is

$$f = \Phi_I^{-1}(p) \tag{2.10}$$

The linear relation, with the momentum as a function of the flow is

$$p = If$$

and with the flow as a function of the momentum is

$$f = \frac{1}{I}p \tag{2.11}$$

where I is the inertance. Figure 2.6b shows a non-linear constitutive relation for an inertia. The inertia is used to model inductance effects in electrical systems and inertial or mass effects in mechanical and fluid systems.

2.4.4 The 1-port effort source and 1-port flow source

The 1-port effort source and 1-port flow source, depicted in Figure 2.7 are idealized versions of voltage supplies, pressure sources, vibration shakers, constant flow systems etc. In each

Figure 2.7: (a) The effort source. (b) The flow source.

case, the effort or flow is a known function of time and is independent of the power supplied or absorbed. This implies that a bond graph source can supply an infinite amount of power, which is not true of real sources. To make a bond graph source accurately represent a real source other bond graph elements must be added it. Source elements are thought of as supplying power to the system, therefore the half arrow is normally directed out of the port, as shown.

2.4.5 The 2-port transformer

The 2-port transformer is a power conserving junction, that is, the power entering the one port is equal to the power leaving the other. Consider the bond graph representation of a transformer in Figure 2.8. If e_1 and f_1 are the power variables at port 1 and e_2 and f_2 are

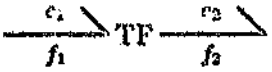


Figure 2.8: The 2-port transformer.

the power variables at port 2 then

$$e_1(t)f_1(t) = e_2(t)f_2(t) \tag{2.12}$$

The power half arrow convention shown in Figure 2.8 is such that the power enters the one port and leaves the other. This is termed a "through power" sign convention.

One possible set of relations which fulfil the power constraint is

$$e_1 = me_2$$

$$mf_1 = f_2 \tag{2.13}$$

These are the constitutive relations of a 2-port transformer, where the parameter m is known as the *transformer modulus*. Typical examples of systems within a single domain, which are modelled using bond graph transformers are electrical transformers, gears and levers. Transformers are also used to represent elements coupling two different energy domains, examples include a rack and pinion and a hydraulic ram. None of the systems mentioned above are ideal transformers, as this would imply that they are 100% efficient. The bond graph transformer is used however, amongst other bond graph elements, to represent them.

2.4.6 The 2 port gyrator

The 2-port gyrator shown in Figure 2.9, like the 2-port transformer is power conserving.

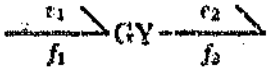


Figure 2.9: The 2-port gyrator.

The other possible set of relations which fulfil the power constraint is

$$e_1 = rf_2$$

$$rf_1 = e_2 \tag{2.14}$$

These are the constitutive relations for the 2-port gyrator. The parameter r is the *gyrator modulus*.

Gyrators are not often found within one energy domain, but are used frequently when linking different physical domains. An example of a system represented by a bond graph gyrator is a d.c. motor, which has a relation between the torque and the current.

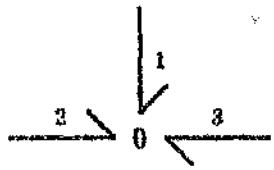


Figure 2.10: The 3-port 0-junction.

2.4.7 The n-port 0-junction

Like the transformer and gyrator, the n-port 0-junction is power conserving. The 0-junction is also termed a flow junction or common effort junction and is equivalent to a parallel connection in an electrical circuit.

The 3-port 0-junction shown in Figure 2.10, has the constitutive relations

$$\begin{aligned} e_1(t) &= e_2(t) = e_3(t) \\ f_1(t) + f_2(t) + f_3(t) &= 0 \end{aligned} \tag{2.15}$$

That is, the efforts on all the ports are equal and the flows sum to zero. The flow variables are summed according to the direction of the half arrows. For example, had the half arrow on bond 2 been directed out of the 0-junction, the constitutive relations would be

$$\begin{aligned} e_1(t) &= e_2(t) = e_3(t) \\ f_1(t) - f_2(t) + f_3(t) &= 0 \end{aligned} \tag{2.16}$$

It is clear that the 3-port 0-junction is easily generalized to an n-port 0-junction.

In [30], the 0-junction is represented by a "p", as opposed to a "0" to emphasize that the 0-junction is equivalent to a parallel junction in an electrical circuit. In this dissertation only a "0" will be used.

2.4.8 The n-port 1-junction

The n-port 1-junction, sometimes called an effort junction or a common flow junction is equivalent to a series connection in an electrical circuit.

Figure 2.11: The 3-port 1-junction.

The 3-port 1-junction, shown in Figure 2.11, has constitutive relations

$$\begin{aligned} f_1(t) &= f_2(t) = f_3(t) \\ e_1(t) + e_2(t) + e_3(t) &= 0 \end{aligned} \tag{2.17}$$

That is, the flows on all the ports are equal and the efforts sum to zero. The effort variables are summed according to the direction of the half arrows. For example, had the half arrow

on bond 3 been directed out of the port, the constitutive relation would be

$$\begin{aligned} f_1(t) &= f_2(t) = f_3(t) \\ e_1(t) + e_2(t) - e_3(t) &= 0 \end{aligned} \tag{2.18}$$

As before, it is clear that the 3-port 1-junction can easily be generalized to form an n-port 1-junction.

In the same way that the 0-junction is occasionally represented by a "p", the 1-junction is sometimes represented by an "s", to represent a series connection [30]. Again this notation will not be adopted here.

2.5 Modulating lines

As discussed previously, a half arrow is used to represent power flow. There are times however, when a pure signal flow must be depicted. Signal flow is represented by a line with a full arrow head, as shown in Figure 2.12. The arrow points in the direction of the signal flow.

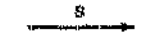


Figure 2.12: A modulating line representing a pure signal flow.

Such a line is termed a "modulating line". In reality, a pure signal flow does not exist, as there is always a transfer of power involved. A pure signal flow can be assumed however, if the power transferred is negligible in comparison to the power in the system itself. An example of this is a measurement instrument, where if there was a significant transfer of power between the instrument and the system, the system behaviour would be affected.

All bond graph elements can be modulated. An example of a modulated resistance is a 4-way hydraulic control valve where the flow rate is modulated by the displacement of the valve rod. Although power is required to displace the valve, it is negligible in comparison to the hydraulic power transferred through the valve. Examples of modulated sources are a dependent current source, a dependent voltage source, a transistor and fluid flowing from one tank into another with no return path.

There are times when a modulating line does not carry any physical signal flow. An example of this is the modulated transformer used to model large angle rotation in mechanical systems [14]. In this case, the transformer is modulated by $l\cos\theta$, where l is the length of the arm and θ, the angle through which it revolves.

2.6 Causality of the basic bond graph elements

The causality assigned to a bond graph element dictates the form of the constitutive relation, that is, it states which variable is the independent variable and which variables are the dependent variables. The constitutive relation in the correct causal form is often termed the causal relation. The convention used for the causal relation is: The output is on the left of the equality sign and the function containing the dependent variables is on the right hand side. For example, the causal relation of a linear resistor, is $e = Rf$ if the effort is the output and $f = \frac{1}{R}e$ if the flow is the output.

The causal forms of the basic multiports are shown in Table 2.3.

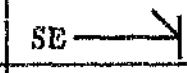
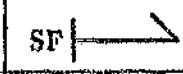
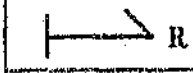
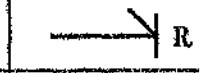
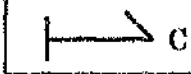
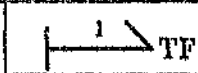
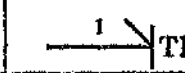
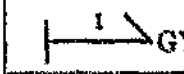
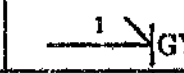
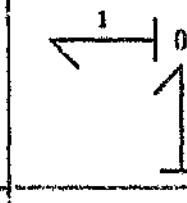
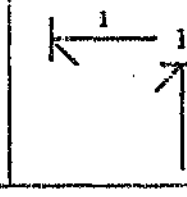
Element	Causal Form	Causal relation
Effort source		$e = E$
Flow source		$f = F$
Resistor		$e = \Phi_R(f)$
		$f = \Phi_R^{-1}(e)$
Capacitor		$e = \Phi_C^{-1}(\int f dt)$
		$f = \frac{d}{dt}(\Phi_C(e))$
Inertia		$f = \Phi_I^{-1}(\int e dt)$
		$e = \frac{d}{dt}(\Phi_I(f))$
Transformer		$e_1 = me_2$ $f_2 = mf_1$
		$f_1 = \frac{1}{m} f_2$ $e_2 = \frac{1}{m} e_1$
Gyrator		$e_1 = rf_2$ $e_2 = rf_1$
		$f_1 = \frac{1}{r} e_2$ $f_2 = \frac{1}{r} e_1$
0-junction		$e_2 = e_1$ $e_3 = e_1$ $f_1 = f_2 - f_3$
1-junction		$f_2 = f_1$ $f_3 = f_1$ $e_1 = e_2 + e_3$

Table 2.3: The causal forms of the basic multiports.

The effort source by definition, impresses an effort on the rest of the system to which it is connected and can therefore only take on the causal form shown in Table 2.3.

Like the effort source, the flow source has only one allowed causal form. The flow source, by definition, forces a known flow on the the system. The only allowed causality is that shown in Table 2.3.

In contrast to the two sources, the resistor is indifferent to the causality imposed upon it. Although either two of the causalities, shown in Table 2.3, are allowed, it does occasionally occur that one of the causal forms is not physically realizable. This happens when the resistor has a non-linear constitutive relation which cannot be inverted (for example, when $y = f(x)$ is one-to-one, but $x = f^{-1}(y)$ is one-to-many). Coulomb friction is an example of an effect represented by a bond graph resistor with only one valid causal form. This also occurs when dealing with some modulated resistors. For example, the flow rate is proportional to the valve offset (the amount it is open) and proportional to the square root of the pressure difference in a 4-way hydraulic control valve. The pressure can therefore not be found as a function of the flow rate if the valve offset is zero.

If a causality that is not physically realizable is assigned to a resistor, it is necessary to change the bond graph structure (normally by including storage or source elements) so that the resistor takes on the only possible causal form. It must be noted that to the author's knowledge, the bond graph is the only modelling technique where the causality is known prior to obtaining the system equations.

The capacitor can take on either of the two causal forms shown in Table 2.3. The first causal form, where the effort is a function of the integral of the flow (the displacement) is referred to as integral causality. The second causal form, where the flow is a function of the time derivative of the capacitor function ($\Phi_C(e)$) is known as derivative causality.

The inertia, like the capacitor can take on either of the two causal forms shown in Table 2.3. If the flow is a function of the integral of the effort (the momentum), the inertia is said to have integral causality. But if the effort is a function of the derivative of the inertia function ($\Phi_I(f)$) the inertia is said to be assigned derivative causality.

The choice of the causality assigned to the storage elements (the inertia and capacitor) has an important effect, because the energy variables of the storage elements (the momentum and displacement) are the state variables. If a storage element is assigned integral causality, the energy variable is an independent state variable. The derivative of an independent state variable is a function of itself, the other independent state variables and the inputs to the system. An independent state variable is included in the final state-space model of the system. If derivative causality is assigned to a storage element, the energy variable is a dependent state variable. A dependent state variable is a function of other independent state variables and source elements. The dependent state variable is not included in the final state-space model of the system.

As depicted in Table 2.3, the 2-port transformer can take on one of two causal forms. There are only two possible causal forms, because if any one of the four variables is constrained to be an input or an output, the remaining three will automatically be constrained. For example, as in the first case in Table 2.3, if effort 2 is an input to the transformer, flow 2 must be an output from it. Also from the constitutive relations, flow 2 is related to flow 1 by the transformer modulus. Therefore flow 1 must be an input to the transformer and effort 1 an output from it. The other causal form is obtained in the same manner except that effort 2 is constrained to be an output from the transformer.

The 2-port gyrator is similar to the transformer in that, for the same reasons, there are only two possible causal forms. Either both efforts are the outputs from the gyrator (and both flows are inputs) or the two flows are the outputs from the gyrator (and the two efforts are the inputs).

There are as many different causal forms for the 0-junction as there are bonds connected to it, but certain requirements have to be met. In a 0-junction, all the efforts are equal and the flows sum to zero, therefore if *one of* the efforts is known the others can be determined. Conversely, if all but one of the flows is given, the remaining one can be calculated. Therefore, an n-port 0-junction can only have one effort input ((n − 1) effort outputs), which implies having (n − 1) flow inputs (one flow output).

The causal requirements of a 1-junction are obtained in a similar manner to that of a 0-junction. In a 1-junction, all the flows are equal and the efforts sum to zero, therefore if one of the flows is given the others can be determined. Conversely, if all but one of the efforts are known, the remaining one can be calculated. Therefore an n-port 1-junction can only have one flow input ((n − 1) flow outputs), which implies having (n − 1) effort inputs (one effort output).

2.7 Multiport fields

Previously only 1-port resistors, capacitors and inertias were discussed. In this section, *fields*, which are multiport generalizations of these 1-port elements are considered.

2.7.1 Capacitor fields

As discussed previously, a capacitor stores and gives back energy without loss. Any single-valued functional relationship between effort and displacement therefore defines an energy conserving capacitor. Likewise, the multiport generalization of a 1-port capacitor, termed a C-field, is energy conserving.

A C-field is denoted by the letter C, with as many bonds as the field has ports, as shown in Figure 2.13.

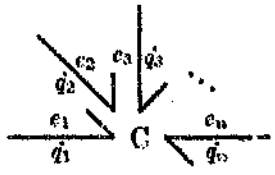


Figure 2.13: An n-port C-field.

C-fields exhibit integral and derivative causality in the same way the 1-port capacitors do, but C-fields may also take on mixed integral-derivative causalities.

The integral form of an n-port C-field has all the efforts as outputs and has constitutive relations

$$\begin{aligned} e_i &= \Phi_{C_i}^{-1}(q_1, q_2, q_3, \ldots, q_n) \\ i &= 1, 2, 3, \ldots, n \end{aligned} \tag{2.19}$$

The derivative form of an n-port C-field has all the efforts as inputs to the C-field and has constitutive relations

$$\begin{aligned} q_i &= \Phi_{C_i}(e_1, e_2, e_3, \ldots, e_n) \\ i &= 1, 2, 3, \ldots, n \end{aligned} \tag{2.20}$$

A C-field with mixed causality has integral causality assigned to some of the bonds and derivative causality assigned to the rest. Consider an n-port C-field, with the first j ports having integral causality and the remaining ports having derivative causality. Such a C-field has constitutive laws

$$\begin{aligned} e_i &= \Phi^{-1}_{C_i}(q_1, q_2, \ldots, q_j, e_{j+1}, \ldots, e_n) \\ i &= 1, 2, 3, \ldots, j \\ q_k &= \Phi_{C_k}(q_1, q_2, \ldots, q_j, e_{j+1}, \ldots, e_n) \\ k &= j+1, \ldots, n \end{aligned} \tag{2.21}$$

An example of a 2-port C-field is a moving plate capacitor, where the voltage and the force on the plates are both functions of the charge on the capacitor and the position of the plates. The bond graph representation of the moving plate capacitor is shown in Figure 2.14.

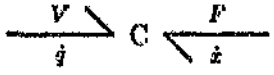


Figure 2.14: Bond graph representation of a moving plate capacitor.

If an element or system can only be described as a field (as is the case for a moving plate capacitor), it is termed an explicit field. However if a field is formed from a group of 1-port (in this case C) elements and junction structure elements (0, 1, TF, GY), then it is termed an implicit field.

Consider the bond graph shown in Figure 2.15a, which consists only of 1-port capacitors and

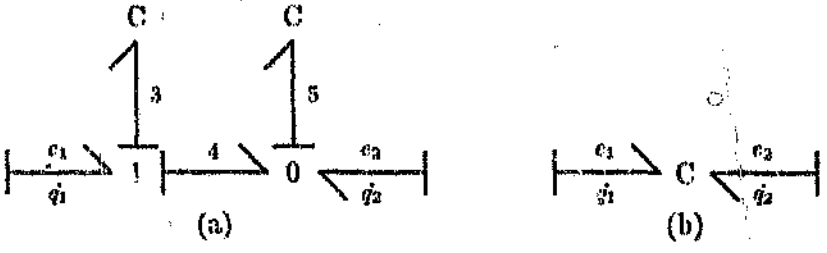


Figure 2.15: A C-field constructed from 1-port capacitors.

0 and 1-junctions. Figure 2.15b is a 2-port implicit C-field representation of the bond graph in Figure 2.15a. Assuming that the 1-port capacitors have linear relations, the constitutive laws of the C-field are

$$e_1 = \left(\frac{1}{C_3} + \frac{1}{C_5}\right) q_1 + \frac{1}{C_5} q_2$$

$$e_2 = \frac{1}{C_5}(q_1 + q_2)$$

or in matrix form

$$\begin{bmatrix} e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{C_3} + \frac{1}{C_5} & \frac{1}{C_5} \\ \frac{1}{C_5} & \frac{1}{C_5} \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} \tag{2.22}$$

The matrix in Equation 2.22 is called the *stiffness matrix* because the effort (force) is a function of the displacement. The stiffness matrix is symmetrical, that is, it is equal to its transpose.

In general the *compliance* and *stiffness matrices*, which are inverses of each other, are symmetrical. This can be proven in a form valid for even nonlinear C-fields [14]. This relation, sometimes termed the *Maxwell symmetry relation*, is

$$\frac{\partial e_i}{\partial q_j} = \frac{\partial^2 \mathbf{E}}{\partial q_j q_i} = \frac{\partial e_j}{\partial q_i} \tag{2.23}$$

$$i, j = 1, 2, 3, \ldots, n$$

This relation makes it possible to ensure that the equations formed for a C-field give rise to an energy conserving field. Also, if the energy as a function of the displacements is known, the constitutive laws can be obtained.

It may not always be useful to form an implicit C-field from a system containing 1-port capacitors, but consider the bond graph in Figure 2.16a, consisting of, amongst other elements, three 1-port capacitors. It is not possible for all three capacitors to have integral causality

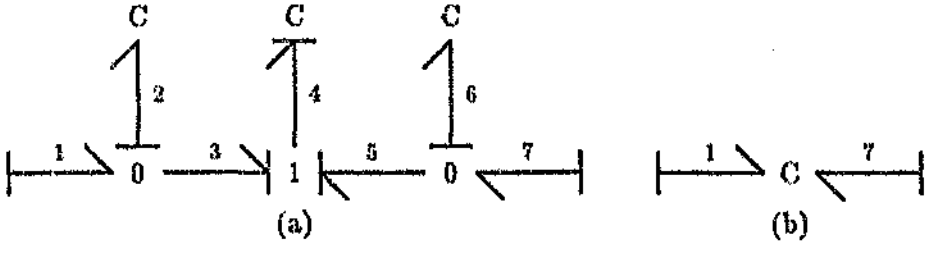


Figure 2.16: Two independent and one dependent 1-port capacitors and the implicit C-field representation.

assigned to them, that is, all the displacements are not independent of one another. In this case, q_1 is related to q_2 and q_6. Notice however that bonds 1 and 7 have efforts as outputs. If an implicit C-field is formed, with bonds 1 and 7 as the two ports, both bonds will have integral causality, as shown in Figure 2.16b. This eliminates the need to perform the algebraic manipulations associated with the derivative causality. However, other manipulation is required to get the equations into a form representative of the field.

2.7.2 Inertia fields

In the same way that the 1-port inertia is analogous to the 1-port capacitor, an I-field is analogous to a C-field. An I-field is therefore also energy conserving. All the results for C-fields hold for I-fields if the flows are substituted for the efforts and the momenta are substituted for the displacements.

An I-field is denoted by the letter I with as many bonds as the field has ports, as depicted in Figure 2.17.

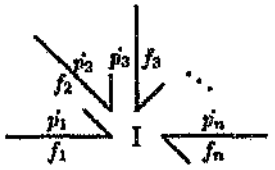


Figure 2.17: An n-port I-field.

I-fields exhibit the same three causal patterns as C-fields, all integral, all derivative and mixed integral-derivative causality.

The integral form of an n-port I-field, which has all the flows as outputs, has constitutive relations

$$\begin{aligned} f_i &= \Phi_{I_i}^{-1}(p_1, p_2, p_3, \ldots, p_n) \\ i &= 1, 2, 3, \ldots, n \end{aligned} \tag{2.24}$$

The derivative form of an n-port I-field, which has all the flows as inputs, has constitutive relations

$$\begin{aligned} p_i &= \Phi_{I_i}(f_1, f_2, f_3, \ldots, f_n) \\ i &= 1, 2, 3, \ldots, n \end{aligned} \tag{2.25}$$

An I-field with mixed causality, has integral causality assigned to some of the bonds and derivative causality assigned to the rest. Consider an n-port I-field with the first j ports having integral causality and the remainder derivative causality. Such an I-field has constitutive relations

$$\begin{aligned} f_i &= \Phi_{I_i}^{-1}(p_1, p_2, \ldots, p_j, f_{j+1}, \ldots, f_n) \\ i &= 1, 2, 3, \ldots, j \\ p_k &= \Phi_{I_k}(p_1, p_2, \ldots, p_j, f_{j+1}, \ldots, f_n) \\ k &= j+1, \ldots, n \end{aligned} \tag{2.26}$$

The reciprocity relation (*Maxwell symmetry relation*) for a non-linear I-field [14] is

$$\begin{aligned} \frac{\partial f_i}{\partial p_j} = \frac{\partial^2 \mathbf{E}}{\partial p_j p_i} = \frac{\partial f_j}{\partial p_i} \\ i, j = 1, 2, 3, \ldots, n \end{aligned} \tag{2.27}$$

Electrical systems containing mutually interacting coils can be represented by an explicit I-field. The constitutive laws for the I-field representing two coupled inductances are

$$\begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} L_1 & M_{12} \\ M_{21} & L_2 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \tag{2.28}$$

where the self-inductance coefficients are denoted by L, the mutual inductance coefficients by M and λ is the flux linkage. The mutual inductance coefficients M_{12} and M_{21} must be equal (as is known from circuit theory) for the I-field to be energy conservative. This is the same result as obtained for the "stiffness" and "compliance" matrices of C-fields.

In the previous section on C-fields, an implicit C-field was used to eliminate the algebraic manipulations associated with derivative causality. The same can be achieved for an I-field.

2.7.3 Mixed energy storing fields

It does occur that an energy storage device cannot be modelled as a pure C-field or I-field, but acts as a C-field at some ports and as an I-field at others. This field will be termed an IC-field. The use of this element within a single energy domain is not evident, but does occur between different domains. An example of such an element is a solenoid, the inertial part being due to the inductance effects and the capacitive part due to the displacement of the core.

An IC-field is shown in Figure 2.18. The ports are numbered so that the first j ports are

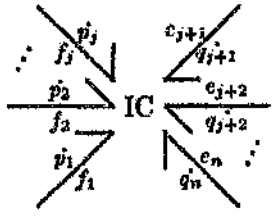


Figure 2.18: A field displaying both capacitive and inertial behavior.

inertial and the remainder capacitive in nature.

The constitutive relations of an n-port IC-field, with all integral causality, are

$$\begin{aligned} f_i &= \Phi_{IC_i}^{-1}(p_1, p_2, \ldots, p_j, q_{j+1}, \ldots, q_n) \\ i &= 1, 2, 3, \ldots, j \\ e_k &= \Phi_{IC_k}^{-1}(p_1, p_2, \ldots, p_j, q_{j+1}, \ldots, q_n) \\ k &= j+1, \ldots, n \end{aligned} \tag{2.29}$$

The reciprocity relation (*Maxwell symmetry relation*) of a non-linear IC-field [14] is

$$\frac{\partial f_i}{\partial q_k} = \frac{\partial^2 \mathbf{E}}{\partial q_k p_i} = \frac{\partial e_k}{\partial p_i} \tag{2.30}$$

$$i = 1, 2, \ldots, j$$

$$k = j+1, \ldots, n$$

It is possible to convert an IC-field into a pure C-field or a pure I-field, by changing the definitions of the efforts and flows in one of the domains, or by attaching gyrators, with a modulus of one, to either the inertial or capacitive bonds.

2.7.4 Resistor fields

An R-field, which is a multiport generalization of a 1-port resistor is shown in Figure 2.19. The constitutive laws of an n-port R-field relate the n efforts to the n flows. In practice, most R-fields are dissipative but in some cases, for example heat transfer, the R-field may act as a source (normally some of the bonds remove energy from the one domain and supply it to another). As in the case of C-fields and I-fields, both explicit and implicit R-fields exist. Explicit R-fields arise frequently when modelling non-linear effects or heat and mass transfer systems. Implicit R-fields may be formed from resistors and junction structure elements for convenience.

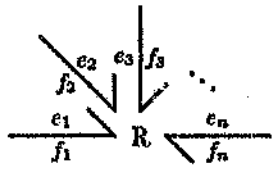


Figure 2.19: A 3-port R-field.

The causality of a R-field is determined by the source and storage elements in the system, because R-fields, like 1-port resistors, are neutral with respect to causality. Although as discussed previously, all the causal forms are not always physically realizable.

An n-port R-field with resistance causality (all the efforts are the outputs from the R-field), has constitutive relations

$$\begin{aligned} e_i &= \Phi_{R_i}(f_1, f_2, f_3, \ldots, f_n) \\ i &= 1, 2, 3, \ldots, n \end{aligned} \tag{2.31}$$

An n-port R-field with conductance causality (all the efforts are the inputs to the R-field), has constitutive relations

$$\begin{aligned} f_i &= \Phi^{-1}_{R_i}(e_1, e_2, e_3, \ldots, e_n) \\ i &= 1, 2, 3, \ldots, n \end{aligned} \tag{2.32}$$

As with the previous fields encountered, the R-field may also take on a mixed causality. Consider an n-port R-field with the first j bonds having efforts (resistance causality) as outputs and the remainder, flows (conductance causality) as outputs. The constitutive relations for such an R-field are

$$\begin{aligned} e_i &= \Phi_{R_i}(f_1, f_2, \ldots, f_j, e_{j+1}, \ldots, e_n) \\ i &= 1, 2, 3, \ldots, j \\ f_k &= \Phi^{-1}_{R_k}(f_1, f_2, \ldots, f_j, e_{j+1}, \ldots, e_n) \\ k &= j+1, \ldots, n \end{aligned} \tag{2.33}$$

It is interesting to note that since the junction structure elements (0, 1, TF, GY) provide relationships between efforts and flows, they could be modelled as R-fields. Although these R-fields would be energy conserving and would not be able to take on all possible causal forms.

2.8 Pseudo bond graphs

There are occasions when the most convenient variables used to model a system do not conform to the definitions of the bond graph variables. The most common problem is that the product of the variables chosen to be the effort and the flow do not equal the instantaneous power. When such variables are used, the bond graph is referred to as a *pseudo bond graph*.

Pseudo bond graphs are often used to model systems involving heat and mass transfer. When modelling heat flow systems, it is common to choose the effort variable to be the temperature and the flow variable to be the heat flow [14]. Because heat flow has the dimensions of power,

the system cannot be represented by a true bond graph. Compressible fluid systems are sometimes modelled with the pressure as the effort variable and the mass flow rate as the flow variable [17]. Again the product of the effort and the flow is not the instantaneous power and a true bond graph cannot be used.

The biggest disadvantage of using pseudo bond graphs, is that they cannot be directly connected to true bond graphs. Other *ad hoc* elements must be used to ensure that the power entering the one bond graph is equal to the power leaving the other. Examples of pseudo bond graphs will be shown in later chapters.

2.9 Assignment of causality

Once the bond graph model of a system has been derived and the causality assigned, the state-space equations defining the system can be methodically obtained from the bond graph. However, prior to writing down the state-space equations, qualitative information about the nature and behaviour of the system can be gained from the bond graph structure and the causality.

In the causal sense, there are three different types of elements:

1. The source and junction structure elements (SE, SF, 0, 1, TF and GY) must meet certain causal conditions, or their basic definitions are no longer valid. If the system model cannot meet these constraints, then there is either an error in the bond graph structure or in the system itself.

2. The storage elements (C and I) have a preferred, but not necessary causal orientation. If a storage element is assigned integral (preferred) causality, the energy variable will be an independent state variable. However, if a storage element is assigned derivative causality, the energy variable will not be included in the final set of system equations.

3. The resistors are neutral with respect to causality. In some cases however, one of the causal forms may not be physically realizable and it is necessary to modify the bond graph structure, so that the resistor can take on the valid causal form.

As the elements differ with respect to their causal requirements there is a procedure for assigning causality, known as the *Sequential Causality Assignment Procedure*, (SCAP) [14]. The procedure is:

1. Choose any unassigned source element (SE or SF), and assign to it, its required causality. Immediately extend the causal implications through the bond graph as far as possible, using the constraint elements (0, 1, TF and GY).

2. Repeat step 1 until all the sources have been assigned.

3. Choose any unassigned storage element (C or I), and assign its preferred (integral) causality. Immediately extend the causal implications through the graph as far as possible, using the constraint elements (0, 1, TF and GY).

4. Repeat step 3 until all storage elements have been assigned a causality. In many practical cases, all the bonds will be causally oriented after this stage. In some cases, however, certain bonds will not be assigned. The causal assignment is then completed as follows:

5. Choose any unassigned resistor and assign an arbitrary causality to it. Immediately extend the causal implications through the graph as far as possible, using the constraint elements (0, 1, TF and GY).

6. Repeat step 5 until all the resistors have been assigned.

7. Choose any remaining unassigned bond (joined to two constraint elements) and assign an arbitrary causality to it. Immediately extend the causal implications through the graph as far as possible, using the constraint elements (0, 1, TF and GY).

8. Repeat step 7 until all the remaining bonds have been assigned.

There are several situations that can arise when applying causality according to the above procedure:

1. All storage elements have integral causality and the procedure is completed after step 4.

2. The assignment procedure is completed by assigning an arbitrary causality to a resistor or bond, that is, after steps 6 or 8.

3. Some of the storage elements may have derivative causality.

These situations may occur individually or together.

2.9.1 Assignment of causality to a system without derivative or arbitrary causalities

Consider the acausal bond graph shown Figure 2.20a. The first step is to assign the required causality to the effort source. Figure 2.20b shows bond 1 assigned a causality according to the conventions of the source (the effort is the output). Since the 1-junction has only one effort input, the other bonds attached to it cannot be assigned. As there are no other sources, the procedure moves to step 3. In Figure 2.20c, inertia 2 is assigned integral causality (the flow is the output from the inertia). Immediately the causality may be assigned to bond 3 as the 1-junction can only have one flow input. This is shown in Figure 2.20d. The procedure cannot be continued as the 0-junction only has one effort output. In Figure 2.20e, capacitor 4 is assigned integral causality (the effort is the output from the capacitor). The causality on bond 5 can be immediately assigned as there may only be one effort input to a 0-junction. The fully augmented bond graph is shown in Figure 2.20f.

The causalities assigned to the bond graph in Figure 2.20f can be interpreted as follows:

1. The system is of second order as both the storage elements have integral causality assigned to them.

2. The two state variables will be momentum, p_2 and displacement, q_4.

3. The resistor has conductance causality assigned to it and it must therefore be possible to write f_5 as a function of e_5.

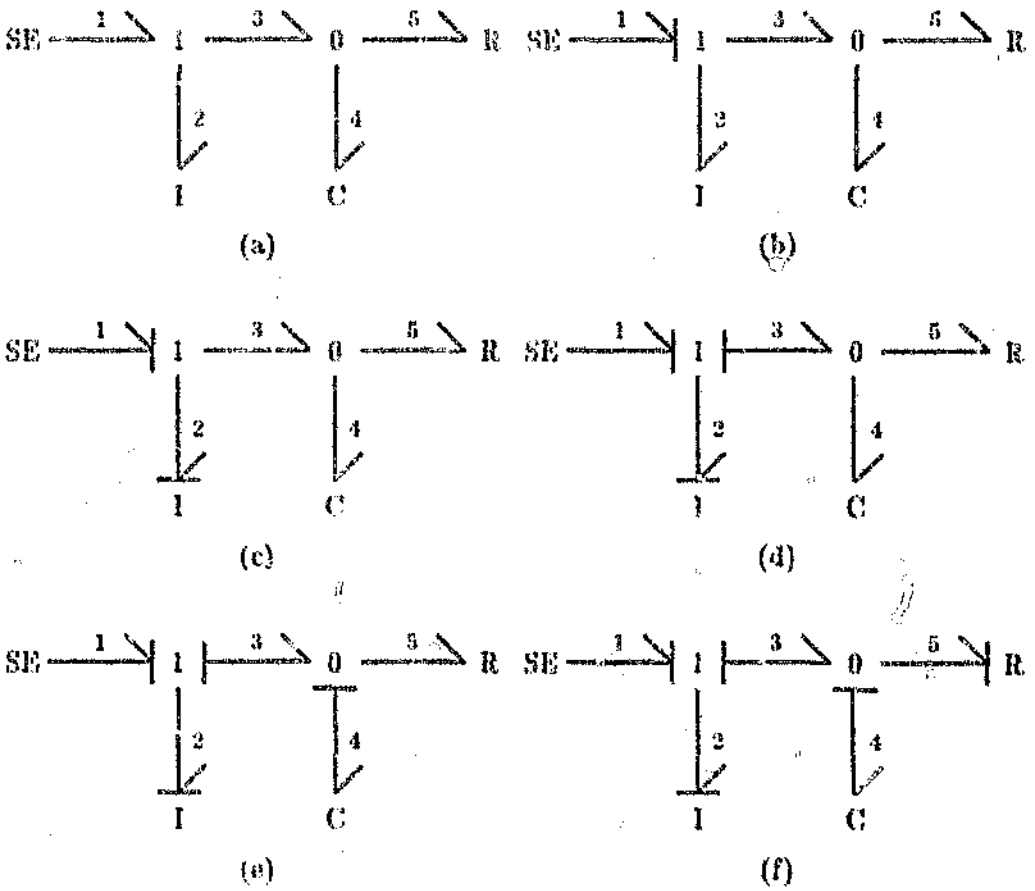


Figure 2.20: Causality assignment example.

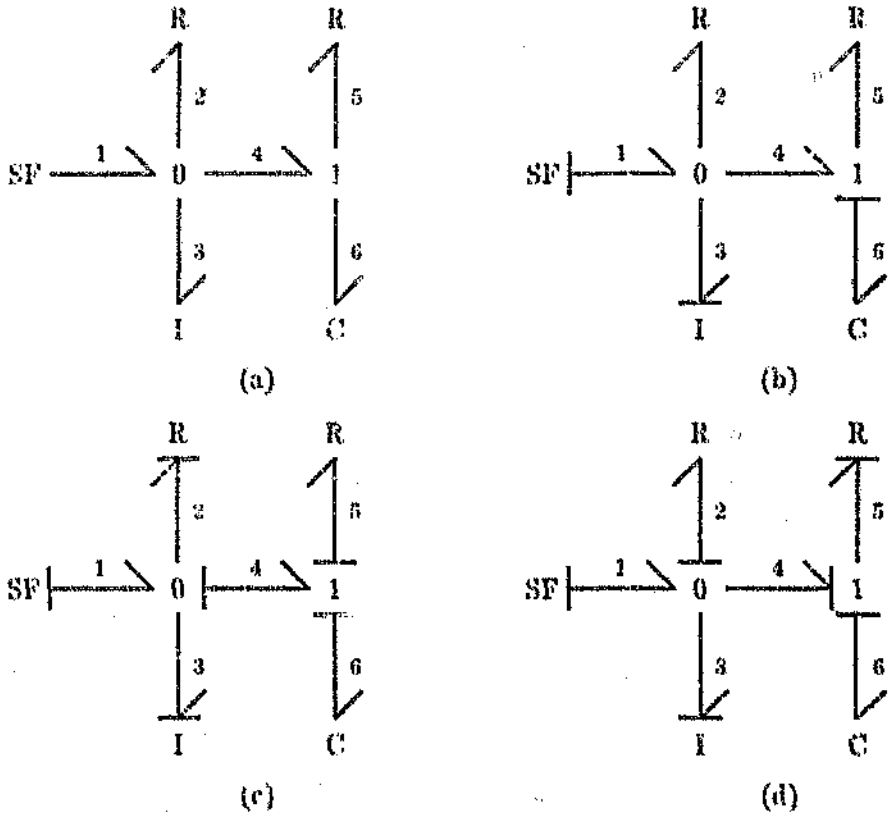


Figure 2.21: A bond graph where the causality is completed by assigning an arbitrary causality to a resistor.

2.9.2 An example of SCAP when the causality is completed by the resistors

Consider the unaugmented bond graph in Figure 2.21a. The flow source is assigned the required causality. As there is only one effort output from the 0-junction, the causality assignment cannot be extended through the bond graph. The next step is to assign the preferred causality to one of the storage elements. Integral causality is assigned to inertia 3. There are now two effort outputs from the 0-junction, but this is still insufficient to continue the assignment. Capacitor 6 is assigned integral causality. The 1-junction to which the capacitor is attached has only one effort input, which is insufficient to extend the causal implications. The assignment of these three causal strokes is shown in Figure 2.21b. In order to complete the procedure, one of the resistors must be assigned an arbitrary causality (provided that the resistor is indifferent to the causality assigned to it). Resistor 2 is assigned conductance causality. This forces resistor 5 to accept resistance causality, because the 0-junction can only have three effort outputs and the 1-junction can only have one flow input. The fully augmented bond graph is shown in Figure 2.21c. Figure 2.21d shows the fully augmented bond graph of the same system but with resistance causality assigned to resistor 2.

The causalities assigned to the bond graph in Figure 2.21c can be interpreted as follows:

1. The system is second order as both the storage elements have integral causality assigned to them.
2. The two state variables will be momentum, p_3 and displacement, q_6.
3. There is an algebraic loop because one of the resistors (resistor 2) had to have an arbitrary causality assigned to it.
4. Resistor 2 has conductance causality assigned to it and it must therefore be possible to write f_2 as a function of e_2.
5. Resistor 5 has resistance causality assigned to it and it must therefore be possible to write e_5 as a function of f_5.

2.9.3 Causality assignment to a bond graph exhibiting derivative causality

Consider the unaugmented bond graph in Figure 2.22a. The effort source is assigned the required causality, but because the 1-junction only has one effort input, the causality cannot be extended through the graph. Inertia 2 is assigned integral causality and because there is now one effort output from the 1-junction, bonds 3 and 4 can be causally oriented. The three remaining bonds however, cannot yet be assigned a causality as the 0-junction has only one effort output from it. This is depicted in Figure 2.22b. Capacitor 5 is assigned integral causality and because the 0-junction now has an effort input, bonds 6 and 7 must be assigned the causality shown in Figure 2.22c.

The causalities assigned to the bond graph in Figure 2.22c can be interpreted as follows:

1. The system is second order as two of the three storage elements have integral causality assigned to them.
2. The two state variables will be momentum, p_2 and displacement, q_5.

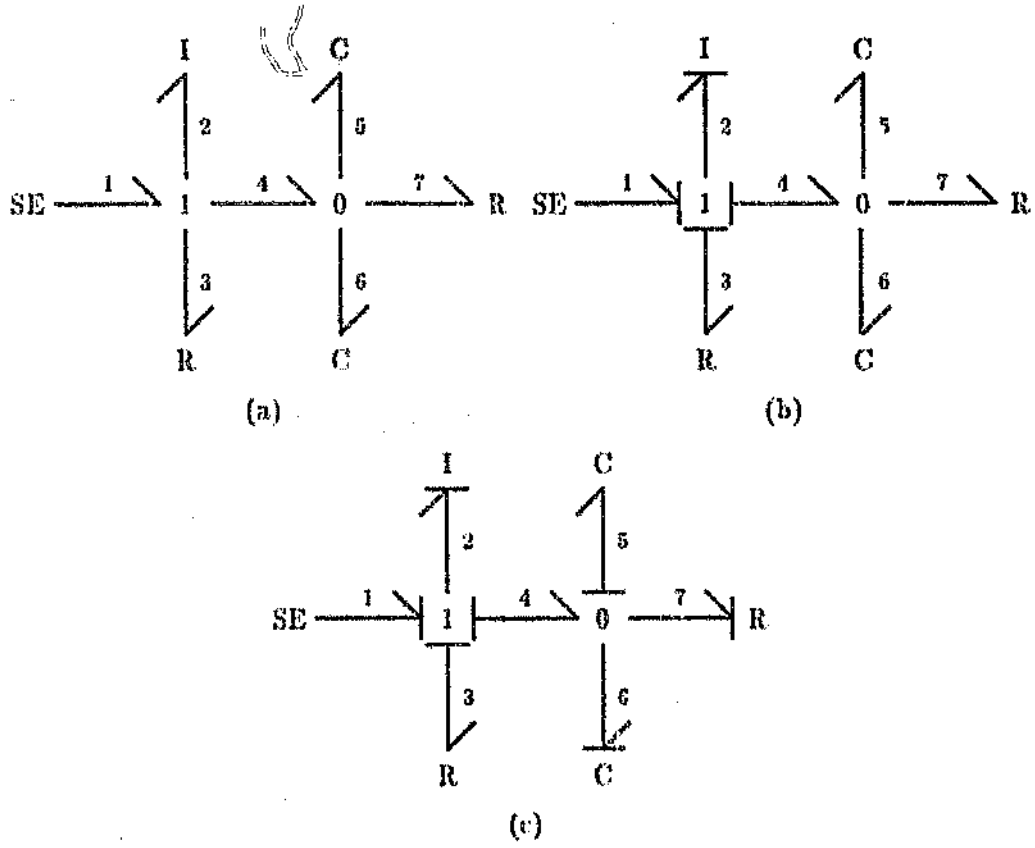


Figure 2.22: Bond graph exhibiting derivative causality.

3. Displacement q_6 will not be included in the final set of system equations because capacitor 6 has derivative causality assigned to it. Because the assignment of integral causality to capacitor 5 caused capacitor 6 to take on derivative causality, q_6 is a function of q_5.

4. There are no algebraic loops.

5. Resistor 3 has resistance causality assigned to it and it must therefore be possible to write e_3 as a function of f_3.

6. Resistor 7 has conductance causality assigned to it and it must therefore be possible to write f_7 as a function of e_7.

2.10 Equation formulation

Once a fully augmented bond graph is available, the state-space equations may be written down in an orderly fashion. The steps to follow in deriving the state space equations are:

1. Select the input and state variables.

2. Write down the causal relations for each element.

3. Reduce the equations to state-space form.

The selection of the input and state variables is straightforward. Each source element provides an input variable and the energy variable of each storage element having integral causality is a state variable.

In practice, it is not necessary to write down all of the causal relations, but it gives insight into the formulation of the equations, particularly to a person without much bond graph experience.

The reduction of the equations to state-space form is usually a simple matter, requiring only the substitution of equations. There are cases, however, when some algebraic manipulation is required. This will be shown later.

In all the examples that follow, the generalized variables; effort, flow, displacement and momentum will be used, as opposed to the standard variables used in a particular domain. This is done for clarity and to avoid an emphasis on any particular domain as a bond graph is a multidomain representation.

2.10.1 The equation formulation of a system without derivative causality and algebraic loops

Consider the fully augmented bond graph in Figure 2.23.

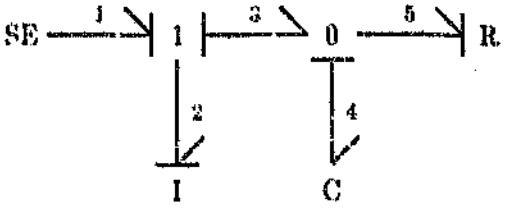


Figure 2.23: A fully augmented bond graph of a system without derivative causality or algebraic loops.

The input to the system is the effort from the effort source, e_1. In this example is is assumed that e_1 is some function E_1.

The state variables of the system are the energy variables of the storage elements with integral causality. In this case, both the inertia and the capacitor are assigned integral causality and the state variables are therefore momentum, p_2 and displacement, q_4.

The general form of a set of state-space equations is

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$$

in this example

$$\mathbf{x} = \begin{bmatrix} p_2 \\ q_4 \end{bmatrix}$$

and

$$\mathbf{u} = [E_1]$$

Step 2 involves writing down the causal relations for all the elements. For convenience, it is assumed that all the elements have linear relations.

Effort source 1 The causality assigned to the effort source implies that the effort e_1 is the output from the effort source. The causal relation is therefore

$$e_1 = E_1$$

Inertia 2 The inertia has integral causality assigned to it, therefore the flow f_2 is a function of the momentum p_2.

$$f_2 = \frac{1}{I_2} p_2$$

and by definition

$$\dot{p}_2 = e_2$$

Capacitor 4 The capacitor also has integral causality assigned to it, therefore the effort e_4 is a function of the displacement q_4.

$$e_4 = \frac{1}{C_4} q_4$$

and by definition

$$\dot{q}_4 = f_4$$

Resistor 5 The resistor has conductance causality assigned to it, therefore the flow f_5 is a function of the effort e_5.

$$f_5 = \frac{1}{R_5} e_5$$

The 1-junction The 1-junction has bonds 1 and 3 as effort inputs (flow outputs) and bond 2 as the effort output (flow input). Therefore, noting the sign convention

$$e_2 = e_1 - e_3$$

$$f_1 = f_2$$

$$f_3 = f_2$$

The 0-junction The 0-junction has bond 4 as the effort input (flow output) and bonds 3 and 5 as the effort outputs (flow inputs). Therefore, noting the sign convention

$$f_4 = f_3 - f_5$$

$$e_3 = e_4$$

$$e_5 = e_4$$

Because the two storage elements have both been assigned integral causality, there are two equations of state (each storage element with integral causality contributes one state variable). To find the first equation one begins with an equation involving the derivative of one of the state variables. Considering the inertia first and using the definition of momentum

$$\dot{p}_2 = e_2 \tag{2.34}$$

but from the causal relations

$$e_2 = e_1 - e_3$$

therefore

$$\dot{p}_2 = e_1 - e_3 \tag{2.35}$$

but from the causal relations

$$e_1 = E_1$$

and

$$e_3 = e_4$$

therefore

$$\dot{p}_2 = E_1 - e_4 \tag{2.36}$$

The term E_1 is an input so it does not have to be pursued any longer, however e_4 does. From the causal relations

$$e_4 = \frac{1}{C_4} q_4$$

therefore

$$\dot{p}_2 = E_1 - \frac{1}{C_4} q_4 \tag{2.37}$$

The derivative of one of the state variables is now in terms of the input and another state variable. This is therefore one of the state-space equations.

The second equation of state is obtained by starting with the capacitor (the first equation was obtained by starting with the inertia). One begins with the definition of the displacement.

$$\dot{q}_4 = f_4 \tag{2.38}$$

but

$$f_4 = f_3 - f_5$$

therefore

$$\dot{q}_4 = f_3 - f_5 \tag{2.39}$$

but

$$f_3 = f_2$$

and

$$f_5 = \frac{1}{R_5} e_5$$

therefore

$$\dot{q}_4 = f_3 - \frac{1}{R_5} e_5 \tag{2.40}$$

but

$$f_2 = \frac{1}{I_2} p_2$$

and

$$e_5 = e_4$$

therefore

$$\dot{q}_4 = \frac{1}{I_2} p_2 - \frac{1}{R_5} e_4 \tag{2.41}$$

As p_2 is a state variable, it need not be pursued any longer, but e_4 is neither an input nor a state variable, so the term must be continued with

$$e_4 = \frac{1}{C_4} q_4$$

therefore

$$\dot{q}_4 = \frac{1}{I_2} p_2 - \frac{1}{R_5 C_4} q_4 \tag{2.42}$$

The derivative of the displacement q_4 is in terms of the two state variables and is therefore the other state-space equation

The state-space equations of the system are

$$\begin{bmatrix} \dot{p}_2 \\ \dot{q}_4 \end{bmatrix} = \begin{bmatrix} 0 & -\frac{1}{C_4} \\ \frac{1}{I_2} & -\frac{1}{R_5 C_4} \end{bmatrix} \begin{bmatrix} p_2 \\ q_4 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} E_1 \end{bmatrix} \tag{2.43}$$

Therefore

$$A = \begin{bmatrix} 0 & -\frac{1}{C_4} \\ \frac{1}{I_2} & -\frac{1}{R_5 C_4} \end{bmatrix}$$

and

$$B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

2.10.2 Equation formulation when the causality is completed by a resistor

The equations of state, for the bond graph in Figure 2.24, are derived as follows:

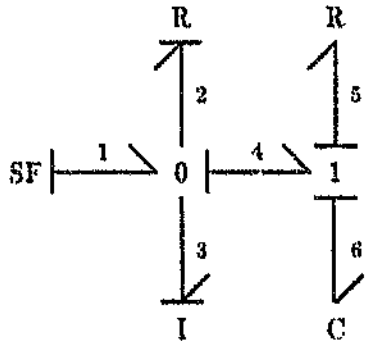


Figure 2.24: A bond graph where the causality is completed by assigning an arbitrary causality to a resistor.

The input to the system is the flow from the flow source, f_1. It is assumed that f_1 is some function F_1.

The state variables of the system are the energy variables of the storage elements with integral causality. In this case, both the capacitor and the inertia have integral causality and therefore the state variables are the momentum, p_3 and the displacement, q_6.

The general form of the state-space equations is

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$$

In this example

$$\mathbf{x} = \begin{bmatrix} p_3 \\ q_6 \end{bmatrix}$$

and

$$\mathbf{u} = [F_1]$$

Step 2 involves writing down the causal relations for all the elements. For convenience, it is assumed that all the elements have linear relations.

Flow source 1 The output from the flow source is by definition the flow f_1

$$f_1 = F_1$$

Inertia 3 The inertia has integral causality assigned to it, therefore the flow f_3 is a function of the momentum p_3.

$$f_3 = \frac{1}{I_3} p_3$$

and by definition

$$\dot{p}_3 = e_3$$

Capacitor 6 The capacitor also has integral causality assigned to it, therefore the effort e_6 is a function of the displacement q_6.

$$e_6 = \frac{1}{C_6} q_6$$

and by definition

$$\dot{q}_6 = f_6$$

Resistor 2 Resistor 2 has conductive causality assigned to it, therefore the flow f_2 is a function of the effort e_2.

$$f_2 = \frac{1}{R_2} e_2$$

Resistor 5 Resistor 5 has resistance causality assigned to it, therefore the effort e_5 is a function of the flow f_5.

$$e_5 = R_5 f_5$$

The 0-junction The 0-junction has bonds 1, 2 and 3 as effort outputs (flow inputs) and bond 4 as the effort input (flow output). Therefore, noting the sign convention

$$f_4 = f_1 - f_2 - f_3$$

$$e_1 = e_4$$

$$e_2 = e_4$$

$$e_3 = e_4$$

The 1-junction The 1-junction has bonds 5 and 6 as effort inputs (flow outputs) and bond 4 as the effort output (flow input). Therefore, noting the sign conventions

$$e_4 = e_5 + e_6$$

$$f_5 = f_4$$

$$f_6 = f_4$$

The first state-space equation is obtained by starting with the definition of the momentum of the inertia.

$$\dot{p}_3 = e_3 \tag{2.44}$$

but from the causal relations

$$e_3 = e_4$$

therefore

$$\dot{p}_3 = e_4 \tag{2.45}$$

but from the causal relations

$$e_4 = e_5 + e_6$$

therefore

$$\dot{p}_3 = e_5 + e_6 \tag{2.46}$$

but from the causal relations

$$e_5 = R_5 f_5$$

and

$$e_6 = \frac{1}{C_6} q_6$$

therefore

$$\dot{p}_3 = R_5 f_5 + \frac{1}{C_6} q_6 \tag{2.47}$$

the term q_6 does not have to be pursued any further as it is a state variable, but

$$f_5 = f_4$$

therefore

$$\dot{p}_3 = R_5 f_4 + \frac{1}{C_6} q_6 \tag{2.48}$$

but

$$f_4 = f_1 - f_2 - f_3$$

therefore

$$\dot{p}_3 = R_5(f_1 - f_2 - f_3) + \frac{1}{C_6} q_6 \tag{2.49}$$

but

$$f_1 = F_1$$

$$f_2 = \frac{1}{R_2} e_2$$

and

$$f_3 = \frac{1}{I_3} p_3$$

therefore

$$\dot{p}_3 = R_5(F_1 - \frac{1}{R_2} e_2 - \frac{1}{I_3} p_3) + \frac{1}{C_6} q_6 \tag{2.50}$$

From Equation 2.50, it can be seen that e_2 is the only term that is neither a state nor an input variable and therefore has to be pursued further, but

$$e_2 = e_4$$

therefore

$$\dot{p}_3 = R_5(F_1 - \frac{1}{R_2} e_4 - \frac{1}{I_3} p_3) + \frac{1}{C_6} q_6 \tag{2.51}$$

In Equation 2.51, e_4 is also not a state or input variable and therefore an expression for e_4 has to be found. But considering Equations 2.45 and 2.46, an expression for e_4 has already been substituted. If $e_4 = e_5 + e_6$ is su' stituted into Equation 2.51, Equation 2.51, will get larger and larger, with e_4 having to be substituted repeatedly.

To get out of this algebraic loop, it is noted that $\dot{p}_3 = e_4$ (Equation 2.45). Reversing the order of the relation to get $e_4 = \dot{p}_3$ and substituting into Equation 2.51 gives

$$\dot{p}_3 = R_5(F_1 - \frac{1}{R_2}\dot{p}_3 - \frac{1}{I_3}p_3) + \frac{1}{C_6}q_6 \tag{2.52}$$

and with some manipulation

$$\dot{p}_3 = \frac{R_2R_5}{R_2 + R_5}F_1 - \frac{R_2R_5}{(R_2 + R_5)I_3}p_3 + \frac{R_2}{(R_2 + R_5)C_6}q_6 \tag{2.53}$$

which is only in terms of the input and state variables.

To obtain the other state-space equation, the definition of the displacement of the capacitor is used

$$\dot{q}_6 = f_6 \tag{2.54}$$

but

$$f_6 = f_4$$

therefore

$$\dot{q}_6 = f_4 \tag{2.55}$$

but

$$f_4 = f_1 - f_2 - f_3$$

therefore

$$\dot{q}_6 = f_1 - f_2 - f_3 \tag{2.56}$$

but

$$f_1 = F_1$$

$$f_2 = \frac{1}{R_2}e_2$$

and

$$f_3 = \frac{1}{I_3}p_3$$

therefore

$$\dot{q}_6 = F_1 - \frac{1}{R_2}e_2 - \frac{1}{I_3}p_3 \tag{2.57}$$

Only e_2 needs to be continued with as p_3 is a state variable and F_1 is an input

$$e_2 = e_4$$

therefore

$$\dot{q}_6 = F_1 - \frac{1}{R_2}e_4 - \frac{1}{I_3}p_3 \tag{2.58}$$

As before, this is in terms of e_4 and again using $e_4 = \dot{p}_3$

$$\dot{q}_6 = F_1 - \frac{1}{R_2}\dot{p}_3 - \frac{1}{I_3}p_3 \tag{2.59}$$

But from Equation 2.53

$$\dot{p}_3 = \frac{R_2R_5}{R_2+R_5}F_1 - \frac{R_2R_5}{(R_2+R_5)I_3}p_3 + \frac{R_2}{(R_2+R_5)C_6}q_6$$

therefore

$$\dot{q}_6 = F_1 - \frac{1}{I_2}\left[\frac{R_2R_5}{R_2+R_5}F_1 - \frac{R_2R_5}{(R_2+R_5)I_3}p_3 + \frac{R_2}{(R_2+R_5)C_6}q_6\right] - \frac{1}{I_3}p_3 \qquad (2.60)$$

and with a little manipulation

$$\dot{q}_6 = \frac{R_2}{R_2+R_5}F_1 - \frac{R_2}{(R_2+R_5)I_3}p_3 - \frac{1}{(R_2+R_5)C_6}q_6 \qquad (2.61)$$

This is in terms of the input and state variables and is therefore an equation of state.

The state-space equations of the system are

$$\begin{bmatrix} \dot{p}_3 \\ \dot{q}_6 \end{bmatrix} = \begin{bmatrix} -\frac{R_2R_5}{(R_2+R_5)I_3} & \frac{R_2}{(R_2+R_5)C_6} \\ -\frac{R_2}{(R_2+R_5)I_3} & -\frac{1}{(R_2+R_5)C_6} \end{bmatrix} \begin{bmatrix} p_3 \\ q_6 \end{bmatrix} + \begin{bmatrix} \frac{R_2R_5}{R_2+R_5} \\ \frac{R_2}{R_2+R_5} \end{bmatrix} \begin{bmatrix} F_1 \end{bmatrix} \qquad (2.62)$$

Therefore

$$A = \begin{bmatrix} -\frac{R_2R_5}{(R_2+R_5)I_3} & \frac{R_2}{(R_2+R_5)C_6} \\ -\frac{R_2}{(R_2+R_5)I_3} & -\frac{1}{(R_2+R_5)C_6} \end{bmatrix}$$

and

$$B = \begin{bmatrix} \frac{R_2R_5}{R_2+R_5} \\ \frac{R_2}{R_2+R_5} \end{bmatrix}$$

The equations of state for the bond graph where resistor 2 is arbitrarily assigned resistance causality are the same as those derived above, but the steps involved in formulating the equations are different.

2.10.3 Equation formulation when the bond graph exhibits derivative causality

The state-space equations of the bond graph depicted in Figure 2.25 are derived as follows:

The input to the system is the effort output from the effort source e_1. It is assumed that e_1 is some function E_1.

The two storage elements with integral causality are inertia 2 and capacitor 5. Capacitor 6 has derivative causality assigned to it and the displacement, q_6 is therefore a dependent state variable. The two state variables are the momentum, p_2 and the displacement, q_5.

The general form of the state-space equation is

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$$

In this example

$$\mathbf{x} = \begin{bmatrix} p_2 \\ q_5 \end{bmatrix}$$

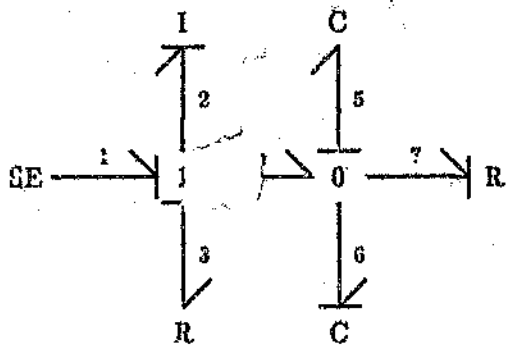


Figure 2.25: A bond graph exhibiting derivative causality.

and

$$\mathbf{u} = [E_1]$$

Step 2 involves writing down the causal relations for all the elements. For convenience, it is assumed that all the elements are described by linear relations.

Effort source 1 The effort source by definition has an effort as the output

$$e_1 = E_1$$

Inertia 2 The inertia has integral causality assigned to it, therefore the flow f_2 is a function of the momentum p_2.

$$f_2 = \frac{1}{I_2} p_2$$

and by definition

$$\dot{p}_2 = e_2$$

Capacitor 5 Capacitor 5 has integral causality assigned to it, therefore the effort e_5 is a function of the displacement q_5.

$$e_5 = \frac{1}{C_5} q_5$$

and by definition

$$\dot{q}_5 = f_5$$

Capacitor 6 Capacitor 6 has derivative causality assigned to it, therefore the displacement q_6 is a function of the effort e_6.

$$q_6 = C_6 e_6$$

and by definition

$$f_6 = \dot{q}_6$$

Resistor 3 Resistor 3 has resistance causality assigned to it, therefore the effort e_3 is a function of the flow f_3.

$$e_3 = R_3 f_3$$

Resistor 7 Resistor 7 has conductance causality assigned to it, therefore the flow f_7 is a function of the effort e_7.

$$f_7 = \frac{1}{R_7} e_7$$

The 1-junction The 1-junction has bonds 1, 3 and 4 as effort inputs (flow outputs) and bond 2 as the effort output (flow input). Therefore, noting the sign conventions

$$e_2 = e_1 - e_3 - e_4$$

$$f_1 = f_2$$

$$f_3 = f_2$$

$$f_4 = f_2$$

The 0-junction The 0-junction has bonds 4, 6 and 7 as effort outputs (flow inputs) and bond 5 as the effort input (flow output). Therefore, noting the sign convention

$$f_5 = f_4 - f_6 - f_7$$

$$e_4 = e_5$$

$$e_6 = e_5$$

$$e_7 = e_5$$

The first state-space equation is obtained as follows:

Using the inertia,

$$\dot{p}_2 = e_2 \tag{2.63}$$

but

$$e_2 = e_1 - e_3 - e_4$$

therefore

$$\dot{p}_2 = e_1 - e_3 - e_4 \tag{2.64}$$

but

$$e_1 = E_1$$

$$e_3 = R_3 f_3$$

and

$$e_4 = e_5$$

therefore

$$\dot{p}_2 = E_1 - R_3 f_3 - e_5 \tag{2.65}$$

The term E_1 does not have to be pursued any further as it is the input, but

$$f_3 = f_2$$

and

$$e_5 = \frac{1}{C_5} q_5$$

therefore

$$\dot{p}_2 = E_1 - R_3 f_2 - \frac{1}{C_5} q_5 \tag{2.66}$$

Only the term in f_2 needs to be pursued further as q_5 is a state variable.

$$f_2 = \frac{1}{I_2} p_2$$

therefore

$$\dot{p}_2 = E_1 - \frac{R_3}{I_2}p_2 - \frac{1}{C_5}q_5 \tag{2.67}$$

$\dot{p}_2$ is in terms of the state variables and inputs only and is therefore a valid equation of state. To get the other state-space equation, the definition of the displacement is used.

$$\dot{q}_5 = f_5 \tag{2.68}$$

but

$$f_5 = f_4 - f_6 - f_7$$

therefore

$$\dot{q}_5 = f_4 - f_6 - f_7 \tag{2.69}$$

but

$$f_4 = f_2$$

and by definition

$$f_6 = \dot{q}_6$$

also

$$f_7 = \frac{1}{R_7}e_7$$

therefore

$$\dot{q}_5 = f_2 - \dot{q}_6 - \frac{1}{R_7}e_7 \tag{2.70}$$

but

$$f_2 = \frac{1}{I_2}p_2$$

There is no term in $\dot{q}_6$ but

$$q_6 = C_6 e_6$$

Taking derivatives on both sides and assuming C_6 to be a constant gives

$$\dot{q}_6 = C_6\dot{e}_6$$

also

$$e_7 = e_5$$

therefore

$$\dot{q}_5 = \frac{1}{I_2}p_2 - C_6\dot{e}_6 - \frac{1}{R_7}e_5 \tag{2.71}$$

p_2 is a state variable, it therefore does not need to be pursued any further, but

$$e_6 = e_5$$

and taking derivatives on both sides gives

$$\dot{e}_6 = \dot{e}_5$$

also

$$\dot{e}_5 = \frac{1}{C_5}q_5$$

therefore

$$\dot{q}_5 = \frac{1}{I_2}p_2 - C_6\dot{e}_5 - \frac{1}{R_7 C_5}q_5 \tag{2.72}$$

but

$$e_5 = \frac{1}{C_5} q_5$$

Taking derivatives on both sides and assuming that C_5 is a constant gives

$$\dot{e}_5 = \frac{1}{C_5} \dot{q}_5$$

therefore

$$\dot{q}_5 = \frac{1}{I_2} p_2 - \frac{C_6}{C_5} \dot{q}_5 - \frac{1}{R_7 C_5} q_5 \tag{2.73}$$

and with a little manipulation

$$\dot{q}_5 = \frac{C_5}{(C_5 + C_6) I_2} p_2 - \frac{1}{(C_5 + C_6) R_7} q_5 \tag{2.74}$$

Therefore the state-space equations of the system are

$$\begin{bmatrix} \dot{p}_2 \\ \dot{q}_5 \end{bmatrix} = \begin{bmatrix} -\frac{R_3}{I_2} & -\frac{1}{C_5} \\ \frac{C_5}{(C_5+C_6)I_2} & -\frac{1}{(C_5+C_6)R_7} \end{bmatrix} \begin{bmatrix} p_2 \\ q_5 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} E_1 \end{bmatrix} \tag{2.75}$$

Therefore

$$A = \begin{bmatrix} -\frac{R_3}{I_2} & -\frac{1}{C_5} \\ \frac{C_5}{(C_5+C_6)I_2} & -\frac{1}{(C_5+C_6)R_7} \end{bmatrix}$$

and

$$B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Had capacitor 6 been assigned integral causality as opposed to capacitor 5, a similar sort of result would have been obtained.

Chapter 3

Bond graph models of systems in various domains

In this chapter, bond graph models of electrical, mechanical translation, mechanical rotation, fluid flow (hydraulic), heat transfer and compressible thermofluid systems will be presented.

The sections on fluid flow, heat transfer and heat and mass transfer will be more detailed than the others, because process systems are of primary importance to the Control Group of the Department of Electrical Engineering at the University of the Witwatersrand.

3.1 Bond graph models of electrical systems

Electrical systems are easily represented using bond graphs, because the electrical domain is the only domain where a formal system representation exists and each bond graph element has an equivalent electrical circuit element. There are cases however, where it is not straightforward to convert an electrical circuit into a bond graph. For such systems however, the bond graph construction method [14], can be used. This method will not be presented here.

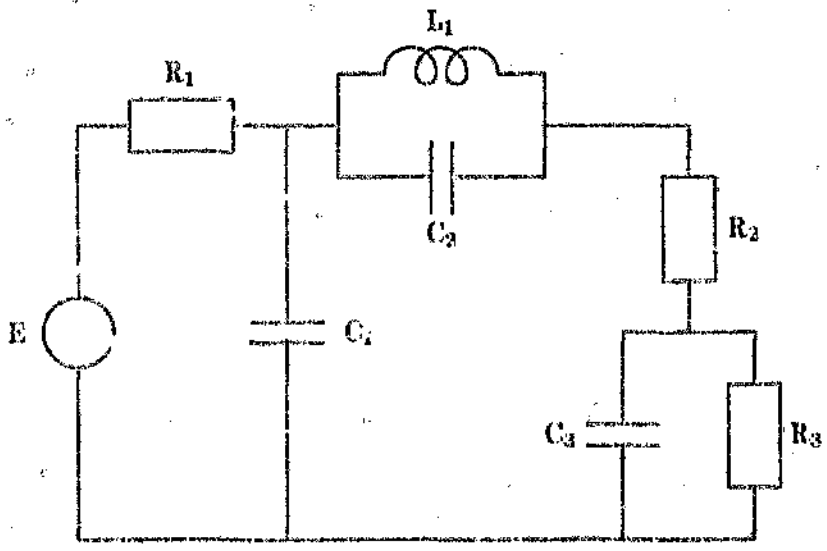
As an example of the method used to model an electrical circuit, consider the circuit in Figure 3.1a and the bond graph model in Figure 3.1b.

The voltage source SE (representing the source E) and resistor R_1 are connected to a 1-junction because they are in series. In series with the voltage source and resistor R_1, is a parallel combination of capacitor C_1 and the rest of the circuit to the right of C_1. Therefore connected to the 1-junction is a 0-junction, representing the parallel connection, to which the capacitor C_1 and the rest of the circuit to the right of C_1 are connected.

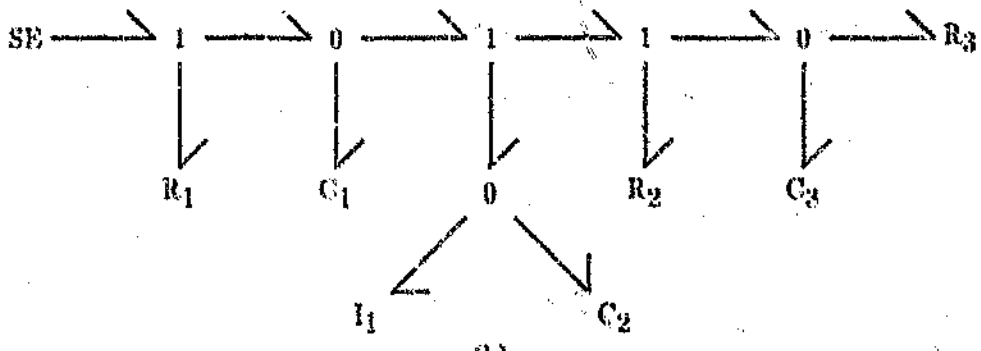
Because the inductor L_1 (represented by I_1 in the bond graph) and capacitor C_2 are in parallel, they are connected to a 0-junction. But because the parallel connection is in series with the remaining circuit, the 0-junction is connected to the rest of the bond graph with a 1-junction.

The resistor R_2 is in series and therefore connected to the bond graph with a 1-junction. Finally the capacitor C_3 and the resistor R_3 are in parallel and they are therefore connected to a 0-junction. Because the parallel combination of C_3 and R_3 is in series with the rest of the circuit, the 0-junction is connected to the 1-junction.

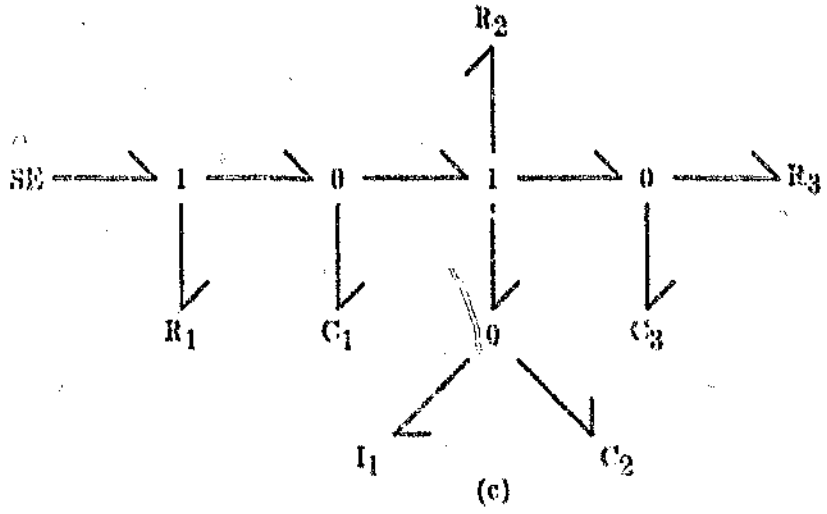
Figure 3.1c shows the two adjacent 3-port, 1-junctions in Figure 3.1b combined into a 4-port 1-junction.



(a)



(b)



(c)

Figure 3.1: An electrical circuit and corresponding bond graph models.

3.2 Bond graph models of mechanical systems

The mechanical domain can be divided into two separate sub-domains, namely mechanical translation and mechanical rotation. These two sub-domains are however modelled in the same way.

3.2.1 Mechanical translation

Consider the mechanical translation system in Figure 3.2 and the corresponding bond graph model in Figure 3.3a. Spring 3 is connected between the position of the force input and

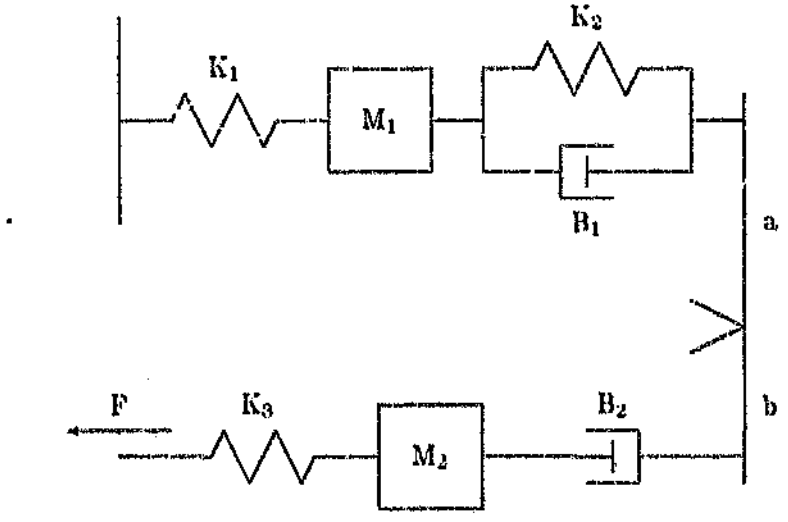
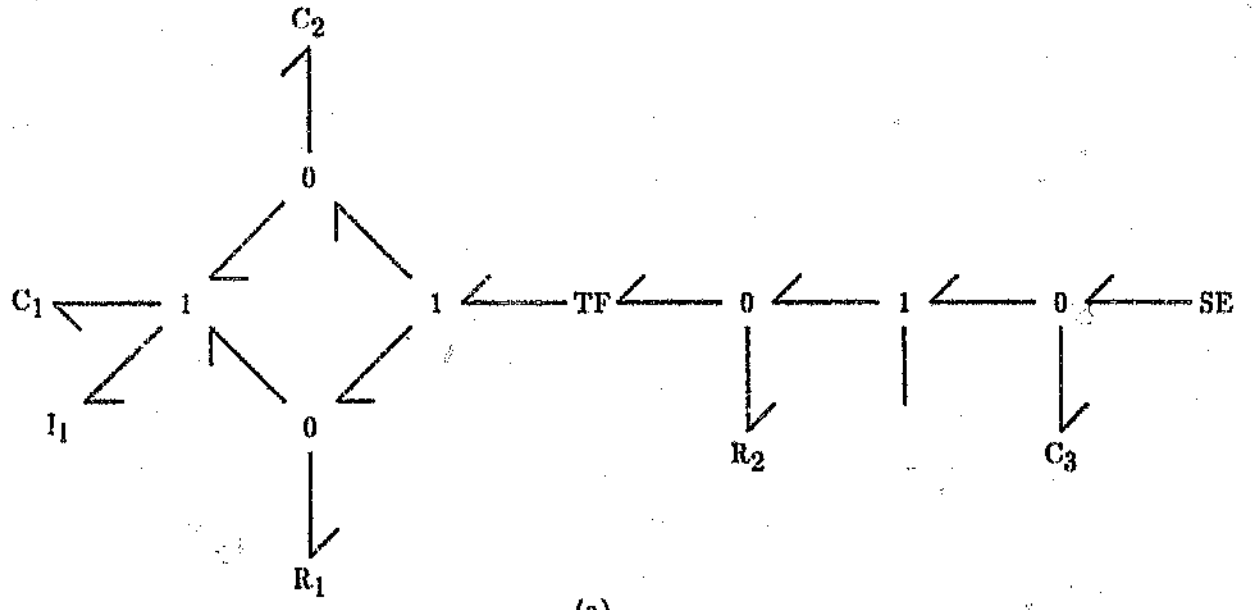


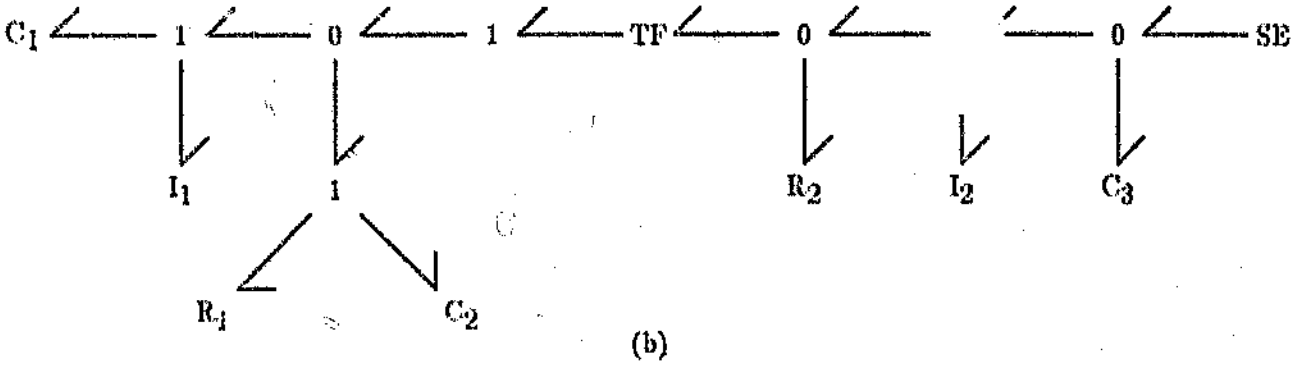
Figure 3.2: A mechanical translation system.

mass 2. The end of the spring, connected to the mass, is moving at a certain velocity (the velocity of mass 2) and the other end is moving at a different velocity. Capacitor 3 (spring 3) is therefore connected to a 0-junction because a 0-junction has different velocities at each of its bonds but the same force on all the bonds. The effort source (force input) is attached to one of the bonds connected to the 0-junction and a 1-junction is connected to the other. The 1-junction, as opposed to a 0-junction, represents a distinct velocity, in this case the velocity of mass 2. Inertia 2 (mass 2) is therefore connected to the 1-junction. Damper 2 is connected between mass 2 and the lever. Because mass 2 is at a particular velocity and the lever is at another, resistor 2 (damper 2) is connected to a 0-junction. The 0-junction is placed between the 1-junction, representing the distinct velocity of mass 2 and the transformer, which represents the lever. A transformer is used to represent the lever because the forces and velocities on either side are related by the ratios of the lengths.

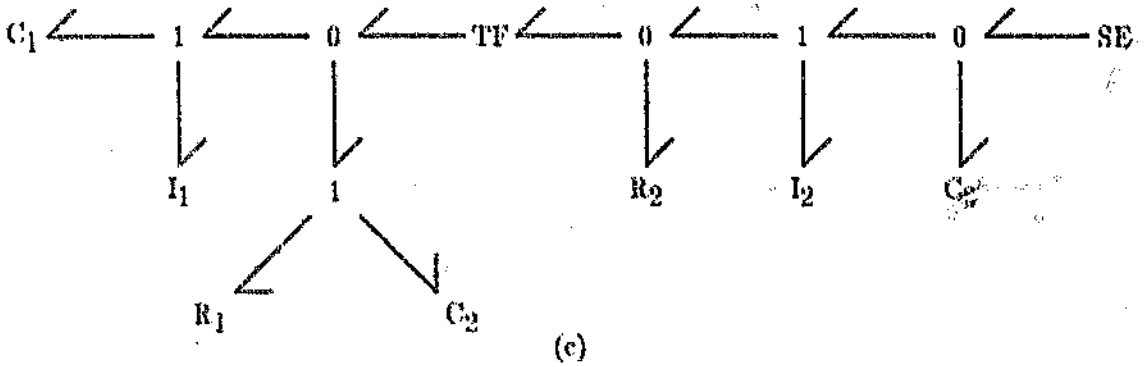
The 1-junction attached to the transformer is used to show that the one side of the lever is at a particular velocity. Both spring 2 and damper 1 are connected between the one side of the lever and mass 1. Because the lever and mass move with two different velocities, Inertia 1 (mass 1) is connected to a 1-junction to show that it is at a distinct velocity. Capacitor 2 (spring 2) and resistor 1 (damper 1) must now be connected between the two 1-junctions. Because the velocity of the spring and the damper is the same and equal to the difference



(a)



(b)



(c)

Figure 3.3: The bond graph of the mechanical translation system.

between the velocity of the lever and the velocity of the mass, capacitor 2 and resistor 1 must be connected to separate 0-junctions. But both the 0-junctions must be connected between the two 1-junctions. Spring 1 is connected between the wall (zero velocity) and mass 1. Because the wall is at zero velocity (reference velocity), it does not have to be included in the bond graph. Capacitor 1 (spring 1) is therefore connected to the 1-junction representing the distinct velocity of mass 1.

Figure 3.3b shows a bond graph "identity". Provided that the power half arrows are as in Figure 3.3a, the loop can be rewritten as shown. In Figure 3.3c, the 1-junction with two bonds connected is removed. A 1 or 0-junction with two bonds attached, can only be removed if the power is flowing through the junction.

3.2.2 Mechanical rotation

As an example of a rotational system, consider the system in Figure 3.4a, where B_1 is a rotational damper, K_1 is a rotational spring and F represents the friction between mass 2 and the surface.

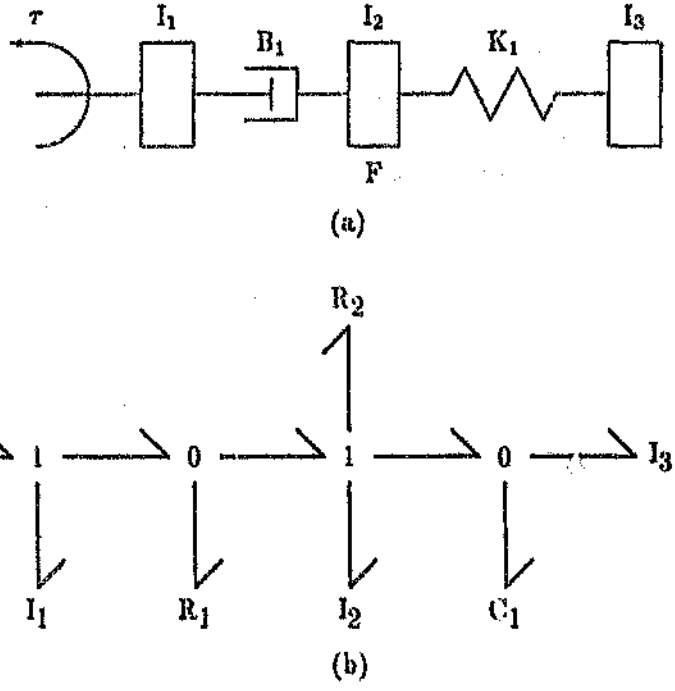


Figure 3.4: A system with mechanical rotation.

The bond graph representation is shown in Figure 3.4b. The torque input occurs at the same angular velocity as mass 1 and both are therefore connected to a 1-junction. Damper 1 is between two distinct velocities, that of mass 1 and mass 2 and is therefore connected to a 0-junction, which is between the two 1-junctions. Inertia 2 is connected to a 1-junction as it is at a distinct velocity. Resistor 2, representing the friction F, is connected to the same 1-junction as mass 2. This is because the friction occurs between the velocity of mass 2 and zero velocity (reference velocity) and the reference does not have to be included. This 1-junction can be explained as follows. The torque entering the 1-junction from the left is required to

accelerate the mass. Some torque is lost due to the friction and the remaining torque can be transferred to the rest of the circuit. Finally capacitor 1 (the spring) is connected to a 0-junction between the 1-junction and inertia 3, because the two ends of the spring are at two different velocities. Inertia 3 does not have to be connected to a 1-junction because its angular velocity occurs between the spring and zero velocity.

3.3 Bond graph models of incompressible fluid systems

In this section, fluid systems, where the static pressure multiplied by the volume flow rate represents most of the transmitted power, will be considered. This implies relatively slow flow rates and large pressures. Although this is mainly the hydraulic systems domain the ideas will be used to model most liquid systems. This is because the majority of the modelling in this dissertation is done for the purposes of controller design and overly complex models are not required. Gas systems cannot usually be modelled effectively using these methods, as the compressibility is an important factor. Gas systems are therefore modelled using the techniques of Section 3.5.

In this section and the following three sections, "fuzzy" statements like *considerable velocity* and *small density changes* will often be encountered. Exact numerical values can often not be given to these statements as they are modelling assumptions and depend on the accuracy required of the model. Similar modelling assumptions are made in all domains.

In general, heat and mass transfer systems are modelled using partial differential equations. This is because the thermal and fluid properties are not only functions of each other, but also of position (that is, they are distributed in space). To use a bond graph model to represent a thermal fluid, it is necessary to spatially divide the fluid into a number of blocks, termed *lumps*. This is because a lumped model is represented by ordinary differential equations. The fluid in each lump is considered to be at the same conditions of temperature, pressure and density. The number of lumps into which a fluid is divided, depends on the accuracy of the model required.

3.3.1 Bond graph variables and elements

When modelling incompressible fluid systems, the *effort* variable is chosen to be the *pressure* (normally gauge pressure), symbol p in units of Pascals, Pa. The *flow* variable is chosen to be the *volume flow rate*, symbol q in units of m^3s^{-1}. The *displacement* variable (the integral of the flow) is therefore the *volume*, symbol V in units of m^3 and the momentum variable (the integral of the effort) is the *pressure momentum*, symbol p_p in units of Pa s [19].

The storage of fluid, in a tank for example, is represented by a capacitor. If a fluid of density ρ is held in a cylindrical tank with cross-sectional area A and the acceleration due to gravity is g, then the pressure [19] is

$$p = \frac{\rho g}{A} V \tag{3.1}$$

The general linear form for a capacitor is

$$p = \frac{1}{C} V \tag{3.2}$$

therefore the capacitance, C for a cylindrical tank is

$$C = \frac{A}{\rho g} \tag{3.3}$$

A pipe or channel through which fluid flows at a considerable velocity is represented by an inertia. If a fluid of density ρ flows through a pipe of cross-sectional area A and length l, the volume flow rate [19] is

$$q = \frac{A}{\rho l} p_p \tag{3.4}$$

The general linear form for an inertia is

$$q = \frac{1}{I} p_p \tag{3.5}$$

therefore the inertance I, is

$$I = \frac{\rho l}{A} \tag{3.6}$$

Inertial elements are therefore important for modelling long pipes, or pipes with a small cross-sectional area.

The flow of a viscous fluid down a pipe [20]) is given by

$$q = K(p_1 - p_2)^{\frac{1}{\alpha}} \tag{3.7}$$

where K is a constant dependent on the restriction and the fluid, and α is a constant in the range 1 to 2. For high flow velocities, where the flow is turbulent, (Reynolds number $Re > 10^5$) $\alpha \approx 2$. When the flow velocity is low enough, so that the flow is laminar, ($Re < 1100$) then $\alpha \approx 1$ [20]. Since the flow rate is a function of the pressure, the flow of fluid through a pipe is represented by a resistor.

3.3.2 A tank with a known inflow and outflow

Consider the system depicted in Figure 3.5a. Fluid enters the tank at a know flow rate q_1 and exits from the bottom of the tank at a known flow rate q_3.

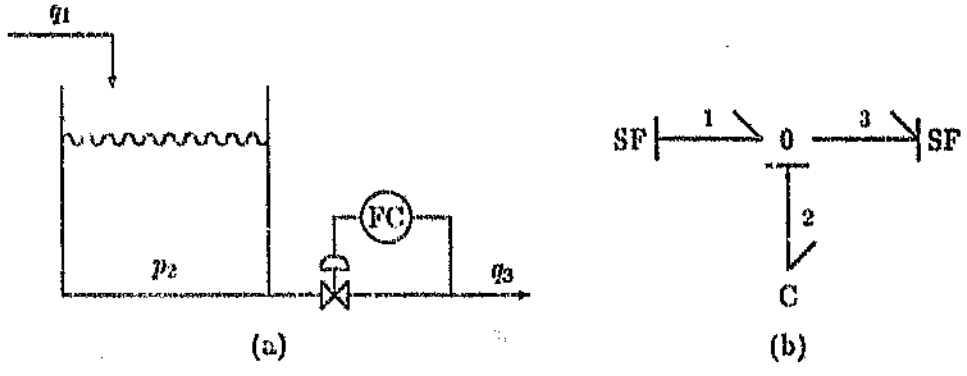


Figure 3.5: A tank filling and emptying, but with the inflow and outflow being known functions of time.

The bond graph of the system is shown in Figure 3.5b. The two flow sources enforce the predefined inflow and outflow rates, while the capacitor represents the storage of fluid in the tank. The constitutive relation for the capacitor is that previously given. The 0-junction represents the conservation of mass, assuming that the density of the fluid flowing into the tank is equal to the density of the fluid leaving.

3.3.3 A tank filling and emptying into the atmosphere

Consider the system depicted in Figure 3.6a, fluid enters the tank with flow rate q_1, and exits from the bottom of the tank into the atmosphere with a flow rate of q_3.

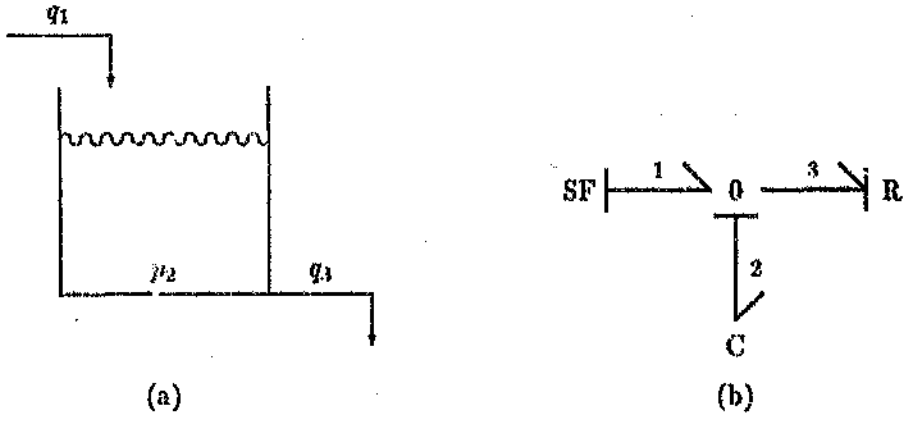


Figure 3.6: A tank filling and emptying into the atmosphere.

The bond graph of the system is shown in Figure 3.6b. The flow source, capacitor and 0-junction are as in the previous example, but a resistor replaces the flow source at the outflow. The resistor has the constitutive law given previously as the outflow rate is a function of the pressure at the bottom of the tank. In the constitutive law given previously, the flow rate was a function of the pressure difference, but because all pressures are defined with respect to atmospheric pressure (termed gauge pressures), the second pressure term is zero.

To emphasize that the outflow is into the atmosphere, the bond graph in Figure 3.6b may be redrawn as in Figure 3.7.

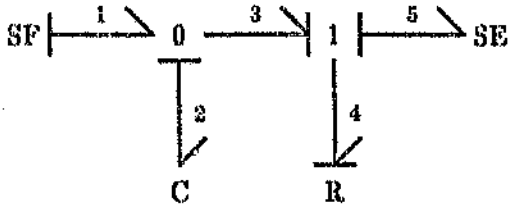


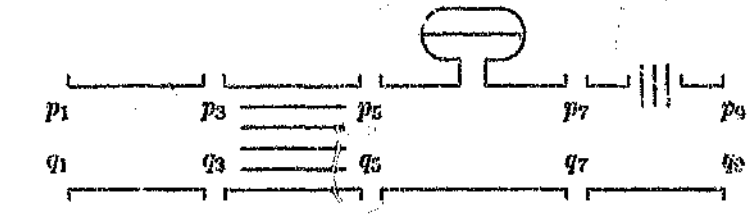
Figure 3.7: An alternative bond graph of the system with the tank filling and emptying into the atmosphere.

In Figure 3.7, the 1-junction is used to shown that the pressure across the resistor is the difference between the pressure at the bottom of the tank and atmospheric pressure. The constitutive relation for the resistor would be

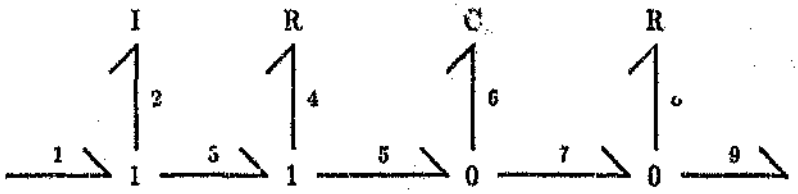
$$q_4 = K p_4^{\frac{1}{n}}$$

3.3.4 A pipe section or fluid lump

Consider the schematic of a pipe section or fluid lump shown in Figure 3.8a, with the corresponding bond graph representation in Figure 3.8b. The inertia, I_2 is used to model the



(a)



(b)

Figure 3.8: A pipe section or fluid lump

acceleration of the fluid and has the constitutive relation discussed previously. The resistor R_4, represents the loss in pressure beyond that required to accelerate the fluid and typically has the constitutive law presented earlier. The capacitor C_6, is intended to model the compliance of the fluid - the compliance of the fluid and the pipe walls. It is feasible to use a capacitor to model the density changes only if these changes are small. If the change in density is large, the thermodynamics of the situation has to be studied. A typical constitutive relation for the capacitor would be

$$p = \frac{1}{C}V$$

where the capacitance C, is the ratio of the volume V, to the bulk modulus B. Finally, resistor R_8, represents the loss of flow in the pipe section due to leakage.

3.3.5 Fluid flow control valves

The flow through a control valve [20] is

$$q = kc_d f(x)(p_1 - p_2)^{\frac{1}{2}} \tag{3.8}$$

where k is a constant dependent on the area of the fully open valve and the density of the fluid.

c_d is the coefficient of discharge of the valve.

$f(x)$ is the fractional area of the valve as a function of valve stem position. When the valve is fully closed, $f(x) = 0$ and when the valve is fully open, $f(x) = 1$.

It is clear that Equation 3.8 is a relation between the flow rate of the fluid passing through the valve and the pressure drop across the valve. A fluid control valve is therefore represented by a resistor. However, because the resistance is a function of the valve stem position and there is no significant power transfer between the stem and the fluid, the effect of $f(x)$ is depicted using a modulating line. The bond graph of the control valve is shown in Figure 3.9.

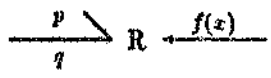


Figure 3.9: A bond graph model of a control valve.

As the control valve relation includes a difference in pressures, it is usually attached to a 1-junction.

If the dynamic characteristics of a valve significantly affect the dynamic behaviour of the system, it is necessary to include these effects in the valve model. This entails the addition of energy storage and possibly other elements.

3.4 Thermal systems

This section discusses heat flow in closed systems where no mass passes into or out of the system boundaries. As mentioned earlier, the fluid in each lump (the boundaries of each lump may or may not be physical) will be assumed to be in pseudoequilibrium.

3.4.1 Bond graph variables

The behaviour of closed thermal systems is often explained using the electric circuit analogy, with heat flow analogous to current and temperature analogous to voltage. The substance through which heat is transferred is modelled as a resistor. The problem with this analogy, for use with bond graphs, is that the product of voltage and current is power, but the product of heat flow and temperature is not (heat flow has the units of power). In order to construct a true bond graph, it is necessary to choose different variables, although the variables heat flow and temperature will be used extensively in the next section by considering what are termed *pseudo bond graphs*.

In this section, the *effort* is chosen to be *temperature*, symbol T in units of Kelvin, K. The *flow* is chosen to be the *entropy flow rate*, symbol $\dot{S}$ in units of $JK^{-1}s^{-1}$. The *displacement* variable is therefore the *entropy*, symbol S in units of JK^{-1}.

As the integral of temperature is physically meaningless there is no momentum variable in thermal systems. The lack of an inertia in the thermal domain has caused some researchers to reconsider some of the initial ideas of having both inertive and capacitive elements in bond graphs [5].

3.4.2 A fluid lump

As shown by Karnopp and Rosenberg[14], a substance in a lump may be represented by a C-field, as shown in Figure 3.10.

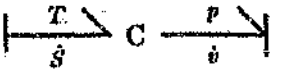


Figure 3.10: A bond graph representation of a substance in a lump.

In Figure 3.10, the hydraulic bond is shown with the power half arrow directed away from the C-field, this is in accordance with the negative sign in the Gibbs equation [14]. Integral causality implies that

$$T = T(s, v)$$

and

$$p = p(s, v)$$

where T is the temperature.

s is the specific entropy ($s = \frac{S}{m}$ where m is the mass of the fluid in the lump and S the entropy of the lump).

v is the specific volume ($v = \frac{V}{m}$ where V is the volume of the lump and m is the mass of the fluid in the lump).

p is the pressure.

Integral causality is obtained by considering the internal energy as a function of the entropy and the volume. The other causal forms are obtained by considering the *enthalpy* h, the *Helmholtz free energy* f, or the *Gibbs free energy* ϕ [14].

3.4.3 Heat transfer with no change of state

Consider as an example, two chambers containing fluid. It is assumed that the fluid in each chamber is considered as a single lump. The fluid in the one chamber is at temperature T_1 and the other at temperature T_2, as shown in Figure 3.11a. Heat is only able to flow through the thermal resistance separating the two chambers. Figure 3.11b shows the bond graph representation of the heat transfer.

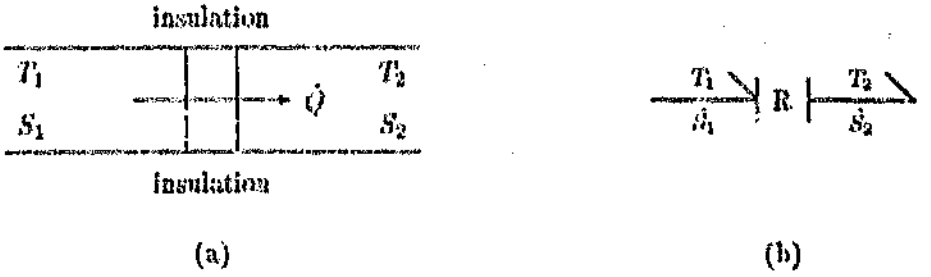


Figure 3.11: Two chambers containing fluids at different temperatures.

A possible constitutive law for the R-field is obtained by considering the heat transfer rate $\dot{Q}$, which is defined as

$$\dot{Q} = HA(T_1 - T_2)$$

where H is the heat transfer coefficient.
A is the area available for heat transfer.

also

$$\frac{k}{l}$$

where k is the thermal conductivity,
l is the thickness.

But $\dot{Q}$ is defined as $T\dot{S}$, therefore

$$T_1\dot{S}_1 = \dot{Q} = T_2\dot{S}_2$$

therefore

$$\dot{S}_1 = \frac{HA(T_1 - T_2)}{T_1}$$

$$\dot{S}_2 = \frac{HA(T_1 - T_2)}{T_2} \tag{3.9}$$

3.5 Compressible thermofluid building blocks

Bond graphs can be used to model thermal systems where a fixed amount of matter is involved [14]. Also, there is a method to model compressible fluid systems based on a Lagrangian rather than the more common Eulerian approach [15]. However, many engineering systems are practically described using a control volume, where matter flows across boundaries carrying energy in several forms with it. Power interaction in such cases needs at least three variables such as pressure, temperature and mass flow rate, rather than the two variables of true bond graphs. Also, the invariant causality of normal bond graphs requires switching as the flow direction reverses [17]. Karnopp[16,17], has shown that it is possible to model such systems using pseudo bond graphs as opposed to true bond graphs, that is, where the efforts and flows used are not complimentary power variables. The remainder of this discussion will be based on these ideas.

3.5.1 Variables and elements for compressible thermofluid pseudo bond graphs

The definitions of the pseudo bond graph variables for compressible thermofluids are shown in Table 3.1. Using these definitions, energy is a *state variable*, whereas in true bond graphs, the energy is a function of the state variables.

The concept of a power bond is now extended to a double bond, where p and $\dot{m}$ are the variables on the one bond and T and $\dot{E}$ or $\dot{Q}$ are the variables on the other. Figure 3.12 shows how two systems, "A" and "B", are connected together using this type of pseudo bond graph.

Karnopp[17], represents the thermal bond (T, $\dot{E}$) as a dotted line, this is done only for clarity. For simplicity, the thermal bonds in this dissertation will be drawn as a solid line.

Variable type	Variable name	Symbol	Units
effort	temperature	T	K
effort	pressure	p	Pa
flow	mass flow rate	$\dot{m}$	kg s^{-1}
flow	energy flow rate	$\dot{E}$	J s^{-1}
flow	heat energy flow rate	$\dot{Q}$	J s^{-1}
flow	rate of change of volume	$\dot{V}$	m^3 s^{-1}
displacement	mass	m	kg
displacement	energy	E	J
displacement	heat energy	Q	J
momentum	pressure momentum	p_p	Pa s

Table 3.1: Compressible thermofluid pseudo bond graph variables.

Figure 3.12: The double bond found in pseudo bond graphs of compressible thermofluid systems.

It must be noted that the energy flow rate is not independent of the mass flow rate and is given by

$$\dot{E} = \dot{m}\left(u + pv + \frac{V_f^2}{2}\right) \tag{3.10}$$

where u is the specific internal energy.
v is the specific volume.
V_f is the flow velocity.

To differentiate between pseudo bond graphs and true bond graphs, Karnopp[17], uses the term accumulator when referring to a capacitor and restrictor when discussing a resistor. In this dissertation the latter two terms will be used.

3.5.2 A junction of two pipes or two lumps in a pipeline, with no corresponding pressure drop

Figure 3.13a shows a pipe junction or the boundary of two fluid lumps in a pipeline. It is assumed that there is no significant pressure drop across the junction.

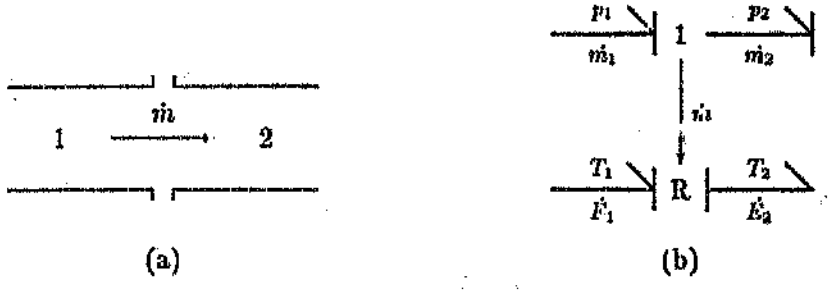


Figure 3.13: A pipe junction or a boundary of two fluid lumps.

From the bond graph in Figure 3.13b, it is clear that the pressures and the mass flow rates are continuous at the point where the two lumps are connected, that is

$$p_1 = p_2$$

and

$$\dot{m}_1 = \dot{m}_2$$

The 1-junction is unnecessary except that it is used to obtain the value of the mass flow rate necessary for the thermal model. Had the bond graph fragment been attached to other bond graph elements, the modulating line could have been connected to any other element which has the particular mass flow rate as an output.

In the thermal model, the fluid in each lump or in each pipe is considered to be at a different temperature. It is also assumed that

$$\dot{E}_1 = \dot{E}_2$$

The convected energy flow rate is given by

$$\dot{E} = c\dot{m}T$$

where c is the specific heat.

$\dot{m}$ is the mass flow rate, either $\dot{m}_1$ or $\dot{m}_2$.

T is the temperature. The temperature will be either T_1 or T_2 depending on whether $\dot{m}$ is positive or negative.

As the temperature is a positive quantity, $\dot{m}_1$ and $\dot{E}_1$ represent the mass and energy leaving lump 1 and $\dot{m}_2$ and $\dot{E}_2$ represents the mass and energy entering lump 2. Assuming that the temperatures are determined internally in lump 1 and lump 2, as discussed in Section 3.5.4, the heat flow rate is given by

$$\dot{E}_1 = \dot{E}_2 = \begin{cases} c\dot{m}T_1 & \text{if } \dot{m} \geq 0 \\ c\dot{m}T_2 & \text{if } \dot{m} < 0 \end{cases} \tag{3.11}$$

One may combine the two previous equations into one constitutive law for the R-field so that only one causal form is necessary. This form is used if the flow changes direction, or if the direction of flow is unknown. The relation for the R-field is

$$\dot{E}_1 = \dot{E}_2 = \frac{c\dot{m}}{2}(T_1 + T_2) + \frac{c|\dot{m}|}{2}(T_1 - T_2) \tag{3.12}$$

As the heat flow rate is a function of both the sum and the difference of the temperatures, the relation cannot be represented using a resistor attached to a 1-junction, as was done previously, but must be represented by an R-field. If this equation is used, the R-field must take on the causality shown in Figure 3.13b. This causality stays fixed regardless of the direction of flow, but the temperature influence changes, as dictated by the equation.

3.5.3 A pipe restriction

Consider the pipe restriction shown in Figure 3.14a. Two bond graph representations are shown in Figure 3.14b and Figure 3.14c. The general form shown in Figure 3.14b is useful when the fluid can choke due to supersonic flow. Under such conditions, the mass flow rate depends not only on the pressure difference, but also on the pressure ratio [16]. For low, subsonic flow, it may be sufficient to assume the mass flow depends only on the pressure difference, then the bond graph in Figure 3.14c can be used. In this dissertation, the bond graph in Figure 3.14c will often be drawn as the 4-port R-field in Figure 3.14b for simplicity.

For such systems

$$\dot{m}_1 = \dot{m}_2$$

and

$$\dot{E}_1 = \dot{E}_2$$

Consider the bond graph in Figure 3.14b, but assume that the mass flow rate is sufficiently slow that it can be calculated using the equation

$$\dot{m} = K(p_1 - p_2)^{\frac{1}{a}} \tag{3.13}$$

Equation 3.13 is similar to Equation 3.7, except that the volumetric flow rate is replaced by the mass flow rate. The constant K now relates the mass flow rate to the square root of the pressure difference and is a function of the square root of the density, whereas before it was a function of the square root of the reciprocal of the density (see Section 5). The term a remains the same.

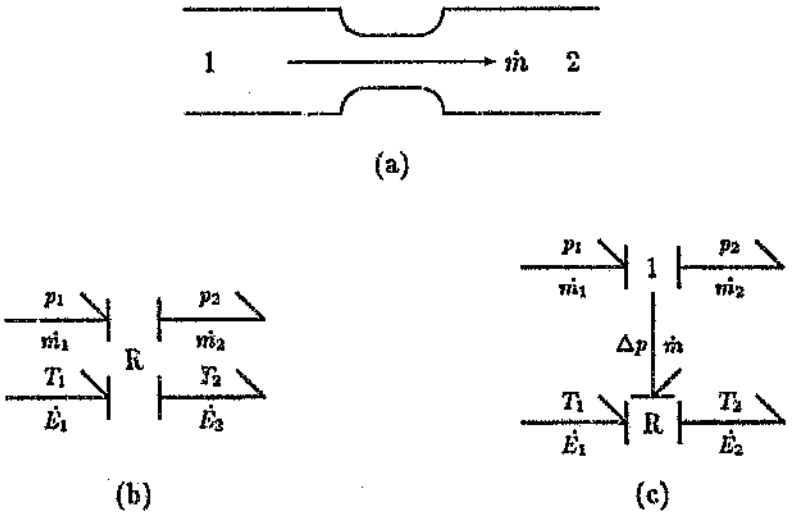


Figure 3.14: A pipe restriction.

As in the previous section, the energy flow rate is defined as

$$\dot{E}_1 = \dot{E}_2 = \begin{cases} c\dot{m}T_1 & \text{if } \dot{m} \geq 0 \\ c\dot{m}T_2 & \text{if } \dot{m} < 0 \end{cases}$$

But as both pressures p_1 and p_2 are inputs to the R-field (note the causality strokes), the heat flow rate has to be written in terms of the pressure difference rather than the mass flow rate. A simple substitution cannot be made, because if $p_2 > p_1$, $\dot{m}$ cannot be calculated, due to the nonlinear characteristic. Therefore

$$\dot{E}_1 = \dot{E}_2 = \begin{cases} cK\,(p_1 - p_2)^{\frac{1}{\alpha}}\,T_1 & \text{if } p_1 \geq p_2 \\ -cK\,(p_2 - p_1)^{\frac{1}{\alpha}}\,T_2 & \text{if } p_1 < p_2 \end{cases} \tag{3.14}$$

If the sign of $(p_1 - p_2)$ is unknown or varies, it is cumbersome to use Equation 3.14. Equations 3.13 and 3.14 can be written

$$\dot{m} = \mathrm{sgn}(p_1 - p_2)K\,|p_1 - p_2|^{\frac{1}{\alpha}} \tag{3.15}$$

$$\dot{E}_1 = \dot{E}_2 = \frac{\mathrm{sgn}(p_1 - p_2)cK\,|p_1 - p_2|^{\frac{1}{\alpha}}}{2}(T_1 + T_2) + \frac{cK\,|p_1 - p_2|^{\frac{1}{\alpha}}}{2}(T_1 - T_2) \tag{3.16}$$

$$\text{where } \mathrm{sgn}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

3.5.4 A pipe segment or fluid lump

Consider a pipe divided into three lumps, labelled 1, 2 and 3, as shown in Figure 3.15a. The bond graph representation of lump 2 is shown in Figure 3.15b. The subscripts in Figure 3.15b correspond to the lump numbers.

The hydraulic portion of the lump model may contain the usual lumped elements discussed in Section 3.3.4. The thermal model contains a thermal capacitor which stores energy $\dot{E}_2$,

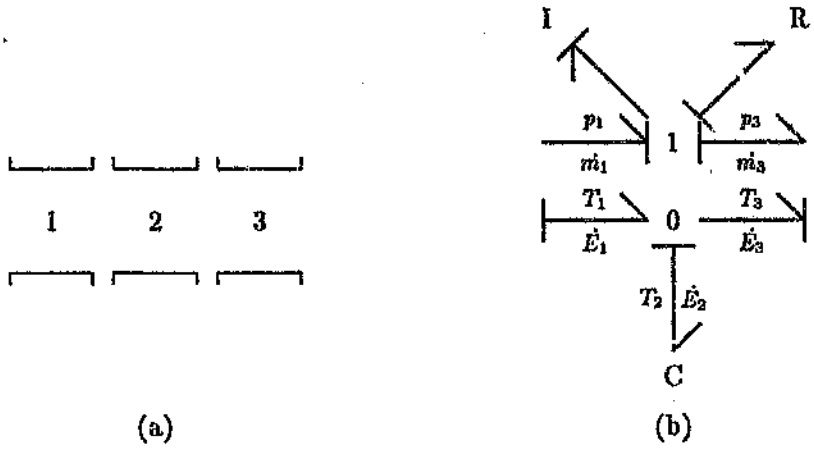


Figure 3.15: The pipe segment or fluid lump.

the difference between $\dot{E}_1$ and $\dot{E}_3$ and "calculates" the temperature of the fluid in the lump. The energy flow rates $\dot{E}_1$ and $\dot{E}_3$ are calculated at the junctions on either side of the lump (see Sections 3.5.2 and 3.5.3).

A possible constitutive relation for the thermal capacitor is

$$T_2 = \frac{E_2}{C_2} = T_o + \frac{E_2}{mc}$$

where T_o is the initial temperature.
E_2 is the energy stored in the lump.
m is the mass of the fluid in the lump ($m = \rho l A$ where ρ is the density of the fluid, l the length of the lump and A the cross sectional area of the pipe).
c is the specific heat of the fluid.

3.5.5 Tank with fluid entering and exiting from the bottom of the tank

Consider the tank shown in Figure 3.16a. It is assumed that the fluid in the tank can be treated as a single lump, that is, there is sufficient mixing to assign a single pressure and temperature to the fluid at all times.

The bond graph of the tank system is shown in Figure 3.16b. The tank is represented by a 2-port C-field, because in general, the temperature and pressure of the fluid in the tank is a function of both the mass and the internal energy.

Although Figure 3.16a shows the tank filling and emptying from the bottom, the set of equations governing the system also hold if the tank is filling from the top. This is because the flow rate of the fluid entering the tank does not depend on the pressure of the fluid in the tank. If the input flow rate is a function of the pressure, as in the coupled tanks example (see Section 3.6.5), then this assumption does not hold.

Liquid systems For many liquid systems, it is assumed that the pressure is a function of the mass only, but the temperature is a function of both the internal energy and mass.

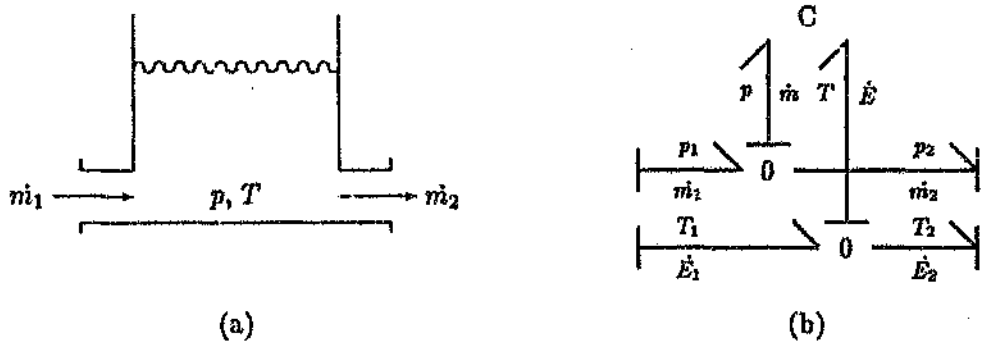


Figure 3.16: A tank filling and emptying through the bottom.

The constitutive relations of such a system, assuming the tank to be cylindrical, are

$$\begin{aligned} p &= \frac{g}{A}m \\ T &= \frac{1}{mc}E \end{aligned} \tag{3.17}$$

Gas systems For gas systems, the pressure and the temperature are functions of both the mass and the energy. For such systems, the tank containing the gas, having a total volume V is assumed to be closed. If the specific internal energy u, and the specific volume v, are defined as

$$\begin{aligned} v &= \frac{V}{m} \\ u &= \frac{E}{m} \end{aligned} \tag{3.18}$$

Then the constitutive relations may be determined from the equation of state of the gas

$$\begin{aligned} T &= T(u, v) \\ p &= p(u, v) \end{aligned} \tag{3.19}$$

3.5.6 Tank with the inflow equal to the outflow

Consider a tank, with the initial height of the fluid equal to the height of the outflow pipe as depicted in Figure 3.17a. It is assumed that the fluid is always at the same height as the outflow pipe. The outflow rate is therefore always equal to the inflow rate.

The bond graph representation of this system is shown in Figure 3.17b. As a fixed mass is assumed, there is no need for a modulating line between the hydraulic bond and the capacitor.

It is clear that this system is identical to the pipe segment discussed in section 3.5.4. It follows, therefore, that a pipe is sometimes said to be modelled as a number of tanks, with the inflow equal to the outflow, connected together. A typical example of this is the model of a heat exchanger.

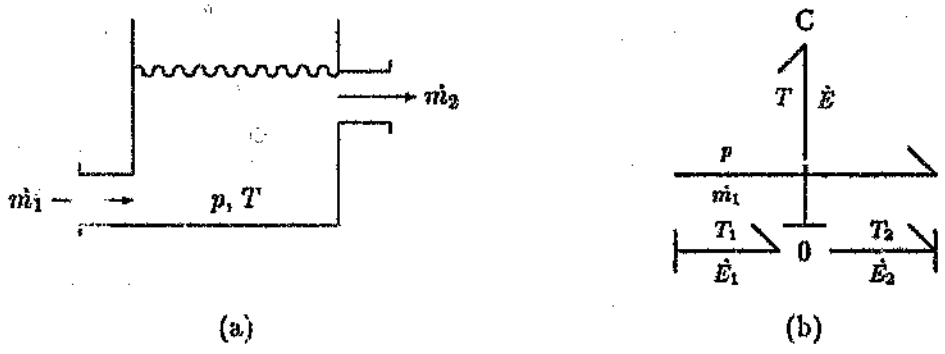


Figure 3.17: A tank with the inflow always equal to the outflow.

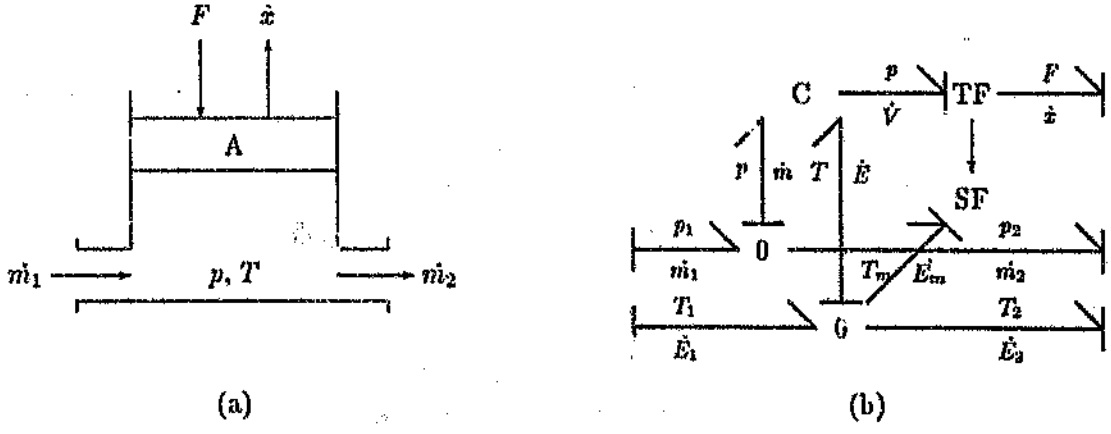


Figure 3.18: A variable volume tank.

3.5.7 A variable volume tank.

In Figure 3.18a, a variable volume tank is shown. This may be used to represent a cylinder of an internal combustion engine, part of a compressor or a hydraulic ram.

The constitutive equations found in Section 3.5.5 still hold, but now the volume V, is variable and the force F, is pA, where p is the pressure and A the cross-sectional area.

The bond graph of the variable volume tank is shown in Figure 3.18b. The equations for this system are:

$$\dot{m} = \dot{m}_1 - \dot{m}_2 \tag{3.20}$$

$$\dot{E} = \dot{E}_1 - \dot{E}_2 - \dot{E}_m \tag{3.21}$$

$$\dot{V} = A\dot{x} \tag{3.22}$$

Where

$$\dot{E}_m = p\dot{V} = F\dot{x} \tag{3.23}$$

is the rate of work and $\dot{x}$ is the piston velocity.

The TF element is essentially a normal bond graph transformer, except that the extracted mechanical power $\dot{E}_m$ must be considered in the calculation of $\dot{E}$. This is the meaning of the modulating line from the transformer to the flow source. The signal interaction and the flow source are needed to connect this pseudo bond graph to a true bond graph.

3.5.8 A heated tank

Consider the heated tank shown in Figure 3.19a. The tank system is that discussed in

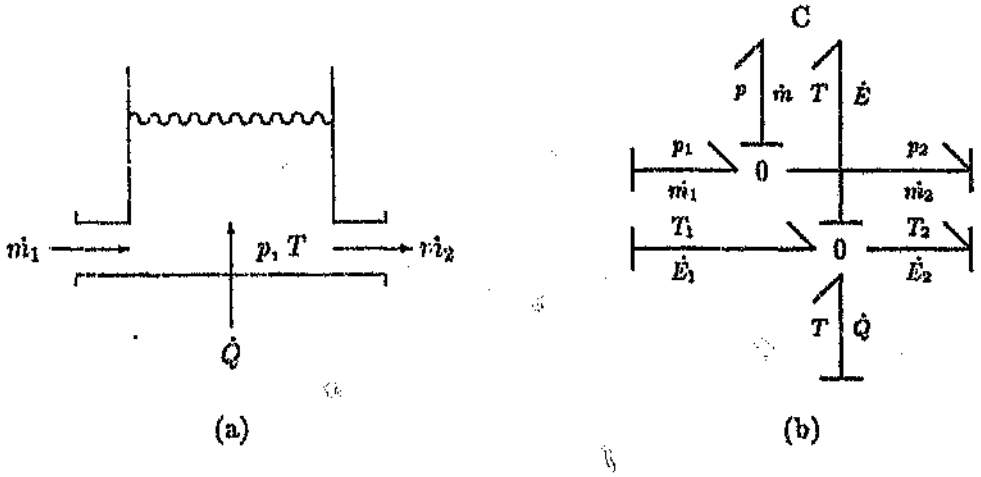


Figure 3.19: A heated tank.

Section 3.5.5, although any of the other tank configurations discussed previously could have been used.

The bond graph of the heated tank is shown in Figure 3.19b. The $T\,\dot{Q}$ bond can be connected to any heat transfer pseudo bond graph.

Conductive heat transfer

Consider the tank containing a fluid at temperature T. In contact with the fluid in the tank is a pipe containing a fluid at temperature T_3. If the temperature of the fluid in the pipe is assumed to be constant, then the equation of this system is

$$\dot{Q} = \dot{Q}_3 = HA(T_3 - T)$$

where H is the heat transfer coefficient.
A is the area for heat transfer.

Two bond graph representations of this conductive heat transfer are shown in Figure 3.20.

(a) (b)

Figure 3.20: Heat transfer pseudo bond graphs.

The bond graph in Figure 3.20a consists of a 2-port R-field and has the constitutive relation

$$\dot{Q} = \dot{Q}_3 = HA(T_3 - T)$$

The bond graph in Figure 3.20b uses a 1-junction to calculate the difference between T_3 and T and ensures that $\dot{Q}$ is equal to $\dot{Q}_3$. The constitutive relation for the resistor is

$$\dot{Q}_4 = HAT_4$$

The direction of the power half arrow and the order of T_3 and T in the equations above, determine the direction of the flow of heat. If $(T_3 - T)$ is positive and the power half arrow is directed towards the tank, then heat is entering the tank and so on.

A tank heated by steam flow through a pipe

Consider the contents of a tank heated by steam flowing thro[illegible] a pipe in contact with the fluid in the tank. It is assumed that the contents of the tank is at a lower temperature than the condensation temperature of steam and that the steam gives off all its latent heat to the tank, that is, it condenses completely afterwards. The heat entering the tank [20] is

$$\dot{Q} = \lambda \dot{m}$$

where λ is the latent h[illegible]at of vaporization.
$\dot{m}$ is the mass flow rate of the steam.

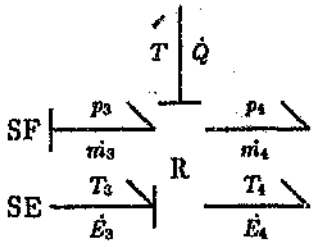


Figure 3.21: The bond graph of heat transfer due to steam flow

The system is represented by the 5-port R-field, shown in Figure 3.21.

The heat flow into the tank is depicted by bond $T\dot{Q}$. The hydraulic bond $p_3\dot{m}_3$ gives the flow rate required to calculate $\dot{Q}$. The latent heat of vaporization λ, is a function of the temperature and the pressure of the steam [20]. If the pressure and temperature are known, then only the bond carrying the flow rate has to be included. If the conditions vary, or if one is required to keep an account of the condensed steam, it is necessary to include all the bonds shown in Figure 3.21.

The constitutive relation for this heat transfer part of the system is

$$\dot{Q} = \lambda \dot{m}_3$$

also

$$\lambda = f(T_3, p_4)$$

3.6 Compressible thermofluid systems

Using the basic "building blocks", discussed in Section 3.5, some common process systems will be modelled.

3.6.1 Tank with a thermofluid entering and exiting through the bottom

Consider the system shown in Figure 3.22.

The model is constructed as follows:

1. The model of the tank in question is discussed in Section 3.5.5.
2. The temperature of the inflow is different to that of the contents, therefore a bond graph fragment to calculate the heat entering the tank is needed. A suitable choice is the pipe lump junction, discussed in Section 3.5.2. Although Section 3.5.2 shows the lump model as a 1-junction and a 2-port R-field, these will be combined into one 4-port R-field for clarity.
3. The mass flow rate of the fluid into the atmosphere depends on the difference between the pressure at the bottom of the tank and atmospheric pressure. The heat flow rate is a function of the temperature of the fluid in the tank and mass flow rate, which is a function of the pressure difference. Although a 1-junction can be used to subtract

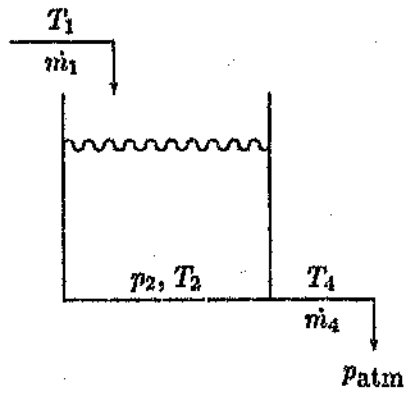


Figure 3.22: Thermofluid entering a tank and exiting through the bottom.

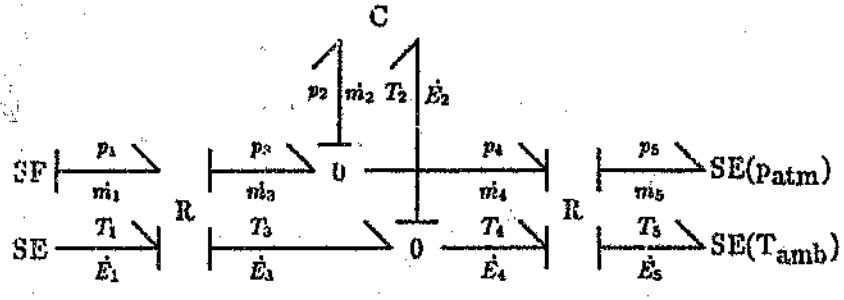


Figure 3.23: Bond graph of a tank with thermofluid entering and exiting through the bottom.

the one pressure from the other, the entire relation can be combined into an R-field as discussed in section 3.5.3. The ambient temperature is not required, but is included for clarity and completeness.

The bond graph of the system is shown in Figure 3.23.

The causal relations are:

The input R-field

$$p_1 = p_3$$

$$\dot{m}_3 = \dot{m}_1$$

$$\dot{E}_1 = c\dot{m}_1 T_1$$

$$\dot{E}_2 = c\dot{m}_1 T_1$$

The C-field

$$p_2 = \frac{g}{A} m_2$$

$$T_2 = \frac{1}{cm_2} E_2$$

The output R-field

$$\dot{m}_4 = K p_4^{\frac{1}{2}}$$

$$\dot{m}_5 = K p_4^{\frac{1}{2}}$$

$$\dot{E}_4 = cK p_4^{\frac{1}{2}} T_4$$

$$\dot{E}_5 = cK p_4^{\frac{1}{2}} T_4$$

As the atmospheric pressure p_5 is the reference pressure (zero), it is not included in the causal relations.

In this example, it is possible to substitute the equation for the mass flow rate into the energy relation for the output R-field as the mass flow rate will always be positive (the fluid always leaves the tank). This cannot be done in general.

3.6.2 A tank with predefined outflow

Consider the system shown in Figure 3.24, where the output flow rate $\dot{m}_3$, is known. The

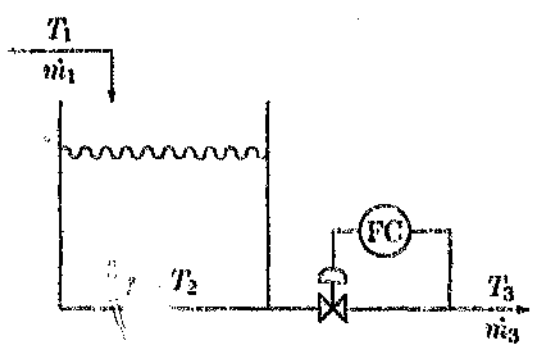


Figure 3.24: Fluid entering and leaving the tank at a known flow rate.

bond graph of the system in Figure 3.24 is shown in Figure 3.25. The flow source connected to bond 3 in Figure 3.25 forces a known output flow rate on the system. With the exception

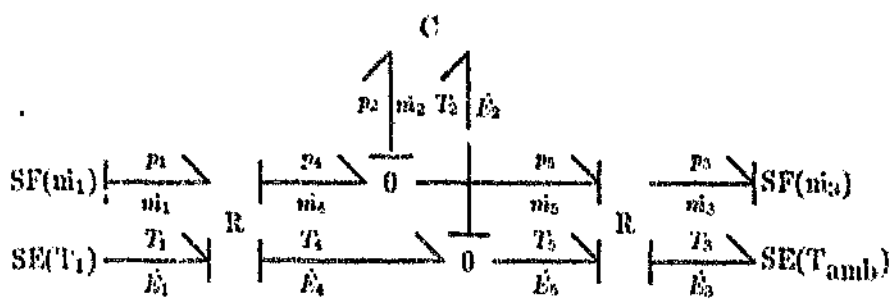


Figure 3.25: A bond graph of a tank with a predetermined outflow rate.

of the flow source connected to hydraulic bond 3 the rest of the bond graph is the same as that in Figure 3.23.

As the R-field between the tank and the output has a different causality assigned to it in comparison with Figure 3.23, the causal relations will be different. A possible set of causal relations are

$$\dot{m}_5 = \dot{m}_3$$

$$p_3 = r_3 - \left(\frac{\dot{m}_3}{K}\right)^{\alpha}$$

$$\dot{E}_3 = c\dot{m}_3 T_5$$

$$\dot{E}_7 = c\dot{m}_3 T_5$$

3.6.3 A tank with thermofluid inflow equal to outflow

Consider the system shown in Figure 3.26, where the inflow and outflow rates are equal.

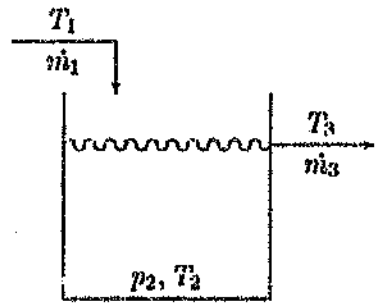


Figure 3.26: Tank with equal inflow and outflow.

The tank model in question is as discussed in Section 3.5.6. As discussed before, 4-port R-fields are used to connect the tank to the inflow and outflow ports. The bond graph of the system in Figure 3.26 is shown in Figure 3.27.

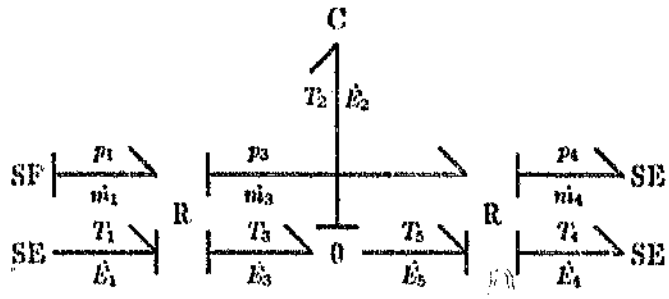


Figure 3.27: Bond graph of a tank with equal inflow and outflow.

A possible set of causal relations for the system are :

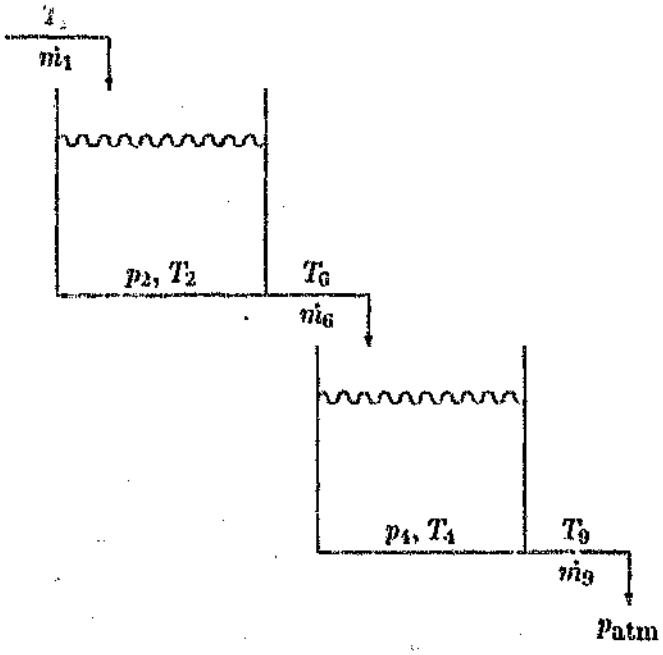


Figure 3.28: Two cascaded tanks.

The inflow R-field

$$\dot{m}_3 = \dot{m}_1$$

$$p_1 = p_3$$

$$\dot{E}_1 = c\dot{m}_1 T_1$$

$$\dot{E}_3 = \dot{m}_1 T_1$$

The capacitor

$$T_2 = \frac{1}{cm} E_2$$

Where m, the mass of the fluid in the tank, is a known constant.

The outflow R-field

$$p_3 = p_4$$

$$\dot{m}_4 = \dot{m}_3$$

$$\dot{E}_5 = c\dot{m}_3 T_4$$

$$\dot{E}_4 = c\dot{m}_3 T_3$$

3.6.4 Two cascaded tanks

Consider the cascaded tank system shown in Figure 3.28. Although the height and temperature of the fluid in the second tank is dependent on the height and temperature of the fluid in the first tank, the two tanks can be treated as separate subsystems. The bond graph of the two-tank system, shown in Figure 3.29, consists of two of the single tank models in Section 3.6.1. The two modulating lines in Figure 3.29 depict that:

1. The inflow to tank 2 is equal to the outflow from tank 1.
2. The temperature of the fluid entering tank 2 is equal to the temperature of the fluid leaving tank 1.

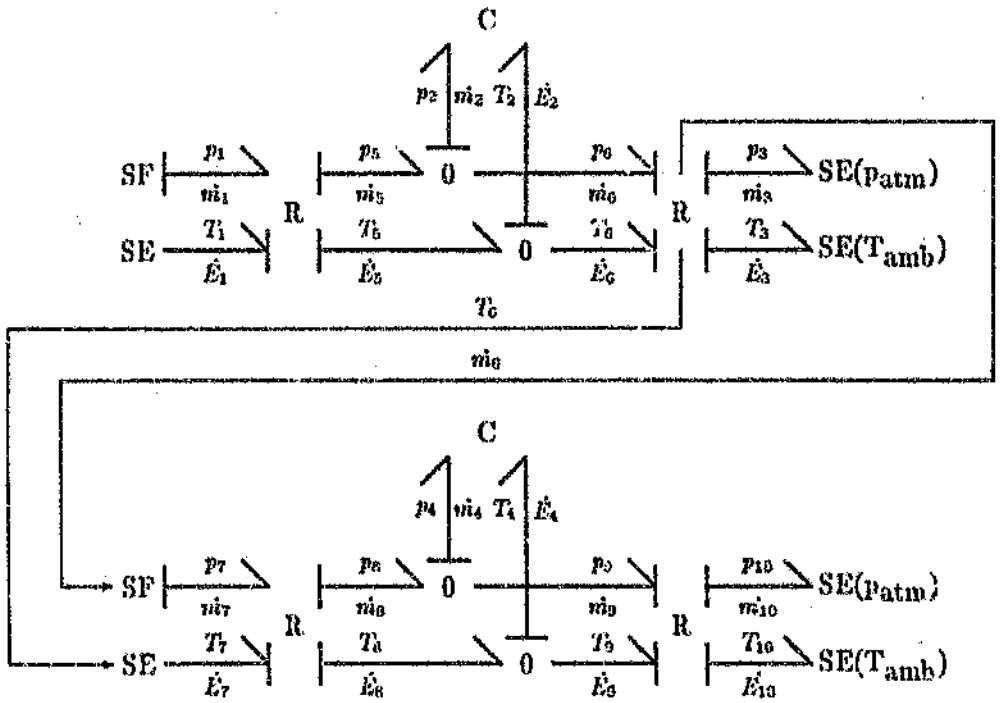


Figure 3.29: The bond graph of the cascaded tank system.

The constitutive relations of the elements have the same form as those in Section 3.6.1, except that the constitutive relations of the flow source and effort source connected to bond 7 are

$$\dot{m}_7 = \dot{m}_6$$

$$T_7 = T_6$$

3.6.5 Two parallel connected tanks

Consider the parallel connected tank system shown in Figure 3.30. The height and temperature of the fluid in the one tank is dependent on the height and temperature of the fluid in the other.

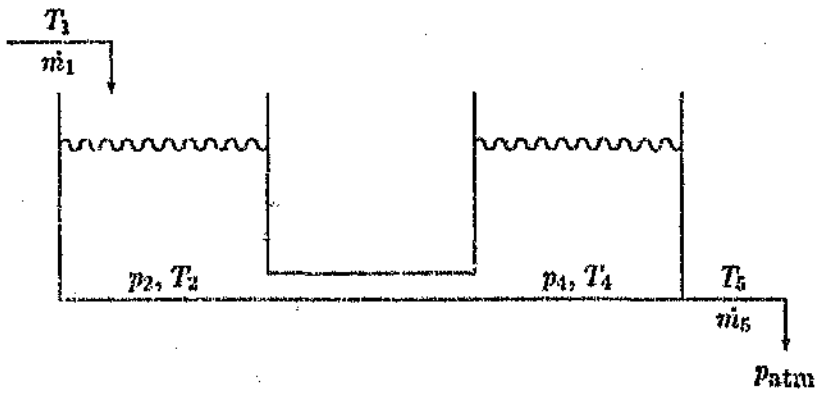


Figure 3.30: Two parallel connected tanks.

The mass flow rate between the two tanks is a function of the pressure difference between the two tanks and the energy flow rate between the two tanks is dependent on the direction of fluid flow and the temperature of the fluid leaving the tank.

Assuming that the pipe connecting the two tanks is sufficiently short to ignore inertial effects, losses due to friction and the heat capacity of the fluid in the pipe, the bond graph of the system is shown in Figure 3.31.

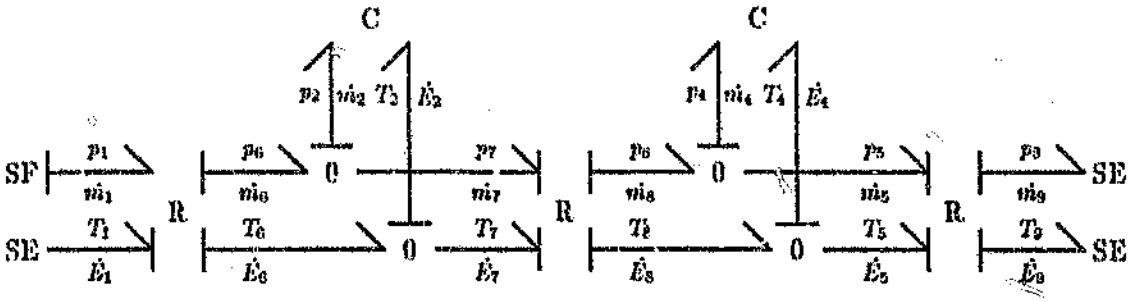


Figure 3.31: The bond graph of the parallel connected tank system.

The two C-fields and the leftmost and rightmost R-fields have the same form as those in Section 3.6.1, but an additional R-field is required to connect the two tanks. The R-field between the two tanks calculates the mass flow rate as a function of the pressure difference and the heat flow rate as a function of the two temperatures and mass flow rate.

Possible causal relations for the middle R-field are

$$\dot{m}_7 = \text{sgn}(p_7 - p_8) K |p_7 - p_8|^{\frac{1}{n}}$$

$$\dot{m}_8 = \text{sgn}(p_7 - p_8) K |p_7 - p_8|^{\frac{1}{n}}$$

$$\dot{E}_7 = \frac{\text{sgn}(p_7 - p_8) c K |p_7 - p_8|^{\frac{1}{n}}}{2}(T_7 + T_8) \quad \frac{c K |p_7 - p_8|^{\frac{1}{n}}}{2}(T_7 - T_8)$$

$$\dot{E}_8 = \frac{\text{sgn}(p_7 - p_8) c K |p_7 - p_8|^{\frac{1}{n}}}{2}(T_7 + T_8) + \frac{c K |p_7 - p_8|^{\frac{1}{n}}}{2}(T_7 - T_8)$$

If the pipe connecting the two tanks is sufficiently long to necessitate including inertial and heat capacity effects, then the bond graph of the system may be drawn as in Figure 3.32.

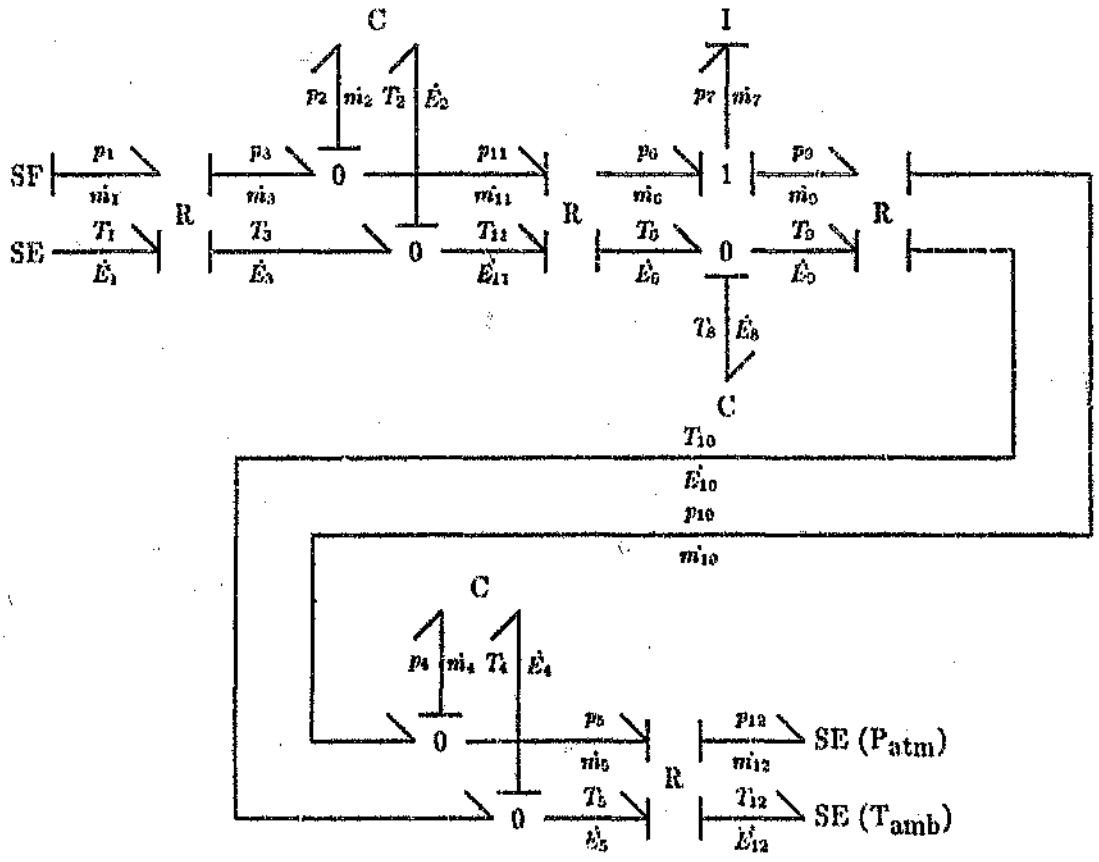


Figure 3.32: A bond graph of two parallel connected tanks with a long pipe connecting the two systems.

In Figure 3.32, it is assumed that the pipe can be represented using only one lump. The C-fields and the R-fields are as before, but the pipe segment discussed in Section 3.5.4 is included.

A possible set of causal relations for the system are:

The first R-field (leftmost)

$$p_1 = p_3$$

$$\dot{m}_3 = \dot{m}_1$$

$$\dot{E}_1 = c\dot{m}_1 T_1$$

$$\dot{E}_3 = c\dot{m}_1 T_1$$

The first C-field (leftmost)

$$p_2 = \frac{g}{A} m_2$$

$$T_2 = \frac{1}{cm_2} E_2$$

The output R-field (second from left)

$$\dot{m_{11}} = \dot{m_6}$$

$$p_6 = p_{11}$$

$$\dot{E_{11}} = \frac{c\dot{m_6}}{2}(T_{11} + T_6) + \frac{c|\dot{m_6}|}{2}(T_{11} - T_6)$$

$$\dot{E_6} = \frac{c\dot{m_6}}{2}(T_{11} + T_6) + \frac{c|\dot{m_6}|}{2}(T_{11} - T_6)$$

The inertia

$$\dot{m_7} = \frac{A}{l} p_{p7}$$

Where p_p is the pressure momentum, A is the cross sectional area and l is the length of the pipe.

The 1-port capacitor

$$T_8 = \frac{E_3}{m_p c}$$

Where m_p is the mass of the fluid in the pipe and equal to the product of the volume of the pipe V, and the density of the fluid ρ.

The output R-field (second from right)

$$\dot{m_{10}} = \dot{m_9}$$

$$p_9 = p_{10}$$

$$\dot{E_9} = \frac{c\dot{m_9}}{2}(T_9 + T_{10}) + \frac{c|\dot{m_9}|}{2}(T_9 - T_{10})$$

$$\dot{E_{10}} = \frac{c\dot{m_9}}{2}(T_9 + T_{10}) + \frac{c|\dot{m_9}|}{2}(T_9 - T_{10})$$

The second C-field (rightmost)

$$p_4 = \frac{g}{A} m_4$$

$$T_4 = \frac{1}{cm_4} E_4$$

The forth R-field (rightmost)

$$\dot{m_5} = K p_5^{\frac{1}{2}}$$

$$\dot{m_{12}} = K p_5^{\frac{1}{2}}$$

$$\dot{E_5} = cK p_5^{\frac{1}{2}} T_5$$

$$\dot{E_{12}} = cK p_5^{\frac{1}{2}} T_5$$

As in the previous case, because the direction of flow is unknown, the long form of the equation to calculate the energy flow rate has to be used. But because the mass flow rate is determined by the inertia and is not a function of the pressure difference, the problem of combining the two equations into one is not encountered.

3.6.6 A heated tank

In this example, a heater is added to the tank in Section 3.6.1. The heater consists of a pipe carrying a hot liquid. It is assumed that the fluid in the pipe is at a uniform temperature T_6. If the temperature is not uniform, a lumped model, similar to that of the heat exchanger presented by Karnopp[17], would have to be used.

The only addition to the system in Section 3.6.1 is the 1-junction, resistor and effort source, discussed in Section 3.5.8. The bond graph is shown in Figure 3.33.

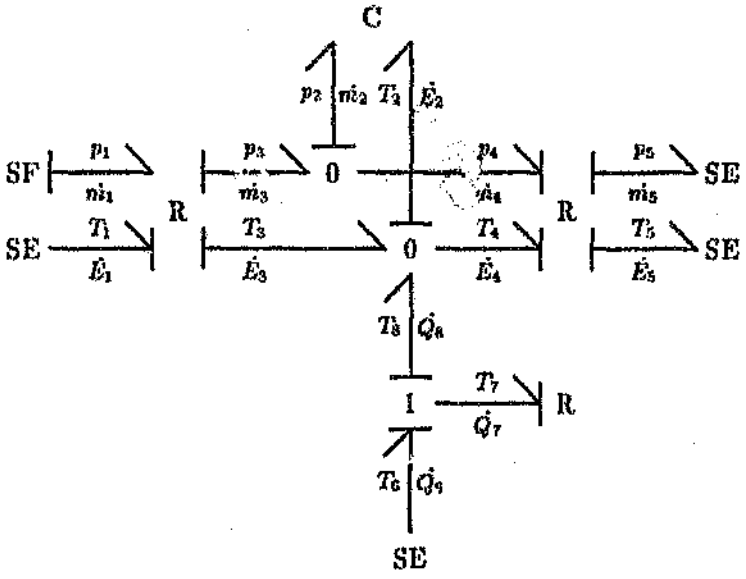


Figure 3.33: The bond graph model of a heated tank.

The constitutive relation for the 1-port resistor is

$$\dot{Q}_7 = HAT_7$$

Chapter 4

The bond graph program

The bond graph modelling program was written to: Allow the user to enter the bond graph model, automatically assign the causality, obtain the equations for each element from the user and run ACSL. Since the bond graph is stored and can be accessed later, rapid structural modification and re-simulation is achieved, allowing the model to evolve progressively.

The computer program is divided into four parts:

1. Entering and modifying the bond graph.
2. Automatic assignment of causality.
3. The creation of the ACSL input file.
4. The simulation of the system using ACSL.

Each of these parts will be discussed in detail in the following sections. A section discussing the limitations of the bond graph package will also be included in this chapter. Prior to these sections, however, sections on the LISP programming language, ACSL, the numbering scheme and the bond graph data structure will be included.

4.1 The LISP programming language

One of the primary advantages of LISP[28,36] is that it is a symbolic manipulation language. This is particularly useful for the implementation of the causality assignment and ACSL input file creation routines, where the manipulation of lists of elements and lists of lists is fundamental to solving the problem. In addition, LISP does not require the variables or the size of the lists to be declared prior to use. Because of this, there is no theoretical limitation to the size or complexity of the bond graph that can be handled (the computer memory size is the limiting factor). Also, this allows large nested list structures to be constructed, which are used extensively in the two routines mentioned above.

4.2 The choice of a simulation package

As shown in Section 2.10, an equation or a number of equations can be written for each bond graph element. Initially, algebraic manipulation of these equations to form the state space

equations was contemplated, but this idea was rejected because algebraic manipulation of complex non-linear equations is difficult to implement, and may be impossible, particularly when algebraic loops and derivative causality exist. Also, if one only has the equations of state, the calculation of each internal variable (for example an effort or flow in the middle of the bond graph) could be tedious.

It was found that ACSL (Advanced Continuous Simulation Language) [1] has the following features:

- It can simulate systems described by time dependent, non-linear differential equations.
- It can sort the equations.
- It can calculate and graphically display the dynamic behaviour of each variable.
- It can calculate the state space equations, the eigenvalues and the eigenvectors (linearizing where necessary).
- It has an implicit integration routine, needed to solve algebraic loops and derivative causality.
- The discrete data points of a continuous function can be entered.

It was decided therefore, to use ACSL as the simulation language.

4.3 Numbering the bond graph

To unambiguously specify all the variables and elements, only the bonds have to be numbered. In this program, however, it is necessary to distinguish between each occurrence of the same element type. This is achieved by numbering all the elements. For simplicity, it was decided that each element should be assigned a unique number. In addition to numbering each element, all the bonds are numbered so that the elements and variables can be referred to in the usual manner.

Consider the fully numbered bond graph in Figure 4.1. Bond 1 is connected to element 1

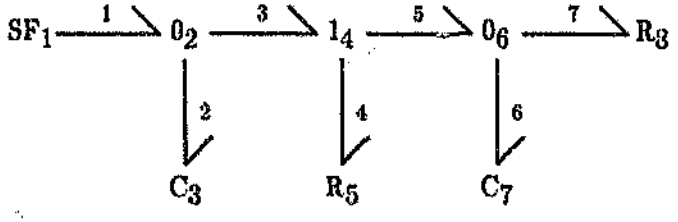


Figure 4.1: A fully numbered bond graph.

which is of type SF and element 2 which is of type 0. Bond 2 is connected to element 2, which is of type 0 and element 3, which is of type C and so on.

4.4 The bond graph data structure

Two types of data structure are used to store the bond graph. The first is a LISP structure and the second, a list of lists. The LISP structure is used in all the modules, but the list of lists is only used in the causality assignment and ACSL input file creation routines.

4.4.1 The LISP structure

The bond graph is stored in two hash tables, *nodes* and *lines*. Each structure in the hash table *nodes* contains three entries: type, the element type (SE, SF, C, I, R, 0, 1, TF or GY), doc-str, the documentation string for the element and equations, the causal equations for the element. Each structure in the hash table *lines* contains two entries: node-pair, the numbers of the elements to which the bond is connected and causality, the causality of the bond. The order of the elements in node-pair is important as this is the direction of the power flow. The half arrow is directed from the first element in node-pair to the second. The causality can be either s, if the causal stroke is at the end of the power half arrow or b, if the causal stroke is at the beginning of the power half arrow.

The two lists, *line-hash-keys* and *node-hash-keys*, are also required to implement the structure. *line-hash-keys* is a list of the bond numbers, used to reference *lines* and *node-hash-keys* is a list of the element numbers, used to reference *nodes*.

Consider the bond graph fragment shown in Figure 4.2. The list *line-hash-keys* would

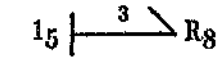


Figure 4.2: A bond graph segment.

be (3) and the list *node-hash-keys* would be (5 8). The structure in *lines*, corresponding to bond 3 would be: node-pair (5 8) and causality b. The structure in *nodes* corresponding to element 5 would be: type 1, doc-str would be some documentation string, possibly nil and equations would be a list of causal relations for the 1-junction also possibly nil. The structure in *nodes* corresponding to element 8 would be: type r, doc-str would be some documentation string, possibly nil and equations would be a list of causal relations for the resistor also possibly nil.

The module that automatically assigns causality to the bond graph assigns a value to causality in each structure in *lines* and the module that creates the CSL input file assigns the equations to equations in each structure in *nodes*. Neither of these two slots can be changed directly by the user.

4.4.2 The list of lists data structure

This data structure consists of three lists:

1. A list of the element numbers.
2. A list of the element types.
3. A list of the bonds connected to each element. Included in this list are variables for the power direction and causality.

As an example of this data structure consider the bond graph in Figure 4.2. In this case, the list of element numbers would be (5 8), the list of element types would be (1 r) and the list of connected bonds would be (((3 out in))((3 in out))). The first element in the list of element numbers is 5, which corresponds to the first element in the list of element types, it being 1 and the first element in the list of connected bonds ((3 out in)). Each element in the list of connected bonds is a bond connected to the element, in this case there is only

one bond connected to the 1-junction. The list of each bond in the list of connected bonds is constructed as follows: The first element is the bond number, in this case 3, the second is the power direction, in this case out of the 1-junction and the third element is the direction of effort flow (the causality), in this case into the 1-junction.

It is necessary to have this data structure as the elements are referenced not only by their element numbers, but also by their element types and the bonds connected to them.

4.5 Entering and modifying the bond graph structure

The bond graph entering and modifying module allows the user to enter the bond graph, either from the keyboard or from a data file containing a previously stored bond graph. It is also possible to enter the bond graph using a combination of these two methods. Once the bond graph has been entered, the structure can be modified, either by adding new bonds and elements, removing existing bonds and elements or re-connecting existing bonds to other existing elements.

If a bond graph fragment, stored in a file, is added to an existing bond graph structure, the bonds and elements are renumbered so that they do not overlap with any of the existing bond or element numbers.

The program searches through the existing bond graph to find the greatest number assigned to any bond and the greatest number assigned to any element. The program increments each of these numbers by one and assigns these numbers to the first bond and element, in the included structure, respectively. Successive numbers are then assigned to each bond and element in the included structure.

For example, assume that the bond graph shown in Figure 4.3a has already been entered into the data structure and the bond graph fragment in Figure 4.3b is to be connected to the first bond graph. In Figure 4.3a, the largest bond number is 6 and the largest element number is 7. The first bond number to be assigned to Figure 4.3b will be 7 and the first element number to be assigned to Figure 4.3b will be 8. The bonds will be assigned the numbers 7, 8, 9 and 10 and the elements will be assigned the numbers 8, 9, 10, 11 and 12. Figure 4.3c shows the bond graph in Figure 4.3b after it has been renumbered. It must be noted that it is possible to renumber the bond graph in many ways. The numbers assigned to the bonds and elements depend on the order in which the bond graph is stored in the data structure.

It is not possible to enter modulating lines into the bond graph data base, but their inclusion is not necessary as the signals they transmit can be included in the constitutive relations of the modulated element. Consider a portion of a conventionally drawn bond graph containing a modulating line as shown in Figure 4.4a. In this example, the modulating line implies that the output from the flow source is equal to f_2, the flow through the resistor.

Figure 4.4b shows the bond graph in a form that can be entered into the program. To include the effect of the modulating line, the user has to, when entering the equations (see Section 4.7) enter that $f_3 = f_2$, where f_2 is the flow through the resistor and f_3 is the output from the flow source.

In addition to allowing the user to enter and modify the bond graph structure, the module also contains functions to clear the data base, undo the last deletion and view the hash table in which the bond graph is stored.

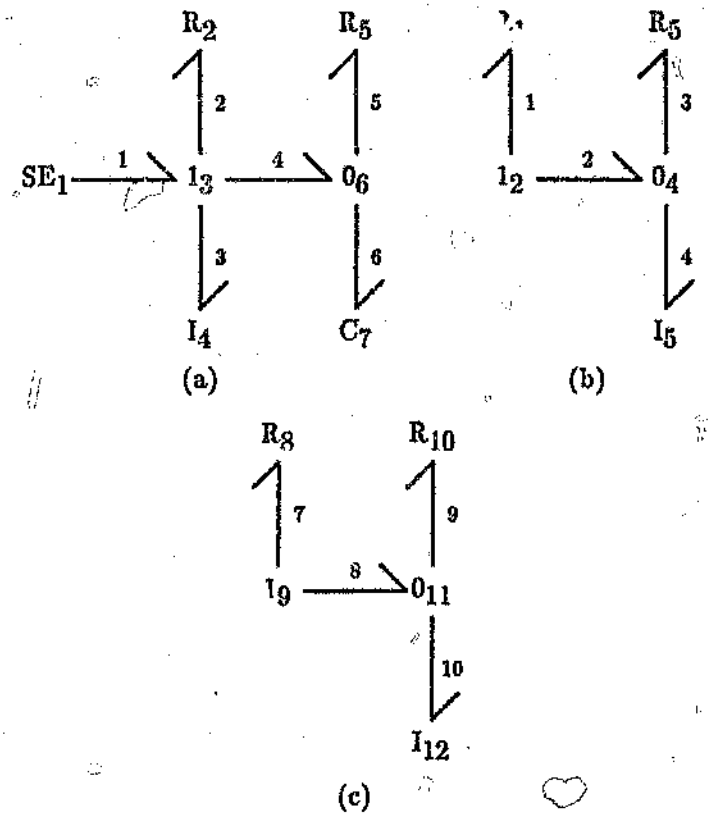


Figure 4.3: The renumbering of a bond graph fragment to be included in the data structure.

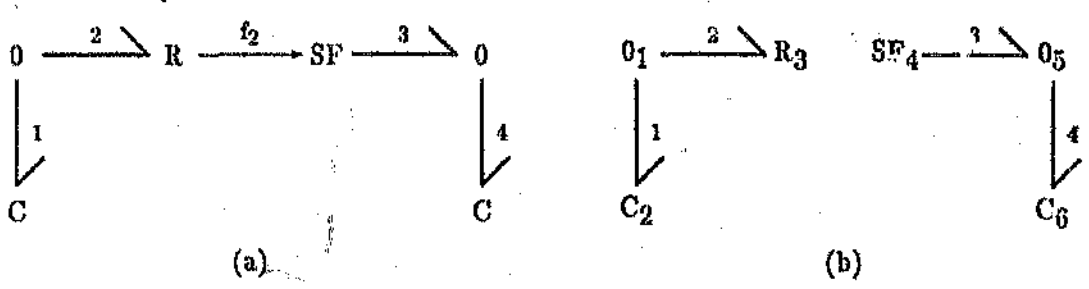


Figure 4.4: The conventional representation of a bond graph with a modulating line and the required representation to be entered into the program.

4.6 Automatic assignment of causality

The *Sequential Causality Assignment Procedure*, SCAP, has been implemented in the bond graph program. An automatic causality assignment procedure was included for a number of reasons:

- To reduce the time required to model and simulate the system.
- To eliminate the human element, which can lead to mistakes.
- To remove the laborious processes and therefore allow the user to concentrate on the parts of the modelling process which require understanding and intuition.

This section will be divided into two subsections, the first will cover the theoretical implementation of SCAP and the second will cover the practical LISP implementation. In the first subsection, reference will be made to procedures, lists and variables. These may or may not be implemented as true LISP functions, lists and variables in the bond graph program discussed in the second subsection. For this reason, the procedure, list and variable names will be written in an *emphasized* font in the first subsection and the true function, list and variable names will be written in a `typewriter` font in the second subsection.

4.6.1 Theoretical implementation of SCAP

The causality assignment procedure chosen, is that devised by Hood, Palmer and Dantzig[10]. The procedure uses the state transition table method, commonly used in asynchronous digital electronics [34], to automate the assignment of causality.

The state transition table, shown in Table 4.1, consists of nine rows and five columns. Each row corresponds to an element type (SE, SF, 0, 1, 1, C, R, TF and GY) and each column corresponds to a different causal condition that can occur at the element. The columns are labelled *start-state, effort-state, flow-state, R-state* and *R-opposite*. Multiport C, I and R-fields are handled, but an IC-field would have to be represented using gyrators and a multiport I or C-field.

Each cell in the state transition table contains the action associated with a particular element and causal condition. For example, "Assign causal stroke next to SF, *get-next, flow-state*". In this case the flow source is assigned a particular causality, a procedure (*get-next*) is called and the variable *state* is assigned *flow-state*.

Two cells (row 1, column 3 and 4) in the state transition table, presented in Table 4.1, differ from those presented in [10]. Hood, Palmer and Dantzig have the cells corresponding to *state* being *start-state* and the element being either a 1 or a 0, giving an error message as the output. These two cells are replaced with a procedure call (*get-next*) and an assignment (*state* is assigned *R-state*). The reasons for this are shown in Appendix A.

In addition to the state transition table, the procedures *get-first, get-next, weak-0, week-1* and *reversal* are also required.

The procedure *get-first* is called when *state* is *start-state*. This occurs at the beginning of the assignment procedure and at the beginning of each assignment path. The procedure searches the bond graph for an element with an unassigned bond attached to it. The elements are chosen according to the usual causality assignment priorities, that is, first SE, then SF followed by I, C, R, TF, GY and finally the 0 and 1-junctions. If an element is found which

	start-state	*effort-state*	*flow-state*	*R-state*	*R-opposite*
SE	assign causal stroke away from SE, *get-next*, *effort-state*	error, stop	*reversal*	error, stop	error, stop
SF	assign causal stroke next to SF, *get-next*, *flow-state*	*reversal*	error, stop	error, stop	error, stop
0	*get-next*, *R-state*	assign causal stroke away from 0, *get-next*, *effort-state*	*weak-0*	assign causal stroke next to 0 on previous bond, *effort-state*	assign causal stroke away from 0 on previous bond, *flow-state*
1	*get next*, *R-state*	*weak-1*	assign causal stroke next to 1, *get-next*, *flow-state*	assign causal stroke away from 1 on previous bond, *flow-state*	assign causal stroke next to 1 on previous bond, *effort-state*
I	assign causal stroke next to I, *get-next*, *flow-state*	*reversal*	*reversal*	error, stop	error, stop
C	assign causal stroke away from C, *get-next*, *effort-state*	*reversal*	*reversal*	error, stop	error, stop
R	*get-next*, *R-state*	*reversal*	*reversal*	error, stop	error, stop
TF	*get-next*, *R-state*	assign causal stroke away from TF, *get-next*, *effort-state*	assign causal stroke next to TF, *get-next*, *flow-state*	assign causal stroke next to TF on previous bond, *effort-state*	assign causal stroke away from TF on previous bond, *flow-state*
GY	*get-next*, *R-state*	assign causal stroke next to GY, *get-next*, *flow-state*	assign causal stroke away from GY, *get-next*, *effort-state*	assign causal stroke next to GY on previous bond, *effort-state*	assign causal stroke away from GY on previous bond, *flow-state*

Table 4.1: Causality assignment state transition table.

meets this criterion, the list, *stack*, is cleared and the element and unassigned bond are assigned to it. If there are no unassigned bonds remaining, the procedure is complete.

The procedure *get-next* gets the element and bond most recently placed in *stack* and searches through the bond graph to find the other element attached to this bond. The function then tries to find an unassigned bond attached to this element. If the element has an unassigned bond attached to it, the element and bond are added to *stack*. If there are no unassigned bonds attached to the element, *reversal* is called.

The procedure *reversal* checks that there are no causal conflicts, that is:

1. An effort source may not have an effort entering.
2. A flow source may not have a flow entering.
3. A 0-junction must have only one effort input.
4. A 1-junction must have only one flow input.
5. A transformer must have one effort entering and one effort leaving.
6. A gyrator must have both efforts entering or leaving.

If there is a causal conflict, the causality assigned to the elements in *stack* is undone. All but the first two elements that were placed in *stack* are removed and *state* is set to *R-opposite*. If there are no causal conflicts, *stack* is searched for a 0 or 1-junction with strong causality and an unassigned bond attached to it (a strong 1-junction is one with a flow entering it and a strong 0-junction is one with an effort input). If there is strong junction with at least one unassigned bond attached to it, the junction and unassigned bond are are added to *stack* and *state* is set to *weak-0* or *weak-1* depending on whether the junction is a 0 or 1. If there are no strong junctions with unassigned bonds attached to them, *state* is set to *start-state*.

The procedure, *weak-1*, checks the 1-junction for one of three cases:

1. There is a flow into the 1-junction.
2. There is no flow into the 1-j - tion and one unassigned bond attached to it.
3. There is no flow into the 1-junction and there is more than one unassigned bond attached to it.

Depending on which of these hold, one of three actions is carried out. The actions are shown in Table 4.2.

The procedure, *weak-0*, checks the 0-junction for one of three cases:

1. There is an effort into the 0-junction.
2. There is no effort into the 0-junction and only one unassigned bond attached to it.
3. There is no effort into the 0-junction and there is more than one unassigned bond attached to it.

Depending on which of these hold, one of three actions is carried out. The actions are shown in Table 4.3.

The meaning of each causal state is:

Decision	Action
flow in	assign causal stroke next to 1, *get-next*, *flow-state*
no flow in and only one unassigned bond attached to the 1-junction	assign causal stroke away from 1, *get-next*, *effort-state*
no flow in and more than one unassigned bond connected to the 1-junction	*reversal*

Table 4.2: The procedure *weak-1*.

Decision	Action
effort in	assign causal stroke away from 0, *get-next*, *effort-state*
no effort in and only one unassigned bond attached to the 0-junction	assign causal stroke next to 0, *get-next*, *flow-state*
no effort in and more than one unassigned bond connected to the 0-junction	*reversal*

Table 4.3: The procedure *weak-0*.

start-state The variable, *state* is set equal to *start-state* at the beginning of each causality assignment path.

effort-state The causality assigned implies that the effort is the output from the current bond graph element.

flow-state The causality assigned implies that the flow is the output from the current bond graph element.

R-state This occurs when an arbitrary causality is assigned to an element.

R-opposite This occurs when the causality assigned while *state* was equal to *R-state*, causes a causal conflict. The variable, *state* is set to *R-opposite* and the other causality is assigned.

To demonstrate the causality assignment procedure, consider the unaugmented bond graph shown in Figure 4.5. The first step in the procedure is to set *state* equal to *start-state* and

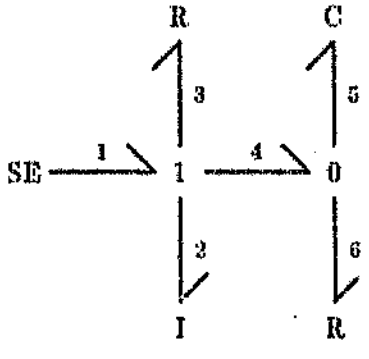


Figure 4.5: An unaugmented bond graph.

therefore call *get-first*. The procedure returns the effort source and unassigned bond 1, which

are entered into *stack*. The state transition table is entered at the first row and the first column, corresponding to the current element being an effort source and *state* being *start-state*. The resulting action is to assign the causal stroke away from SE, call *get-next*, which retrieves the 1-junction and bond 2 (although bond 3 or bond 4 may also be used as they too are unassigned) and set *state* equal to *effort-state*. The 1-junction and bond 2 are added to *stack*.

The transition table is now entered at row 4 and column 2 corresponding to a 1-junction and *effort-state*, resulting in *weak-1* being called. The procedure *weak-1* enters at row three (Table 4.2), the consequence being that reversal is called. The procedure tries to find a causal conflict. As there are none, the procedure searches through *stack* to see if there is a strong 1 or 0-junction with an unassigned bond attached to it. There are none and *state* is therefore set equal to *start-state*.

Because *state* is equal to *start-state*, *get-first* is called. The procedure returns the inertia and unassigned bond 2 (although the capacitor and bond 5 could also be the output). The list, *stack* is reset and the inertia and bond 2 are assigned to it. The state transition table is entered at column 1 and row 5. The result is that a causal stroke is assigned next to the inertia and *get-next* is called. The procedure *get-next* returns the 1-junction and the unassigned bond 3 (bond 4 may also be chosen). The variable *state* is set to *flow-state* and the 1-junction and bond 3 are added to *stack*.

The state transition table is entered at row 4 column 3, with the effect that the causal stroke is assigned next to the 1-junction on bond 3 and *get-next* is called. The procedure *get-next* looks to see if there are any unassigned bonds attached to the resistor (attached to the 1-junction). As there are no other unassigned bonds, *reversal* is called. There are no causal conflicts, but the 1-junction is strong (due to bond 2) and bond 4 is unassigned, these are therefore added to *stack*. The variable, *state* set to *weak-1*. The procedure *weak-1* is entered at the first row and the resulting action is to assign the causal stroke next to the 1-junction on bond 4 and to call *get-next*. The procedure returns the 0-junction and unassigned bond 5 (bond 6 may also be the output) and these are added to *stack*. The variable *state* is set to *flow-state*.

The transition table is entered at row 3 column 3, calling *weak-0*. Because there is no effort input to the 0-junction and there are more than one unassigned bonds attached, *reversal* is called. There are no causal conflicts and also no strong junctions with unassigned bonds in *stack*. Therefore *state* is set to *start-state*.

The procedure *get-first* is called. It returns the capacitor and unassigned bond 5. The list *stack* is reset and the capacitor and bond 5 are assigned to it. The state transition table is entered at row 6 column 1, with the result that the causal stroke is assigned away from the capacitor and *get-next* is called. The procedure adds the 0-junction and unassigned bond 6 to *stack* and *state* is set to *effort-state*.

The state transition table is entered at row 3 column 2. The causal stroke is placed away from the 0-junction on bond 6. The procedure *get-next* is unable to find another unassigned bond attached to the resistor and therefore calls *reversal*. Because there are no causal conflicts and no strong 0 or 1-junctions with unassigned bonds in *stack*, *state* is set to *start state*. As *get-first* is unable to find another unassigned bond, the assignment procedure is complete.

The fully augmented bond graph is shown in Figure 4.6.

The causality assignment procedure described above is complete in that it attempts to resolve all causal conflicts. When a causal conflict is encountered which cannot be resolved the user is told that there is an error in the bond graph structure. Hood, Palmer and Dantzig[10]

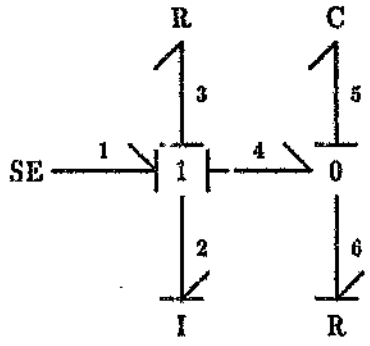


Figure 4.6: The fully augmented bond graph.

state that this method is more efficient and requires less code than conventional *if-then-else* programming techniques.

4.6.2 Program implementation

The state transition table, shown in Table 4.1, has been simplified so that only a few LISP routines are needed. The state transition table used in the program is shown in Table 4.4.

The LISP function `away-next-effort` assigns a causal stroke away from the element in the first list in the global list `*stack*`, sets the global variable `*state*` equal to `effort-state` and calls `get-next`.

The function `next-next-flow` assigns a causal stroke next to the element in the first list in `*stack*`, sets `*state*` equal to `flow-state` and calls `get-next`.

The function `error-stop` informs the user that there is an error in the bond graph structure stating where it occurred and requests that the structure be altered before re-assigning the causality.

The function `away-previous-effort` assigns a causal stroke away from the element in the second list in `*stack*` and sets `*state*` equal to `effort-state`. It must be noted that this is equivalent to assigning the causal stroke *next* to the bond graph element in the first list in `*stack*`, but the bond to which the causal stroke is being assigned must be that in the second list in `*stack*`. This is what is implied in the state transition table, Table 4.1.

The function `next-previous-flow` assigns a causal stroke next to the element in the second list in `*stack*` and sets `*state*` equal to `flow-state`. As before, this is equivalent to assigning the causal stroke *away* from the bond graph element in the first list in `*stack*`, but the bond to which the causal stroke is being assigned must be that in the second list in `*stack*`. This is what is implied in the state transition table, Table 4.1.

The function `next-r` sets `*state*` equal to `r-state` and calls `get-next`.

The function `reversal` checks to see if there is a causal conflict. If there is, the causality assigned to each of the elements in `*stack*` is undone. All but the last two elements in `*stack*` are removed and `*state*` is set to `r-opposite`. If there are no causal conflicts, the bond graph is searched for a 0-junction with strong causality, having one or more unassigned bonds attached to it. If such an element is found, it and the unassigned bond are added to `*stack*` and `weak-0` is called. If not, the bond graph is searched for a 1-junction with strong causality, having one or more unassigned bonds attached to it. If such an element is found,

	start-state	effort-state	flow-state	R-state	R-opposite
se	away-next-effort	error-stop	reversal	error-stop	error-stop
sf	next-next-flow	reversal	error-stop	error-stop	error-stop
0	next-r	away-next-effort	weak-0	away-previous-effort	next-previous-flow
1	next-r	weak-1	next-next-flow	next-previous-flow	away-previous-effort
i	next-next-flow	reversal	reversal	error-stop	error-stop
c	away-next-effort	reversal	reversal	error-stop	error-stop
r	next-r	reversal	reversal	error-stop	error-stop
tf	next-r	away-next-effort	next-next-flow	away-previous-effort	next-previous-flow
gy	next-r	next-next-flow	away-next-effort	away-previous-effort	next-previous-flow

Table 4.4: The causality assignment state transition table, used in the computer implementation.

it and the unassigned bond are added to *stack* and weak-1 is called. If no such element is found, *state* is set to start-state.

The function weak-1 checks to see whether there is a flow input to the 1-junction and counts the number of unassigned bonds. If there is a flow input, next-next-flow is called. If there is only one unassigned bond and no flow input, away-next-effort is called. Finally, if there is no flow input and more than one unassigned bond, reversal is called.

The function weak-0 checks to see whether there is an effort input to the 0-junction and counts the number of unassigned bonds. If there is an effort input, away-next-effort is called. If there is only one unassigned bond and no effort input, next-next-flow is called. Finally, if there is no effort input and more than one unassigned bond, reversal is called.

The function get-next finds the other element attached to the last bond in *stack* and checks to see if there are any unassigned bonds connected to the element. If there are, the element and unassigned bond are added to *stack*. If not, reversal is called.

The function get-first resets *stack*, gets an element with an unassigned bond attached to it and places the element and bond in *stack*. The elements are chosen using the normal SCAP priority, first SE, then SF followed by C, I, R, TF, GY, 0 and finally 1. If a resistor is chosen, an algebraic loop will result and therefore in addition to placing the element and bond in *stack*, they are also added to the global list *r-algebraic-loop*. In the similar manner, if get-first has, as the output, either a transformer, gyrator, 1 or 0-junction an algebraic loop will result. In this case however, in addition to adding the element and bond to *stack*, they are included in the global list *bond-algebraic-loop*. It is necessary to have both lists because the module that creates the ACSL input file differentiates between the two causes of algebraic loop.

The state transition table is implemented as a forward chaining algorithm, with two antecedents and one consequent in the rules. The antecedents are *state* and the type of the element in the first list in *stack*. The consequent is the function that has to be called. The rule base used, is that presented by Winston and Horn[36].

Once the causality has been assigned to the bond graph, the bond graph is searched for storage elements with derivative causality. If such elements are found, the storage elements and the bonds assigned derivative causality are added to the list *deriv-causality*.

The lists *r-algebraic-loop*, *bond-algebraic-loop* and *deriv-causality* are used to form the ACSL input file.

4.7 Creation of an ACSL input file

This module takes the bond graph stored in the data file and creates an ACSL input file from it. The module uses the equations entered by the user, the equations in the data file (if there are any) or a combination of both to form the input file.

4.7.1 Bond graph program variables

By convention, the bond graph variables, are written with the bond number as the subscript. For example, the effort on bond 5 would be referred to as e_5. However, because the program and ACSL are unable to accept subscripts, the bond graph variable is followed by the bond number. For example, the effort on bond 5, would be referred to as e5.

The program is only able to handle the generalized bond graph variables, effort e, flow f, displacement q and momentum p. The variables, natural to modelling systems in a particular domain, therefore have to be converted to the generalized variables, before the equations can be entered. .

4.7.2 ACSL data required by the equation formulation routine

The user is required to enter two variables needed by ACSL, the duration of the simulation, `simtim` and the integration step size, `ccalc`.

The following initializations are also made:

1. Each ACSL plot contains 1000 data points (`points = 1000`).
2. The communication interval, `cint` is set equal to the duration of the simulation divided by the number of points (`cint = simtim / points`).
3. The number of integration steps in each communication interval `nstp` is calculated as the communication interval divided by the integration step size (`nstp = cint / ccalc`).
4. On each iteration through the **derivative** section, ACSL checks to see if the simulation should be terminated. The statement for this, as written by the computer program is `termt(t .gt. simend)`, that is, when the time is *greater* than `simend`. To ensure that the simulation ends at `simtim`, the end time, `simend` is calculated as the simulation duration less the integration step size (`simend = simtim - ccalc`).

4.7.3 Bond graph data required by the routine

As mentioned earlier, modelling is an iterative process and a number of modifications might have to be made, before the model is a satisfactory representation of the system. An iteration could involve changing the form or the coefficients of the equations. Because every equation is not modified at each iteration, it is advantageous to store the equations so that they do not have to be re-entered each time. Included in each structure in the the hash table *nodes*, is an entry for the equations. If this variable does not have a value assigned to it, the user is prompted to enter the equations. If it does have a value, the user is asked whether the stored equations are still valid. If the equations still hold, they are used, if not, the user is prompted to re-enter the correct equations. Possible reasons for the equations being incorrect are:

- The magnitude of one of the coefficients may need to be changed.
- The form of the equation may be not represent the behaviour of the element sufficiently well.
- Another bond may have been connected to the element and another equation and another term are therefore needed.
- The causality of the bonds attached to the element may have changed.
- The bond numbering may have changed.

When entering the equations, the user is given:

1. The element for which the equations are to be entered. This information includes the element type, the number assigned to the element as well as the nu· ·bers of the bonds connected to the element.

2. The documentation string for the element.

3. The causality assigned to the bonds attached to the element. For example if bond 6 is connected to a resistor and bond 6 is assigned conductive causality, then the program would inform the user that for the resistor, the causality assigned implies that `f6` is a function of `e6`.

When entering the data for a resistor, inertia or capacitor, the user may enter either a linear or non-linear relation governing the behaviour of the element. If the relation is linear, it is only necessary to enter the linear coefficient. For example if one is entering the linear equation for a capacitor attached to bond 7, then regardless whether the causal relation is $e_7 = \frac{1}{C_7} q_7$ or $q_7 = C_7 e_7$, only the value of the capacitance (C_7) must be entered. If an element has a non-linear equation governing its behaviour, it is necessary to enter the entire equation. For example, if the causal relation for a resistor, assigned resistance causality, attached to bond 3 is $e_3 = 3.5 f_3 \sqrt{4.78 f_3}$, the user is required to enter `e3 = 3.5 * f3 * sqrt(4.78 * f3)`.

If a resistor, capacitor or inertia has more than one bond attached to it, it is not possible to enter just the coefficients. Instead it is necessary to enter the equations in full. Consider the resistor field in Figure 4.7. Bonds 1 and 2 have resistance causality and bond 3 has conductance causality. The user is required to enter three different equations, one with effort 1

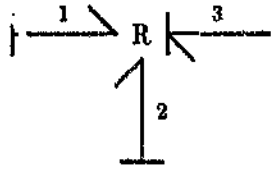


Figure 4.7: A resistor field.

as the output, another with effort 2 as the output and the last with flow 3 as the output. Assume that the relations governing the behaviour of the R-field are $e_1 = 5.3 f_1 f_2 + 6.9 e_3$, $e_2 = 0.134 f_1 \sqrt{f_2} - 5 e_3$ and $f_3 = 2.6 e_3 + 7.9 f_1$. The user will be prompted to enter `e1` as a function of `f1`, `f2` and `e3` and must enter `e1 = 5.3 * f1 * f2 + 6.9 * e3`. The user will then be prompted to enter `e2` as a function of `f1`, `f2` and `e3` and must enter `e2 = 0.134 * f1 * sqrt(f2) - 5 * e3`. Finally the user will be prompted to enter `f3` as a function of `f1`, `f2` and `e3` and must enter `f3 = 2.6 * e3 + 7.9 * f1`.

If a storage element has integral causality assigned to it, in addition to entering the causal relations, the user is required to enter the initial value (at time $t = 0$) of the energy variable. For example, consider an inertia attached to bond 10, assigned integral causality. The user is required to enter the momentum p_{10} at time $t = 0$. This value may either be a constant or an expression.

If the output from either of the two sources is anything but a step at time $t = 0$, it is necessary to enter the full form of the equation. For example, if an effort source attached to bond 25, has the sinusoid $5 \sin 2t$ as its output, the the user must enter `e25 = 5 * sin(2 * t)`. If the input to the system is a step at time $t = 0$, the user is required to enter only the height of the step. For example if the output from a flow source attached to bond 32 is $f_{32} = 6.7u(t)$, then the user must enter `6.7`.

When entering the data for either a transformer or gyrator, the user is required to enter only the value of the modulus. The user is given the equation form so that the definition of the modulus is clear. For example, consider a transformer connected to bonds 5 and 6. The user will be told that the modulus, m is defined as e6 = m e5. The user may enter a number if the modulus is constant or an equation string if the transformer or gyrator is modulated. For example if a transformer modulus, $m = 32$, the user must enter 32. If the modulus of a modulated gyrator $r = 6.7f_2$, the user must enter `6.7 * f2`.

If the user enters only a number and not an equation (when entering a linear equation for a resistor, capacitor or inertia, a constant for the initial value of an energy variable, the modulus of a non-modulated transformer or gyrator or a step input at $t = 0$) then this constant is placed in the `initial` [1] section in ACSL, with the ACSL statement `constant` preceding it. By doing this, it is possible to change these values at run-time without the use of the bond graph program (although if these values are changed at run-time, they are not changed in the bond graph data structure). Also the value is not re-initialized at each iteration of the `derivative` section. As an example, consider an inertia connected to bond 46, assigned integral causality and having the constitutive relation $f_{46} = \frac{1}{78.9}p_{46}$, that is, the inertance is 78.9. When prompted to enter `f46` as a function of p46, or just the inertance, if the relation is linear, the user would enter `78.0`. The program would then write two equations, the first would be `constant i46 = 78.9` and the second `f46 = (1.0 / i46) * p46`. The first equation would be placed in the `initial` section.

In addition to entering either a linear coefficient or a non-linear equation, the user is able to describe the relation using a set of discrete data points. ACSL linearly interpolates between the data points and linearly extrapolates on either end to get the continuous form of the relation.

If the user chooses to enter only the data points, the program lists the dependent variables of the element, assuming that it is not modulated. The user is then prompted to enter all the dependent variables. If the element is not modulated, the dependent variables entered will be those displayed. If the element is modulated, the modulating variables and the displayed variables must be entered. ACSL limits the table to having three dependent variables, if there are more than three dependent variables, the user has to fit an equation to the data. Once the user has entered the dependent variables, the user is prompted to enter the number of data points for the first dependent variable. The data points for this variable are then entered. The process is repeated for each of the remaining dependent variables. Finally the user is prompted to enter the data for the independent variable for each combination of dependent variable data. The program automatically puts the data entered in the required ACSL format, breaking the lines and inserting an ellipsis where necessary.

The user is not required to enter any data for the 1 or 0-junctions as the standard equations are written by the program.

4.7.4 Algebraic loops

If an arbitrary causality is assigned to a bond connected to a resistor or a bond between two junction structure elements, an algebraic loop will result.

If all the elements have linear relations, it is possible to algebraically manipulate the equations to remove the algebraic loop, as shown in Section 2.10.2. However, there are some non-linear relations that can't be solved explicitly. Even if the relations are linear, excessive user intervention may be required, which could lead to errors. It was therefore decided to treat

linear and non-linear equations in the same way. The ACSL implicit operator (IMPL), which consists of a Newton-Rapson integration, is used to try to find the solution to the algebraic loop [1]. The disadvantage of the IMPL operator is that it can significantly increase the computing time needed for the simulation.

The general form of an ACSL implicit statement is
`y = impl(yz, e, m, efl, expr, ydl)`

where `yz` is the initial guess.
`e` is the error bound (`e` can be real constant or an arithmetic expression).
`m` is the maximum number of iterations to try (`m` must be an integer constant or variable, not an expression).
`efl` is the error flag (a variable only) which is set non-zero if the iteration does not converge.
`expr` is the expression that contains or eventually leads back to `y`.
`ydl` is the value used to increment the initial value to estimate the derivative.

The program prompts the user to enter the values for `yz`, `e`, `m` and `ydl`. Each of these variables has a constant default value, which the user may use, if the values are satisfactory. The default values assigned to the variables are:

```
yz = 0.0
e = 0.0001
m = 50
ydl = 0.001
```

Numerous tests were done on systems containing algebraic loops and the default constants assigned, resulted in the majority of iterations converging.

Every time the implicit operator is needed, the modelling package increments the error flag counter by one, so that there is a unique error flag variable for each implicit statement in the ACSL input file. That is, the first time the implicit statement is used the error flag variable will be `ef1`, on the second occurrence it will be `ef2` and so on. The user is informed of the variable names assigned to the error flags, so that, during the simulation, the user can check that all the implicit integrations converge.

It may occur, although infrequently, that the implicit integration does not converge and the error flag is set non-zero by ACSL. The user is then be required to change, either the values in the implicit statement, or the structure of the bond graph.

4.7.5 An algebraic loop caused by assigning an arbitrary causality to a resistor

Consider the bond graph in Figure 2.21c, redrawn in Figure 4.8. The causality assignment can be completed, once an arbitrarily causality is assigned to resistor 2 or resistor 5. For a resistor, a relation exists, by definition, between the effort and the flow. Therefore if either the effort or the flow is known, the other can be calculated. In this example, once an arbitrary causality is assigned to one of the resistors, the causality on the other resistor is determined (because of the junction structure constraints). Therefore if any one of effort 2, flow 2, effort 5 or flow 5 is known and assuming that all the other parameters of the circuit are known, the remaining three variables can be calculated. Therefore to solve the algebraic loop, one is required to calculate one of effort 2, flow 2, effort 5 or flow 5 implicitly. For simplicity the

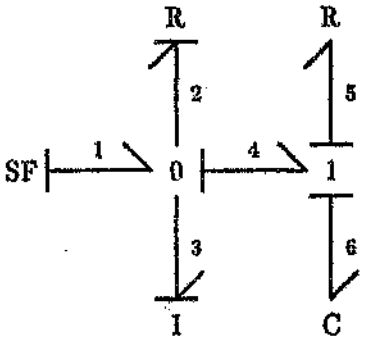


Figure 4.8: A bond graph of a system with an algebraic loop.

program always makes the effort variable on the resistor, to which the an arbitrary causality was assigned, implicit.

Consider the system in Figure 4.8. The linear causal equations, in the form suitable as inputs to ACSL, prior to being altered to account for the algebraic loop, are:

```
f1 = F1
f3 = (1.0 / I3) * p3
p3 = integ (e3, p3i)
e6 = (1.0 / C6) * q6
q6 = integ(f6, q6i)
f2 = (1.0 / R2) * e2
e5 = R5 * f5
f4 = f1 - f2 - f3
e1 = e4
e2 = e4
e3 = e4
e4 = e5 + e6
f5 = f4
f6 = f4
```

Where `F1` is the input function to the system.
`I3` is the inertance.
`p3i` is the value of momentum p3 at time t=0.
`C6` is the capacitance.
`q6i` is the value of displacement q6 at time t=0.
`R2` is the resistance of resistor 2.
`R5` is the resistance of resistor 5.

The numerical values of which would have been entered by the user, but are left in the general form for clarity.

Because there is an algebraic loop and the equations can therefore not be sorted by ACSL, one of the variables must be calculated implicitly. Assuming that the causality was completed after an arbitrary causality was assigned to resistor 2, the program will calculate the effort, `e2` implicitly. The equation `e2 = e4`, is replaced by `e2 = impl(0.0, 0.0001, 50, ef1, e4, 0.001)` (assuming the constant default parameters are used).

The equations of the system, as contained in the ACSL input file, are therefore:

```
f1 = F1
f3 = (1.0 / I3) * p3
p3 = integ (e3, e3i)
e6 = (1.0 / C6) * q6
q6 = integ(i6, q6i)
f2 = (1.0 / R2) * e2
e5 = R5 * f5
f4 = f1 - f2 - f3
e1 = e4
e2 = impl(0.0, 0.0001, 50, ef1, e4, 0.001)
e3 = e4
e4 = e5 + e6
f5 = f4
f6 = f4
```

Which ACSL can sort and simulate.

4.7.6 An algebraic loop caused by assigning an arbitrary causality to a bond

When an algebraic loop is the result of assigning an arbitrary causality to a resistor, only one ACSL implicit statement is required as discussed in the previous section. However, when an algebraic loop is the result of assigning an arbitrary causality to a bond, attached to a junction structure element, there is no immediate relation between the effort and the flow on that bond, as there is when a resistor is involved. It is therefore necessary to include two implicit statements, one for the effort and one for the flow on the bond to which an arbitrary causality was assigned.

As an example of a system where the causality is completed by assigning an arbitrary causality to a bond connected to a junction structure element, consider the unaugmented bond graph in Figure 4.9a. Figure 4.9b shows the bond graph once the flow source has been assigned its required causality and the inertia and capacitor have been assigned their preferred causalities. None of these causalities can be continued through the graph, as the requirements for the three junction structure elements have not yet been met.

As there are no resistors, a bond attached to a junction structure element must be assigned an arbitrary causality. Bond 2 is assigned a causality such that the effort e_2, is the output from the transformer (the input to the 0-junction). The causality can now be continued through the graph, completing the causality assignment. The fully augmented bond graph is shown in Figure 4.9c. Figure 4.9d depicts the fully augmented bond graph of the system had effort e_2, been the input to the transformer (the output from the 0-junction).

Assuming that all the relations are linear, the causal relations of the system in Figure 4.9c, in a form acceptable as ACSL inputs are:

```
f1 = F1
f4 = (1.0 / I4) * p4
p4 = integ(e4, p4i)
e7 = (1.0 / C7) * q7
q7 = integ(f7, q7i)
f3 = m * f2
```

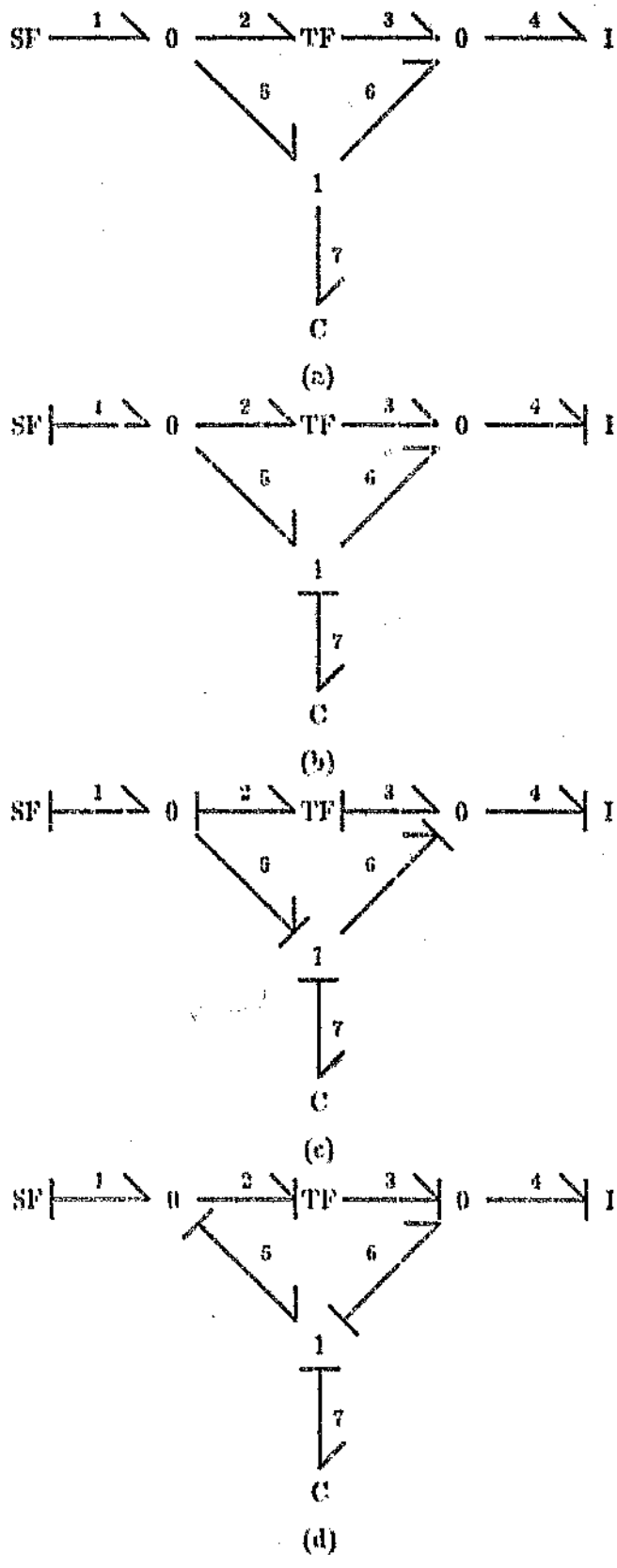


Figure 4.9: A bond graph where the causality is completed once an arbitrary causality is assigned to a bond.

```
e2 = m * e3
f2 = f1 - f5
e1 = e2
e5 = e2
e6 = e5 - e7
f5 = f6
f7 = f6
f6 = -f3 + f4
e3 = e6
e4 = e6
```

Where F1 is the input function to the system.
I4 is the inertance.
p4i is the value of momentum, p4 at time t=0.
C7 is the capacitance.
q7i is the value of displacement, q7 at time t=0.
m is the value of the transformer modulus.

The numerical values of which would have been entered by the user, but are left in the general form for clarity.

As an algebraic loop exists, due to one of the bonds being assigned an arbitrary causality, these equations cannot be sorted by ACSL. It is necessary to calculate two of the variables implicitly. As the causality was completed by assigning an arbitrary causality to bond 2, the program will make effort e2, and flow f2, implicit. The equations `e2 = m * e3` and `f2 = f1 - f5`, will be replaced by `e2 = impl(0.0, 0.0001, 50, ef1, m * e3, 0.001)` and `f2 = impl(0.0, 0.0001, 50, ef2, f1 - f5, 0.001)` respectively (assuming the constant default parameters are chosen).

The equations of the system, as contained in the ACSL input file, are therefore:

```
f1 = F1
f4 = (1.0 / I4) * p4
p4 = integ(e4, p4i)
e7 = (1.0 / C7) * q7
q7 = integ(f7, q7i)
f3 = m * f2
e2 = impl(0.0, 0.0001, 50, ef1, m * e3, 0.001)
f2 = impl(0.0, 0.0001, 50, ef2, f1 - f5, 0.001)
e1 = e2
e5 = e2
e6 = e5 - e7
f5 = f6
f7 = f6
f6 = -f3 + f4
e3 = e6
e4 = e6
```

Which ACSL can sort and simulate.

4.7.7 Derivative causality

When a storage element is not assigned its preferred causality, it is said to have derivative causality, because the causal relationship has to be written in the derivative form.

The method used to form an ACSL input file of a system, with derivative causality, is best illustrated using an example. Consider the system, in Figure 2.22c, redrawn in Figure 4.10. Inertia 2 and capacitor 5 have integral causality assigned to them, but capacitor 6 has derivative causality assigned to it. Capacitor 6 has derivative causality because capacitor 5 is assigned integral causality and both capacitors are connected to the same 0-junction.

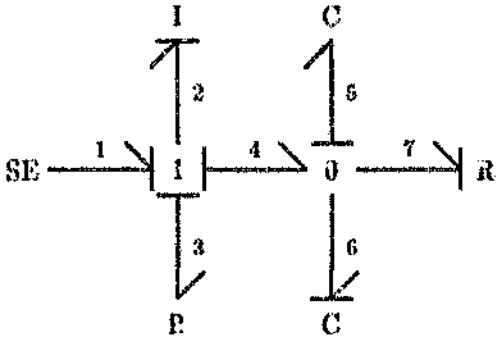


Figure 4.10: The bond graph of a system displaying derivative causality

The linear causal relations before iteration, in ACSL format are:

```
e1 = E1
f2 = (1.0 / I2) * p2
p2 = integ(e2, p2i)
e5 = (1.0 / C5) * q5
q5 = integ(f5, q5i)
q6 = C6 * e6
f6 = derivt(f6i, q6)
e3 = R3 * f3
f7 = (1.0 / R7) * e7
e2 = e1 - e3 - e7
f1 = f2
f3 = f2
f4 = f2
f5 = f4 - f6 - f7
e4 = e5
e6 = e5
e7 = e5
```

Where	E1	is the input function to the system.
	I2	is the inertance.
	p2i	is the value of momentum, p2 at time t=0.
	C5	is the capacitance of capacitor 5.
	q5i	is the value of displacement, q5 at time t=0.
	C6	is the capacitance of capacitor 6.
	f6i	is the initial value of flow, f6.
	R3	is the resistance of resistor 3.
	R7	is the resistance of resistor 7.

The numerical values of which would have been entered by the user, but are left in the general form for clarity.

For each of the two storage elements with integral causality there is an `integ` (ACSL integration) statement, but for the capacitor with derivative causality there is a `derivt` (ACSL derivative) statement, hence the terms integral and derivative causality.

It is important to realize that the ACSL `derivt` operator, should only be included if *absolutely* necessary, as the derivative operator is a first order approximation and can lead to instability if used to represent a major loop [1]. Systems with derivative causality were simulated using the `derivt` operator, but the results seldom corresponded with those expected. It was therefore decided to implement the derivative operator, using some other method. The technique used is as follows.

The derivative term `f6 = derivt(f6i, q6)` is rewritten as `f6 = q6d`, where `d` represents the differential and it is read as either `f6` *is equal to* `q6` *dot*, or `f6` *is equal to the derivative of* `q6`. The program then searches through the list of equations to find an equation with `q6` on the left hand side of the equality. The equation returned is `q6 = C6 * e6`. The aim is then to differentiate both sides of this equation. Differentiating `q6` gives `q6d`, as required and differentiating `C6 * e6`, assuming that `C6` is a constant, gives `C6 * e6d`. The equation `q6d = C6 * e6d` is included in the list of equations. The program then searches through the right hand side of the latest equation, to find any bond graph variables (`e`, `f`, `p` or `q`). The program finds `e6d`. The program then searches through all the equations to find an equation with `e6` on the left hand side. The equation returned is `e6 = e5`. Differentiating both sides gives `e6d = e5d`. This equation is included in the list of equations. The program looks at the right hand side for any bond graph variables and finds `e5d`. The program then searches through the list of equations to find an equation with `e5` on the left hand side. The program returns the equation `e5 = (1.0 / C5) * q5`. Differentiating both sides gives `e5d = (1.0 / C5) * q5d`, assuming that `C5` is a constant. This equation is added to the list of equations. When searching through the right hand side of the latest equation the bond graph variable `q5d` is found. The program then searches through the list of equations to find an equation with `q5` on the left hand side. The equation returned is `q5 = integ(f5, q5i)`. Differentiating both sides we get `q5d = f5`. This equation is included in the list of equations. As an equation for `f5` already exists, the procedure is halted. The equations of the system are now:

```
e1 = F1
f2 = (1.0 / I2) * p2
p2 = integ(e2, p2i)
e5 = (1.0 / C5) * q5
q5 = integ(f5, q5i)
q6 = C6 * e6
```

```
f6 = q6d
e3 = R3 * f3
f7 = (1.0 / R7) * e7
e2 = e1 - e3 - e7
f1 = f2
f3 = f2
f4 = f2
f5 = f4 - f6 - f7
e4 = e5
e6 = e5
e7 = e5
q6d = C6 * e6d
e6d = e5d
e5d = (1.0 / C5) * q5d
q5d = f5
```

In this example flow 6 is related to flow 5 by the ratio of the capacitances, `f6 = (C6 / C5) * f5`, but `f5 = f4 - f6 - f7`, therefore `f5 = f4 - (C6 / C5) * f5 - f7`. That is, `f5` is a function of itself. Therefore, there is an algebraic loop. To remove the algebraic loop it is necessary to include an implicit function, as discussed in the previous section. The question that arises is: Which new equation should be calculated implicitly?

The problem is solved by considering at another example. Consider the fully augmented bond graph in Figure 4.11.

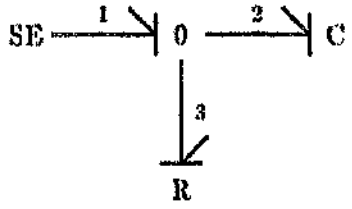


Figure 4.11: A system displaying derivative causality, due to the constraints of the source and junction structure elements.

The capacitor is assigned derivative causality because of the constraints of the source and 0-junction. Assuming all the relations to be linear, the equations of the system (including those equations containing derivative terms) in the ACSL format discussed above are.

```
e1 = E1
f3 = (1.0 / R3) * e3
q2 = C2 * e2
f2 = q2d
f1 = f2 + f3
e2 = e1
e3 = e1
q2d = C2 * e2d
e2d = e1d
e1d = E1d
```

Because `E1` is not a function of any bond graph variables (there is no modulation), `E1d` is known. Therefore in this example there are *no* algebraic loops. This system can be simulated

as it is.

In the first example, the last equation added onto the list of equations was `q5d = f5`. Therefore, for any of the artificial derivatives (`q5d`, `e5d`, `e6d` or `q6d`) to be calculated, `f5` must be known. But `f5 = f4 - f6 - f7` and `f6` is related to `f5` through the artificial derivatives. Therefore `f5` is a function of itself.

In the second example, the last equation added was `e1d = E1d`. Because `E1` is a known function of time, it can be differentiated to give `E1d`, from which `q2d` can be calculated. The second system therefore does not contain an algebraic loop.

A storage element can be assigned derivative causality for two reasons:

1. Due to source and junction structure constraints.
2. Due to another storage element being assigned preferred causality and the junction structure constraints.

The last equation to be differentiated will therefore always be an equation of a source or it will contain an ACSL integration. Therefore if the equation to be differentiated contains an `integ` operator, it should be made implicit.

Returning to the first example, the last term was `q5 = integ(f5, q5i)`, differentiating both sides and making the equation implicit gives `q5d = impl(0.0, 0.0001 50, ef1, f5, 0.001)`, if the default constants are chosen. This equation is added to the list of equations.

The list of equations for the first example are:

```
e1 = E1
f2 = (1.0 / I2) * p2
p2 = integ(e2, p2i)
e5 = (1.0 / C5) * q5
q5 = integ(f5, q5i)
q6 = C6 * e6
f6 = q6d
e3 = R3 * f3
f7 = (1.0 / R7) * e7
e2 = e1 - e3 - e7
f1 = f2
f3 = f2
f4 = f2
f5 = f4 - f6 - f7
e4 = e5
e6 = e5
e7 = e5
q6d = C6 * e6d
e6d = e5d
e5d = (1.0 / C5) * q5d
q5d = impl(0.0, 0.0001, 50, ef1, f5, 0.001)
```

Which ACSL can sort and simulate.

4.8 Running ACSL

The ACSL simulation package is run from the bond graph program using the ACSL file, (*filename*.csl), created by the equation formulation routine, as the input. The file *filename*.csl is the DOS file obtained from the UNIX file *filename*.acsl.

The program does not create a file of ACSL runtime commands (*filename*.cmd). Instead, the runtime execution of ACSL is left to the user. It was decided not to include a file of runtime commands as this would make ACSL less flexible. Because modelling and simulation are iterative processes, the more the user can interact with the simulation package the better. For example, the user may want to simulate the system over a different time interval, view other variables that were perhaps not considered to be important earlier or even change the values of some of the constants assigned earlier.

The ACSL user manual [1], can be consulted for the use of the runtime commands.

When the user completes the simulation, using the ACSL stop command, the user will be returned to the main menu of the bond graph program.

4.9 Program limitations

The limitations of the bond graph program are:

- If the system contains an algebraic loop or derivative causality, implicit integration is used to solve the algebraic loop. There are occasions, especially when the system is highly nonlinear, that the integration does not converge. The user is then required to enter other bond graph elements (usually source or energy storage elements), or combine elements together to remove the algebraic loop or derivative causality. In addition, the inclusion of the implicit operator can increase the simulation time drastically [1].
- As was seen in Section 4.7.7, some of the equations of a system with derivative causality are artificially differentiated, to form the ACSL input file. Because the user is required to differentiate some of the equations, it is not possible to enter data in the form of a table, if the system has derivative causality.
- If a table is used to store the data of a particular function, there can be no more than three dependent variables. This is a limitation of ACSL, not the modelling package. If there are more than three dependent variables, an equation has to be found to represent the data.
- IC-fields cannot be entered into the bond graph program because of the limitations of the automatic causality assignment procedure.

Chapter 5

An application example

In this chapter, the development and simulation of a bond graph model of a coupled tank system will be presented. The entering of the bond graph structure, the assignment of causality, the entering of the equations and the use of ACSL will be shown. However, these processe will not be shown in full as they are repetitive. This example, together with the three complete examples presented in Appendix B, illustrate the capabilities of the bond graph program.

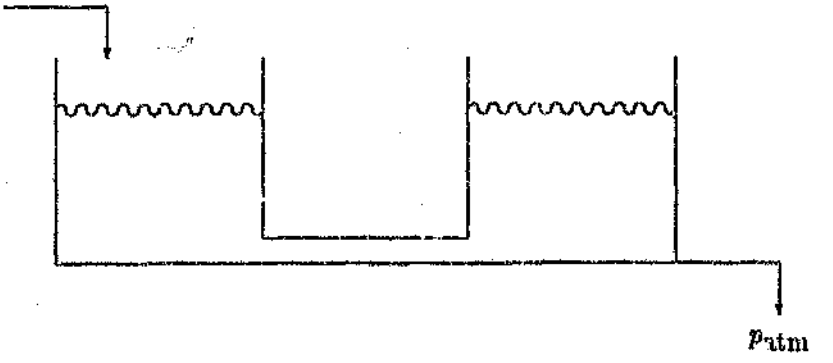
The system to be modelled is the parallel connected tank system, presented in Section 3.6.5. The system is redrawn in Figure 5.1a. The bond graph structure, developed in Section 3.6.5, is shown in Figure 5.1b. As yet, neither the system in Figure 5.1a nor the bond graph in Figure 5.1b have been labelled.

When numbering the bond graph, it is necessary to uniquely number each element and each bond, as discussed in Section 4.3. In Section 3.6.5, the hydraulic and thermal bonds are assigned the same numbers. This is not possible for use with the bond graph program, because the program uses the generalized variables effort and flow and not the variables specific to each domain. In addition, entering the bond graph structure, from the keyboard is simplified by numbering each bond uniquely.

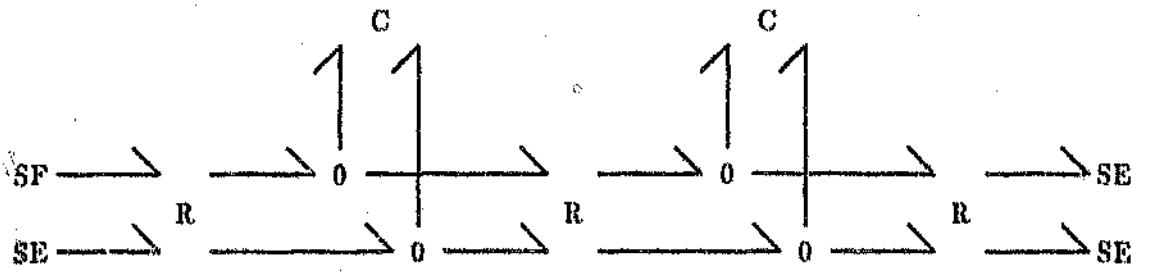
The fully numbered bond graph, in the form required by the bond graph program, is shown in Figure 5.2. Figure 5.3 shows the two tank system, labelled to correspond with the bond graph in Figure 5.2.

Bonds 3 and 5 are entered into the bond graph program as follows. Two fonts are used to distinguish between the program output and the user input. The program output is written using the typewriter font and the user input is written using the Sans Serif font. It is assumed that bonds 1, 2 and 4 have already been entered.

```
Enter the bond number
3

Enter the number of the element at the start of the power half arrow
3

This element has been previously defined

Enter the number of the element at the end of the power half arrow
5

Enter the element type
0
```



(a)



(b)

Figure 5.1: Two parallel connected tanks.

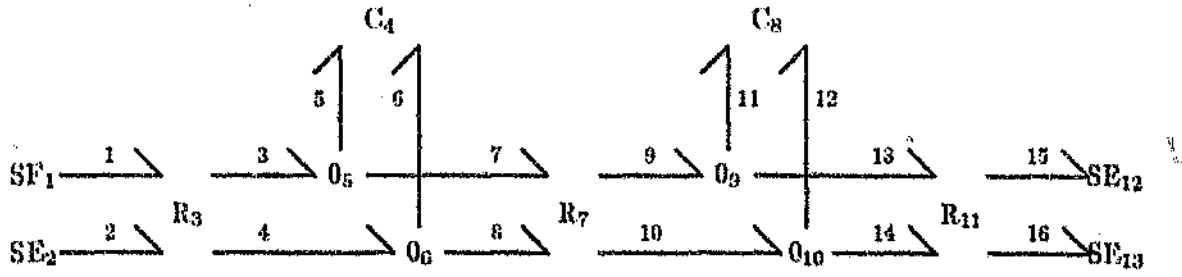


Figure 5.2: The fully numbered bond graph.

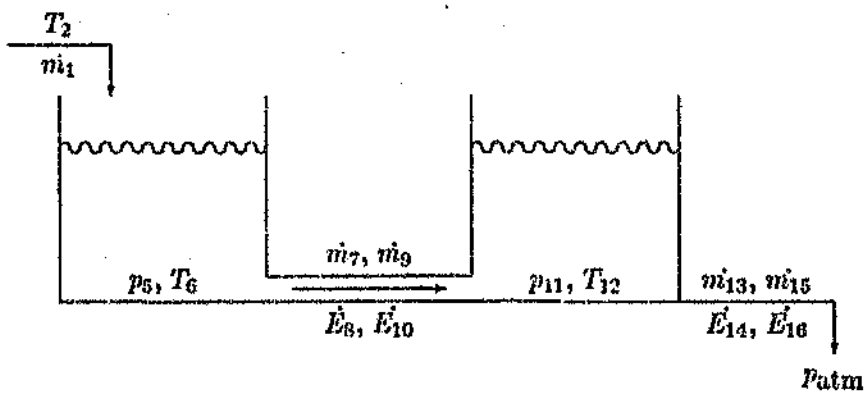


Figure 5.3: Labelled two-tank system.

```
Enter the documentation string for this element
0-junction representing the conservation of mass in the left hand tank

Enter F to finish or any other key to continue
c

Enter the bond number
5

Enter the number of the element at the start of the power half arrow
5

This element has been previously defined

Enter the number of the element at the end of the power half arrow
4

Enter the element type
c

Enter the documentation string for this element
C-field representing the storage of mass and energy in the left hand tank

Enter F to finish or any other key to continue
c
```

The remaining bonds are entered in a similar manner.

Causality is automatically assigned to the bond graph. The causal implications, as deduced by the program, are

```
EFFORT SOURCE, (ELEMENT NUMBER 13) CONNECTED TO
BOND 16

The causality assigned implies that e16 is the output

EFFORT SOURCE (ELEMENT NUMBER 12) CONNECTED TO
BOND 15
```

```
The causality assigned implies that e15 is the output

RESISTOR (ELEMENT NUMBER 11) CONNECTED TO
BOND 13 BOND 14 BOND 15 BOND 16

The causality assigned implies that
f13 is a function of e13 e14 e15 e16
f14 is a function of e13 e14 e15 e16
f15 is a function of e13 e14 e15 e16
f16 is a function of e13 e14 e15 e16

CAPACITOR (ELEMENT NUMBER 8) CONNECTED TO
BOND 11 BOND 12

The causality assigned implies that
e11 is a function of q11 q12
e12 is a function of q11 q12

RESISTOR (ELEMENT NUMBER 7) CONNECTED TO
BOND 7 BOND 8 BOND 9 BOND 10

The causality assigned implies that
f7 is a function of e7 e8 e9 e10
f8 is a function of e7 e8 e9 e10
f9 is a function of e7 e8 e9 e10
f10 is a function of e7 e8 e9 e10

CAPACITOR (ELEMENT NUMBER 4) CONNECTED TO
BOND 5 BOND 6

The causality assigned implies that
e5 is a function of q5 q6
e6 is a function of q5 q6

EFFORT SOURCE (ELEMENT NUMBER 2) CONNECTED TO
BOND 2

The causality assigned implies that e2 is the output

RESISTOR (ELEMENT NUMBER 3) CONNECTED TO
BOND 1 BOND 2 BOND 3 BOND 4

The causality assigned implies that
e1 is a function of f1 e2 e3 e4
f2 is a function of f1 e2 e3 e4
f3 is a function of f1 e2 e3 e4
f4 is a function of f1 e2 e3 e4

FLOW SOURCE (ELEMENT NUMBER 1) CONNECTED TO
BOND 1
```

The causality assigned implies that f1 is the output

Based on the causal implications or the bond graph structure, which can also be obtained from the program, the fully augmented bond graph, shown in Figure 5.4, is obtained.

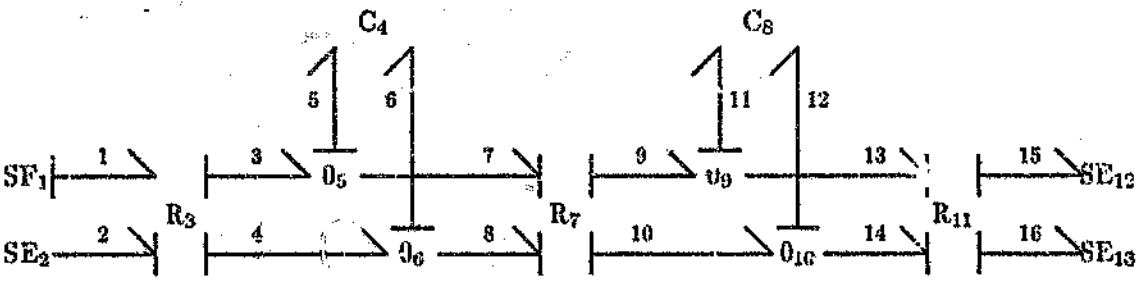


Figure 5.4: The fully augmented bond graph.

If the mass flow rate of the fluid flowing into the left hand tank is a known function $\dot{M}(t)$ and the temperature of the fluid flowing into the left hand tank is a known function $T(t)$, then the equations for each bond graph element, using the information in Section 3.5 and noting the causality shown in Figure 5.4 are:

The flow source connected to bond 1

$$\dot{m}_1 = \dot{M}(t)$$

The effort source connected to bond 2

$$T_2 = T(t)$$

The R-field connected to bonds 1, 2, 3 and 4

$$p_1 = p_3$$

$$\dot{m}_3 = \dot{m}_1$$

$$\dot{E}_2 = c\dot{m}_1 T_2$$

$$\dot{E}_4 = c\dot{m}_1 T_2$$

It must be noted that p_1 is not necessarily equal to p_3, but depends on the position of the pipe through which the fluid enters the tank.

The C-field connected to bonds 5 and 6

$$v_5 = \frac{g}{A_1} m_5$$

$$T_6 = \frac{1}{c} \frac{E_6}{m_5}$$

The R-field connected to bonds 7, 8, 9 and 10

$$\dot{m}_7 = \mathrm{sgn}(p_7 - p_9)K\,|p_7 - p_9|^{\frac{1}{2}}$$

$$\dot{m}_9 = \mathrm{sgn}(p_7 - p_9)K\,|p_7 - p_9|^{\frac{1}{2}}$$

$$\dot{E}_8 = \frac{\mathrm{sgn}(p_7 - p_9)cK\,|p_7 - p_9|^{\frac{1}{2}}}{2}(T_8 + T_{10}) + \frac{cK\,|p_7 - p_9|^{\frac{1}{2}}}{2}(T_8 - T_{10})$$

$$\dot{E}_{10} = \frac{\mathrm{sgn}(p_7 - p_9)cK\,|p_7 - p_9|^{\frac{1}{2}}}{2}(T_8 + T_{10}) + \frac{cK\,|p_7 - p_9|^{\frac{1}{2}}}{2}(T_8 - T_{10})$$

The C-field connected to bonds 11 and 12

$$p_{11} = \frac{q}{A_2}m_{11}$$

$$T_{12} = \frac{1}{c}\frac{E_{12}}{m_{11}}$$

The R-field connected to bonds 13, 14, 15 and 16

$$\dot{m}_{13} = Kp_{13}^{\frac{1}{2}}$$

$$\dot{m}_{15} = Kp_{13}^{\frac{1}{2}}$$

$$\dot{E}_{14} = cKp_{13}^{\frac{1}{2}}T_{14}$$

$$\dot{E}_{16} = cKp_{13}^{\frac{1}{2}}T_{14}$$

The effort source connected to bond 15

$$p_{15} = 0 \quad \text{(atmospheric pressure)}$$

The flow source connected to bond 16

$$T_{16} = T_{\text{ambient}}$$

where c is the specific heat.
q is the acceleration due to gravity.
A_1 is the area of the left hand tank.
A_2 is the area of the right hand tank.
K is a constant for the pipe, relating the mass flow rate to the square root of the pressure difference. The relation for K will be given later.

As the bond graph modelling package can only accept the generalized bond graph variables, namely e, f, q and p, all the variables above must be replaced with an e, if they are effort variables, an f if they are flow variables, a q if they are displacement variables or a p if they are momentum variables.

The equations for the system in a form compatible with the bond graph modelling package are therefore

The flow source connected to bond 1

$$f_1 = \dot{M}$$

The effort source connected to bond 2

$$e_2 = T$$

The R-field connected to bonds 1, 2, 3 and 4

$$e_1 = e_3$$

$$f_3 = f_1$$

$$f_2 = ef_1e_2$$

$$f_4 = ef_1e_2$$

The C-field connected to bonds 5 and 6

$$e_5 = \frac{g}{A_1}q_5$$

$$e_6 = \frac{1}{e}\frac{q_6}{q_5}$$

The R-field connected to bonds 7, 8, 9 and 10

$$f_7 = \mathrm{sgn}(e_7 - e_9)K\,|e_7 - e_9|^{\frac{1}{2}}$$

$$f_9 = \mathrm{sgn}(e_7 - e_9)K\,|e_7 - e_9|^{\frac{1}{2}}$$

$$f_8 = \frac{\mathrm{sgn}(e_7 - e_9)eK\,|e_7 - e_9|^{\frac{1}{2}}}{2}(e_8 + e_{10}) + \frac{eK\,|e_7 - e_9|^{\frac{1}{2}}}{2}(e_8 - e_{10})$$

$$f_{10} = \frac{\mathrm{sgn}(e_7 - e_9)eK\,|e_7 - e_9|^{\frac{1}{2}}}{2}(e_8 + e_{10}) + \frac{eK\,|e_7 - e_9|^{\frac{1}{2}}}{2}(e_8 - e_{10})$$

The C-field connected to bonds 11 and 12

$$e_{11} = \frac{g}{A_2}q_{11}$$

$$e_{12} = \frac{1}{e}\frac{q_{12}}{q_{11}}$$

The R-field connected to bonds 13, 14, 15 and 16

$$f_{13} = Ke_{13}^{\frac{1}{2}}$$

$$f_{15} = Ke_{13}^{\frac{1}{2}}$$

$$f_{14} = eKe_{13}^{\frac{1}{2}}e_{14}$$

$$f_{16} = eKe_{13}^{\frac{1}{2}}e_{14}$$

The effort source connected to bond 15

$$e_{15} = 0$$

The flow source connected to bond 16

$$e_{16} = T_{\text{ambient}}$$

The values of the constants, required for the simulation are obtained by considering the details of the system.

It is assumed that the fluid is water and the density $\rho = 1000\text{kg m}^{-3}$. The specific heat is assumed equal to the specific heat at constant pressure and independent of temperature. The specific heat at constant pressure is $c_p = 4180\text{J kg}^{-1}\text{K}^{-1}$ [12].

The area of the left hand tank is known to be $A_1 = 0.8\text{m}^2$ and the area of the tank on the right is known to be $A_2 = 1.8\text{m}^2$.

The acceleration due to gravity is assumed constant and equal to $g = 9.81\text{m s}^{-2}$.

The flow rate of fluid from a small orifice into the atmosphere is defined [7], as

$$q = C_d A_p \sqrt{2gh}$$

where C_d is the coefficient of discharge and approximately equal to 0.621 [7].
A_p is the area of the orifice.
h is the height of the fluid above the orifice.

Multiplying both sides by the density, to get a relation for the mass flow rate

$$\dot{m} = \rho C_d A_p \sqrt{2gh}$$

but the pressure p at the orifice is ρgh, therefore

$$\dot{m} = C_d A_p \sqrt{2\rho p}$$

The constant, K, relating the mass flow rate to the square root of the pressure difference is therefore

$$K = C_d A_p \sqrt{2\rho}$$

It is assumed that the pipe is sufficiently short, so that the above relations apply, not only for an orifice, but for the pipe itself. The area of the pipe is known to be $A_p = 707 \times 10^{-6}\text{m}^2$, corresponding to a diameter of 30mm. The constant K is therefore $K = 19.6 \times 10^{-3}\text{kg}^{\frac{1}{2}}\text{m}^{\frac{1}{2}}$.

It is known that the initial height of the water in tank 1 is 0.25m and the initial height of the water in tank 2 is 0.5m. The temperature of the water in both tanks is the same and equal to the ambient temperature of 300K. Therefore

1. The initial mass of water in tank 1 is $q_{5_i} = 200\text{kg}$.
2. The initial mass of water in tank 2 is $q_{11_i} = 900\text{kg}$.
3. The initial energy stored in tank 1 is $q_{6_i} = 250.8 \times 10^6\text{J}$.
4. The initial energy stored in tank 2 is $q_{12_i} = 1.129 \times 10^9\text{J}$.

At time $t = 0$, the water flowing into tank 1, $f_1 = 1\text{kgs}^{-1}$. The water entering tank 1 is at a temperature of 323K.

The equations of the flow source connected to bond 1, the capacitor connected to bonds 5 and 6 and the resistor connected to bonds 7, 8, 9 and 10 are entered as follows

```
FLOW SOURCE (ELEMENT NUMBER 1) CONNECTED TO
BOND 1
```

```
The documentation for this element is:
"Mass flow rate into left hand tank"

Enter the data for flow source SF1.
If the flow input is a step at t=0 then enter only the height of the step.
If the flow input is anything but a step at t=0 then enter the entire
equation, using the form f1=.....
If the data is in the form of a table, enter "table"
1.0

CAPACITOR (ELEMENT NUMBER 4) CONNECTED TO
BOND 5 BOND 6

The documentation for this element is:
"C-field representing the storage of mass and energy in the left hand tank"

Enter e5 as a function of q5 q6
If the data is in the form of a table, enter "table"
e5=9.81/0.8*q5

Enter the value of q5 at time t=0
200.0

Enter e6 as a function of q5 q6
If the data i  in the form of a table, enter "table"
e6=1.0/4180*q6/q5

Enter the value of q6 at time t=0
250.8e6

RESISTOR (ELEMENT NUMBER 7) CONNECTED TO
BOND 7 BOND 8 BOND 9 BOND 10

The documentation for this element is:
"R-field used to calculate the mass flow rate due to the pressure difference
and the energy flow rate due to the temperature and mass flow rate"

Enter f7 as a function of e7 e8 e9 e10
If the data is in the form of a table enter "table"
f7=sign(1.0,e7-e9)*19.6e-3*sqrt(abs(e7-e9))

Enter f8 as a function of e7 e8 e9 e10
If the data is in the form of a table enter "table"
f8=0.5*4180*sign(1.0,e7-e9)*19.6e-3*sqrt(abs(e7-e9))*(e8+e10)+0.5*4180*19.6e-3* sqrt(abs(e7-
e9))*(e8-e10)

Enter f9 as a function of e7 e8 e9 e10
If the data is in the form of a table enter "table"
f9=sign(1.0,e7-e9)*19.6e-3*sqrt(abs(e7-e9))

Enter f10 as a function of e7 e8 e9 e10
If the data is in the form of a table enter "table"
f10=0.5*4180*sign(1.0,e7-e9)*19.6e-3*sqrt(abs(e7-e9))*(e8+e10)+0.5*4180*19.6e-3*sqrt(abs(e7-
e9))*(e8-e10)
```

The ACSL input file (two-tank.acsl) is created by the bond graph program from the bond graph structure, stored in the file two-tank.data and the equations entered by the user. The UNIX file two-tank.acsl is converted to the DOS file two-tank.csl. The ACSL input file is:

```
program Two parallel connected tanks
initial
constant points=1000
constant ccalc=5.0
constant simtim=5000
simend=simtim-ccalc
cint=simtim/points
nstp=cint/ccalc
constant f1=1.0
constant e2=323.0
constant q6i=250.8e6
constant q5i=200.0
constant q12i=1.129e9
constant q11i=900.0
constart e15=0.0
constant e16=300.0
end $"initial"
dynamic
derivative
f4=4180*f1*e2
f3=f1
f2=4180*f1*e2
e1=e3
e7=e5
e3=e5
f5=-f7+f3
e8=e6
e4=e6
f6=-f8+f4
e6=1.0/4180*q6/q5
q6=integ(f6,q6i)
e5=9.81/0.8*q5
q5=integ(f5,q5i)
f10=0.5*4180*sign(1.0,e7-e9)*19.6e-3*sqrt(abs(e7-e9))*(e8+e10)+
0.5*4180*19.6e-3*sqrt(abs(e7-e9))*(e8-e10)
f9=sign(1.6,e7-e9)*19.6e-3*sqrt(abs(e7-e9))
f8=0.5*4180*sign(1.0,e7-e9)*19.6e-3*sqrt(abs(e7-e9))*(e8+e10)+
0.5*4180*19.6e-3*sqrt(abs(e7-e9))*(e3-e10)
f7=sign(1.0,e7-e9)*19.6e-3*sqrt(abs(e7-e9))
e13=e11
e9=e11
f11=-f13+f9
e14=e12
e10=e12
f12=-f14+f10
e12=1.0/4180*q12/q11
q12=integ(f12,q12i)
e11=9.81/1.8*q11
q11=integ(f11,q11i)
f16=4180*19.6e-3*sqrt(e13)*e14
```

```
f15=19.6e-3*sqrt(e13)
f14=4180*19.6e-3*sqrt(e13)*e14
f13=19.6e-3*sqrt(e13)
termt(t.gt.simend)
end $"derivative"
end $"dynamic"
terminal
end $"terminal"
end $"program"
```

The equations for f8 and f10 above, span two lines. This has been done for clarity only, as in the ACSL input file, each equation is written in a single line.

The coupled tank system is simulated using ACSL.

The pressure at the bottom of tank 1, varies with time as shown in Figure 5.5.

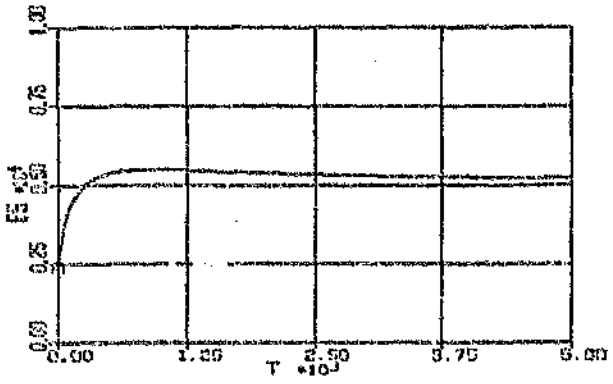


Figure 5.5: The pressure at the bottom of tank 1, e5 as a function of time.

The pressure at the bottom of tank 2, varies with time as shown in Figure 5.6.

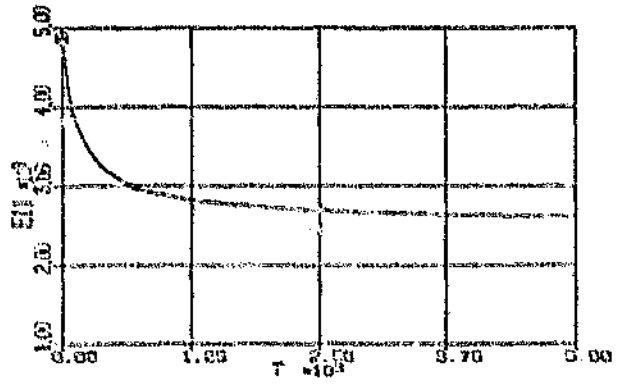


Figure 5.6: The pressure at the bottom of tank 2, e11 as a function of time.

The temperature of the water in tank 1, varies with time as shown in Figure 5.7.

The temperature of the water in tank 2, varies with time as shown in Figure 5.8.

The mass flow rate of the water leaving tank 1, varies with time as shown in Figure 5.9.

The energy flow rate leaving tank 1, varies with time as shown in Figure 5.10.

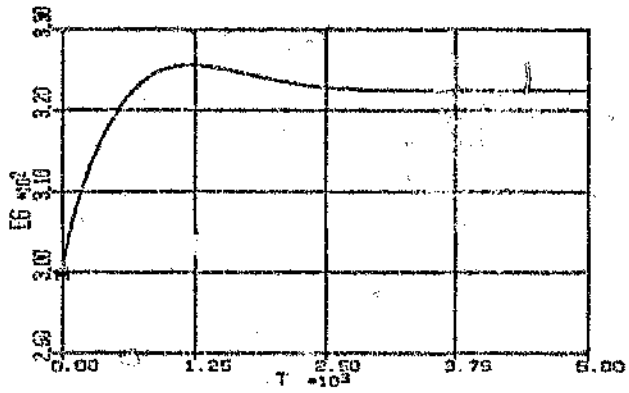


Figure 5.7: The temperature of the water in tank 1, e6 as a function of time.

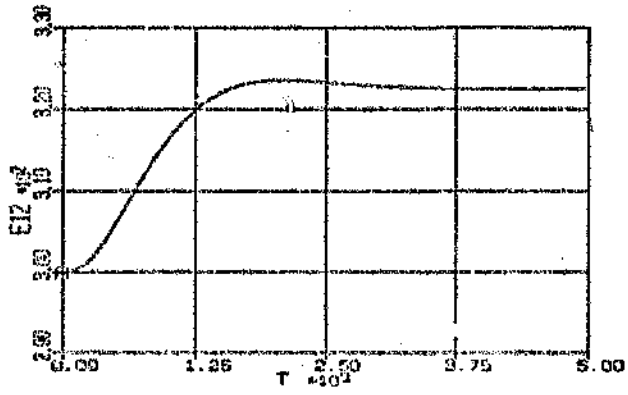


Figure 5.8: The temperature of the water in tank 2, e12 as a function of time.

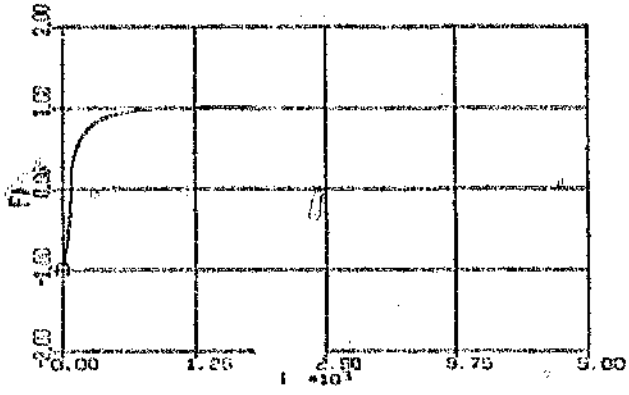


Figure 5.9: The mass flow rate of the water leaving tank 1, f7 as a function of time.

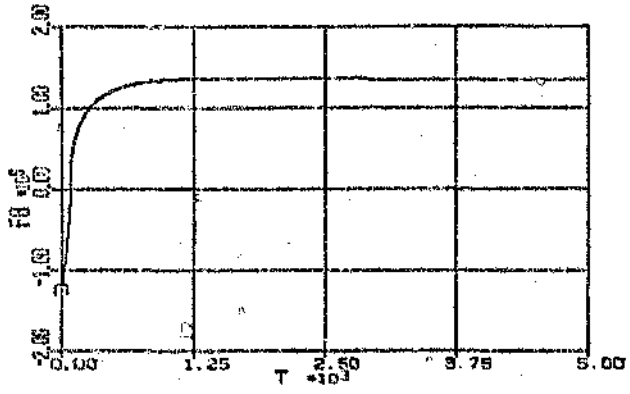


Figure 5.10: The energy flow rate between the two tanks, f8 as a function of time.

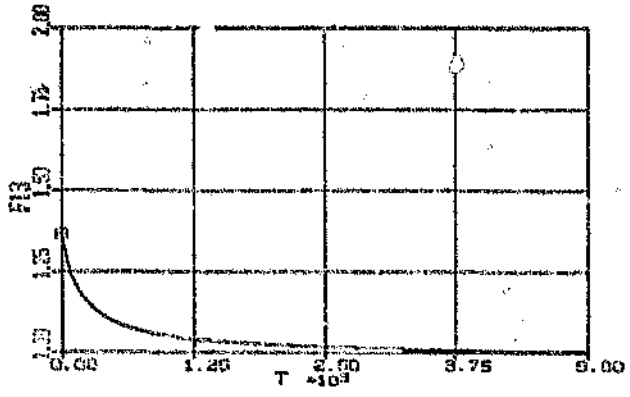


Figure 5.11: The mass flow rate of the water between the two tanks, f13 as a function of time.

The mass flow rate of the water leaving tank 2, varies with time as shown in Figure 5.11.

The energy flow rate leaving tank 2, varies with time as shown in Figure 5.12.

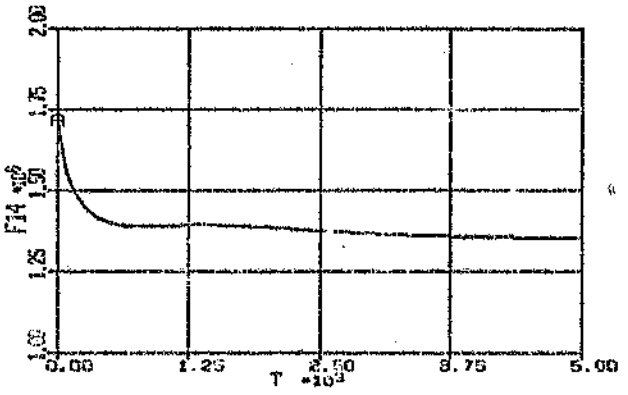


Figure 5.12: The energy flow rate leaving tank 2, f14 as a function of time.

Chapter 6

Causality as a modelling tool

Together with the bond graph structure, the causality plays a major role in describing the qualitative behaviour of the elements of a system and the behaviour of the system as a whole. In this chapter, an example will be presented to show how a knowledge of the causality, prior to writing the equations, can assist in obtaining the bond graph model. Consider a system consisting of a 4-way hydraulic valve connected to ram with an inertial load, as shown in Figure 6.1.

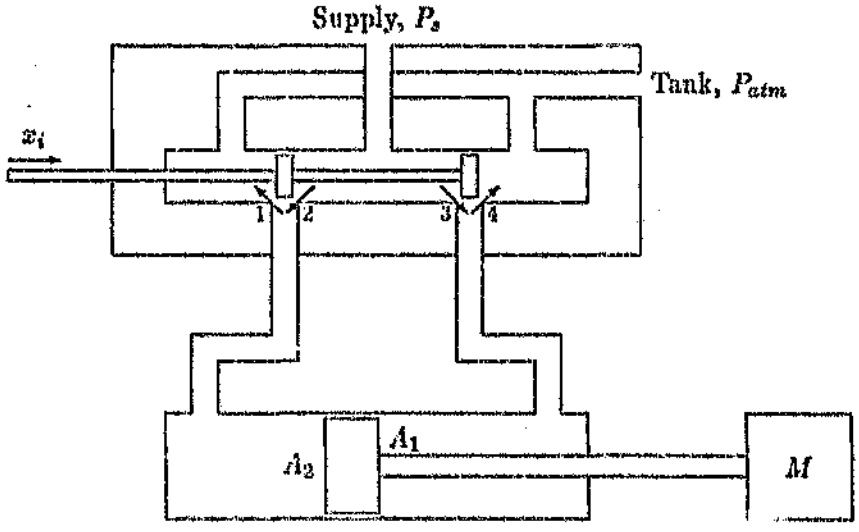


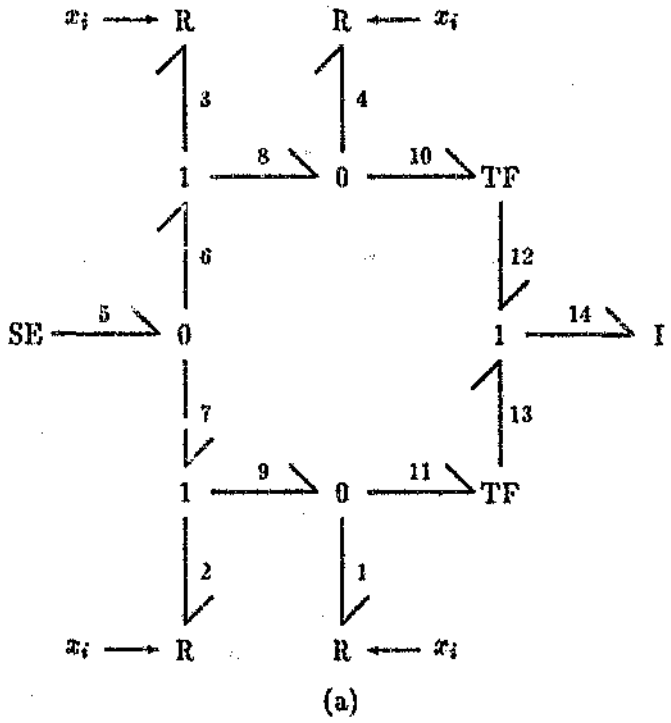
Figure 6.1: A system consisting of a 4-way hydraulic valve, a ram and a mass.

The acausal bond graph, assuming the fluid to be incompressible, is shown in Figure 6.2a.

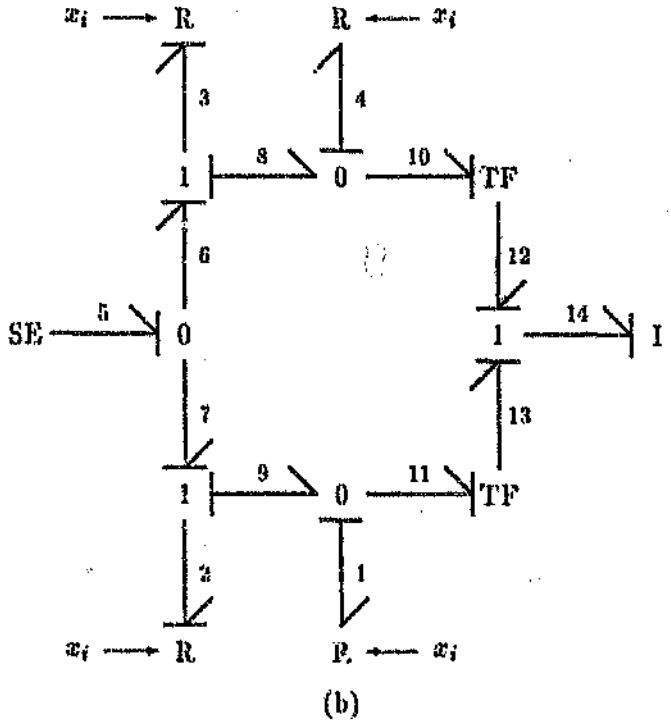
The bond graph is interpreted as follows.

The effort source represents the fluid being supplied to the valve at a known pressure, P_s. The tank pressure is not included in the bond graph, because it is assumed that the fluid leaving the valve exits into the atmosphere (reference pressure).

The leftmost 0-junction constrains the pressures at entrance to orifices 2 and 3, to be equal to the supply pressure. Resistors 1, 2, 3 and 4 represent the fluid flow (the relationship between the volumetric flow rate and the pressure difference) at the positions labelled 1, 2, 3 and 4 in Figure 6.1 respectively. The modulating lines indicate that the flow rate is a function of the displacement of the valve rod, x_i.



(a)



(b)

Figure 6.2: The bond graph of a system consisting of a 4-way hydraulic valve, a ram and a mass.

The 1-junction connected to bond 2 constrains flow rate 2 to be a function of the difference between the supply pressure and the pressure in the cylinder. Likewise for the 1-junction connected to bond 3. The 0-junction connected to bond 4 constrains the flow rate between the cylinder and the tank to be a function of the pressure in the cylinder only (the tank is assumed to be at zero pressure). Likewise for the 0-junction attached to bond 1.

The two transformers represent the conversion of the fluid energy to mechanical energy on either side of the piston. Because the piston is a solid body, the velocity of the one surface is equal to the velocity of the other. This is the interpretation of the 1-junction attached to bonds 12, 13 and 14.

The 1-junction could have been connected directly to the two 0-junctions and a single transformer connected between the 0-junction and the inertia. The 1-junction would then constrain the flow rate of the fluid entering the one side of the cylinder to be equal to the flow rate of the fluid leaving the other side. The transformer would represent the conversion of the net fluid energy to mechanical energy. Using this configuration, an assumption about the piston area would have to be made.

The inertia represents the mass attached to the rod.

The constitutive relations of the bond graph elements are:

Effort source

$$p_5 = P_s$$

Resistor 1

$$q_1 = \begin{cases} K_1(x_o + x_i)p_1^{\frac{1}{2}} & x_i > -x_o \\ 0 & \text{otherwise} \end{cases}$$

Resistor 2

$$q_2 = \begin{cases} K_2(x_o - x_i)p_2^{\frac{1}{2}} & x_i < x_o \\ 0 & \text{otherwise} \end{cases}$$

Resistor 3

$$q_3 = \begin{cases} K_3(x_o + x_i)p_3^{\frac{1}{2}} & x_i > -x_o \\ 0 & \text{otherwise} \end{cases}$$

Resistor 4

$$q_4 = \begin{cases} K_4(x_o - x_i)p_4^{\frac{1}{2}} & x_i < x_o \\ 0 & \text{otherwise} \end{cases}$$

Transformer connected to bonds 10 and 12

$$F_{12} = A_1 p_{10}$$

$$q_{10} = A_1 v_{12}$$

Transformer connected to bonds 11 and 13

$$F_{13} = A_2 p_{11}$$

$$q_{11} = A_2 v_{13}$$

Inertia

$$v = \frac{1}{M}p$$

Where	P_s	is the supply pressure.
	K_1 to K_4	are orifice flow constants [2].
	x_o	is the valve underlap, assumed equal for all four orifices [2].
	x_i	is the position of the ram. $x_i = 0$ in central position.
	A_1	is the area of the right side of the piston.
	A_2	is the area of the left side of the piston.
	M	is the mass of the load.

Consider the four resistors. if the constitutive relations are written with the pressure as a function of the flow rate, the relations hold only when a flow exists. When the flow rate is zero, the pressure cannot be calculated. Therefore, the flow must be the output from all the resistors (conductance causality).

Assigning causality to the bond graph in Figure 6.2a gives the bond graph in Figure 6.2b. Two algebraic loops exist, if resistor 3 is assigned conductance causality, resistor 4 is forced to take on resistance causality. Likewise for resistors 2 and 1. For resistors 1 and 4 to be assigned conductance causality, additional elements must be added to the bond graph.

If the incompressibility assumption is removed, the system can be represented by the bond graph in Figure 6.3a. The two capacitors represent the compressibility of the fluid on either side of the piston.

The capacitors have the relations

Capacitor 17

$$p_{17} = \frac{1}{C_1}V_{17}$$

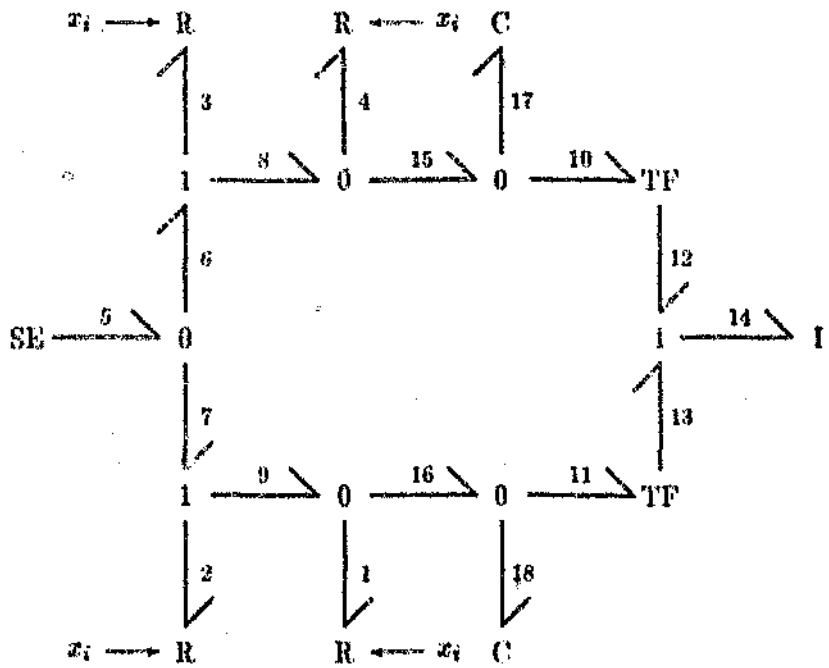
Capacitor 18

$$p_{18} = \frac{1}{C_2}V_{18}$$

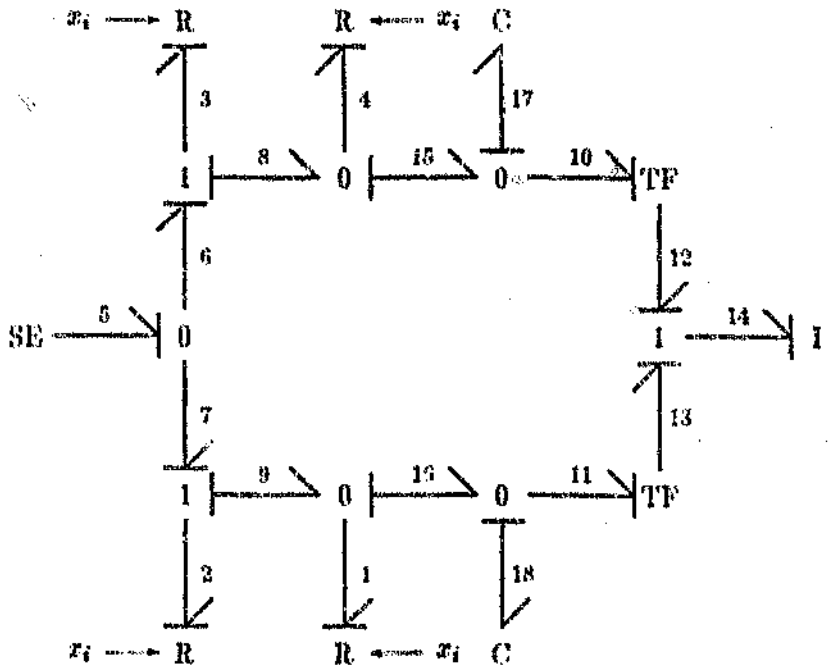
where	C_1 and C_2	are the ratio of the mean volumes to the bulk modulus. Instead of using the mean volumes, the displacement of the rod could be taken into account to get the actual volumes, this would require modulating the two capacitors by the derivative of the velocity of the load.
	V_{17}	is the volume of the fluid to the left of the piston.
	V_{18}	is the volume of the fluid to the right of the piston.

Figure 6.3b, shows the bond graph after assigning the causality. Taking the compressibility into account, ensures that the four resistors are all assigned conductance causality, as required.

As can be seen from this example, had the modeller assumed that the fluid was incompressible and was not aware of causal implications, it would not be possible to obtain the equations of state for all values of x_i. Also, there would be no clear indication of the reasons for this modelling error.



(a)



(b)

Figure 6.3: The bond graph of a system consisting of a 4-way hydraulic valve, a ram and a mass, taking the compressibility into account.

The causality displayed by the bond graph, which is obtained from the structure, gives the reasons for a system's behaviour. As an example of this, consider the bond graph in Figure 6.3b. The causality on bond 14 shows that a force is applied to the mass and the mass therefore moves at a certain velocity. The force is due to the pressure on the piston, which arises because of the compressibility of the fluid. This is shown by the causality on bonds 17, 10, 12, 18, 11 and 13 and the bond graph elements to which these bonds are attached. The amount of fluid in the piston depends on the difference between the mass flow rate of the fluid next to the piston and the mass flow rate of the fluid leaving or entering the valve. This is shown by the causality on bonds 15, 17, 10, 16, 18 and 11 and the 0-junctions to which the bonds are attached. The fluid leaving the piston is a function of the pressure (due to compressibility) as shown by the causality on bonds 15 and 16. The fluid entering the piston is a function of the difference between the supply pressure and the pressure on the piston side of the valve (due to the compressibility). This is shown by the causality on bond 5, 6, 3, 8, 7, 2 and 3 and the 0 and 1-junctions to which these bonds are connected.

Because the bond graph consists of a number of elements, each having a particular meaning and because the bond graph displays the cause-effect relationships, it constitutes a powerful modelling tool.

Chapter 7

Conclusions

This dissertation has shown that the bond graph is a powerful multidomain model representation. In addition to being able to obtain the state-space equations from the bond graph, the reasons for a system's behaviour can be obtained from the causality and the implication of each bond graph element.

It has also been shown that the bond graph is particularly well suited as a model representation for a computer aided modelling package. Such a package allows rapid model construction, validation and modification.

7.1 Bond graph theory

The bond graph is a multidomain modelling technique, which uses symbols to characterize the behaviour of a system. Because the bond graph is a multidomain representation, analogical reasoning plays an important role in deriving and understanding the model. That is, two systems in different domains can have identical bond graphs and will therefore behave in the same way. If the behaviour of one of the systems is well understood, the behaviour of the other system can be explained in terms of it. This allows the modeller to construct dynamic models of systems in domains which are not familiar.

Because there are a limited number of different bond graph elements, the implication of each element is well understood. If a number of bond graph elements are connected together, to form a bond graph fragment, its behaviour can be explained from an understanding of each of the elements making up the fragment. If all the fragments are combined to form the bond graph model of the system, the behaviour can be explained in terms of the fragments for which the behaviour is understood. Therefore, the bond graph is a tool used to understand the reasons for a system's behaviour.

When modelling a system using a bond graph, the bond graph elements represent the behaviour of a particular part of a system. A combination of these elements, forming the bond graph, represents the behaviour of the system as a whole. As a bond graph is constructed from the behaviour of the individual parts of a system, one may be inclined to think that the bond graph modelling technique is incompatible with modelling for the purposes of controller design (where the model is not usually very complex and is not necessarily related to the structure of the plant). It must be noted, however, that the individual part, referred to, does not have to be the smallest part of the system, but can be one of the major subsystems, or for that matter, the entire system. This implies that a large subsystem can sometimes be

represented by a single bond graph element or a few bond graph elements connected together, depending on the accuracy required. Therefore, a bond graph model can be made as simple or as complex as desired.

To the author's knowledge, the bond graph, is the only modelling technique to display the causality prior to writing the equations. As shown in Section 6, a knowledge of the causality constitutes a powerful tool. Together with the structure, the causality is used to understand the qualitative behaviour of the system.

The bond graph can be converted to a block diagram (which is the most common representation of a control system) or to a signal flow graph (which is primarily a technique for representing linear systems). But, because each bond has associated with it two variables, an effort and a flow, the bond graph is the most compact of the three representations. The block diagram and the signal flow graph are both causal, but unless they have been formed from a bond graph, the causality is due to the order of the blocks or nodes and does not depend on the structure of the system. Which is the case for the bond graph.

Unlike block diagrams and signal flow graphs, bond graphs do not display the feedback present in many systems. A tank with fluid entering at the top and leaving through the bottom is an example of a self-regulating system. That is, it is a system containing negative feedback. The fluid flow rate into the atmosphere is a function of the height of the fluid in the tank, which is a function of the difference between the inflow and outflow rates. Feedback is shown by block diagrams and signal flow graphs because they depict only signal flow, whereas the bond graphs shows the *energy* interactions. The feedback in the bond graph is modelled by 0 and 1-junctions and in some cases R-fields, together with the direction of power flow. If a bond graph is converted into a block diagram, the signal feedback will be evident.

7.1.1 The use of pseudo bond graphs for modelling compressible thermofluid systems

The pseudo bond graph representation of a compressible thermofluid system is valuable, because, although the constitutive relations of many of the elements, particularly the resistors connecting the fluid lumps, are extremely complex, the bond graph representation is uncluttered and clearly depicts the system components and their interactions.

Although the pseudo bond graph model does not represent the flow of power, the variables used, are those traditionally used for modelling control volume and heat transfer systems. However, by not making the effort and flow variables complimentary power variables, new types of elements, often containing modulating lines and block diagram segments, are required to connected the pseudo bond graph to a true bond graph. This is necessary to ensure that the energy is conserved. Although these additional elements are required, they are usualy very similar. Apart from making the bond graph more complex, the elements do not normally pose much of a problem. Also, by not using complimentary power variables, the Maxwell symmetry check for true C-fields is not available and it is easier to assume constitutive laws which may not be physically valid.

Because the pseudo bond graph uses standard bond graph elements, bonds and modulating lines, the structure of the system is clearly shown and the modelling process is unchanged.

Karnopp[17] states that despite some potential or actual disadvantages of the compressible thermofluid pseudo bond graphs when compared to true bond graphs, the importance of control volume analysis of compressible thermofluid systems warrants their use in combination with true bond graphs.

7.2 Computer aided modelling using bond graphs

The bond graph is an ideal model representation for a Computer Aided Modelling (CAM) package because:

- The bond graph is a multidomain modelling technique.
- The bond graph is a graphical representation of the behaviour of a system and the structure is therefore easily represented and manipulated.
- The cause-effect relationships, obtained from the structure, are shown. Also, the assignment of causality is a sequential procedure, governed by a well defined set of rules. Such an algorithm is easily implemented on a computer.
- Most of the state-space equations are obtained from systematic equation substitutions. This algorithm is also easily implemented.

It is possible to construct a bond graph model of a system by connecting together the bond graph models of the parts which make up the system. This is because the bond graph models of the parts have the same form as the bond graph model of the system. Because a computer has the ability to store and retrieve large amounts of data, the construction of a bond graph model from a number of existing stored bond graph fragments is easily accomplished.

7.2.1 The bond graph program

The bond graph program allows the user to enter each element separately from the keyboard and include a previously stored bond graph fragment. The program also includes numerous routines to manipulate the bond graph structure.

The causality is assigned by the program and is complete in that it attempts to resolve all causal conflicts. If the causality assignment procedure cannot be completed, the bond graph or system is structurally incorrect. The causality assignment routine uses a state transition table to automate the procedure. Hood, Palmer and Dantzig[10] state that this algorithm is more efficient and requires less code than conventional *if-then-else* programming.

The user is given the causal implications and is required to enter the causal relations for each element. If the equations are linear, only the linear coefficients need be entered. Also if the user has only a number of data points describing the relation, these can be entered and the program creates an ACSL table from them. The equations entered by the user are stored so that they do not have to re-entered each time the routine is called.

It has been stated, that there are occasions when the implicit integration fails to converge and the algebraic loop cannot be solved without algebraic manipulation. It is then necessary to include in other bond graph elements (normally source and storage elements) to remove the algebraic loop or derivative causality. Thoma[30] states that the inclusion of these storage elements, which are usually small in comparison to the major storage elements of the system, can lead to inaccuracies in computation. This is because the dynamics of these elements are usually significantly faster than the dynamics of the system as a whole and hence, the integration step length has to be made sufficiently small so that these transients are calculated accurately. Such a system is termed a stiff system.

The bond graph program allows rapid structural modification, causality assignment, equation modification and re-simulation. For these reasons and because the model can be construc'd

from a number of stored "building blocks", computer aided bond graph modelling is superior to conventional pen and paper methods.

Chapter 8

Recommendations for further work

This chapter will be divided into two sections, the first section will discuss topics for further research in bond graph theory and the second will deal with possible improvements and additions to the bond graph program.

8.1 Bond graph theory

In this dissertation, pseudo bond graphs are used to model compressible thermofluid systems. As shown in Section 3.5.7, the compressible thermofluid pseudo bond graph cannot be directly connected to a conventional bond graph, where the product of the effort and the flow variables is the instantaneous power. It would therefore be advantageous to combine the double bond of the pseudo bond graph into the single bond of a true bond graph.

Le Sueur[19] uses true bond graphs to model compressible thermofluid systems, by defining the effort variable to be the specific enthalpy and the flow variable to be the mass flow rate. Because the enthalpy is a function of the pressure and temperature, all the elements in the bond graph must be modulated by either the pressure or the temperature so that the other can be calculated. Le Sueur modulates all the elements by the pressure. It is the author's opinion that although the product of specific enthalpy and mass flow rate is power, it is as inconvenient to "carry around" the pressure as it is to use a pseudo bond graph. In addition, the enthalpy is not a natural modelling variable as it cannot be directly measured, merely calculated. The pressure, temperature and mass flow rates of the compressible thermofluid pseudo bond graph, however, are the variables normally used to model such systems.

Other research has been conducted on modelling open systems [3,5,27], but the bond graph models derived are complex and offer little or no insight into the system structure or behaviour. The problem that arises, is that three variables are needed to model compressible thermofluid systems, as opposed to the two used in true bond graph models. More research, into the modelling of compressible thermofluid systems using bond graphs is required.

In order to model chemical process systems, chemical bonds must be added to the existing bonds used to model compressible thermofluids. Research into this area is required.

Karnopp[18], states that it is easy enough to include an inertia in a compressible thermofluid pseudo bond graph, but its properties are difficult to justify with any precision. More research is required to fully understand the implication of using such an element.

8.2 The bond graph program

8.2.1 The gr phical interface

In the author's opinion, a graphical interface is the most important unit that needs to be included in the bond graph program. What is envisaged, is a graphical editor, where the user can position the cursor, using a "mouse", on an element or bond in a side window and place it where it is required in the bond graph. The user should, in a similar manner, be able to move and delete bonds and elements.

The accessibility of information, about the elements, bonds or the system as a whole, can be improved by including a graphical editor. For example, the user should be able to position the cursor on an element and request all the information on that element. The user should be given the element number, the numbers of the bonds attached to the element, the causality and the causal implications (if the causality has already been assigned), the equations governing the elements behaviour (if the equations have been entered by the user) and the documentation for the element.

At present, modulating lines cannot be included in the bond graph data base, but the signals carried can be included in the equations of the modulated element. With a graphical interface, however, the user should be able to enter a modulating line. When entering the equations of an element connected to a modulating line, the dependent variables should include the signal carried by the line.

Once all of the elements have been entered, the causality assignment procedure should be invoked from the graphical interface. The causal strokes assigned must be displayed.

The user should have continual access to the graphical editor so that information about the individual elements can be retrieved and the bond graph structure can be seen. This is necessary, so that a particular element can be visualized within the context of the entire system. Having the structure in front of the user, when simulating the system, aids the user in choosing the correct variables for a particular task. The "windows" concept can be used to implement this, with the simulation or equation input routine in one window and the graphical interface in another.

It is envisaged that the graphical editor will have the form shown in Figure 8.1.

A graphical interface, clearly displays and simplifies the modification of the bond graph.

8.2.2 Miscellaneous additions

At present, the bond graph program can only use the generalized variables effort, flow, displacement and momentum. It is envisaged that the user should be able to enter the equations using variables natural to the particular domain. For example, when modelling mechanical systems, the user should be able to use the variables; force F, velocity v, displacement x and momentum p. For example, consider a 2-port C-field, where the one port (bond 4) is an electrical port and the other (bond 5) is a mechanical port. The user should be able to define the effort, flow and displacement on bond 4 to be the voltage, V_4, current, i_4 and charge, q_4 respectively. In a similar manner, the user should be able to define the effort, flow and displacement on bond 5 to be the force, F_5, velocity, v_5 and displacement, x_5 respectively.

As mentioned previously, the bond graph structure is compatible with the the "building block" concept. That is, a large system can be constructed by connecting together previously

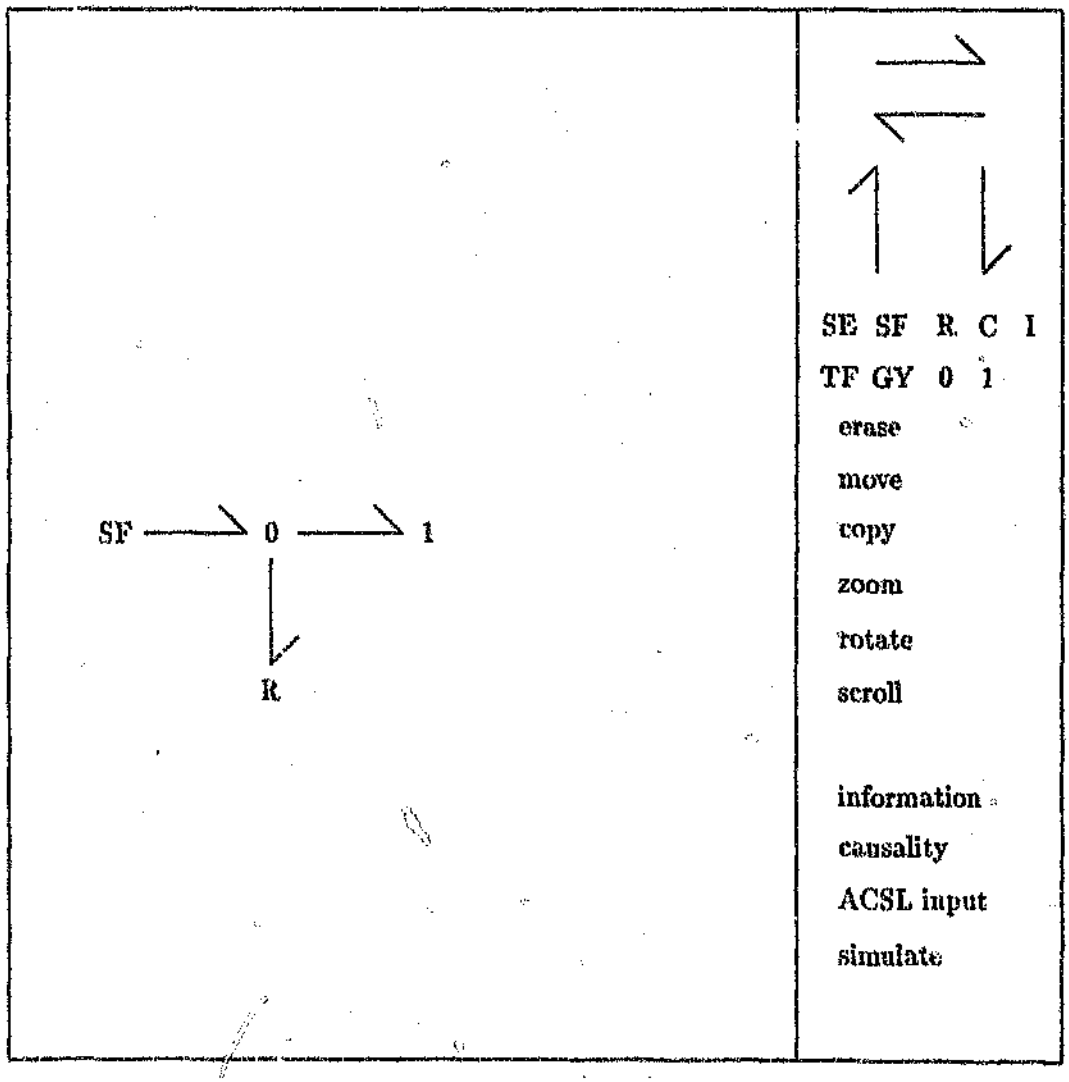


Figure 8.1: The proposed bond graph graphical editor.

stored bond graphs of smaller systems and subsystems. Because the bond graph of a large system can be extremely complex and so big that it becomes difficult to interpret, it is possible to represent the system using a *word bond graph*. Each word, will represent a smaller system or subsystem and will have a bond graph associated with it.

The word bond graph clearly represents the system structure, but the underlying and possibly complicating detail is hidden from the user. Normally the word bond graph is a tool used by the modeller to focus on the interconnections between the various subsystems [30] and is therefore used at the beginning of the modelling stage. In this context, the word bond graph is used to remove the detail, so that the bond graphs of the subsystems are not shown all the time. That is, the word bond graph shows the process from a systems viewpoint and is used throughout the modelling stage. The bond graph model of each subsystem should however, still be visible to the user if desired, perhaps the user could position the cursor on one of the words and the subsystem model could be displayed in one of the windows. This type of modelling would be extremely powerful, as the user can get an overview of the system or concentrate on the details if desired. In addition, because of the bond graph structure, it is still easy to modify the models of the subsystems, making them as complex or simple as the specifications require. The causality would be assigned as before, because the bonds connecting the subsystems in the word bond graph are the same bonds as those that would connect the bond graph fragments. This representation could therefore form the major plant data base, as all the individual elements are documented as well as each word representing a system.

Thoma[30], describes a procedure for estimating the eigenvalues of a linear system, prior to formulating the equations for the system. The use of this technique, within the modelling program, should be investigated as it would then be possible to estimate the integration step length required to simulate the system accurately.

Techniques exist [6], for separating the bond graph into two distinct parts, one part containing the fast transients and the other containing the slower transients. This would be beneficial in that the two bond graphs could be studied separately, depending on where the emphasis of the problem lies. In addition, by separating the system into two bond graphs, simulation, particularly of stiff systems would be more accurate. Such techniques could be included in the modeling package.

A state transition table is used to automate the assignment of causality. Hood, Palmer and Dantzig[10], state that this method is more efficient, requires less code and is more easily maintained than conventional *if-then-else* programming. Other methods may prove to be more efficient. Research into other algorithms, possibly using constraint propagation techniques [35], is required.

At present, ACSL is only able to construct numerical, linear state-space equations from the data entered by the user. It is envisaged that symbolic equations, entered by the user, could be manipulated to obtain the symbolic state-space equations. Typical packages that could be used include Mathematica and MACSYMA.

References

[1] "Advanced Continuous Simulation Language (ACSL) Reference Manual". Mitchell and Gauthier Associates, 1986.

[2] Barnard B.W. and Dransfield P. "Predicting Response of a Proposed Hydraulic Control System Using Bond Graphs". Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, March 1977, pp 1-8.

[3] Beaman J.J. and Breedveld P.C. "Physical Modelling with Eulerian Frames and Bond Graphs". Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, vol. 110, June 1988, pp 182-188.

[4] Bos A.M. and Breedveld P.C. "1985 Update of the Bond Graph Bibliography". Journal of the Franklin Institute, vol. 319, no. 1/2, January/February 1985, pp 269-286.

[5] Breedveld P.C. "Thermodynamic Bond Graphs and the Problem of Thermal Inertance". Journal of the Franklin Institute, vol. 314, no. 1, July 1982, pp 15-40.

[6] Dauphin-Tanguy G. and Borne P. "Order Reduction of Multi-time Scale Systems Using Bond Graphs, the Reciprocal System and the Singular Perturbation Method". Journal of the Franklin Institute, vol. 319, no. 1/2, 1985, pp 157-171.

[7] Douglas J.F., Gasiorek J.M., Swaffield J.A. "Fluid Mechanics". Longman Scientific and Technical, 1985.

[8] Engja H. "Bond Graph Model of a Reciprocating Compressor". Journal of the Franklin Institute, vol. 319, no. 1/2, 1985, pp 115-124.

[9] Granda J.J. "Computer Generation of Physical System Differential Equations Using Bond Graphs". Journal of the Franklin Institute, vol. 319, no. 1/2, 1985, pp 243-255.

[10] Hood S.J., Palmer E.R. and Dantzig F. "A Fast, Complete Method for Automatically Assigning Causality to Bond Graphs". Journal of the Franklin Institute, vol. 326, no. 1, 1989, pp 83-92.

[11] Hood S.J., Palmer E.R. and Withers D.H. "Automating Physical System Modelling Using Bond Graphs". Computer Aided Design, vol. 21, no. 9, november 1989, pp 584-588.

[12] Janna W.S. "Engineering Heat Transfer". Van nostrand Reinhold, 1988.

[13] Karnopp D.C. and Rosenberg R.C "Guest Editorial". Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, September 1972, pp 177-178.

[14] Karnopp D C. and Rosenberg R.C. "System Dynamics: A Unified Approach". John Wiley and Sons, 1975.

[15] Karnopp D.C. "A Bond Graph Modelling Philosophy for Thermofluid Systems". Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, vol. 100, March 1978, pp 70–75.

[16] Karnopp D.C. "Pseudo Bond Graphs for Thermal Energy Transport". Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, vol. 100, September 1978, pp 165–169.

[17] Karnopp D.C. "State Variables and Pseudo Bond Graphs for Compressible Thermofluid Systems". Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, vol. 101, 1979, pp 201–204.

[18] Karnopp D.C. Personal Communication, 1990.

[19] Le Sueur P.K. "An Introduction to Bond Graphs". LGI Short Course no. 0681, 1990.

[20] MacLeod I.M. "ELEN522, Process Control". Course notes, Department of Electrical Engineering, University of the Witwatersrand, 1990.

[21] Margolis D.L. "Bond Graph Fluid Line Models for Inclusion with Dynamic Systems Simulations". Journal of the Franklin Institute, vol. 308, no. 3, September 1979, pp 255–268.

[22] Margolis D.L. "A Survey of Bond Graph Modelling for Interacting Lumped and Distributed Systems". Journal of the Franklin Institute, vol. 319, no. 1/2, January/Febuary 1985, pp 125–135.

[23] Perelson A.S. "Bond Graph Junction Structures". Transactions of ASME, Journal of Dynamic Systems, Measuremant and Control, June 1975, pp 189–195.

[24] Rosenberg R.C. "A Users Guide to Enport-4". Wiley, 1974.

[25] Rosenberg R.C. "Exploiting Bond Graph Causality in Physical System Models". Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, vol. 109, December 1987, pp 378–383.

[26] The Santa Cruz Operation. "SCO UNIX System V/386", 1989.

[27] Shoureshi R. and McLaughlin K. "Application of Bond Graphs to Thermofluid Processes and Systems". Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, vol. 107, December 1985, pp 241–245.

[28] Steele G.L. Jr. "Common Lisp: The Language". Digital Press, 1984.

[29] Stevens A.L. and MacLeod I.M. "Research Issues in Computer Aided Process Modelling and Control System Design". Transactions of the South African Institute of Electrical Engineers, vol. 82, no. 1, March 1991, pp 22–27.

[30] Thoma J.U. "Simulation by bond graphs". Version 8810, Thoma Consulting, Bellevueweg 23, 6300 Zug, Switzerland, 1988.

[31] Tylee J.L. "Pseudo Bond Graph Representation of PWR Pressurizer Dynamics". Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, vol. 105, December 1983, pp 255–261.

[32] van Dixhoorn J.J. "Simulation of Bond Graphs on Minicomputers". Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, March 1977, pp 9–14.

[33] Wellstead P.E. "Introduction to Physical System Modelling". Academic Press, 1979.

[34] Wild B.G. "ELEN304, Digital and Microprocessor Engineering". Course notes, Department of Electrical Engineering, University of the Witwatersrand, 1988.

[35] Winston P.H. "Artificial Intelligence". 2nd edition, Addison–Wesley, USA, 1984.

[36] Winston P.H. and Horn B.K.P. "Lisp". 3rd edition, Addison–Wesley, USA, 1989.

[37] Zeid A. "Some Bond Graph Structural Properties: Eigen Spectra and Stability". Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, vol. 111, September 1989, pp 382–288.

[38] Zhou T., Rosenberg R.C. and Breedveld P.C. "Bibliography of Bond Graph Theory and Applications". Personal Communication, 1990.

Appendix A

Alteration of the causality assignment state transition table

In the state transition table presented by Hood, Palmer and Dantzig[10], when *state* is equal to *start-state* and the element being considered is either a 0 or a 1, an error results.

As an example of why this should not be so, consider the bond graph in Figure A.1a. The causality assignment proceeds as follows.

The required causality is assigned to the flow source, but because the 0-junction only has one effort output the causality can not be propagated any further. The preferred causality is assigned to the capacitor, again the causality can not be continued any further as the 1-junction only has one effort input. The preferred causality is then assigned to the inertia, but as in the two previous steps, the causality can not be extended further as the 0-junction only has one effort output. The assignment of these three causalities is shown in Figure A.1b. It is now necessary to assign an arbitrary causality to one of the three bonds 2, 4 or 5. Had the state transition table, devised by Hood, Palmer and Dantzig, been used an error would now result, because all the unassigned bonds are connected only to 0 or 1-junctions.

Continuing with the causality assignment procedure.

Bond 2 is assigned a causality such that the effort is the input to the left hand 0-junction (the output from the right hand 0-junction) and the causality of the remaining two bonds follows automatically. The causally oriented bond graph is shown in Figure A.1c. Note that bond 2 could also have been assigned a causality so that the effort is the output from the left hand 0-junction (the input to the right hand 0-junction). Had this been the case the causally oriented bond graph would have been that shown in Figure A.1d.

Therefore if a bond has to have an arbitrary causality assigned to it and it is attached only to 0 or 1-junctions, it should be treated in the same way as if it were connected to a transformer, a gyrator or a resistor. The cells corresponding to *state* being *start-state* and the element being either a 0 or a 1, are therefore changed to *get-next, R-state*.

The assignment of causality to a bond graph, such as that presented in Figure A.1a, using the corrected state transition table has been found to be correct.

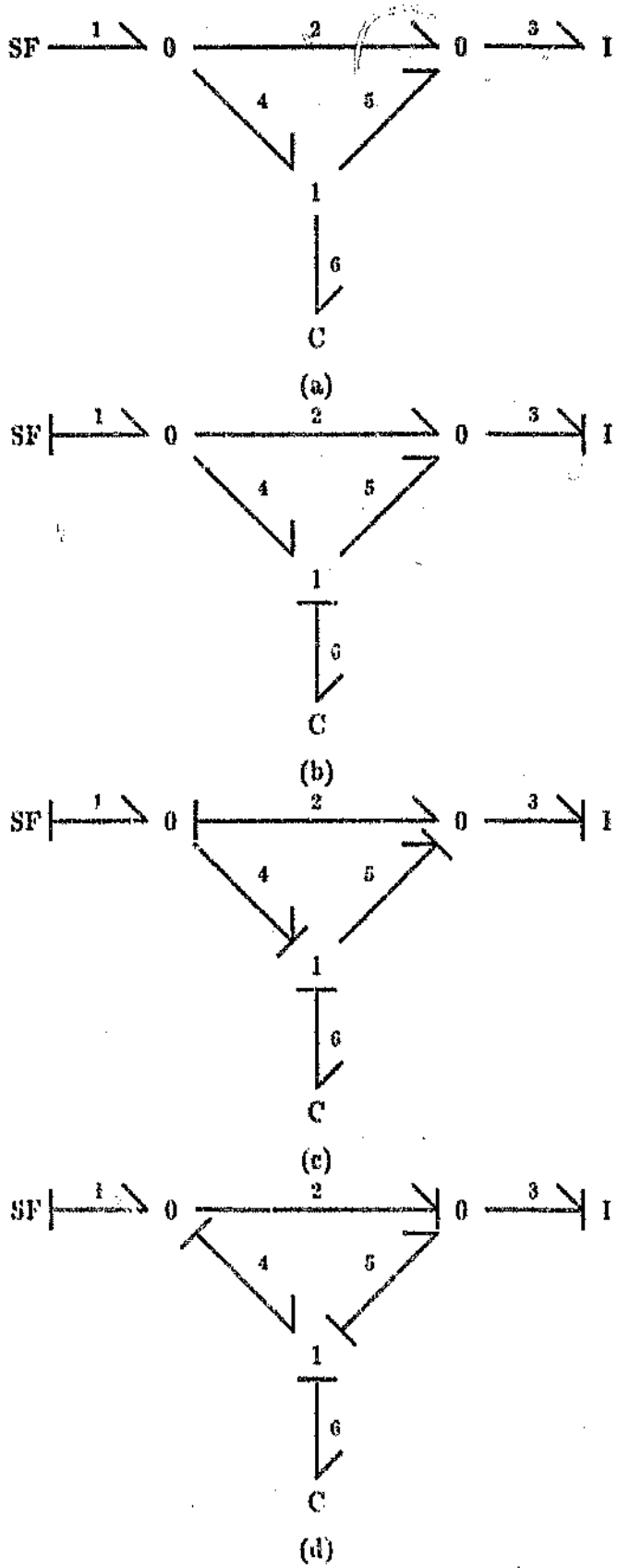


Figure A.1: A bond graph where the causality assignment procedure is completed by assigning an arbitrary causality to a bond attached to a 1 or 0-junction.

Appendix B

Further examples using the bond graph program

In this appendix, further examples using the bond graph program will be shown. Three systems will be simulated and the equations found. The examples do not correspond to any particular systems but are useful to demonstrate the operation of the program.

B.1 A system with all the storage elements assigned preferred causality and no algebraic loops

The aim of this example is to demonstrate the use of the modelling program to enter the bond graph from the keyboard, assign the causality, enter the equations and run ACSL.

The system considered in this example, will be that presented in Section 2.10.1, redrawn in Figure B.1. The difference between Figure B.1 and Figure 2.23 in Section 2.10.1, is that

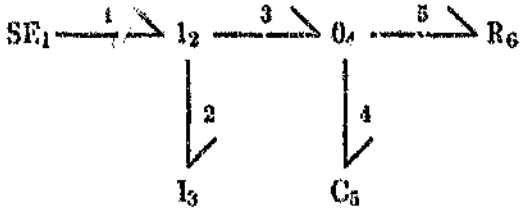


Figure B.1: The bond graph of a system without derivative causality and algebraic loops.

in Figure B.1, all the elements are numbered so that the system can be entered into the program.

It is assumed, that all the elements have the same equations as those presented in Section 2.10.1 and the values of the constants are:

$E_1 = u(t)$
$I_2 = 2.0$
$C_4 = 4.0$
$R_5 = 5.0$

Because the bond graph in Figure B.1 does not correspond to any particular system, no documentation strings are entered. The examples in Section 5 and Appendix C clearly show the use of the documentation string.

The execution of the bond graph program is as follows.

```
> (load "bond.lisp")

Loading the necessary programs, won't be a second

Do you want to

A Enter or modify the bond graph structure

B Assign or view the causality

C Create an ACSL input file

D Run ACSL

E Quit the program

Press the key corresponding to your choice and hit enter a

Do you want to

A Enter the bond graph, either by entering each element separately from
the keyboard, by reading the bond graph or bond graph fragment from a
data file or by connecting existing bond graphs elements or fragments.

B Display all the bonds currently connected.

C Delete all the bonds in the bond list.

D Modify the variables of a bond.

E Delete a bond from the bond list.

F Undo the most recent bond deletion.

G Leave the bond graph entering routine.

Press the key corresponding to your choice and hit enter a

Do you want to

A Enter the bonds and elements separately from the keyboard

B Include an existing bond graph

Press the desired key and hit return a

Enter the bond number
1

Enter the number of the element at the start of the power half arrow
1

Enter the element type
se
```

```
Enter the documentation string for this element

Enter the number of the element at the end of the power half arrow
2

Enter the element type
1

Enter the documentation string for this element

Enter F to finish or any other key to continue
c

Enter the bond number
2

Enter the number of the element at the start of the power half arrow
2

This element has been previously defined


Enter the number of the element at the end of the power half arrow
3

Enter the element type
i

Enter the documentation string for this element

Enter F to finish or any other key to continue
c

Enter the bond number
3

Enter the number of the element at the start of the power half arrow
2

This element has been previously defined

Enter the number of the element at the end of the power half arrow
4

Enter the element type
0

Enter the documentation string for this element

Enter F to finish or any other key to continue
c

Enter the bond number
4

Enter the number of the element at the start of the power half arrow
4

This element has been previously defined

Enter the number of the element at the end of the power half arrow
5
```

```
Enter the element type
c

Enter the documentation string for this element

Enter F to finish or any other key to continue
c

Enter the bond number
5

Enter the number of the element at the start of the power half arrow
4

This element has been previously defined

Enter the number of the element at the end of the power half arrow
6

Enter the element type
r

Enter the documentation string for this element

Enter F to finish or any other key to continue
f

FINISHED


Do you want to

A Enter the bond graph, either by entering each element separately from
the keyboard, by reading the bond graph or bond graph fragment from a
data file or by connecting existing bond graphs elements or fragments.

B Display all the bonds currently connected.

C Delete all the bonds in the bond list.

D Modify the variables of a bond.

E Delete a bond from the bond list.

F Undo the most recent bond deletion.

G Leave the bond graph entering routine.

Press the key corresponding to your choice and hit enter g


Enter the filename in which to store the hash tables,
do not include an extension, it will be set to '.data'
egl

Data written to file /usr/bondgraph/data/egl.data


Do you want to

A Enter or modify the bond graph structure

B Assign or view the causality
```

```
C Create an ACSL input file
D Run ACSL
E Quit the program
Press the key corresponding to your choice and hit enter b

Do you want to
A Assign causality to a bond graph.
B View the causal implications.
C View the bond graph data structure
D Leave the causality assignment routine.
Press the key corresponding to your choice and hit enter a

Do you want to assign or view the causality for the bond graph stored in
the file EG1.data (Y or N) y

Getting the first element
Assigning causality
Getting the next element in the stack
Getting the first element
Assigning causality
Getting the next element in the stack
Assigning causality
Getting the next element in the stack
Assigning causality
Getting the next element in the stack
Assigning causality
Getting the next element in the stack
DONE
Data written to file /usr/bondgraph/data/eg1.data

Do you want to
A Assign causality to a bond graph.
B View the causal implications.
C View the bond graph data structure
D Leave the causality assignment routine.
Press the key corresponding to your choice and hit enter b
```

```
Do you want to assign or view the causality for the bond graph stored in the
file EG1.data (Y or N) y


RESISTOR (ELEMENT NUMBER 6) CONNECTED TO
BOND 5

The causality assigned implies that
f5 is a function of e5

CAPACITOR (ELEMENT NUMBER 5) CONNECTED TO
BOND 4

The causality assigned implies that
e4 is a function of q4

INERTIA (ELEMENT NUMBER 3) CONNECTED TO
BOND 2

The causality assigned implies that
f2 is a function of p2

EFFORT SOURCE (ELEMENT NUMBER 1) CONNECTED TO
BOND 1

The causality assigned implies that e1 is the output

Press ENTER to continue


Do you want to

A Assign causality to a bond graph.

B View the causal implications.

C View the bond graph data structure

D Leave the causality assignment routine.

Press the key corresponding to your choice and hit enter c


Do you want to assign or view the causality for the bond graph stored in the
file EG1.data (Y or N) y


Bond 5 is #S(LINE NODE-PAIR (4 6) CAUSALITY E)
Element 4 is #S(NODE TYPE 0 DOC-STR  EQUATIONS NIL)
Element 6 is #S(NODE TYPE R DOC-STR  EQUATIONS NIL)

Bond 4 is #S(LINE NODE-PAIR (4 5) CAUSALITY B)
Element 4 is #S(NODE TYPE 0 DOC-STR  EQUATIONS NIL)
Element 5 is #S(NODE TYPE C DOC-STR  EQUATIONS NIL)

Bond 3 is #S(LINE NODE-PAIR (2 4) CAUSALITY B)
Element 2 is #S(NODE TYPE 1 DOC-STR  EQUATIONS NIL)
Element 4 is #S(NODE TYPE 0 DOC-STR  EQUATIONS NIL)
```

```
Bond 2 is #S(LINE NODE-PAIR (2 3) CAUSALITY E)
Element 2 is #S(NODE TYPE 1 DOC-STR  EQUATIONS NIL)
Element 3 is #S(NODE TYPE I DOC-STR  EQUATIONS NIL)

Bond 1 is #S(LINE NODE-PAIR (1 2) CAUSALITY E)
Element 1 is #S(NODE TYPE SE DOC-STR  EQUATIONS NIL)
Element 2 is #S(NODE TYPE 1 DOC-STR  EQUATIONS NIL)

Press ENTER to continue


Do you want to
A Assign causality to a bond graph.
B View the causal implications.
C View the bond graph data structure
D Leave the causality assignment routine.
Press the key corresponding to your choice     hit enter d

Do you want to
A Enter or modify the bond graph structure
B Assign or view the causality
C Create an ACSL input file
D Run ACSL
E Quit the program
Press the key corresponding to your choice and hit enter c

Do you want to get the acsl equations of the bond graph stored in the file
EG1.data (Y or N) y

Enter the acsl program name.
A system without algebraic loops and derivative causality
Enter the integration step size, ccalc
0.01
Enter the duration of the simulation, simtim.
200
RESISTOR (ELEMENT NUMBER 6) CONNECTED TO
BOND 5
The documentation for this element is:
""
Enter the data for resistor R5.
If the equation is linear, enter only the resistance.
```

```
If the equation is non-linear enter the entire equation with
f5 as a function of e5
If the data is in the form of a table enter "table"
5.0

CAPACITOR (ELEMENT NUMBER 5) CONNECTED TO
BOND 4

The documentation for this element is:
""

Enter the data for capacitor C4.
If the equation is linear, enter only the capacitance.
If the equation is non-linear enter the entire equation with
e4 as a function of q4
If the data is in the form of a table, enter "table"
4.0

Enter the value of q4 at time t=0
0.0

INERTIA (ELEMENT NUMBER 3) CONNECTED TO
BOND 2

The documentation for this element is:
""

Enter the data for inertia I2.
If the equation is linear, enter only the inertance.
If the equation is non-linear enter the entire equation with
f2 as a function of p2
If the data is in the form of a table, enter "table"
2.0

Enter the value of p2 at time t=0
0.0

EFFORT SOURCE (ELEMENT NUMBER 1) CONNECTED TO
BOND 1

The documentation for this element is:
""

Enter the data for effort source SE1.
If the effort input is a step at t=0 then enter only the height of the step.
If the effort input is anything but a step at t=0 then enter the entire
equation, using the form e1=.....
If the data is in the form of a table enter "table"
1.0

Acsl input file dumped to the file /usr/bondgraph/acslfile/eg1.acsl

Hash table written to file /usr/bondgraph/data/eg1.data


Do you want to

A Enter or modify the bond graph structure
```

```
B Assign or view the causality

C Create an ACSL input file

D Run ACSL

E Quit the program

Press the key corresponding to your choice and hit enter d


Do you want to simulate the system stored in the file
EG1.acsl (Y or N) y
```

The ACSL input file created by the program is:

```
program A system without algebraic loops and derivative causality
initial
constant points=1000
constant ccalc=0.01
constant simtim=200
simend=simtim-ccalc
cint=simtim/points
nstp=cint/ccalc
constant e1=1.0
constant i2=2.0
constant p2i=0.0
constant c4=4.0
constant q4i=0.0
constant r5=5.0
end $"initial"
dynamic
derivative
f3=f2
f1=f2
e2=-e3+e1
f2=(1.0/i2)*p2
p2=integ(e2,p2i)
e5=e4
e3=e4
f4=-f5+f3
e4=(1.0/c4)*q4
q4=integ(f4,q4i)
f5=(1.0/r5)*e5
termt(t.gt.simend)
end $"derivative"
end $"dynamic"
terminal
end $"terminal"
end $"program"
```

The system is then simulated using ACSL. The momentum p2 varies with time as shown in Figure B.2 and the displacement q4 varies with time as shown in Figure B.3.

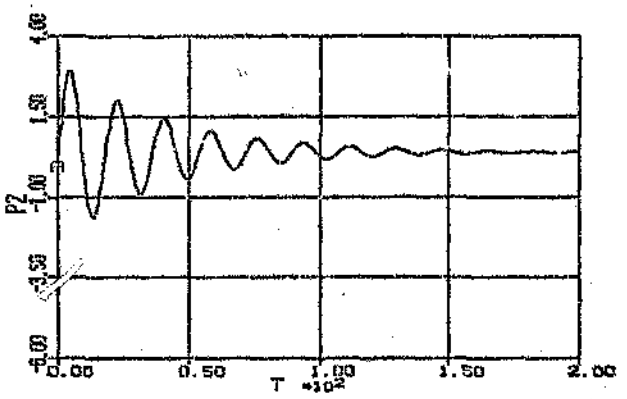


Figure B.2: Momentum p2 as a function of time.

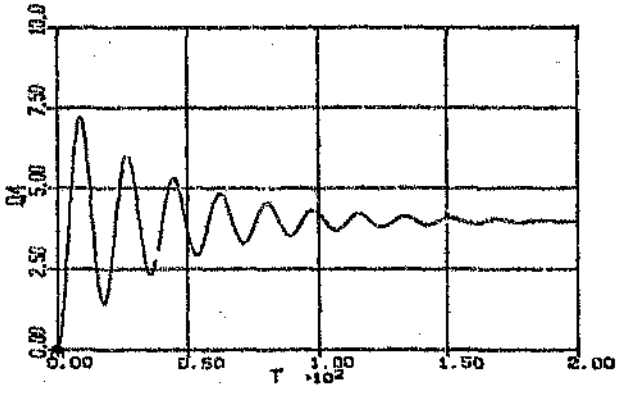


Figure B.3: Displacement q4 as a function of time.

The linear state space equations, rounded to three decimal places, as calculated by ACSL are

$$\begin{bmatrix} \dot{p}_2 \\ \dot{q}_4 \end{bmatrix} = \begin{bmatrix} 0.000 & -0.250 \\ 0.500 & -0.050 \end{bmatrix} \begin{bmatrix} p_2 \\ q_4 \end{bmatrix} + \begin{bmatrix} 1.000 \\ 0.000 \end{bmatrix} \begin{bmatrix} e_1 \end{bmatrix}$$

B.2 A system with all integral causality, but exhibiting an algebraic loop

The aim of this example is to show the inclusion of a bond graph from a data file, the assignment of causality, the entering of the equations, including a table of values and the ACSL output.

The system considered in this example, is that presented in Section 2.10.2. In this example, the bond graph is read in from a data file called `algeloop.data`. The bond and element numbers are assigned by the modelling package.

The numbered bond graph, previously stored, is shown in Figure B.4.

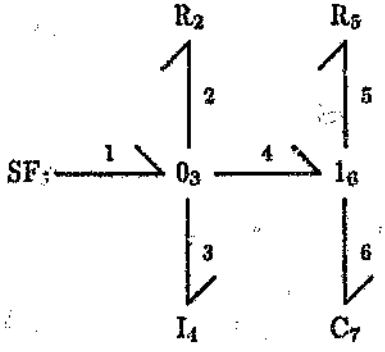


Figure B.4: The bond graph read in by the modelling package.

The causality is assigned as in Section B.1.

It is assumed, that all the elements, with the exception of capacitor 6, have the same equations as those presented in Section 2.10.1. The values of the cons ants are:

$F_1 = 28.0u(t)$
$I_3 = 3.0$
$R_2 = 2.0$
$R_5 = 5.0$

Capacitor 6 has the relation shown in Figure B.5. Table B.1, shows the discrete values chosen from Figure B.5, to be entered as data for capacitor 6.

The execution of the bond graph program is as follows:

```
> (load "bond.lisp")
```

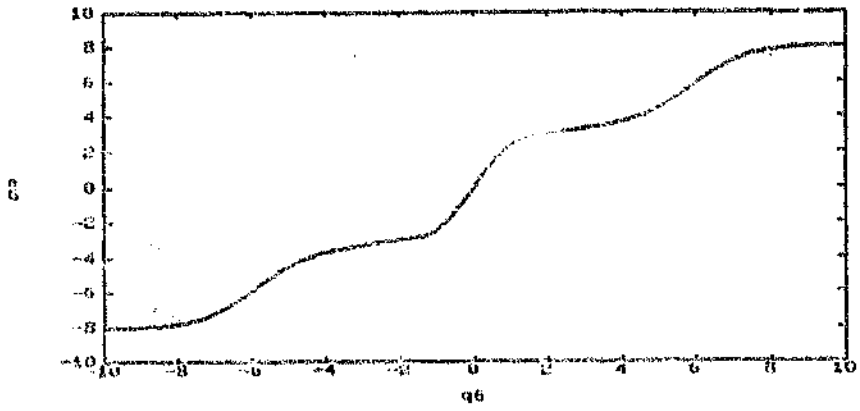


Figure B.5: The relation for the capacitor.

q_6	c_6
−10.0	−8.0
−9.0	−8.0
−8.0	−7.8
−7.0	−7.2
−6.0	−5.9
−5.0	−4.5
−4.0	−3.7
−3.0	−3.3
−2.0	−3.0
−1.0	−2.4
0.0	0.0
1.0	2.4
2.0	3.0
3.0	3.3
4.0	3.7
5.0	4.5
6.0	5.9
7.0	7.2
8.0	7.8
9.0	8.0
10.0	8.0

Table B.1: Data for capacitor 6.

Loading the necessary programs, won't be a second

Do you want to

A Enter or modify the bond graph structure

B Assign or view the causality

C Create an ACSL input file

D Run ACSL

E Quit the program

Press the key corresponding to your choice and hit enter a

Do you want to

A Enter the bond graph, either by entering each element separately from the keyboard, by reading the bond graph or bond graph fragment from a data file or by connecting existing bond graphs elements or fragments.

B Display all the bonds currently connected.

C Delete all the bonds in the bond list.

D Modify the variables of a bond.

E Delete a bond from the bond list.

F Undo the most recent bond deletion.

G Leave the bond graph entering routine.

Press the key corresponding to your choice and hit enter a

Do you want to

A Enter the bonds and elements separately from the keyboard

B Include an existing bond graph

Press the desired key and hit return b

Enter the filename of the bond graph you want to include
algeloop.data

Do you want to

A Enter the bond graph, either by entering each element separately from the keyboard, by reading the bond graph or bond graph fragment from a data file or by connecting existing bond graphs elements or fragments.

B Display all the bonds currently connected.

C Delete all the bonds in the bond list.

D Modify the variables of a bond.

E Delete a bond from the bond list.

```
F Undo the most recent bond deletion.

G Leave the bond graph entering routine.

Press the key corresponding to your choice and hit enter g

Enter the filename in which to store the hash tables,
do not include an extension, it will be set to '.data'
eg2

Data written to file /usr/bondgraph/data/eg2.data

Do you want to

A Enter or modify the bond graph structure

B Assign or view the causality

C Create an ACSL input file

D Run ACSL

E Quit the program

Press the key corresponding to your choice and hit enter b

Do you want to

A Assign causality to a bond graph.

B View the causal implications.

C View the bond graph data structure

D Leave the causality assignment routine.

Press the key corresponding to your choice and hit enter a

Do you want to assign or view the causality for the bond graph stored in the
file EG2.data (Y or N) y

Getting the first element

Assigning causality

Getting the next element in the stack

Getting the first element

Assigning causality

Getting the next element in the stack

Getting the first element

Assigning causality

Getting the next element in the stack

Getting the first element
```

```
Getting the next element in the stack
Assigning causality
Assigning causality
Getting the next element in the stack
Assigning causality
Getting the next element in the stack
DONE
Data written to file /usr/bondgraph/data/eg2.data

Do you want to
A Assign causality to a bond graph.
B View the causal implications.
C View the bond graph data structure
D Leave the causality assignment routine.
Press the key corresponding to your choice and hit enter b

Do you want to assign or view the causality for the bond graph stored in the
file EG2.data (Y or N) y

CAPACITOR (ELEMENT NUMBER 7) CONNECTED TO
BOND 6
The causality assigned implies that
e6 is a function of q6
RESISTOR (ELEMENT NUMBER 5) CONNECTED TO
BOND 5
The causality assigned implies that
f5 is a function of e5
INERTIA (ELEMENT NUMBER 4) CONNECTED TO
BOND 3
The causality assigned implies that
f3 is a function of p3
RESISTOR (ELEMENT NUMBER 2) CONNECTED TO
BOND 2
The causality assigned implies that
e2 is a function of f2
FLOW SOURCE (ELEMENT NUMBER 1) CONNECTED TO
BOND 1
The causality assigned implies that f1 is the output
Press ENTER to continue
```

```
Do you want to
A Assign causality to a bond graph.
B View the causal implications.
C View the bond graph data structure
D Leave the causality assignment routine.
Press the key corresponding to your choice and hit enter c

Do you want to assign or view the causality for the bond graph stored in the
file EG2.data (Y or N) y


Bond 6 is #S(LINE NODE-PAIR (6 7) CAUSALITY B)
Element 6 is #S(NODE TYPE 1 DOC-STR  EQUATIONS NIL)
Element 7 is #S(NODE TYPE C DOC-STR  EQUATIONS NIL)

Bond 5 is #S(LINE NODE-PAIR (6 5) CAUSALITY E)
Element 6 is #S(NODE TYPE 1 DOC-STR  EQUATIONS NIL)
Element 5 is #S(NODE TYPE R DOC-STR  EQUATIONS NIL)

Bond 4 is #S(LINE NODE-PAIR (3 6) CAUSALITY E)
Element 3 is #S(NODE TYPE 0 DOC-STR  EQUATIONS NIL)
Element 6 is #S(NODE TYPE 1 DOC-STR  EQUATIONS NIL)

Bond 3 is #S(LINE NODE-PAIR (3 4) CAUSALITY E)
Element 3 is #S(NODE TYPE 0 DOC-STR  EQUATIONS NIL)
Element 4 is #S(NODE TYPE I DOC-STR  EQUATIONS NIL)

Bond 2 is #S(LINE NODE-PAIR (3 2) CAUSALITY B)
Element 3 is #S(NODE TYPE 0 DOC-STR  EQUATIONS NIL)
Element 2 is #S(NODE TYPE R DOC-STR  EQUATIONS NIL)

Bond 1 is #S(LINE NODE-PAIR (1 3) CAUSALITY B)
Element 1 is #S(NODE TYPE SF DOC-STR  EQUATIONS NIL)
Element 3 is #S(NODE TYPE 0 DOC-STR  EQUATIONS NIL)

Press ENTER to continue


Do you want to
A Assign causality to a bond graph.
B View the causal implicat ons.
C View the bond graph data structure
D Leave the causality assignment routine.
Press the key corresponding to your choice and hit enter d
```

```
Do you want to

A Enter or modify the bond graph structure

B Assign or view the causality

C Create an ACSL input file

D Run ACSL

E Quit the program

Press the key corresponding to your choice and hit enter c


Do you want to get the acsl equations of the bond graph stored in the file
EG2.data (Y or N) y


Enter the acsl program name.
A system with an algebraic loop

Enter the integration step size, ccalc
0.01

Enter the duration of the simulation, simtim.
25

CAPACITOR (ELEMENT NUMBER 7) CONNECTED TO
BOND 6

The documentation for this element is:
""

Enter the data for capacitor C6.
If the equation is linear, enter only the capacitance.
If the equation is non-linear enter the entire equation with
e6 as a function of q6
If the data is in the form of a table, enter "table"
table

The table will be named tab1

If the capacitor is not modulated the independent variables are normally q6

Enter the independent variables in the form of a list eg (f3 e5 f7)
(q6)

Enter the number of data points for q6
21

Enter value 1 for q6
-10.0

Enter value 2 for q6
-9.0

Enter value 3 for q6
-8.0

Enter value 4 for q6
-7.0
```

```
Enter value 5 for q6
-6.0

Enter value 6 for q6
-5.0

Enter value 7 for q6
-4.0

Enter value 8 for q6
-3.0

Enter value 9 for q6
-2.0

Enter value 10 for q6
-1.0

Enter value 11 for q6
0.0

Enter value 12 for q6
1.0

Enter value 13 for q6
2.0

Enter value 14 for q6
3.0

Enter value 15 for q6
4.0

Enter value 16 for q6
5.0

Enter value 17 for q6
6.0

Enter value 18 for q6
7.0

Enter value 19 for q6
8.0

Enter value 20 for q6
9.0

Enter value 21 for q6
10.0

Enter the value of e6 when q6 is -10.0
-8.0

Enter the value of e6 when q6 is -9.0
-8.0

Enter the value of e6 when q6 is -8.0
-7.8

Enter the value of e6 when q6 is -7.0
-7.2
```

```
Enter the value of e6 when q6 is -6.0
-5.9

Enter the value of e6 when q6 is -5.0
-4.5

Enter the value of e6 when q6 is -4.0
-3.7

Enter the value of e6 when q6 is -3.0
-3.3

Enter the value of e6 when q6 is -2.0
-3.0

Enter the value of e6 when q6 is -1.0
-2.4

Enter the value of e6 when q6 is 0.0
0.0

Enter the value of e6 when q6 is 1.0
2.4

Enter the value of e6 when q6 is 2.0
3.0

Enter the value of e6 when q6 is 3.0
3.3

Enter the value of e6 when q6 is 4.0
3.7

Enter the value of e6 when q6 is 5.0
4.5

Enter the value of e6 when q6 is 6.0
5.9

Enter the value of e6 when q6 is 7.0
7.2

Enter the value of e6 when q6 is 8.0
7.8

Enter the value of e6 when q6 is 9.0
8.0

Enter the value of e6 when q6 is 10.0
8.0

Enter the value of q6 at time t=0
0.0

RESISTOR (ELEMENT NUMBER 5) CONNECTED TO
BOND 5

The documentation for this element is:
""

Enter the data for resistor R5.
If the equation is linear, enter only the resistance.
```

```
If the equation is non-linear enter the entire equation with
f5 as a function of e5
If the data is in the form of a table enter "table"
5.0

INERTIA (ELEMENT NUMBER 4) CONNECTED TO
BOND 3

The documentation for this element is:
""

Enter the data for inertia I3.
If the equation is linear, enter only the inertance.
If the equation is non-linear enter the entire equation with
f3 as a function of p3
If the data is in the form of a table, enter "table"
3.0

Enter the value of p3 at time t=0
0.0

RESISTOR (ELEMENT NUMBER 2) CONNECTED TO
BOND 2

The documentation for this element is:
""

Enter the data for resistor R2.
If the equation is linear, enter only the resistance.
If the equation is non-linear enter the entire equation with
e2 as a function of f2
If the data is in the form of a table enter "table"
2.0

FLOW SOURCE (ELEMENT NUMBER 1) CONNECTED TO
BOND 1

The documentation for this element is:
""

Enter the data for flow source SF1.
If the flow input is a step at t=0 then enter only the height of the step.
If the flow input is anything but a step at t=0 then enter the entire
equation, using the form f1=.....
If the data is in the form of a table, enter "table"
28.0


The equation e5=-e6+e4 has to be calculated implicitly

That is it will be written in the form

e5 = impl(yz, e, m, ef1, -e6+e4, ydl)

where yz is the initial guess
```

```
e is the error bound
m is the maximum number of iterations to try
efi is the error flag variable
ydl is the value used to increment the initial value to estimate the
    derivative


Please enter yz (a constant or expression) default 0.0

Please enter e (a constant or expression) default 0.0001

Please enter m (an integer constant) default 50

Please enter ydl (a constant or expression) default 0.001

Acsl input file dumped to the file /usr/bondgraph/acslfile/eg2.acsl

Hash table written to file /usr/bondgraph/data/eg2.data


Do you want to

A Enter or modify the bond graph structure

B Assign or view the causality

C Create an ACSL input file

D Run ACSL

E Quit the program

Press the key corresponding to your choice and hit enter d


Do you want to simulate the system stored in the file
EG2.acsl (Y or N) y
```

The ACSL input file, created by the modelling package from the bond graph structure stored in the file eg2.data and the equations entered by the user above is:

```
program A system with an algebraic loop
initial
constant points=1000
constant ccalc=0.01
constant simtim=25
simend=simtim-ccalc
cint=simtim/points
nstp=cint/ccalc
constant f1=28.0
constant r2=2.0
constant i3=3.0
constant p3i=0.0
constant r5=5.0
```

```
constant q6i=0.0
end $"initial"
dynamic
derivative
e5=impl(0.0  ,0.0001  ,50  ,ef1,-e6+e4,0.001)
e4=e2
e3=e2
e1=e2
f2=-f4-f3+f1
e2=r2*f2
f3=(1.0/i3)*p3
p3=integ(e3,p3i)
f6=f5
f4=f5
f5=(1.0/r5)*e5
table tab1,1,21/-10.0,-9.0,-8.0,-7.0,-6.0,-5.0,-4.0,-3.0...
,-2.0,-1.0,0.0,1.0,2.0,3.0,4.0,5.0,6.0...
,7.0,8.0,9.0,10.0,-8.0,-8.0,-7.8,-7.2,-5.9...
,-4.5,-3.7,-3.3,-3.0,-2.4,0.0,2.4,3.0,3.3...
,3.7,4.5,5.9,7.2,7.8,8.0,8.0/
e6=tab1(q6)
q6=integ(f6,q6i)
termt(t.gt.simend)
end $"derivative"
end $"dynamic"
terminal
end $"terminal"
end $"program"
```

The system is then simulated using ACSL. The momentum, p3 varies with time as shown in Figure B.6 and the displacement, q6 varies with time as shown in Figure B.7.

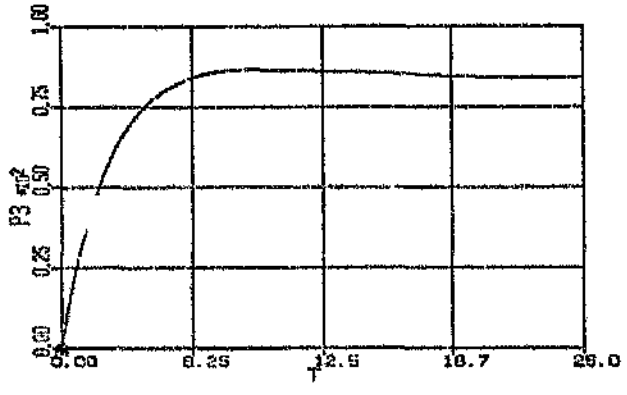


Figure B.6: Momentum, p3 as a function of time.

The linear state space equations, rounded to three decimal places, as calculated by ACSL are

$$\begin{bmatrix} \dot{p}_3 \\ \dot{q}_6 \end{bmatrix} = \begin{bmatrix} -0.476 & 0.686 \\ -0.095 & -0.343 \end{bmatrix} \begin{bmatrix} p_3 \\ q_6 \end{bmatrix} + \begin{bmatrix} 1.429 \\ 0.286 \end{bmatrix} \begin{bmatrix} f_1 \end{bmatrix}$$

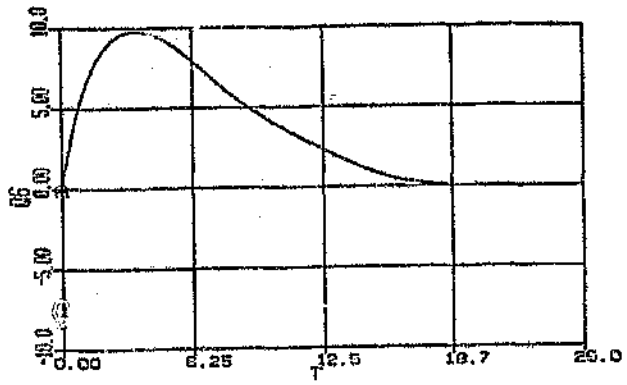


Figure B.7: Displacement, q6 as a function tim

B.3 A system with derivative causality

The aim of this example is to demonstrate entering the equations, simulation of the system, re-entering one of the equations and obtaining an ACSL output.

The system considered in this example, is that presented in Section 2.10.3, redrawn in Figure B.8 for convienience.

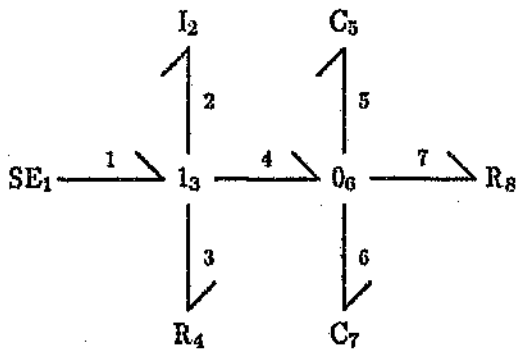


Figure B.8: The bond graph of a system displaying derivative causality

The bond graph is read in from the keyboard and the causality assigned automatically. This will not be shown here as the procedure is the same as that shown in Section B.1

It is assumed, that all the elements have the same equations as those presented in Section 2.10.3. The values of the constants are chosen to be:

$E_1 = u(t)$
$I_2 = 2.0$
$C_5 = 5.0$
$C_6 = 6.0$
$R_3 = 3.0$
$R_7 = 7.0$

```
Do you want to

A Assign causality to a bond graph.

B View the causal implications.

C View the bond graph data structure
```

```
D Leave the causality assignment routine.

Press the key corresponding to your choice and hit enter b


Do you want to assign or view the causality for the bond graph stored in the
file EG3.data (Y or N) y

RESISTOR (ELEMENT NUMBER 8) CONNECTED TO
BOND 7

The causality assigned implies that
f7 is a function of e7

CAPACITOR (ELEMENT NUMBER 7) CONNECTED TO
BOND 6

The causality assigned implies that
e6 is a function of q6

CAPACITOR (ELEMENT NUMBER 5) CONNECTED TO
BOND 5

The causality assigned implies that
q5 is a function of e5

RESISTOR (ELEMENT NUMBER 4) CONNECTED TO
BOND 3

The causality assigned implies that
e3 is a function of f3

INERTIA (ELEMENT NUMBER 2) CONNECTED TO
BOND 2

The causality assigned implies that
f2 is a function of p2

EFFORT SOURCE (ELEMENT NUMBER 1) CONNECTED TO
BOND 1

The causality assigned implies that e1 is the output

Press ENTER to continue


Do you want to

A Assign causality to a bond graph.

B View the causal implications.

C View the bond graph data structure

D Leave the causality assignment routine.

Press the key corresponding to your choice and hit enter c


Do you want to assign or view the causality for the bond graph stored in the
file EG3.data (Y or N) y
```

```
Bond 7 is #S(LINE NODE-PAIR (6 8) CAUSALITY E)
Element 6 is #S(NODE TYPE 0 DOC-STR  EQUATIONS NIL)
Element 8 is #S(NODE TYPE R DOC-STR  EQUATIONS NIL)

Bond 6 is #S(LINE NODE-PAIR (6 7) CAUSALITY B)
Element 6 is #S(NODE TYPE 0 DOC-STR  EQUATIONS NIL)
Element 7 is #S(NODE TYPE C DOC-STR  EQUATIONS NIL)

Bond 5 is #S(LINE NODE-PAIR (6 5) CAUSALITY E)
Element 6 is #S(NODE TYPE 0 DOC-STR  EQUATIONS NIL)
Element 5 is #S(NODE TYPE C DOC-STR  EQUATIONS NIL)

Bond 4 is #S(LINE NODE-PAIR (3 6) CAUSALITY B)
Element 3 is #S(NODE TYPE 1 DOC-STR  EQUATIONS NIL)
Element 6 is #S(NODE TYPE 0 DOC-STR  EQUATIONS NIL)

Bond 3 is #S(LINE NODE-PAIR (3 4) CAUSALITY B)
Element 3 is #S(NODE TYPE 1 DOC-STR  EQUATIONS NIL)
Element 4 is #S(NODE TYPE R DOC-STR  EQUATIONS NIL)

Bond 2 is #S(LINE NODE-PAIR (3 2) CAUSALITY E)
Element 3 is #S(NODE TYPE 1 DOC-STR  EQUATIONS NIL)
Element 2 is #S(NODE TYPE I DOC-STR  EQUATIONS NIL)

Bond 1 is #S(LINE NODE-PAIR (1 3) CAUSALITY E)
Element 1 is #S(NODE TYPE SE DOC-STR  EQUATIONS NIL)
Element 3 is #S(NODE TYPE 1 DOC-STR  EQUATIONS NIL)

Press ENTER to continue


Do you want to
A Assign causality to a bond graph.
B View the causal ir lications.
C View the bond graph data structure
D Leave the causality assignment routine.
Press the key corresponding to your choice and hit enter d

Do you want to
A Enter or modify the bond graph structure
B Assign or view the causality
C Create an ACSL input file
D Run ACSL
E Quit the program
Press the key corresponding to your choice and hit enter c
```

```
Do you want to get the acsl equations of the bond graph stored in the file
EG3.data (Y or N) y

Enter the acsl program name.
A system with derivative causality
Enter the integration step size, ccalc
0.01
Enter the duration of the simulation, simtim. 100
RESISTOR (ELEMENT NUMBER 8) CONNECTED TO
BOND 7
The documentation for this element is:
""
Enter the data for resistor R7.
If the equation is linear, enter only the resistance.
If the equation is non-linear enter the entire equation with
f7 as a function of e7
If the data is in the form of a table enter "table"
7.0
CAPACITOR (ELEMENT NUMBER 7) CONNECTED TO
BOND 6
The documentation for this element is:
""
Enter the data for capacitor C6.
If the equation is linear, enter only the capacitance.
If the equation is non-linear enter the entire equation with
e6 as a function of q6
If the data is in the form of a table, enter "table"
6.0
Enter the value of q6 at time t=0
0.0
CAPACITOR (ELEMENT NUMBER 5) CONNECTED TO
BOND 5
The documentation for this element is:
""
Enter the data for capacitor C5.
If the equation is linear, enter only the capacitance.
If the equation is non-linear enter the entire equation with
q5 as a function of e5
If the data is in the form of a table, enter "table"
5.0
RESISTOR (ELEMENT NUMBER 4) CONNECTED TO
BOND 3
The documentation for this element is:
""
```

```
Enter the data for resistor R3.
If the equation is linear, enter only the resistance.
If the equation is non-linear enter the entire equation with
e3 as a function of f3
If the data is in the form of a table enter "table"
3.0

INERTIA (ELEMENT NUMBER 2) CONNECTED TO
BOND 2

The documentation for this element is:
""

Enter the data for inertia I2.
If the equation is linear, enter only the inertance.
If the equation is non-linear enter the entire equation with
f2 as a function of p2
If the data is in the form of a table, enter "table"
2.0

Enter the value of p2 at time t=0
0.0

EFFORT SOURCE (ELEMENT NUMBER 1) CONNECTED TO
BOND 1

The documentation for this element is:
""

Enter the data for effort source SE1.
If the effort input is a step at t=0 then enter only the height of the step.
If the effort input is anything but a step at t=0 then enter the entire
equation, using the form e1=.....
If the data is in the form of a table enter "table"
1.0


Enter the time derivative of the function c5*e5
c5*e5d

Enter the time derivative of the function e6
e6d

Enter the time derivative of the function (1.0/c6)*q6
(1.0/c6)*q6d


The equation q6d = f6 has to be calculated implicitly

That is it will be written in the form

q6d = impl(yz, e, m, ef1, f6, yd1)

where yz is the initial guess
      e is the error bound
```

```
m is the maximum number of iterations to try
efl is the error flag variable
ydl is the value used to increment the initial value to estimate the
    derivative

Please enter yz (a constant or expression)  default 0.0

Please enter e (a constant or expression) default 0.0001

Please enter m (an integer constant) default 50

Please enter ydl (a constant or expression) default 0.001

Acsl input file dumped to the file /usr/bondgraph/acslfile/eg3.acsl

Hash table written to file /usr/bondgraph/data/eg3.data


Do you want to

A Enter or modify the bond graph structure

B Assign or view the causality

C Create an ACSL input file

D Run ACSL

E Quit the program

Press the key corresponding to your choice and hit enter d


Do you want to simulate the system stored in the file
EG3.acsl (Y or N) y
```

The ACSL input file, created by the modelling package from the bond graph structure stored in the file eg3.data and the equations entered by the user above is:

```
program A system with derivative causality
initial
constant points=1000
constant ccalc=0.01
constant simtim=100
simend=simtim-ccalc
cint=simtim/points
nstp=cint/ccalc
constant e1=1.0
constant i2=2.0
constant p2i=0.0
constant r3=3.0
constant c5=5.0
constant c6=6.0
```

```
constant q6i=0.0
constant r7=7.0
end $"initial"
dynamic
derivative
q6d=impl(0.0,0.0001,50,ef1,f6,0.001)
e6d=(1.0/c6)*q6d
e5d=e6d
q5d=c5*e5d
f4=f2
f3=f2
f1=f2
e2=-e4-e3+e1
f2=(1.0/i2)*p2
p2=integ(e2,p2i)
e3=r3*f3
e7=e6
e5=e6
e4=e6
f6=-f7-f5+f4
q5=c5*e5
f5=q5d
e6=(1.0/c6)*q6
q6=integ(f6,q6i)
f7=(1.0/r7)*e7
termt(t.gt.simend)
end $"derivative"
end $"dynamic"
terminal
end $"terminal"
end $"program"
```

The system is then simulated using ACSL. The momentum, p2 varies with time as shown in Figure B.9, the displacement, q6 varies with time as shown in Figure B.10 and the displace-

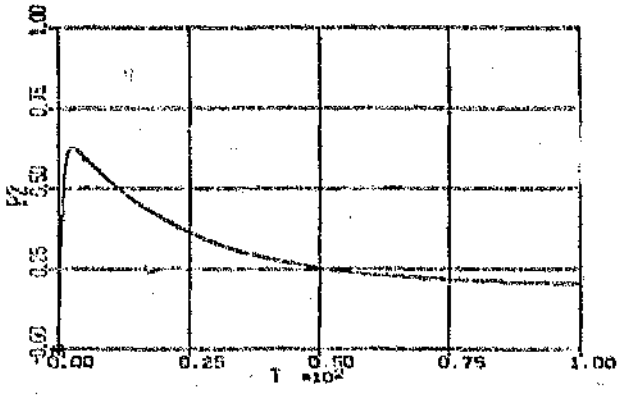


Figure B.9: Momentum, p2 as a function of time.

ment, q5 varies with time as shown in Figure B.11.

Returning to the entering of the equations and making $I_2 = 10.0$

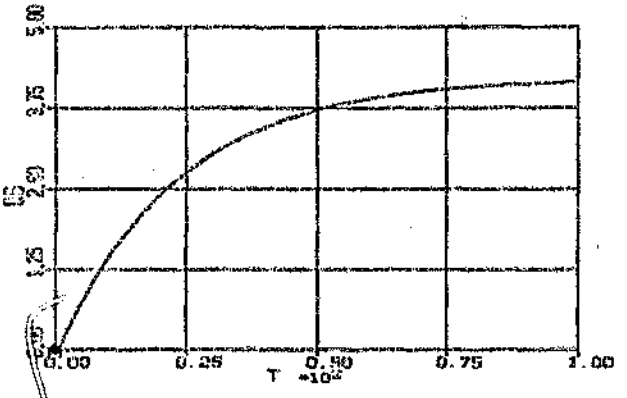


Figure B.10: Displacement, q6 as a function of time.

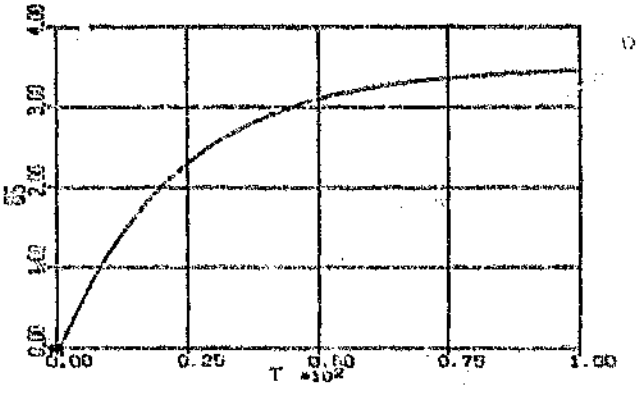


Figure B.11: Displacement, q5 as a function of time.

```
Do you want to

A Enter or modify the bond graph structure

B Assign or view the causality

C Create an ACSL input file

D Run ACSL

E Quit the program

Press the key corresponding to your choice and hit enter c


Do you want to get the acsl equations of the bond graph stored in the file
EG3.data (Y or N) y


Enter the acsl program name. A system with derivative causality

Enter the integration step size, ccalc 0.01

Enter the duration of the simulation, simtim. 100

RESISTOR (ELEMENT NUMBER 8) CONNECTED TO
BOND 7

The documentation for this element is:
""

The causality assigned implies that
f7 is a function of e7

The last equations stored were
f7=(1.0/r7)*e7
constant r7=7.0

Do these equations still hold (Y or N) y

CAPACITOR (ELEMENT NUMBER 7) CONNECTED TO
BOND 6

The documentation for this element is:
""

The causality assigned implies that
e6 is a function of q6

The last equations stored were
e6=(1.0/c6)*q6
constant c6=6.0
q6=integ(f6,q6i)
constant q6i=0.0

Do these equations still hold (Y or N) y

CAPACITOR (ELEMENT NUMBER 5) CONNECTED TO
BOND 5

The documentation for this element is:
""
```

```
The causality assigned implies that
q5 is a function of e5

The last equations stored were
q5=c5*e5
constant c5=5.0
f5=q5d

Do these equations still hold (Y or N) y

RESISTOR (ELEMENT NUMBER 4) CONNECTED TO
BOND 3

The documentation for this element is:
""

The causality assigned implies that
e3 is a function of f3

The last equations stored were
e3=r3*f3
constant r3=3.0

Do these equations still hold (Y or N) y

INERTIA (ELEMENT NUMBER 2) CONNECTED TO
BOND 2

The documentation for this element is:
""

The causality assigned implies that
f2 is a function of p2

The last equations stored were
f2=(1.0/i2)*p2
constant i2=2.0
p2=integ(e2,p2i)
constant p2i=0.0

Do these equations still hold (Y or N) n

Enter the data for inertia I2.
If the equation is linear, enter only the inertance.
If the equation is non-linear enter the entire equation with
f2 as a function of p2
If the data is in the form of a table, enter "table"
10.0

Enter the value of p2 at time t=0
0.0

EFFORT SOURCE (ELEMENT NUMBER 1) CONNECTED TO
BOND 1

The documentation for this element is:
""

The causality assigned implies that e1 is the output
```

```
The last equations stored were
constant e1=1.0

Do these equations still hold (Y or N) y


Enter the time derivative of the function c5*e5
c5*e5d

Enter the time derivative of the function e6
e6d

Enter the time derivative of the function (1.0/c6)*q6
(1.0/c6)*q6d


The equation q6d = f6 has to be calculated implicitly

That is it will be written in the form

q6d = impl(yz, e, m, ef1, f6, yd1)

where yz is the initial guess
      e is the error bound
      m is the maximum number of iterations to try
      ef1 is the error flag variable
      yd1 is the value used to increment the initial value to estimate the
          derivative


Please enter yz (a constant or expression)  default 0.0

Please enter e (a constant or expression) default 0.0001

Please enter m (an integer constant) default 50

Please enter yd1 (a constant or expression) default 0.001

Acsl input file dumped to the file /usr/bondgraph/acslfile/eg3.acsl

Hash table written to file /usr/bondgraph/data/eg3.data


Do you want to

A Enter or modify the bond graph structure

B Assign or view the causality

C Create an ACSL input file

D Run ACSL

E Quit the program
```

```
Press the key corresponding to your choice and hit enter d

Do you want to simulate the system stored in the file
EG3.acsl (Y or N) y
```

The system is again simulated using ACSL. The momentum, p2 now varies with time as shown in Figure B.12, the displacement, q6 now varies with time as shown in Figure B.13

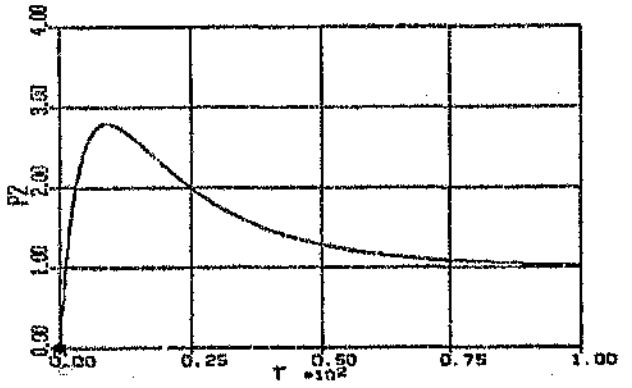


Figure B.12: Momentum, p2 as a function of time.

and the displacement, q5 now varies with time as shown in Figure B.14.

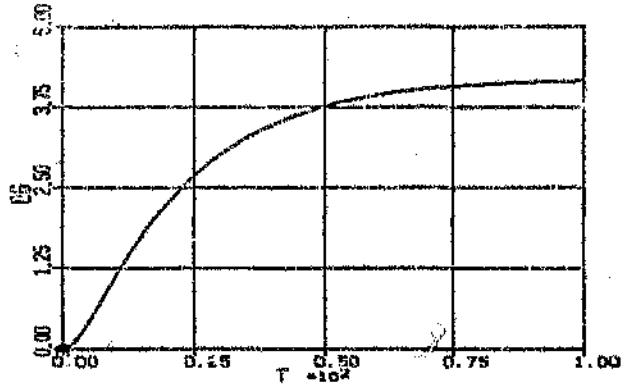


Figure B.13: Displacement, q6 as a function of time.

The linear state space equations of the system, rounded to three decimal places, as calculated by ACSL are

$$\begin{bmatrix} \dot{p_2} \\ \dot{q_6} \end{bmatrix} = \begin{bmatrix} -0.300 & -3.167 \\ 0.055 & -0.013 \end{bmatrix} \begin{bmatrix} p_2 \\ q_6 \end{bmatrix} + \begin{bmatrix} 1.000 \\ 0.000 \end{bmatrix} \begin{bmatrix} e_1 \end{bmatrix}$$

and

$$\begin{bmatrix} q_5 \end{bmatrix} = \begin{bmatrix} 0.833 \end{bmatrix} \begin{bmatrix} q_6 \end{bmatrix}$$

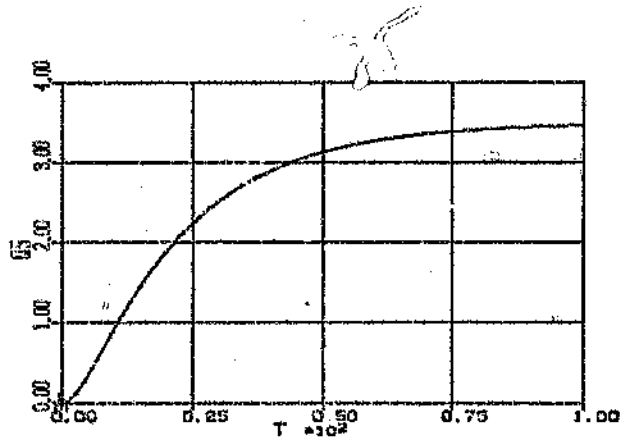


Figure B.14: Displacement, q5 as a function of time.

Appendix C

User manual

The bond graph modelling program is designed to allow the user to rapidly construct a bond graph model of a system. The system represented by the bond graph can be simulated and the linear equations of state obtained using ACSL.

The program is divided into four parts:

1. A module to enter and modify the structure of the bond graph.
2. An automatic sequential causality assignment procedure.
3. A module that creates an ACSL input file from the bond graph structure and the equations either entered by the user or stored in a data file.
4. A module to automatically run ACSL from the program.

C.1 Getting started

The bond graph program is contained in the directory /usr/bondgraph. The user is required to move to this directory, or to have this directory in the path. In addition, the user is required to ensure that the directory containing LISP is in the path. LISP is invoked by typing `lisp`. The user should now see the LISP prompt, >. The modelling package is started by typing (load "bond"). This function loads all the required modules and presents the user with the main menu:

```
Do you want to

A  Enter or modify the bond graph structure

B  Assign or view the causality

C  Create an ACSL input file

D  Run ACSL

E  Quit the program

Press the key corresponding to your choice and hit enter
```

Notes
The directory /usr/bondgraph contains both the LISP code and the compiled code of all modules in the bond graph modelling package. There are two subdirectories within /usr/bondgraph, these are /usr/bondgraph/data and /usr/bondgraph/acslfile. The first subdirectory contains all the bond graph data files (bond graph structures) and the second, all the ACSL input files (.acsl files which are converted to .csl files by the module which runs ACSL).

C.2 Entering, modifying and viewing the bond graph structure

The user is able to move from the main menu to the entering and modifying menu by typing the letter a and pressing return. The menu, controlling the entering and modifying module is then displayed on the screen:

```
Do you want to

A  Enter the bond graph, either by entering each element separatoly from
   the keyboard, by reading the bond graph or bond graph fragment from a data
   file or by connecting existing bond graphs elements or fragments.

B  Display all the bonds currently connected.

C  Delete all the bonds in the bond list.

D  Modify the variables of a bond.

E  Delete a bond from the bond list.

F  Undo the most recent bond deletion.

G  Leave the bond graph entering routine.

Press the key corresponding to your choice and hit enter
```

Assuming that the bond graph in Figure C.1 is to be entered from keyboard, the user would

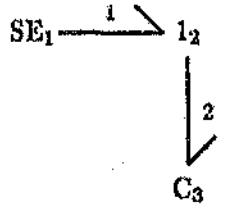


Figure C.1: A bond graph fragment.

proceed as follows.

The user would type the letter a followed by return to indicate that a bond graph is to be entered. The menu with the choice of whether the bond graph is to be entered from the keyboard or from a file is then displayed.

```
Do you want to

A  Enter the bonds and elements separately from the keyboard

B  Include an existing bond graph

Press the desired key and hit return
```

Because the bond graph is to be entered from the keyboard, the user would type a followed by return.

The user is prompted to enter a bond number

```
Enter the bond number
```

The user can enter either bond first. Considering bond 1 first, the user enters 1 and presses return.

The user is then prompted to enter the number of the element from which the power leaves.

```
Enter the number of the element at the start of the power half arrow
```

The user must enter 1 and press return. The user is now asked to enter the type of element 1

```
Enter the element type
```

The user enters se and presses return. Next the user is prompted to enter the documentation string corresponding to element 1.

```
Enter the documentation string for this element
```

The user enters Effort source attached to bond 1. Note that any documentation string could have been entered, in this example, an arbitrary string was chosen to demonstrate the procedure.

Next the user is prompted to enter the number of the element into which the power flows.

```
Enter the number of the element at the end of the power half arrow
```

The user enters 2. The user is now asked to enter the type of element 2

```
Enter the element type
```

The user enters 1. Next the user is prompted to enter the documentation string corresponding to element 2.

```
Enter the documentation string for this element
```

The user enters 1-junction attached to bonds 1 and 2.

The user is then asked if all the bonds have been entered

```
Enter F to finish or any other key to continue
```

Because bond 2 must still be entered, the user must type any letter, other than f and press return.

The process is repeated for bond 2:

```
Enter the bond number
2
```

```
Enter the number of the element at the start of the power half arrow
2

This element has been previously defined
```

Because the user has entered the type and documentation string for element 2, it is not necessary to enter them again

```
Enter the number of the element at the end of the power half arrow
3

Enter the element type
c

Enter the documentation string for this element
Capacitor connected to bond 2

Enter F to finish or any other key to continue
```

The user must now enter f, because the entering of the bond graph is complete. The user will be returned to the menu controlling the entering and modifying module.

Assume now that the bond graph, shown in Figure C.2, is stored in the file inc.data in

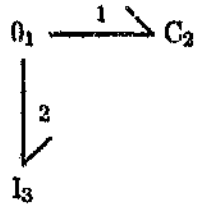


Figure C.2: The bond graph fragment stored in inc.data.

the directory /usr/bondgraph/data. To add the bond graph fragment to the bond graph already entered, the user proceeds as follows.

The user must now enter a. As before, the user is given the choice of entering the bond graph from the keyboard or from a file. Now the user is required to enter b. The user is then prompted to enter the filename of the file where the bond graph is stored

```
Enter the filename of the bond graph you want to include
```

The user must enter inc.data. Control is then returned to the main entering and modifying menu. As discussed in previously, when a bond graph, contained in a file is included in an existing bond graph structure, the bond and element numbers of the bond graph in the file are changed so as not to overlap with existing bond and element numbers. Because of this it is necessary to view the entire structure, this is achieved by entering the letter b.

The bond graph entered thus far is:

```
Bond 4 is #S(LINE NODE-PAIR (4 6) CAUSALITY NIL)
Element 4 is #S(NODE TYPE 0 DOC-STR A 0-junction EQUATIONS NIL)
Element 6 is #S(NODE TYPE I DOC-STR An inertia EQUATIONS NIL)

Bond 3 is #S(LINE NODE-PAIR (4 5) CAUSALITY NIL)
Element 4 is #S(NODE TYPE 0 DOC-STR A 0-junction EQUATIONS NIL)
Element 5 is #S(NODE TYPE C DOC-STR A capacitor EQUATIONS NIL)
```

```
Bond 2 is #S(LINE NODE-PAIR (2 3) CAUSALITY NIL)
Element 2 is #S(NODE TYPE 1 DOC-STR 1-junction attached to bonds 1 and 2
EQUATIONS NIL)
Element 3 is #S(NODE TYPE C DOC-STR Capacitor connected to bond 2
EQUATIONS NIL)

Bond 1 is #S(LINE NODE-PAIR (1 2) CAUSALITY NIL)
Element 1 is #S(NODE TYPE SE DOC-STR Effort source attached to bond 1
EQUATIONS NIL)
Element 2 is #S(NODE TYPE 1 DOC-STR 1-junction attached to bonds 1 and 2
EQUATIONS NIL)
```

The structure is viewed using the UNIX "more" command [26]. The screen can be scrolled line by line, by pressing the return key, or can be scrolled page by page by pressing the space bar. When the user has completed viewing the entire structure, the user is required to press enter to acknowledge this. Control is then transferred to the main menu.

Assuming that the 1-junction (element 2) and 0-junction (element 4) are to be connected, the user follows the same procedure for entering the bond graph from the keyboard:

```
Do you want to

A Enter the bond graph, either by entering each element separately from
the keyboard, by reading the bond graph or bond graph fragment from a
data file or by connecting existing bond graphs elements or fragments.

B Display all the bonds currently connected.

C Delete all the bonds in the bond list.

D Modify the variables of a bond.

E Delete a bond from the bond list.

F Undo the most recent bond deletion

G Leave the bond graph entering routine.

Press the key corresponding to your choice and hit enter a


Do you want to

A Enter the bonds and elements separately from the keyboard

B Include an existing bond graph

Press the desired key and hit return a


Enter the bond number
5

Enter the number of the element at the start of the power half arrow
2

This element has been previously defined

Enter the number of the element at the end of the power half arrow
4
```

```
This element has been previously defined

Enter F to finish or any other key to continue f
```

Viewing the structure again

```
Bond 5 is #S(LINE NODE-PAIR (2 4) CAUSALITY NIL)
Element 2 is #S(NODE TYPE 1 DOC-STR 1-junction attached to bonds 1 and 2
EQUATIONS NIL)
Element 4 is #S(NODE TYPE 0 DOC-STR A 0-junction EQUATIONS NIL)

Bond 4 is #S(LINE NODE-PAIR (4 6) CAUSALITY NIL)
Element 4 is #S(NODE TYPE 0 DOC-STR A 0-junction EQUATIONS NIL)
Element 6 is #S(NODE TYPE I DOC-STR An inertia EQUATIONS NIL)

Bond 3 is #S(LINE NODE-PAIR (4 5) CAUSALITY NIL)
Element 4 is #S(NODE TYPE 0 DOC-STR A 0-junction EQUATIONS NIL)
Element 5 is #S(NODE TYPE C DOC-STR A capacitor EQUATIONS NIL)

Bond 2 is #S(LINE NODE-PAIR (2 3) CAUSALITY NIL)
Element 2 is #S(NODE TYPE 1 DOC-STR 1-junction attached to bonds 1 and 2
EQUATIONS NIL)
Element 3 is #S(NODE TYPE C DOC-STR Capacitor connected to bond 2
EQUATIONS NIL)

Bond 1 is #S(LINE NODE-PAIR (1 2) CAUSALITY NIL)
Element 1 is #S(NODE TYPE SE DOC-STR Effort source attached to bond 1
EQUATIONS NIL)
Element 2 is #S(NODE TYPE 1 DOC-STR 1-junction attached to bonds 1 and 2
EQUATIONS NIL)
```

The bond graph is therefore that shown in Figure C.3.

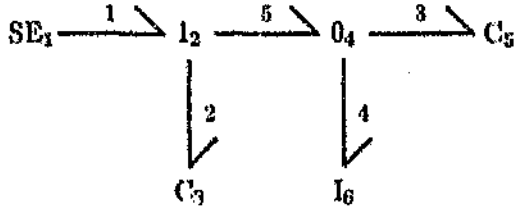


Figure C.3: The complete bond graph.

Assume that for some or other reason, the capacitor attached to bond 2, should have been a resistor. Bond 2 must be removed (which will remove the capacitor as well) and replaced with a bond attached to a resistor (the bond can also be numbered 2) as shown.

```
Do you want to

A Enter the bond graph, either by entering each element separately from
the keyboard, by reading the bond graph or bond graph fragment from a
data file or by connecting existing bond graphs elements or fragments.

B Display all the bonds currently connected.
```

```
C Delete all the bonds in the bond list.

D Modify the variables of a bond.

E Delete a bond from the bond list.

F Undo the most recent bond deletion.

G Leave the bond graph entering routine.

Press the key corresponding to your choice and hit enter e


Enter the bond you want to delete [e.g.  (6 13)]
(2 3)

Are you sure ?
y

used-elements (2 4 6 5 1) start-node 2 end-node 3


Do you want to

A Enter the bond graph, either by entering each element separately from
the keyboard, by reading the bond graph or bond graph fragment from a
data file or by connecting existing bond graphs elements or fragments.

B Display all the bonds currently connected.

C Delete all the bonds in the bond list.

D Modify the variables of a bond.

E Delete a bond from the bond list.

F Undo the most recent bond deletion.

G Leave the bond graph entering routine.

Press the key corresponding to your choice and hit enter a


Do you want to

A Enter the bonds and elements separately from the keyboard

B Include an existing bond graph

Press the desired key and hit return a


Enter the bond number
2

Enter the number of the element at the start of the power half arrow
2

This element has been previously defined

Enter the number of the element at the end of the power half arrow
3

Enter the element type
r
```

```
Enter the documentation string for this element
A resistor, which replaces the capacitor

Enter F to finish or any other key to continue f


Do you want to

A Enter the bond graph, either by entering each element separately from
the keyboard, by reading the bond graph or bond graph fragment from a
data file or by connecting existing bond graphs elements or fragments.

B Display all the bonds currently connected.

C Delete all the bonds in the bond list.

D Modify the variables of a bond.

E Delete a bond from the bond list.

F Undo the most recent bond deletion.

G Leave the bond graph entering routine.

Press the key corresponding to your choice and hit enter b

Bond 2 is #S(LINE NODE-PAIR (2 3) CAUSALITY NIL)
Element 2 is #S(NODE TYPE 1 DOC-STR 1-junction attached to bonds 1 and 2
EQUATIONS NIL)
Element 3 is #S(NODE TYPE R DOC-STR A resistor, which replaces the capacitor
EQUATIONS NIL)

Bond 5 is #S(LINE NODE-PAIR (2 4) CAUSALITY NIL)
Element 2 is #S(NODE TYPE 1 DOC-STR 1-junction attached to bonds 1 and 2
EQUATIONS NIL)
Element 4 is #S(NODE TYPE 0 DOC-STR A 0-junction EQUATIONS NIL)

Bond 4 is #S(LINE NODE-PAIR (4 6) CAUSALITY NIL)
Element 4 is #S(NODE TYPE 0 DOC-STR A 0-junction EQUATIONS NIL)
Element 6 is #S(NODE TYPE I DOC-STR An inertia EQUATIONS NIL)

Bond 3 is #S(LINE NODE-PAIR (4 5) CAUSALITY NIL)
Element 4 is #S(NODE TYPE 0 DOC-STR A 0-junction EQUATIONS NIL)
Element 5 is #S(NODE TYPE C DOC-STR A capacitor EQUATIONS NIL)

Bond 1 is #S(LINE NODE-PAIR (1 2) CAUSALITY NIL)
Element 1 is #S(NODE TYPE SE DOC-STR Effort source attached to bond 1
EQUATIONS NIL)
Element 2 is #S(NODE TYPE 1 DOC-STR 1-junction attached to bonds 1 and 2
EQUATIONS NIL)

Press ENTER to continue
```

Control is then returned to the main entering and modifying menu. The bond graph contained in the data structure is shown in Figure C.4.

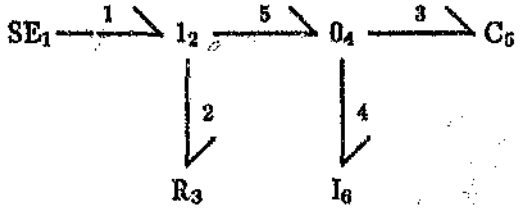


Figure C.4: The final acausal bond graph.

Assuming that the bond graph is now correct, the entering and modifying module can be exited. This is done by entering the letter g. The user is then prompted to enter the filename in which to store the bond graph structure.

```
Enter the filename in which to store the hash tables, do not include an extension,
it will be set to '.data'
```

The user should not enter an extension, as this will automatically be set to .data. The file will be stored in the directory /usr/bondgraph/data. To store the structure in the file /usr/bondgraph/data/bg1.data the user must enter bg1. The user will now be returned to the main menu.

Notes There are a few other options available in the entering an modifying module, which will not be demonstrated here as they are relatively obvious, they are:

1. The user can change the start and end element numbers. This is necessary if a bond is accidentally connected to the incorrect element.
2. The user can delete all the bonds and elements from the bond graph structure.
3. If a bond is accidentally deleted, the most recently deleted bond can be re-inserted in the structure.

C.3 Assigning causality and viewing the causal implications

The user is able to enter the causality assignment routine by entering b from the main menu. The user is now given three options, to assign the causality, to view the hash tables in which the bond graph structure is stored or to view the causal implications:

```
Do you want to

A Assign causality to a bond graph.

B View the causal implications.

C View the bond graph data structure

D Leave the causality assignment routine.

Press the key corresponding to your choice and hit enter
```

Assuming that the user wants to assign the causality to the bond graph stored in the file /usr/bondgraph/data/bg1.data, the user proceeds as follows:

The user enters a from the causality menu and is then asked if the file bg1.data contains the bond graph structure

```
Do you want to assign or view the causality for the bond graph stored in the file
BG1.data (Y or N)
```

The user is required to enter y. If the causality is to be assigned to any file other than the one displayed, the user is required to enter n and will then be prompted to enter the filename (again an extension must not be entered, it will be assumed to be .data).

The causality is assigned automatically:

```
Getting the first element

Assigning causality

Getting the next element in the stack

Getting the first element

Assigning causality

Getting the next element in the stack

Assigning causality

Getting the next element in the stack

Assigning causality

Getting the next element in the stack

Assigning causality

Getting the next element in the stack

DONE

Data written to file /usr/bondgraph/data/bg1.data
```

Control is then returned to the causality menu. To view the implications of the causality assigned to the bond graph the user must enter b. The causal implications are viewed using the UNIX "more" command [26]. The screen can be scrolled line by line, by pressing the return key, or can be scrolled page by page by pressing the space bar. When the user has completed viewing the implications, the user is required to press enter to acknowledge this.

```
RESISTOR (ELEMENT NUMBER 3) CONNECTED TO
BOND 2

The causality assigned implies that
f2 is a function of e2

EFFORT SOURCE (ELEMENT NUMBER 1) CONNECTED TO
BOND 1
```

```
The causality assigned implies that e1 is the output

CAPACITOR (ELEMENT NUMBER 5) CONNECTED TO
BOND 3

The causality assigned implies that
e3 is a function of q3

INERTIA (ELEMENT NUMBER 6) CONNECTED TO
BOND 4

The causality assigned implies that
f4 is a function of p4
```

Once the user has pressed enter when prompted to do so, the user is returned to the causality menu. If the user wishes to view the bond graph hash tables, the user must enter c. This function is the same as that provided in the entering and modifying module and its output will not be shown here. To leave the causality assignment module, the user must enter d. Control is then returned to the main menu.

C.4 Creating the ACSL input file

The module used to construct the ACSL input file can be invoked from the main menu by entering c.

This module asks the user to enter whether or not the data file currently being used is the one for which the ACSL input file must be formed.

```
Do you want to get the acsl equations of the bond graph stored in the file
BG1.data (Y or N)
```

Assuming that the file /usr/bondgraph/data/bg1.data contains the bond graph structure to be simulated, the user must enter y. If an ACSL input file is to be formed from any file other than the one displayed, the user is required to enter n and will then be prompted to enter the filename (again an extension must not be entered, it will be assumed to be .data).

The user is required to enter the ACSL program name

```
Enter the acsl program name.
```

In this example, the ACSl program can be given any arbitrary name for example, A bond graph model.

The user is then required to enter the integration step size

```
Enter the integration step size, ccalc
```

It is assumed that an integration step length equal to 0.01 seconds is known to be sufficiently accurate. This is then entered. The user is then required to enter the duration of the simulation.

```
Enter the duration of the simulation, simtim.
```

It is assumed that the user is only interested in the behaviour for the first 75 seconds. The user then enters 75.

The user is then required to enter the data for resistor 2.

```
RESISTOR (ELEMENT NUMBER 3) CONNECTED TO
BOND 2

The documentation for this element is:
"A resistor, which replaces the capacitor"

Enter the data for resistor R2.
If the equation is linear, enter only the resistance.
If the equation is non-linear enter the entire equation with
f2 as a function of e2
If the data is in the form of a table enter "table"
```

Assuming resistor 2 is governed by a linear relation of the form $e_2 = 2.0 f_2$, it is only necessary to enter the resistance, 2.0.

The user is then required to enter the data for capacitor 3.

```
CAPACITOR (ELEMENT NUMBER 5) CONNECTED TO
BOND 3

The documentation for this element is:
"A capacitor"

Enter the data for capacitor C3.
If the equation is linear, enter only the capacitance.
If the equation is non-linear enter the entire equation with
e3 as a function of q3
If the data is in the form of a table, enter "table"
```

Assuming capacitor 3 is governed by a non-linear relation of the form $e_3 = \frac{1.0}{3.0} q_3^2$, it is necessary to enter the entire relation, e3=1.0/3.0*q3*q3.

The user is required to enter the initial value of the energy variable for each storage element with integral causality. For the capacitor:

```
Enter the value of q3 at time t=0
```

Assuming the the initial value of q3, q3i to be zero, the user enters 0.0.

The remaining elements are entered in the same way.

```
EFFORT SOURCE (ELEMENT NUMBER 1) CONNECTED TO
BOND 1

The documentation for this element is:
"Effort source"

Enter the data for effort source SE1.
If the effort input is a step at t=0 then enter only the height of the step.
If the effort input is anything but a step at t=0 then enter the entire
equation, using the form e1=.....
If the data is in the form of a table enter "table"
1.0
```

```
INERTIA (ELEMENT NUMBER 6) CONNECTED TO
BOND 4

The documentation for this element is:
"An inertia"

Enter the data for inertia I4.
If the equation is linear, enter only the inertance.
If the equation is non-linear enter the entire equation with
f4 as a function of p4
If the data is in the form of a table, enter "table"
4.0

Enter the value of p4 at time t=0
0.0

Acsl input file dumped to the file /usr/bondgraph/acslfile/bg1.acsl

Hash table written to file /usr/bondgraph/data/bg1.data
```

The ACSL input file created is:

```
program A bond graph model
initial
constant points=1000
constant ccalc=0.01
constant simtim=75
simend=simtim-ccalc
cint=simtim/points
nstp=cint/ccalc
constant i4=4.0
constant p4i=0.0
constant q3i=0.0
constant e1=1.0
constant r2=2.0
end $"initial"
dynamic
derivative
f4=(1.0/i4)*p4
p4=integ(e4,p4i)
e3=1.0/3.0*q3*q3
q3=integ(f3,q3i)
e5=e3
e4=e3
f3=+f5-f4
f5=f2
f1=f2
e2=-e5+e1
f2=(1.0/r2)*e2
termt(t.gt.simend)
end $"derivative"
end $"dynamic"
terminal
```

```
end $"terminal"
end $"program"
```

Notes
If the equations of the system had been entered previously, the user would be given the option of specifying whether the equations are still valid, if they are, it is not necessary to re-enter them. The user also has the option of entering the data for each element in the form of a table. Neither of these two options are demonstrated here as they are clearly shown in Appendix B.

C.5 Running ACSL

The module used to run ACSL from the modelling package, can be invoked from the main menu by entering d.

This module prompts the user to enter whether the ACSL input file, just created, must be used as the input to ACSL.

```
Do you want to simulate the system stored in the file
BG1.acsl (Y or N)
```

The input file is compiled and then executed. The modelling package does not construct a ACSL command file, but instead the user is required to enter the commands from the command line. This is clearly shown in the ACSL manual [1].

The ACSL plot of effort 2 as a function of time is shown in Figure C.5.

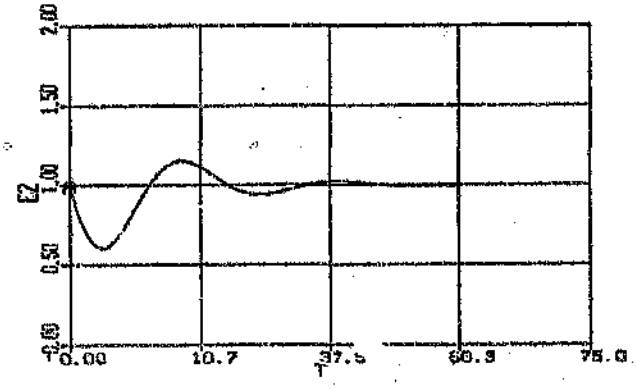


Figure C.5: Effort 2 as a function of time.

The ACSL plot of displacement 3 as a function of time is shown in Figure C.6.

The ACSL plot of momentum 4 as a function of time is shown in Figure C.7.

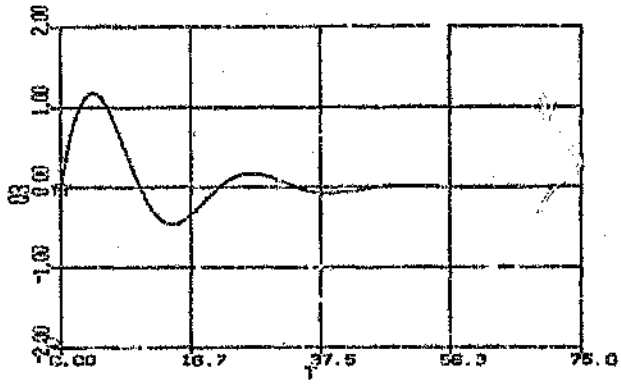


Figure C.6: Displacement 3 as a function of time.

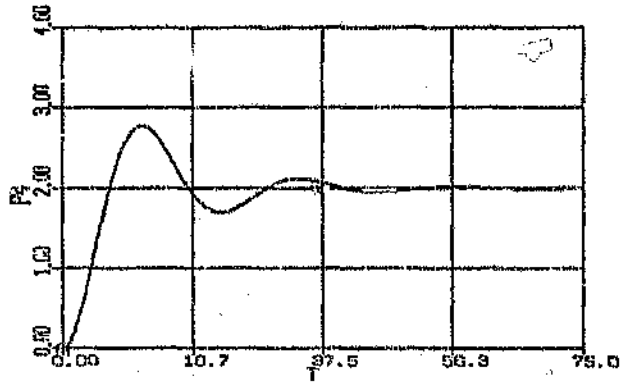


Figure C.7: Momentum 4 as a function of time.

Appendix D

Bond graph program Lisp code

D.1 The bond graph program controlling module (bond.lisp)

```
(defun run-bond-graph-programs ()
  (load-bond-graph-programs)
  (loop
   (format t "~|Do you want to~%~%A  Enter or modify the bond graph structure~
           ~%~%B  Assign or view the causality~%~%C  Create an ACSL input file~
           ~%~%D  Run ACSL~%~%E  Quit the program~%~%Press the key ~
           corresponding to your choice and hit enter  ")
   (setf ans (read))
   (cond ((equal ans 'a)
          (enter-bond-graph))
         ((equal ans 'b)
          (causality-main-function))
         ((equal ans 'c)
          (dump-acsl-equations))
         ((equal ans 'd)
          (run-acsl))
         (t (format t "~%~%Thanks for using the program")
            (return)))))


(defun load-bond-graph-programs ()
  (format t "~|Loading the necessary programs, won't be a second~%")
  (load "enter.lisp")
  (load "shell.lisp")
  (load "causal.lisp")
  (load "bondacsl.lisp")
  (load "run-acsl.lisp"))

(run-bond-graph-programs)
```

D.2 The entering and modifying module (enter.lisp)

```
;-------------------------------------------------------------------------
;-------------------------------------------------------------------------
;                    Bond Graph Entering Module
;-------------------------------------------------------------------------
;-------------------------------------------------------------------------
;                    Written by: A.L. Stevens
;                    Modified and added to by: C.J. Hartmann
;                    Modified and added to by: G.B. Grobbelaar
;                    Last modified on: 07/06/91
;-------------------------------------------------------------------------
;-------------------------------------------------------------------------

;;; The aim of this module is to allow the user to enter or modify the bond
;;; graph structure. The user is able to enter the bond graph either from the
;;; keyboard or the bond graph can be read in from a file. It is also possible
;;; to use a combination of both methods, this is particularly useful and
;;; necessary if the system model is constructed of numerous bond graph
;;; building blocks. The user also has the ability to move bonds around and
;;; to delete bonds and elements. A function also exists which allows the
;;; user to view the two hash tables.

;;; The data file that is written by this module is of the form:
;;; The first line contains the statement 'causality-not-assigned, stating
;;; the the causality is not assigned by this module, and the causality that
;;; was assigned previously is probably not correct anymore.
;;; The second line contains the global variable *line-hash-keys*
;;; The next couple of lines contain the hash table *lines*. *lines* takes
;;; up one line for each bond in the hash table.
;;; Following the hash table *lines* is the global variable *node-hash-keys*.
;;; The next couple of lines contain the hash table *nodes*. *nodes* takes
;;; up one line for each bond graph element in the hash table.

(defvar *filename* nil)                  ; A global variable containing the
                                         ; filename, without the path, of the
                                         ; most recently read bond graph data
                                         ; file.
(defvar *lines* nil)                     ; The hash table of the bonds, contains
                                         ; the element numbers at either end of
                                         ; the bond and the causality assigned
                                         ; to the bond.
(defvar *line-hash-keys* nil)            ; The list of bond numbers by which the
                                         ; hash table is referenced.
(defvar *nodes* nil)                     ; The hash table containing all the
                                         ; bond graph elements, contains the
                                         ; element type, the documentation
                                         ; string and the equations for that
                                         ; element.
(defvar *node-hash-keys* nil)            ; A list of element numbers, used to
                                         ; reference the hash table *nodes*.
(defvar *temp-line-hash-keys* nil)       ; This global variable is the same as
                                         ; *line-hash-keys*, but is used when
                                         ; the data is read in from a file,
                                         ; once the elements and bonds have
                                         ; been renumbered, this is added on
```

```
                                        ; to *line-hash-keys*.
(defvar *temp-node-hash-keys* nil)      ; This global variable is the same as
                                        ; *node-hash-keys*, but is used when
                                        ; the data is read in from a file,
                                        ; once the elements and bonds have
                                        ; been renumbered, this is added on
                                        ; to *node-hash-keys*.
(defvar *temp-lines* nil)               ; This hash table is the same as
                                        ; *lines*, but is used when
                                        ; the data is read in from a file,
                                        ; once the elements and bonds have
                                        ; been renumbered, this is added on
                                        ; to *lines*.
(defvar *temp-nodes* nil)               ; This hash table is the same as
                                        ; *nodes*, but is used when
                                        ; the data is read in from a file,
                                        ; once the elements and bonds have
                                        ; been renumbered, this is added on
                                        ; to *nodes*.

(defstruct line

;;; This lisp structure is referenced by the list *line-hash-keys*. The hash
;;; table contains the two elements that the bond joins in node-pair. The
;;; first element in node-pair is the element at the end of the bond without
;;; the power half arrow and the second element in node-pair is the element
;;; that the power is entering. The causality can either be nil, e or b. It is
;;; nil is no causality has been assigned, e is the causal stroke is at the
;;; same end as the power half arrow or b if the causal stroke is at the
;;; opposite end to the power half arrow.

  (node-pair '(0 0))
  (causality nil))

(defstruct node

;;; This list structure contains the data for each bond graph element and is
;;; referenced by *node-hash-keys*. The hash table contains three entries,
;;; the element type, the documentation string for the bond graph element and
;;; the equations for the element. The equations slot is nil if no equations
;;; have yet been obtained.

  (type nil)
  (doc-str nil)
  (equations nil))


(defun reset-enter ()
  (clrhash *lines*)
  (clrhash *nodes*))


(defun init-enter ()

;;; Both the hash table's will be cleared and a new one created
```

```
  (setq *lines* (make-hash-table))
  (setq *nodes* (make-hash-table))
  (reset-enter)
  (setq *line-hash-keys* nil
        *node-hash-keys* nil
        *directed-lines* nil))


(defun read-a-line ()

;;; This function is called by add-new-line and prompts the user to enter the
;;; bond number, and the two elements to which the bond is connected. The
;;; elements are entered in the order of the direction of power flow, i.e. the
;;; first element number entered corresponds to the element that the power is
;;; leaving and the second element corresponds to the element the power is
;;; entering.

  (let ((line-number nil)
        (this-line-struc nil)
        (s-n-n nil)
        (e-n-n nil))
    (   ;
      ormat t "~%~%Enter the bond number ~%")
     (setq line-number (read))
     (if (member line-number *line-hash-keys*)
         (format t "~%This bond already exists, please re-enter~%")
         (return)))
    (setq this-line-struc line-number)
    (setq this-line-struc (make-line))
    (format t "~%Enter the number of the element at the start of the power "
            half arrow~%")
    (setq s-n-n (read))
    (node-information-read s-n-n)
    (format t "~%Enter the number of the element at the end of the power "
            half arrow~%")
    (setq e-n-n (read))
    (node-information-read e-n-n)
    (setf (line-node-pair this-line-struc)
          (list s-n-n e-n-n))
    (values line-number this-line-struc)))


(defun node-information-read (node)

;;; This function is called by read-a-line and prompts the user to enter
;;; the element type and the documentation string for the element. The
;;; function writes this information to the hash table *nodes*. If the element
;;; has been defined previously, it is not necessary to re-enter the data.

  (let ((allowed-node-types '(ac sf i c r tr gy 1 0)))
    (if (member node *node-hash-keys*)
        (format t "~%This element has been previously defined~%")
        (let ((this-node-struct (make-node)))
          (setf *node-hash-keys* (cons node *node-hash-keys*))
          (loop
```

```
          (format t "~%Enter the element type~%")
          (setf (node-type this-node-struct) (read))
          (if (not (member (node-type this-node-struct) allowed-node-types))
              (format t "~%This is not a valid bond graph element, please ~
                      re-enter~%")
              (return)))
        (format t "~%Enter the documentation string for this element~%")
        (setf (node-doc-str this-node-struct) (read-line))
        (setf (gethash node *nodes*) this-node-struct))
    (values)))


(defun add-new-line ()

;;; This function adds a new bond to the hash table *lines* and adds the bond
;;; to the variable *line-hash-keys*

  (multiple-value-bind (line-number new-line) (read-a-line)
    (setf *line-hash-keys* (cons line-number *line-hash-keys*)) ;a LIFO list.
    (setf (gethash line-number *lines*) new-line))
  (values))


(defun view-nodes ()

;;; This function prints each entry in the hash table *nodes* to the screen

  (dolist (node-number *node-hash-keys*)
    (format t "~%Element ~a is ~a~%" node-number (gethash node-number *nodes*)))
  (values))


(defun view-lines ()

;;; This function prints each entry in the hash table *lines* to the screen

  (dolist (line-number *line-hash-keys*)
    (format t "~%Bond ~a is ~a~" line-number (gethash line-number *lines*)))
  (values))


(defun view ()

;;; This function prints the contents of the hash tables *nodes* and
;;; *lines* to the screen. The contents of both hash tables is written to a
;;; file, then the contents of the file is displayed on the screen using the
;;; unix command "more". "more" is used to control the scrolling of the file
;;; contents.

  (with-open-file (a-stream "/usr/bondgraph/data/view.output"
                            :direction :output :if-exists :supersede)
                  (dolist (line-number *line-hash-keys*)
                    (let* ((node-pair (line-node-pair
                                       (gethash line-number *lines*)))
                           (start-node (first node-pair))
```

```
                        (end-node (second node-pair)))
                    (format a-stream "~%~%Bond ~a is ~a"
                            line-number (gethash line-number *lines*))
                    (format a-stream "~%Element ~a is ~a"
                            start-node (gethash start-node *nodes*))
                    (if (eq start-node end-node)
                        (format a-stream "~%End element is the same as "
                                start element")
                        (format a-stream "~%Element ~a is ~a" end-node
                                (gethash end-node *nodes*))))))
  (run-unix-program "more" :input "/usr/bondgraph/data/view.output")
  (format t "~%~%Press ENTER to continue~%")
  (read-char)
  (values))


(defun write-hash-table-to-file-enter (filename)

;;; This function writes the hash tables *lines* and *nodes* to a file
;;; *filename*.data. In addition to the hash tables, *line-hash-keys* and
;;; *node-hash-keys*, which reference the hash tables are also written to the
;;; file, as is a statement, informing the modules that the causality has not
;;; been assigned.

  (setq filename (concatenate 'string "/usr/bondgraph/data/"
                              (string-downcase filename) ".data"))
  (format t (concatenate 'string "~%Data written to file " filename "~%"))
  (with-open-file (line-stream filename
                               :direction :output)
                  (print 'causality-not-assigned line-stream)
                  (print *line-hash-keys* line-stream)
                  (dolist (line-number *line-hash-keys*)
                    (print (gethash line-number *lines*) line-stream))
                  (print *node-hash-keys* line-stream)
                  (dolist (node *node-hash-keys*)
                    (print (gethash node *nodes*) line-stream))))


(defun read-in-bond-graph ()

;;; This function is called when the user wants to include a bond graph that
;;; has previously been written to a file. The bond graph structure is read
;;; in by the function read-data-from-file-enter, which sets the two global
;;; lists *temp-line-hash-keys* and *temp-node-hash-keys* and reads the two hash
;;; tables into *temp-lines* and *temp-nodes*. If *line-hash-keys* is not nil
;;; i.e. part of a bond graph has already been entered then the functions
;;; greatest-line and greatest-node are called, which find the greatest bond
;;; number in *line-hash-keys* and the greatest element number in
;;; *node-hash-keys* respectively. These two number are both incremented by
;;; one so that none of the new bond and element numbers will overlap with the
;;; existing bond and element numbers. All the new bonds and elements are now
;;; renumbered sequentially i.e. if there are four new bonds and the largest
;;; number in *line-hash-keys* is 7, then the new bonds will be numbered 8, 9,
;;; 10 and 11. The new hash tables are then included in the existing hash
;;; tables and the newly renumbered bond and element numbers are included in
;;; *line-hash-keys* and *bond-hash-keys* respectively. The variable node-pair
```

```
;;; in the structure *lines* is then changed to take into account the new
;;; element numbers. If originally *line-hash-keys* had the value nil, i.e.
;;; none of the bond graph had already been entered, then *line-hash-keys*,
;;; *node-hash-keys*, *lines* and *nodes* take on the values of
;;; *temp-line-hash-keys*, *temp-node-hash-keys*, *temp-lines* and
;;; *temp-nodes* respectively.

  (let ((greatest-line 1)
        (greatest-node 1)
        (replace-temp-node nil)
        (replace-temp-line nil))
    (format t "~|Enter the filename of the bond graph you want to include~%")
    (read-data-from-file-enter (read))
    (if *line-hash-keys*
        (progn (setf greatest-line (+ 1 (find-greatest-number-in-list
                                          *line-hash-keys*)))
               (setf greatest-node (+ 1 (find-greatest-number-in-list
                                          *node-hash-keys*)))
               (dolist (anum *temp-line-hash-keys*)
                 (setf replace-temp-line (cons greatest-line replace-temp-line))
                 (setf greatest-line (+ 1 greatest-line)))
               (dolist (anum *temp-node-hash-keys*)
                 (setf replace-temp-node (cons greatest-node replace-temp-node))
                 (setf greatest-node (+ 1 greatest-node)))
               (dotimes (pos (length *temp-line-hash-keys*))
                 (setf (gethash (nth pos replace-temp-line) *lines*)
                       (gethash (nth pos *temp-line-hash-keys*) *temp-lines*)))
               (dotimes (pos (length *temp-node-hash-keys*))
                 (setf (gethash (nth pos replace-temp-node) *nodes*)
                       (gethash (nth pos *temp-node-hash-keys*) *temp-nodes*)))
               (setf *line-hash-keys* (append replace-temp-line
                                              *line-hash-keys*))
               (setf *node-hash-keys* (append replace-temp-node
                                              *node-hash-keys*))
               (dolist (a-key replace-temp-line)
                 (setf s-n-n (nth (position (first (line-node-pair
                                                     (gethash a-key *lines*)))
                                            *temp-node-hash-keys*)
                                  replace-temp-node))
                 (setf e-n-n (nth (position (second (line-node-pair
                                                      (gethash a-key *lines*)))
                                            *temp-node-hash-keys*)
                                  replace-temp-node))
;;; These last two statements find the position of the old node number in
;;; the old list of node numbers. The function then replaces the old node number with
;;; the new node number from the new list of nodes (they occur in the same
;;; place in the list) i.e.
;;; If one enters a bond graph it completely renumbers it so that the node
;;; and bond numbers do not coincide, but the hash table *lines* contains
;;; the node pair. This node pair has to be renumbered. s-n-n is the new
;;; start-node-number, e-n-n is the new end-node-number
                 (setf a-struc a-key)
                 (setf a-struc (make-line))
                 (setf (line-node-pair a-struc)
                       (list s-n-n e-n-n))
                 (setf (gethash a-key *lines*) a-struc)))
```

```
        (progn (setf *line-hash-keys* *temp-line-hash-keys*)
               (setf *node-hash-keys* *temp-node-hash-keys*)
               (dolist (a-line-number *line-hash-keys*)
                 (setf (gethash a-line-number *lines*)
                       (gethash a-line-number *temp-lines*)))
               (dolist (a-node-number *node-hash-keys*)
                 (setf (gethash a-node-number *nodes*)
                       (gethash a-node-number *temp-nodes*)))))))


(defun find-greatest-number-in-list (a-list)

;;; This function searches through a list and finds the largest number
;;; contained in it.

  (let ((greatest-number (first a-list)))
    (dolist (a-number a-list)
     (if (> a-number greatest-number)
         (setf greatest-number a-number)))
    (values greatest-number)))


(defun reset-temp-hash-tables ()

;;; This function resets the hash tables *temp-lines* and *temp-nodes*

  (setq *temp-lines* (make-hash-table))
  (setq *temp-nodes* (make-hash-table))
  (clrhash *temp-lines*)
  (clrhash *temp-nodes*))


(defun read-data-from-file-enter (filename)

;;; This function reads in a bond graph stored in a file. The two hash tables
;;; read in are assigned to *temp-lines* and *temp-nodes*, the bond and
;;; element numbers which reference the two hash tables are also read in, they
;;; are assigned to *temp-line-hash-keys* and *temp-node-hash-keys*
;;; respectively.

  (reset-temp-hash-tables)
  (setq filename (concatenate 'string "/usr/bondgraph/data/"
                              (string-downcase filename)))
  (with-open-file (line-stream filename :direction :input)
                  (setf causality-assigned? (read line-stream))
                  (setf *temp-line-hash-keys* (read line-stream))
                  (setf line-keys *temp-line-hash-keys*)
                  (dolist (a-line-number line-keys)
                    (setf (gethash a-line-number *temp-lines*)
                          (read line-stream)))
                  (setf *temp-node-hash-keys* (read line-stream))
                  (setf node-keys *temp-node-hash-keys*)
                  (dolist (a-node-number node-keys)
                    (setf (gethash a-node-number *temp-nodes*)
                          (read line-stream)))))
```

```
  (values))


(defun delete-line ()

;;; This function allows the user to remove a bond from the hash table
;;; *lines*. The bond is then written to a file line-del.data so that it can
;;; be "undone" if necessary. If there are elements, which are no longer
;;; connected to other bonds, then these too are removed and placed in a file
;;; node-del.data.

  (format t "~% Enter the bond you want to delete [e.g. (6 13)]~%")
  (let ((line-to-be-deleted (read)))
    (format t "~% Are you sure ?~%")
    (when (eq (read) 'y)
      (multiple-value-bind (line-hash-key position)
                           (get-line-hash-key-from-node-pair line-to-be-deleted)
          ((lambda ()
;;; Write the bond to a file for later "undoing".
          (with-open-file (del-stream "/usr/bondgraph/data/line-del.data"
                                      :direction :output)
                          (print line-hash-key del-stream)
                          (print (gethash line-hash-key *lines*) del-stream))
          (remhash line-hash-key *lines*)
          (setf *line-hash-keys*
                (delete (get-nth-element position *line-hash-keys*)
                        *line-hash-keys* :test #'equal))))))
   (get-directed-lines)
;;; Now we must test if the nodes are still used by other lines. If both the
;;; nodes are not used by other lines then the information for both nodes is
;;; written into on file, preceded by the number two, while if only one of the
;;; nodes is not used it is written into the file the first entry will be 1.
    (let ((used-nodes (get-number-list *directed-lines*))
          (start-node (first line-to-be-deleted))
          (end-node (second line-to-be-deleted)))
      (format t "used-elements ~a start-node ~a end-node ~a~" used-nodes
              start-node end-node)
      (if (subsetp line-to-be-deleted used-nodes)
;;; The file containing the a previous node information must be deleted to
;;; prevent the reading of spurious node data with the next function call of
;;; undo.
          ((lambda ()
            (if (probe-file "/usr/bondgraph/data/node-del.data")
                (delete-file "/usr/bondgraph/data/node-del.data"))))
;;; otherwise data must be written to the file node-del.data
          (with-open-file (del-stream "/usr/bondgraph/data/node-del.data"
                                      :direction :output)
                          (cond
                           ((not (intersection line-to-be-deleted used-nodes))
                            (print '2 del-stream)
                            (print-delete-node start-node del-stream)
                            (print-delete-node end-node del-stream))
                           ((not (member start-node used-nodes))
                            (print '1 del-stream)
                            (print-delete-node start-node del-stream))
                           ((not (member end-node used-nodes))
```

```
                        (print '1 del-stream)
                        (print-delete-node end-node del-stream)))))))
  (values))


(defun print-delete-node (node del-stream)

;;; This function prints the node information to the stream del-stream and
;;; removes the information from the hash table *nodes* and removes the node
;;; number from the *node-hash-keys*.

  (print node del-stream)
  (print (gethash node *nodes*) del-stream)
  (remhash node *nodes*)
  (setq *node-hash-keys* (set-difference *node-hash-keys* (list node))))


(defun undo ()

;;; This function urdoes the most recent bond deletion. If there is
;;; information about the nodes that were deleted, then this is also read into
;;; the hash table.

  (let ((del-stream nil))
    (with-open-file (del-stream "/usr/bondgraph/data/line-del.data"
                                :direction :input)
                    (let ((line-key (read del-stream))
                          (the-line (read del-stream)))
                      (format t "~%line-key is ~a" line-key)
                      (format t "~%the-bond is ~a" the-line)
                      (setq *line-hash-keys* (cons line-key *line-hash-keys*))
                      (format t "~%*line-hash-keys* is ~a" *line-hash-keys*)
                      (setf (gethash line-key *lines*) the-line)
;;; Puts the line back in the table.
                      (get-directed-lines)))
;;; If there is node information to be entered then we will have to read the
;;; information.
     (with-open-file
     (del-stream "/usr/bondgraph/data/node-del.data"
                 :direction :input
                 :if-does-not-exist nil)
     (let ((nodes (read del-stream))
           (node1-key (read del-stream))
           (node1-info (read del-stream)))
       (format t "~% elements ~a" nodes)
       (setq *node-hash-keys* (cons node1-key *node-hash-keys*))
       (format t "~% element1-key ~a hash ~a" node1-key *node-hash-keys*)
       (setf (gethash node1-key *nodes*) node1-info)
;;; If there is the node information of two nodes the second node information
;;; is read.
       (if (eq '2 nodes)
           (let ((node2-key (read del-stream))
                 (node2-info (read del-stream)))
             (setq *node-hash-keys* (cons node2-key *node-hash-keys*))
             (setf (gethash node2-key *nodes*) node2-info)
             (format t "~% hello"))
```

```
          ())))))
   (values))


(defun modify-line ()

;;; This function allows the user to connect the bond to another element or
;;; elements.

  (format t "~%Enter the bond you wish to modify [e.g. (8 13)]~%")
  (let* ((line-number (get-line-hash-key-from-node-pair (read)))
         (print line-number)
         (the-line-being-changed (gethash line-number *lines*))
         (print line-number)
         (desired-change nil)
         (used-nodes (get-number-list *directed-lines*))
         (node-pair (line-node-pair the-line-being-changed))
         (s-n-n (first node-pair))
         (e-n-n (second node-pair)))
    (format t "~%What do you want to change?~%")
    (format t "~%Enter s-n, e-n, for start-element, end-element respectively~%")
    (setq desired-change (read))
    (case desired-change
      (s-n (format t "~%Enter the new start-element-number~%")
           (setq s-n-n (read))
           (if (member s-n-n used-nodes)
               ()
               (node-information-read s-n-n)))
      (e-n (format t "~%Enter the new end-element-number~%")
           (setq e-n-n (read))
           (if (member e-n-n used-nodes)
               ()
               (node-information-read s-n-n)))
      (otherwise (format t "~%Option not understood, try again!~%")))
    (setf (line-node-pair the-line-being-changed)
          (list s-n-n e-n-n))
    (values)))


(defun enter-bond-graph ()

;;; This function starts and controls the entering and modifying of the bond
;;; graph structure.
  (init-enter)
  (loop    .
   (options)
   (setf ans (read))
   (cond ((equal ans 'a)
          (connect))
         ((equal ans 'b)
          (view))
         ((equal ans 'c)
          (reset-enter))
         ((equal ans 'd)
          (modify-line))
         ((equal ans 'e)
```

```
        (delete-line))
       ((equal ans 'f)
        (undo))
       (t (goodbye)
          (return))))
  (val  s))


(defun goodbye ()
  (for    t "~|Enter the filename in which to store the hash tables,~%do not ~
          include an extension, it will be set to '.data'~%")
  (setf *filename* (string (read)))
  (write-hash-table-to-file-enter *filename*))


(defun connect ()

;;; This function gives the user the option of entering the bond graph from
;;; the keyboard or from a file that already exists.

  (format t "~|Do you want to
          ~%~%A  Enter the bonds and elements separately from the keyboard
          ~%~%B  Include an existing bond graph
          ~%~%Press the desired key and hit return  ")
  (if (equal (read) 'b)
      (read-in-bond-graph)
      (loop (add-new-line)
            (format t "~%Enter F to finish or any other key to continue~%")
            (when (eq 'f (read))
              (return (print 'finished)))))
  (get-directed-lines)
  (values))


(defun get-directed-lines ()

;;; This function forms a list of all the node pairs in the hash table
;;; *lines*.

  (setq *directed-lines* nil)
  ;perhaps we could do this better using MAPHASH! (see page 285)
  (dolist (line-key *line-hash-keys*
                    (setf *directed-lines* (reverse *directed-lines*)))
    (setq *directed-lines* (cons (line-node-pair (gethash line-key *lines*))
                                 *directed-lines*))))


(defun get-line-hash-key-from-node-pair (node-pair)

;;; Given the node pair (e.g. '(2 3)), this function returns the corresponding
;;; bond's hash table key from *line-hash-keys*.

  (let* ((posi (first (get-positions-of-occurrence node-pair
                                                   *directed-lines*)))
         (line-hash-key (get-nth-element posi *line-hash-keys*)))
```

```
;;; The order of values was chosen to avoid using m-v-b's when only the
;;; line-hash-key is required.
    (values line hash-key posi)))


(defun get-positions-of-occurrence (chosen-element L)

;;; This function returns a list, positions-of-occurrence, containing the
;;; positions of chosen-element in the list L.

  (let ((positions-of-occurrence nil)
        (n 0))
    (dolist (temp L (reverse positions-of-occurrence))
      (setf n (+ n 1))
      (when (equal temp chosen-element)
        (setf positions-of-occurrence
              (cons n positions-of-occurrence))))))


(defun get-nth-element (n L)

;;; This function gets the nth element of a list. Where n=1 corresponds to the
;;; first element.

  (nth (- n 1) L))


(defun get-number-list (L &aux number-list)
  (dolist (temp L (reverse number-list))
    (setq number-list
          (remove-duplicates (append (reverse temp) number-list) :test #'=))))


(defun options ()

;;; This function displays the options available in this module. The output is
;;; formatted for the 80 col vga screen.

  (format t "~| Do you want to~%")
  (format t "~% A  Enter the bond graph, either by entering each "
          element separately from~%    the keyboard, by reading the bond "
          graph or bond graph fragment from a data~%    file or by "
          connecting existing bond graphs elements or fragments.~%")
  (format t "~% B  Display all the bonds currently connected.~%")
  (format t "~% C  Delete all the bonds in the bond list.~%")
  (format t "~% D  Modify the variables of a bond.~%")
  (format t "~% E  Delete a bond from the bond list.~%")
  (format t "~% F  Undo the most recent bond deletion.~%")
  (format t "~% G  Leave the bond graph entering routine.~%")
  (format t "~% Press the key corresponding to your choice and hit enter ~%))
```

D.3 The automatic causality assignment module (causal.lisp)

```
;-----------------------------------------------------------------------------
;-----------------------------------------------------------------------------
;                    Bond Graph Causality Assignment Module
;-----------------------------------------------------------------------------
;-----------------------------------------------------------------------------
;                    Written by: Grant Grobbelaar
;                    Last modified on: 07/06/91
;-----------------------------------------------------------------------------
;-----------------------------------------------------------------------------

;;; This program assigns the causality to an unangmented bond graph. The
;;; module uses the sequential causality procedure proposed by Hood, Palmer
;;; and Daatzig, with alterations and variations making it more suitable for
;;; lisp (and computer) implementation by Grobbelaar. The state transition
;;; table is implemented using Winston's forward chaining rule-based.

;;; The data file that is written by this module is of the form:
;;; The first line contains the statement 'causality-assigned, stating
;;; the the causality is assigned by this module and is correct. The second
;;; line contains the global variable *line-hash-keys*. The next couple of
;;; lines contain the hash table *lines*. *lines* takes up one line for each
;;; bond in the hash table. Following the hash table *lines* is the global
;;; variable *node-hash-keys*. The next couple of lines contain the hash
;;; table *nodes*. *nodes* takes up one line for each bond graph element in the
;;; hash table. Following this, three lists are written, *r-algebraic-loop*,
;;; *bond-algebraic-loop* and *deriv-causality*.

;top

(defvar *r-algerbraic-loop* nil)          ; A global list with information
                                          ; about which bonds attached to
                                          ; resistors had the causality assigned
                                          ; arbitrarily, it is nil if there are
                                          ; no algebraic loops.
(defvar *bond-algerbraic-loop* nil)       ; A global list with information
                                          ; about which bonds attached junction
                                          ; structure elements had the causality
                                          ; assigned arbitrarily, it is nil if
                                          ; there are no algebraic loops.
(defvar *deriv-causality* nil)            ; A global list with information about
                                          ; bonds attached to storage elements
                                          ; that have derivative causality
                                          ; assigned to them.
(defvar *causality-assigned?* nil)        ; This global variable is not nil if
                                          ; the causality has already been
                                          ; assigned to the bond graph
(defvar *error-flag-num* 1)               ; If there is an algebraic loop or
                                          ; derivative causality, then an ACSL
                                          ; implicit statement (IMPL) is
                                          ; required. Within this statement is
                                          ; an error flag, so that the flag
                                          ; variable is not repeated, it is
                                          ; indexed. The variable is ef and
                                          ; *error-flag-num* is the
```

```
                                        ; index.
(defvar *table-var* 1)                  ; As for the error flag, if the data
                                        ; is in the form of a table, the table
                                        ; is named tab and indexed with the
                                        ; variable *table-var*.
(defvar *list-of-nodes* nil)            ; This is a global list, which is
                                        ; initially created and remains
                                        ; unaltered. It contains the numbers of
                                        ; each bond graph element. More
                                        ; information on *list-of-nodes* can
                                        ; be found in the introduction to the
                                        ; function get-node-data-causal.
(defvar *list-of-n ie-types* nil)       ; Corresponding to each element number
                                        ; in *list-of-nodes*, is the element
                                        ; type in *list-of-node-types*.
                                        ; corresponding to the number. More
                                        ; information on *list-of-node-types*
                                        ; can be found in the introduction
                                        ; to the function
                                        ; get-node-data-causal.
(defvar *list-of-connected-lines* nil)  ; Corresponding to each element number
                                        ; in *list-of-nodes* are all the bonds
                                        , connected to that element and are
                                        ; stored in *list-of-connected-lines*.
                                        ; More information on
                                        ; *list-of-connected-lines* can
                                        ; be found in the introduction to the
                                        ; function get-node-data-causal.
(defvar *filename* nil)                 ; A global variable containing the
                                        ; filename, without the path, of the
                                        ; most recently read bond graph data
                                        ; file.
(defvar *lines* nil)                    ; The hash table of the bonds, contains
                                        ; the element numbers at either end of
                                        ; the bond and the causality assigned
                                        ; to the bond.
(defvar *line-hash-keys* nil)           ; The list of bond numbers by which the
                                        ; hash table is referenced.
(defvar *nodes* nil)                    ; The hash table containing all the
                                        ; bond graph elements, contains the
                                        ; element type, the documentation
                                        ; string and the equations for that
                                        ; element.
(defvar *node-hash-keys* nil)           ; A list of element numbers, used to
                                        ; reference the hash table *nodes*.
(defvar *error-flag* nil)               ; *error-flag* is set equal to 'error
                                        ; if there is an error in the bond
                                        ; graph structure. The assignment
                                        ; procedure is halted and the user
                                        ; is required to correct the bond
                                        ; graph.
(defvar *stack* nil)                    ; The global variable *stack* is a
                                        ; list of all the bonds to which
                                        ; causality has been assigned, or is
                                        ; in the process of being assigned to
                                        ; since the last time *state* was
```

```
                                        ; 'start-state. The exact form of
                                        ; *stack* will be shown later.
(defvar *unused-nodes* nil)             ; This list has the same format as
                                        ; *list-of-nodes*, except that
                                        ; *unused-nodes* is a list of all the
                                        ; bond graph elements that have not
                                        ; yet had a causality assigned to all
                                        ; the bonds connected them.
(defvar *unused-node-types* nil)        ; This list has the same format as
                                        ; *list-of-node-types*, except that
                                        ; *unused-node-types* is a list of
                                        ; the element types corresponding to
                                        ; the bond graph elements in
                                        ; *unused-nodes*.
(defvar *unused-connected-lines* nil)   ; This list has the same format as
                                        ; *list-of-connected-lines*, except
                                        ; that *unused-connected-lines*
                                        ; contains a list of all the bonds
                                        ; attached to an element that have
                                        ; not had a causality assigned to
                                        ; them yet.
(defvar *state* 'start-state)           ; The global variable *state* is the
                                        ; current state as defined by Hood,
                                        ; Palmer and Dantzig.


(defstruct line

;;; This lisp structure is referenced by the list *line-hash-keys*. The hash
;;; table contains the two elements that the bond joins in node-pair. The
;;; first element in node-pair is the element at the end of the bond without
;;; the power half arrow and the second element in node-pair is the element
;;; that the power is entering. The causality can either be nil, e or b. It is
;;; nil is no causality has been assigned, e is the causal stroke is at the
;;; same end as the power half arrow or b if the causal stroke is at the
;;; opposite end to the power half arrow.

  (node-pair '(0 0))
  (causality nil))


(defstruct node

;;; This list structure contains the data for each bond graph element and is
;;; referenced by *node-hash-keys*. There hash table contain three entries,
;;; the element type, the documentation string for the bond graph element and
;;; the equations for the element. The equations slot is nil if no equations
;;; have yet been obtained.

  (type nil)
  (doc-str nil)
  (equations nil))


(defun reset-causal ()
```

```
  (clrhash *lines*)
  (clrhash *nodes*))


(defun init-causal ()

;;; Both hash tables are cleared and new ones created

  (setq *lines* (make-hash-table))
  (setq *nodes* (make-hash-table))
  (reset-causal)
  (setq *line-hash-keys* nil
        *node-hash-keys* nil
        *directed-lines* nil
        *r-algerbraic-loop* nil
        *bond-algerbraic-loop* nil
        *error-flag* nil))


(defun get-first ()

;;; This function is called whenever *state* is 'start-state. The function
;;; resets *stack*, and calls the function update-unused-lists, which
;;; updates the lists *unused-nodes*, *unused-node-types* and
;;; *unused-connected-lines*, so that the most recent causal assignments are
;;; excluded. The function then gets an unassigned bond and the bond graph
;;; element to which it is connected. The elements have the usual priorities
;;; (first se, then sf, c, i, tf, gy, 0 and finally 1). If a resistor is
;;; returned by the function, a list of the form
;;; (element-type element-number bond-number) is consed onto the list
;;; *r-algebraic-loop*. If the function returns any of the junction structure
;;; elements (tf, gy, 0, 1), a list of the form
;;; (element-type element-number bond-number) is consed onto the list
;;; *bond-algebraic-loop*. Finally *stack* is set equal to
;;; ((element-number element-type bond-number)).

  (format t "~%Getting the first element~%")
  (setf *stack* (list nil))
;;; update-unused-lists does not work if *stack* is () as *stack* must be of
;;; the form (())
  (update-unused-lists)
;;; cond was chosen as it only evaluates the first form found that matches.
;;; Therefore we can implement the correct priorities
  (cond ((member 'se *unused-node-types*)
         (setf element-type 'se)
         (setf element-number (get-node-number-from-type 'se))
         (setf bond-number (get-unassigned-bond 'se)))
        ((member 'sf *unused-node-types*)
         (setf element-type 'sf)
         (setf element-number (get-node-number-from-type 'sf))
         (setf bond-number (get-unassigned-bond 'sf)))
        ((member 'c *unused-node-types*)
         (setf element-type 'c)
         (setf element-number (get-node-number-from-type 'c))
         (setf bond-number (get-unassigned-bond 'c)))
        ((member 'i *unused-node-types*)
```

```
        (setf element-type 'i)
        (setf element-number (get-node-number-from-type 'i))
        (setf bond-number (get-unassigned-bond 'i)))
       ((member 'r *unused-node-types*)
        (setf element-type 'r)
        (setf element-number (get-node-number-from-type 'r))
        (setf bond-number (get-unassigned-bond 'r))
        (setf *r-algerbraic-loop* (cons (list element-type element-number
                                              bond-number)
                                        *r-algerbraic-loop*)))
       ((member 'tf *unused-node-types*)
        (setf element-type 'tf)
        (setf element-number (get-node-number-from-type 'tf))
        (setf bond-number (get-unassigned-bond 'tf))
        (setf *bond-algerbraic-loop* (cons (list element-type element-number
                                                 bond-number)
                                           *bond-algerbraic-loop*)))
       ((member 'gy *unused-node-types*)
        (setf element-type 'gy)
        (setf element-number (get-node-number-from-type 'gy))
        (setf bond-number (get-unassigned-bond 'gy))
        (setf *bond-algerbraic-loop* (cons (list element-type element-number
                                                 bond-number)
                                           *bond-algerbraic-loop*)))
       ((member 'O *unused-node-types*)
        (setf element-type 'O)
        (setf element-number (get-node-number-from-type 'O))
        (setf bond-number (get-unassigned-bond 'O))
        (setf *bond-algerbraic-loop* (cons (list element-type element-number
                                                 bond-number)
                                           *bond-algerbraic-loop*)))
       ((member '1 *unused-node-types*)
        (setf element-type '1)
        (setf element-number (get-node-number-from-type '1))
        (setf bond-number (get-unassigned-bond '1))
        (setf *bond-algerbraic-loop* (cons (list element-type element-number
                                                 bond-number)
                                           *bond-algerbraic-loop*))))
  (setf *stack* (list (list element-number element-type bond-number)))
  (values))


(defun get-node-number-from-type (type)

;;; This function, given the element type gets the element number of the first
;;; occurrence of type in *unused-node-types*.

  (nth (position type *unused-node-types*) *unused-nodes*))


(defun get-unassigned-bond (type)

;;; Given the element type, this function gets an unassigned bond attached
;;; to the element the corresponds to the first occurrence of type in
;;; *unused-node-types*.
```

```
  (setf unassigned-bond (first (first
                                (nth (position type *unused-node-types*)
                                     *unused-connected-lines*)))))


(defun update-unused-lists ()

;;; This function forms a list *unused-connected-lines*, which has the same
;;; form as *list-of-connected-lines* except that all the bonds that have
;;; a causality assigned to them or are contained in *stack* are removed. If
;;; all the bonds attached to a bond graph element have causality assigned
;;; to them or if they are contained in *stack*, then this element number and
;;; element type are removed from *list-of-nodes* and *list-of-node-types* to
;;; form *unused-nodes* and *unused-node-types* respectively.

  (let ((temp-connected-lines nil)
        (count-list nil)
        (counter 0))
    (dolist (node-bonds *list-of-connected-lines*)
      (dolist (a-bond node-bonds)
        (dolist (stack-var *stack*)
          (if (or (not (equal (third a-bond) 'none))
                  (equal (third stack-var) (first a-bond)))
              (setf node-bonds (remove a-bond node-bonds :test #'equal)))))
      (setf temp-connected-lines (cons node-bonds temp-connected-lines)))
    (setf *unused-connected-lines* (reverse temp-connected-lines))
    (dolist (node-bonds *unused-connected-lines*)
      (if (equal node-bonds nil)
          (setf count-list (cons counter count-list)))
      (setf counter (+ 1 counter)))
    (remove-elements-from-*unused-nodes*-and-*unused-node-types* count-list)
; Note that here the lisp function position cannot be used, because if there
; are multiple occurrences of nil, position only finds the first one.
    (setf *unused-connected-lines*
          (remove nil *unused-connected-lines* :test #'equal))
    (values)))


(defun remove-elements-from-*unused-nodes*-and-*unused-node-types* (count-list)

;;; This function is called by update-unused-lists and is sent a list of
;;; positions of elements to remove. The elements corresponding to every
;;; position in count-list are removed from *list-of-nodes* and
;;; *list-of-node-types* to form *unused-nodes* and *unused-node-types*
;;; respectively.

  (let ((temp-list-nodes nil)
        (temp-list-types nil)
        (count 0))
    (dolist (node *list-of-nodes*)
      (if (not (member count count-list))
          (progn (setf temp-list-nodes (cons node temp-list-nodes))
                 (setf temp-list-types (cons (nth (position node
                                                            *list-of-nodes*)
                                                  *list-of-node-types*)
                                             temp-list-types))))
```

```
      (setf count (+ 1 count)))
    (setf *unused-nodes* (reverse temp-list-nodes))
    (setf *unused-node-types* (reverse temp-list-types))
    (values)))


(defun get-next ()

;;; The function get-next looks at the first list in *stack* and gets the most
;;; recently considered bond number (last-bond-number) and element number
;;; (last-element-number). The function then searches through
;;; *list-of-connected-lines*, to find the two occurrences of last-bond-number.
;;; the-position is the position of the bond graph element, attached to
;;; last-bond-number where this element number is not equal to
;;; last-element-number, in *list-of-nodes*.
;;; That is the function gets the element attached to the other end of the
;;; bond presently being considered. The function then goes through all the
;;; bonds connected to this element. If the causality on any bond is
;;; unassigned and the bond is not the same one being considered at present,
;;; then this bond is chosen. A list of the form
;;; (element-number element-type bond-number) is consed onto *stack*. If no
;;; bond is chosen the function reversal is called.

  (let ((last-bond-number (third (first *stack*)))
        (last-element-number (first (first *stack*)))
        (function-call 'reversal))
    (format t "~%Getting the next element in the stack~%")
    (dolist (node-bond *list-of-connected-lines*)
      (dolist (a-bond node-bond)
        (if (and (member last-bond-number a-bond)
                 (not (equal (nth (position node-bond *list-of-connected-lines*)
                                  *list-of-nodes*) last-element-number)))
            (setf the-position (position node-bond
                                         *list-of-connected-lines*)))))
    (dolist (a-bond (nth the-position *list-of-connected-lines*))
      (if (and (not (member last-bond-number a-bond)) (member 'none a-bond))
          (progn (setf function-call nil)
                 (setf element-number (nth the-position *list-of-nodes*))
                 (setf element-type (nth the-position *list-of-node-types*))
                 (setf bond-number (first a-bond))
                 (setf *stack*
                       (cons (list element-number element-type bond-number)
                             *stack*))
                 (return))))
    (if (not (null function-call))
        (reversal))
    (values)))


(defun away-next-effort ()

;;; This function assigns the causal stroke away from the current bond graph
;;; element (last-element-number) on the current bond (last-bond-number). It
;;; sets *state* equal to effort-state and calls the function get-next.

  (let ((last-bond-number (third (first *stack*)))
```

```
      (last-element-number (first (first *stack*))))
  (format t "~%Assigning causality~%")
  (dolist (node-bond *list-of-connected-lines*)
    (setf list-a-bond nil)
    (dolist (a-bond node-bond)
      (if (member last-bond-number a-bond)
          (if (equal (nth (position node-bond *list-of-connected-lines*)
                          *list-of-nodes*) last-element-number)
              (setf a-bond (list (first a-bond) (second a-bond) 'out))
              (setf a-bond (list (first a-bond) (second a-bond) 'in))))
      (setf list-a-bond (cons a-bond list-a-bond)))
    (setf *list-of-connected-lines* (substitute
                                     (reverse list-a-bond) node-bond
                                     *list-of-connected-lines*)))
  (setf *state* 'effort-state)
  (update-unused-lists)
  (get-next)
  (values)))


(defun next-next-flow ()

;;; This function assigns the causal stroke next to the current bond graph
;;; element (last-element-number) on the current bond (last-bond-number). It
;;; sets *state* equal to flow-state and calls the function get-next.

  (let ((last-bond-number (third (first *stack*)))
        (last-element-number (first (first *stack*))))
    (format t "~%Assigning causality~%")
    (dolist (node-bond *list-of-connected-lines*)
      (setf list-a-bond nil)
      (dolist (a-bond node-bond)
        (if (member last-bond-number a-bond)
            (if (equal (nth (position node-bond *list-of-connected-lines*)
                            *list-of-nodes*) last-element-number)
                (setf a-bond (list (first a-bond) (second a-bond) 'in))
                (setf a-bond (list (first a-bond) (second a-bond) 'out))))
        (setf list-a-bond (cons a-bond list-a-bond)))
      (setf *list-of-connected-lines* (substitute (reverse list-a-bond)
                                                  node-bond
                                                  *list-of-connected-lines*)))
    (setf *state* 'flow-state)
    (update-unused-lists)
    (get-next)
    (values)))


(defun error-stop ()

;;; If there is an error in the bond graph structure, this function is called.
;;; The function sets *error-flag* to 'error

  (format t "~|There is an ERROR in the bond graph structure.~%"
          The error occurred when dealing with bond ~a,~%"
          connected to element ~a, of type ~a~%" (third (first *stack*))
          (first (first *stack*)) (second (first *stack*)))
```

```
  (format t "~%Please rectify this error~%")
  (loop
   (if (y-or-n-p "~%Press y to continue")
       (return)))
  (setf *error-flag* 'error))


(defun away-previous-effort ()

;;; This function assigns the causal stroke away from the previous bond graph
;;; element (previous-element-number) on the previous bond
;;; (previous-bond-number). The function then sets *state* equal to
;;; effort-state.

  (let ((previous-bond-number (third (second *stack*)))
        (previous-element-number (first (second *stack*))))
    (format t "~%Assigning causality~%")
    (dolist (node-bond *list-of-connected-lines*)
      (setf list-a-bond nil)
      (dolist (a-bond node-bond)
        (if (member previous-bond-number a-bond)
            (if (equal (nth (position node-bond *list-of-connected-lines*)
                            *list-of-nodes*) previous-element-number)
                (setf a-bond (list (first a-bond) (second a-bond) 'out))
                (setf a-bond (list (first a-bond) (second a-bond) 'in))))
        (setf list-a-bond (cons a-bond list-a-bond)))
      (setf *list-of-connected-lines* (substitute (reverse list-a-bond)
                                                  node-bond
                                                  *list-of-connected-lines*)))
    (setf *state* 'effort-state)
    (update-unused-lists)
    (values)))


(defun next-previous-flow ()

;;; This function assigns the causal stroke next to the previous bond graph
;;; element (previous-element-number) on the previous bond
;;; (previous-bond-number). The function then sets *state* equal to
;;; flow-state.

  (let ((previous-bond-number (third (second *stack*)))
        (previous-element-number (first (second *stack*))))
    (format t "~%Assigning causality~%")
    (dolist (node-bond *list-of-connected-lines*)
      (setf list-a-bond nil)
      (dolist (a-bond node-bond)
        (if (member previous-bond-number a-bond)
            (if (equal (nth (position node-bond *list-of-connected-lines*)
                            *list-of-nodes*) previous-element-number)
                (setf a-bond (list (first a-bond) (second a-bond) 'in))
                (setf a-bond (list (first a-bond) (second a-bond) 'out))))
        (setf list-a-bond (cons a-bond list-a-bond)))
      (setf *list-of-connected-lines* (substitute (reverse list-a-bond)
                                                  node-bond
                                                  *list-of-connected-lines*)))
```

```
    (setf *state* 'flow-state)
    (update-unused-lists)
    (values)))


(defun next-r ()

;;; This function sets *state* equal to 'r-state and calls the function
;;; get-next

  (setf *state* 'r-state)
  (get-next)
  (values))


(defun weak-1 ()

;;; This function counts how many unassigned bonds are attached to  .e current
;;; element and checks to see whether there is a flow flowing into the element
;;; on any of the bonds. If there is a flow, flowing into the element the
;;; function next-next-flow is called. If there is no flow into the element,
;;; but there is one unassigned bond attached then the function
;;; away-next-effort is called, otherwise the function reversal is called.

  (let* ((last-element-number (first (first *stack*)))
         (the-position (position last-element-number *list-of-nodes*))
         (flow-in? 'no-flow-in)
         (number-unassigned 0))
    (dolist (a-bond (nth the-position *list-of-connected-lines*))
      (if (equal (third a-bond) 'out)
          (setf flow-in? 'flow-in)
          (if (member 'none a-bond)
              (setf number-unassigned (+ 1 number-unassigned)))))
    (if (equal flow-in? 'flow-in)
        (next-next-flow)
        (if (equal number-unassigned 1)
            (away-next-effort)
            (reversal)))))


(defun weak-0 ()

;;; This function counts how many unassigned bonds are attached to the current
;;; element and checks to see whether there is an effort flowing into the
;;; element on any bond. If there is an effort flowing into the element the
;;; function away-next-effort is called. If there is no effort flowing into
;;; the junction but there is one unassigned bond attached to the element,
;;; then the function next-next-flow is called, otherwise the
;;; function reversal is called.

  (let* ((last-element-number (first (first *stack*)))
         (the-position (position last-element-number *list-of-nodes*))
         (effort-in? 'no-effort-in)
         (number-unassigned 0))
    (dolist (a-bond (nth the-position *list-of-connected-lines*))
```

```
      (if (equal (third a-bond) 'in)
          (setf effort-in? 'effort-in)
          (if (member 'none a-bond)
              (setf number-unassigned (+ 1 number-unassigned)))))
    (if (equal effort-in? 'effort-in)
        (away-next-effort)
        (if (equal number-unassigned 1)
            (next-next-flow)
            (reversal)))))


(defun reversal ()
  (if (causal-conflict)
      (progn (undo-causality)
             (remove-all-but-last-two-elements-from-*stack*)
             (update-unused-lists)
             (setf *state* 'r-opposite))
      (progn (find-0-with-strong-causality)
             (if (equal *state* 'start-state)
                 (find-1-with-strong-causality))))
  (values))


(defun causal-conflict ()

;;; This function is called by reversal to see whether there is a causal
;;; conflict. The function counts the number of efforts entering and leaving
;;; each bond. If the element is a 1-junction and there is more than one
;;; effort leaving, then conflict is set true. If the element is a 0-junction
;;; and there is more than one effort entering, then conflict is set true. If
;;; the element is a transformer and there is more than one effort entering or
;;; more than one effort leaving, then conflict is set true. If the element is
;;; a gyrator and the number of efforts entering is equal to the number of
;;; efforts leaving, then conflict is set true.

  (let ((conflict nil)
        (number-in 0)
        (number-out 0))
    (dolist (node-bond *list-of-connected-lines*)
      (setf number-in 0)
      (setf number-out 0)
      (dolist (a-bond node-bond)
        (if (equal (third a-bond) 'in)
            (setf number-in (+ number-in 1)))
        (if (equal (third a-bond) 'out)
            (setf number-out (+ number-out 1))))
      (if (and (equal (nth (position node-bond *list-of-connected-lines*)
                           *list-of-node-types*) '1)
               (> number-out 1))
          (setf conflict 't))
      (if (and (equal (nth (position node-bond *list-of-connected-lines*)
                           *list-of-node-types*) '0)
               (> number-in 1))
          (setf conflict 't))
      (if (and (equal (nth (position node-bond *list-of-connected-lines*)
                           *list-of-node-types*) '
```

```
            (or (> number-out 1)
                (> number-in 1)))
        (setf conflict 't))
      (if (and (equal (nth (position node-bond *list-of-connected-lines*)
                           *list-o2-node-types*) 'gy)
               (= number-out number-in))
          (setf conflict 't)))
    (values conflict)))


(defun undo-causality ()

;;; This function is called by reversal, it takes each list in *stack* and
;;; sets the causality on that bond in *list-of-connected-lines* to 'none.

  (format t "~%Undoing conflicting causality~%")
  (dolist (stack-var *stack*)
    (dolist (node-bond *list-of-connected-lines*)
      (setf list-a-bond nil)
      (dolist (a-bond node-bond)
        (if (member (third stack-var) a-bond)
            (progn (setf a-bond (list (first a-bond) (second a-bond) 'none))
                   (setf a-bond (list (first a-bond) (second a-bond) 'none))))
        (setf list-a-bond (cons a-bond list-a-bond)))
      (setf *list-of-connected-lines* (substitute (reverse list-a-bond)
                                                  node-bond
                                                  *list-of-connected-lines*))))
  (values))


(defun remove-all-but-last-two-elements-from-*stack* ()
  (dotimes (count (- (length *stack*) 2))
    (pop *stack*))
  (values))


(defun find-0-with-strong-causality ()

;;; This function is called by reversal, it checks to see whether any
;;; 0-junctions in *stack* have a strong causality assigned to them and have
;;; at least one unassigned bond attached (calls the function
;;; strong-0-and-unassigned). If there are such 0-junctions, the function
;;; get-the-bond conses that bond onto *stack* and *state* is set to
;;; weak-0. If there are no such 0-junctions *state* is set to start-state.

  (dolist (stack-var *stack*)
    (if (and (equal (second stack-var) '0)
             (strong-0-and-unassigned? (first stack-var)))
;;; If in an "and" clause such as that above if the first thing does not
;;; succeed then the others are never tried i.e. if the element is not a
;;; 0-junction it will not call the function strong-0-and-unassigned?
        (progn (get-the-bond (first stack-var))
               (setf *state* 'weak-0)
               (return))
        (setf *state* 'start-state))))
```

```
(defun find-1-with-strong-causality ()

;;; This function is called by reversal, it checks to see whether any
;;; 1-junctions in *stack* have a strong causality assigned to them and have
;;; at least one unassigned bond attached (calls the function
;;; strong-1-and-unassigned). If there are such 1-junctions, the function
;;; get-the-bond conses that bond onto *stack* and *state* is set to
;;; weak-1. If there are no such 0-junctions *state* is set to start-state.

  (dolist (stack-var *stack*)
    (if (and (equal (second stack-var) '1)
             (strong-1-and-unassigned? (first stack-var)))
;;; See comment in previous function for discussion of "and"
        (progn (get-the-bond (first stack-var))
               (setf *state* 'weak-1)
               (return))
        (setf *state* 'start-state))))


(defun strong-1-and-unassigned? (element-number)

;;; This function is called by find-1-with-strong-causality, it checks to see
;;; whether there is a flow entering the junction, if so, strong? is set equal
;;; to strong. The number of unassigned bonds attached to the junction are
;;; then counted. If strong? is strong and there are more than 1 unassigned
;;; bonds then strong-and-unassigned is set true.

  (let ((number-unassigned 0)
        (strong? 'weak)
        (strong-and-unassigned nil))
    (dolist (a-bond (nth (position element-number *list-of-nodes*)
                         *list-of-connected-lines*))
      (if (equal (third a-bond) 'out)
          (setf strong? 'strong))
      (if (member 'none a-bond)
          (setf number-unassigned (+ 1 number-unassigned))))
    (if (and (equal strong? 'strong) (> number-unassigned 0))
        (setf strong-and-unassigned 't))))


(defun strong-0-and-unassigned? (element-number)

;;; This function is called by find-0-with-strong-causality, it checks to see
;;; whether there is an effort entering the junction, if so, strong? is set
;;; equal to strong. The number of unassigned bonds attached to the junction
;;; are then counted. If strong? is strong and there are more than 1
;;; unassigned bonds then strong-and-unassigned is set true.

  (let ((number-unassigned 0)
        (strong? 'weak)
        (strong-and-unassigned nil))
    (dolist (a-bond (nth (position element-number *list-of-nodes*)
                         *list-of-connected-lines*))
      (if (equal (third a-bond) 'in)
          (setf strong? 'strong))
      (if (member 'none a-bond)
```

```
        (setf number-unassigned (+ 1 number-unassigned))))
    (if (and (equal strong? 'strong) (> number-unassigned 0))
        (setf strong-and-unassigned 't))))


(defun get-the-bond (element-number)

;;; This function is called by find-0-with-strong-causality and
;;; find-1-with-strong-causality, if either the 0-junction or 1-junction has a
;;; strong causality and at least one unassigned bond. This function gets the
;;; unassigned bond attached to the junction with strong causality and conses
;;; a list of the form (element-number element-typebond-number) onto *stack*.

  (dolist (a-bond (nth (position element-number *list-of-nodes*)
                       *list-of-connected-lines*))
    (if (member 'none a-bond)
        (progn (setf *stack* (cons (list element-number
                                         (nth (position element-number
                                                        *list-of-nodes*)
                                              *list-of-node-types*)
                                         (first a-bond)) *stack*))
               (return))))
;;; The cons construct is used because there may be other 0 or 1-junctions
;;; in * stack* which are strong and have unassigned bonds, so we cannot just
;;; clear *stack*.
  (values))


(defun causality-main-function ()

;;; This is the controlling function in the causality assignment module, it
;;; displays the options available to the user and depending on the user's
;;; choice calls one of three functions

  (init-causal)
  (loop
   (causal-options)
   (setf ans (read))
   (cond ((equal ans 'a)
          (assign-causality))
         ((equal ans 'b)
          (view-causal-implications))
         ((equal ans 'c)
          (view-bond-graph-structure))
         (t (return)))))


(defun assign-causality ()

;;; This function controls the assignment of the causality. The function calls
;;; read-data-causal, which reads the hash tables from a file. The function
;;; get-node-causal is then called, which converts the hash tables into three
;;; global lists *list-of-nodes*, *list-of-node-types* and
;;; *list-of-connected-lines*. *state* is set to start state and
;;; *unused-nodes*, *unused-node-types* and *unused-connected-lines* are set
```

```
;;; equal to *list-of-nodes*, *list-of-node-types* and
;;; *list-of-connected-lines* respectively. The function then enters a loop,
;;; which is exited if there is an error in the bond graph structure or if
;;; *state* is start-state and *unused-nodes* is nil. If *state* is equal
;;; to start-state and *unused-nodes* is not nil then the function get-first
;;; is called. The function invokes the rule-base, sending it *state* and
;;; the element type. The function response returned is then evaluated. Also
;;; if *state* is equal to weak-1 the function weak-1 is called and if the
;;; state is weak-0, the function weak-0 is called. Once the causality
;;; assignment procedure is completed, the storage elements are checked for
;;; derivative causality (find-derivative-causality) and the function
;;; update-and-write-hash-table is called.

  (read-data-causal)
  (get-node-data-causal)
  (set-causality-to-none)
  (let ((*state* 'start-state))
    (setf *unused-nodes* *list-of-nodes*)
    (setf *unused-node-types* *list-of-node-types*)
    (setf *unused-connected-lines* *list-of-connected-lines*)
    (loop
     (if (equal *error-flag* 'error)
         (return))
     (if (and (equal *unused-nodes* nil) (equal *state*  start-state))
         (return))
     (if (and (equal *state* 'start-state) (not (equal *unused-nodes* nil)))
         (get-first))
     (eval (run (list *state*) (list (second (first *stack*)))))
     (if (equal *state* 'weak-1)
         (weak-1))
     (if (equal *state* 'weak-0)
         (weak-0)))
    (if (not (equal *error-flag* 'error))
        (progn (format t "~%DONE~%")
               (find-derivative-causality)
               (update-and-write-hash-table)))
    (values)))


(defun set-causality-to-none ()

;;; This function takes the list *list-of-connected-lines* and makes all the
;;; causalities equal to 'none

  (dolist (node-bond *list-of-connected-lines*)
    (setf list-a-bond nil)
    (dolist (a-bond node-bond)
      (setf a-bond (list (first a-bond) (second a-bond) 'none))
      (setf list-a-bond (cons a-bond list-a-bond)))
    (setf *list-of-connected-lines* (substitute
                                     (reverse list-a-bond) node-bond
                                      *list-of-connected-lines*))))


(defun read-data-causal ()
```

```
;;; This function finds which file the bond graph to be assigned causality
;;; is stored in

  (if *filename*
      (if (not (y-or-n-p "~|Do you want to assign or view the causality for ~
                          the bond graph stored in the ~%file ~a.data"
                          *filename*))
          (progn (format t "~%~%Enter the file name of the file to read from, ~
                          don't include an extension~%it will be taken to be ~
                          taken to be 'data'~%")
                 (setf *filename* (read))))
      (progn (format t "~|Enter the file name of the file to read from, ~
                      don't include an extension~%it will be taken to be ~
                      taken to be 'data'~%")
             (setf *filename* (read))))
  (read-data-from-file-causal *filename*)
  (values))


(defun get-node-data-causal ()

;;; This function takes the data stored in the two hash tables and converts
;;; it into a form that is more more suited to this program. This function
;;; creates three global lists, *list-of-nodes*, *list-of-node-types* and
;;; *list-of-connected-lines*. *list-of-nodes* is the same as *node-hash-keys*
;;; i.e. it contains a list of all the bond graph element numbers.
;;; *list-of-node-types* contains a list of all element types. The first
;;; element in *list-of-node-types* corresponds to the first element in
;;; *list-of-nodes*. The second element in *list-of-node-types* corresponds to
;;; the second element in *list-of-nodes* and so on.
;;; *list-of-connected-lines* contains a list of all the bonds attached to each
;;; bond graph element. *list-of-connected-lines* has the following format:
;;; ((..(..)..)...(...(bond-number power-direction causality-direction)...)...)
;;; *list-of-connected-lines* therefore, consists of a list of lists. The first
;;; list in *list-of-connected-lines* corresponds to the first element in
;;; *list-of-nodes*. The second list in *list-of-connected-lines*
;;; corresponds to the second element in *list-of-nodes* and so on. In each
;;; of these lists are numerous other lists. Each of these lists corresponds
;;; to a bond attached to the element, if there are two bonds attached to the
;;; bond graph element, there are two lists, if there is one bond attached,
;;; there is one list. In each of these lists corresponding to a bond there
;;; are three elements, the first is the bond number, the second is the power
;;; direction and the third is the causality direction (the  direction of the
;;; effort).
;;; Consider an example
;;;              1                2
;;;     se(1)--------\| 1(2) |---------\ r(3)
;;;                   |
;;;                   | 3
;;;                   |
;;;                   /
;;;                  ---
;;;                  i(4)
;;; The three lists are therefore
;;; *list-of-nodes* is (1 2 3 4)
```

```
;;; *list-of-node-types* is (se 1 r 1)
;;; *list-of-connected-lines* is (((1 out out))((1 in in)(2 out in)(3 out out))
;;;                               ((2 in out))((3 in in)))
;;; That is the first element in *list-of-nodes* is number one, it is of type
;;; se (the first element in *list-of-node-types*) and has one bond connected
;;; (the first list in *list-of-connected-lines*), the bond is number 1, the
;;; power is flowing out of the effort source and the effort is also flowing
;;; out of the effort source etc. The second element in *list-of-nodes* is
;;; number 2, it is of type 1 and has three bonds connected, bond 1, bond 2
;;; and bond 3. The power and the effort is the input to the 1-junction on
;;; bond 1, the power is leaving and the effort entering on bond 2 and both
;;; the power and the effort are leaving on bond 3. And so on for the remaining
;;; 2 elements.

  (let ((current-node nil)
        (current-node-type nil)
        (node-lines nil))
    (setf *list-of-nodes* nil)
    (setf *list-of-node-types* nil)
    (setf *list-of-connected-lines* nil)
    (setf *list-of-nodes* *node-hash-keys*)
    (dolist (node-number *node-hash-keys*)
      (setf *list-of-node-types* (cons (node-type
                                        (gethash node-number *nodes*))
                                       *list-of-node-types*)))
    (setf *list-of-node-types* (reverse *list-of-node-types*))
    (dolist (node-number *node-hash-keys*)
      (dolist (line-number *line-hash-keys*)
        (if (member node-number (line-node-pair (gethash line-number *lines*)))
            (setf node-lines
                  (cons (list line-number
                              (power-direction-causal
                               (position node-number (line-node-pair
                                                      (gethash line-number
                                                               *lines*))))
                              (causality-direction-causal
                               (position node-number (line-node-pair
                                                      (gethash line-number
                                                               *lines*)))
                               (line-causality (gethash line-number
                                                        *lines*))))
                        node-lines))))
      (setf *list-of-connected-lines*
            (cons node-lines *list-of-connected-lines*))
      (setf node-lines nil))
    (setf *list-of-connected-lines* (reverse *list-of-connected-lines*))
    (values))


(defun power-direction-causal (num)

;;; This function is called by get-node-data-causal, where num is the position
;;; of the node in line-node-pair. If num is zero i.e. the first node is being
;;; considered then the power is flowing out of that node, otherwise (if
;;; num=1 i.e. the second node is being considered) the power is flowing into
;;; the node.
```

```
  (if (= num 0)
      (setf power-direction 'out)
      (setf power-direction 'in)))


(defun causality-direction-causal (num direc)

;;; This function is called by get-node-data-causal, where num is the position
;;; of the node in line-node-pair and direc is line-causality. If num is zero
;;; the first node is being considered and therefore power is flowing out
;;; of the node, if direc is 'e i.e. the causal stroke is at the same end
;;; as the power half arrow, then the causality is 'out, but if direc is 'b
;;; i.e. the causal stroke is at the opposite end to the power half arrow then
;;; causality is 'in. If num is 1 the power is flowing into the node, then if
;;; direc is 'e the causality is 'in otherwise if direc is 'b the causality is
;;; 'out

(cond ((and (equal num 0)
            (equal direc 'e))
       (setf causality-direction 'out))
      ((and (equal num 0)
            (equal direc 'b))
       (setf causality-direction 'in))
      ((and (equal num 1)
            (equal direc 'e))
       (setf causality-direction 'in))
      ((and (equal num 1)
            (equal direc 'b))
       (setf causality-direction 'out))))


(defun read-data-from-file-causal (filename)

;;; This function reads the data stored in the file filename.data. The format
;;; of the data file is discussed at the beginning of the file enter.lisp.
;;; A portion of the function was originally written by C. Hartmann. The
;;; function was modified and added to by G. Grobbelaar

  (setf filename (concatenate 'string "/usr/bondgraph/data/"
                              (string-downcase filename) ".data"))
  (with-open-file (line-stream filename :direction :input)
                  (setf causality-assigned? (read line-stream))
                  (setf *line-hash-keys* (read line-stream))
                  (setf line-keys *line-hash-keys*)
                  (dolist (a-line-number line-keys)
                    (setf (gethash a-line-number *lines*)
                          (read line-stream)))
                  (setf *node-hash-keys* (read line-stream))
                  (setf node-keys *node-hash-keys*)
                  (dolist (a-node-number node-keys)
                    (setf (gethash a-node-number *nodes*)
                          (read line-stream))))
  (values))
```

```
(defun find-derivative-causality ()

;;; This function searches through  *list-of-connected-lines* and
;;; *list-of-node-types* to find all storage elements with derivative
;;; causality assigned to them. If there are such elements, a list of the form
;;; (element-type element-number bond-number) is consed onto the global list
;;; *deriv-causality*

  (let ((counter 0))
    (setf *deriv-causality* nil)
    (dolist (a-node *list-of-node-types*)
      (if (equal a-node 'c)
          (dolist (a-bond (nth counter *list-of-connected-lines*))
;;; Note we must use a counter, we cannot use position, because if the element
;;; appears more than once in the list we may get the incorrect one using
;;; the lisp function position because
;;; (position 'c '(1 2 c 3 c)) is 2
            (if (equal (third a-bond) 'in)
                (setf *deriv-causality*
                      (cons (list 'c (nth counter *list-of-nodes*)
                                  (first a-bond)) *deriv-causality*)))))
      (if (equal a-node 'i)
          (dolist (a-bond (nth counter *list-of-connected-lines*))
;;; Note we must use a counter, we cannot use position, because if the element
;;; appears more than once in the list we may get the incorrect one using
;;; the lisp function position because
;;; (position 'c '(1 2 c 3 c)) is 2
            (if (equal (third a-bond) 'out)
                (setf *deriv-causality*
                      (cons (list 'i (nth counter *list-of-nodes*)
                                  (first a-bond)) *deriv-causality*)))))
      (setf counter (+ 1 counter)))))


(defun update-and-write-hash-table ()
  (add-causality-to-hash-table)
  (write-hash-table-to-file-causal *filename*))


(defun add-causality-to-hash-table ()

;;; This function takes the causality assignments in the list
;;; *list-of-connected-lines* and transfers them to the hash table *lines*
;;; using the predefined format, i.e. if the causal stroke is at the same
;;; end as the power half arrow, the causality is 'e, otherwise it is 'b.

  (let ((list-of-bonds-used nil)
        (current-node nil)
        (first-element-in-line-pair nil)
        (second-element-in-line-pair nil))
    (dolist (node-bonds *list-of-connected-lines*)
      (setf current-node (nth (position node-bonds *list-of-connected-lines*)
                              *list-of-nodes*))
      (dolist (a-bond node-bonds)
        (setf first-element-in-line-pair (first (line-node-pair
                                                 (gethash (first a-bond)
```

```
                                                        *lines*))))
        (setf second-element-in-line-pair (second (line-node-pair
                                                    (gethash (first a-bond)
                                                        *lines*))))
        (if (not (member (first a-bond) list-of-bonds-used))
            (progn
             (cond ((and (equal (third a-bond) 'out)
                         (equal current-node first-element-in-line-pair))
                    (setf (line-causality (gethash (first a-bond) *lines*)) 'e))
                   ((and (equal (third a-bond) 'in)
                         (equal current-node first-element-in-line-pair))
                    (setf (line-causality (gethash (first a-bond) *lines*)) 'b))
                   ((and (equal (third a-bond) 'out)
                         (equal current-node second-element-in-line-pair))
                    (setf (line-causality (gethash (first a-bond) *lines*)) 'b))
                   ((and (equal (third a-bond) 'in)
                         (equal current-node second-element-in-line-pair))
                    (setf (line-causality (gethash (first a-bond) *lines*))
                          'e)))
              (setf list-of-bonds-used (cons (first a-bond)
                                             list-of-bonds-used)))))))


(defun write-hash-table-to-file-causal (filename)

;;; This function writes the line and the node hash tables to a file
;;; *filename*.data. In addition to the hash tables, *line-hash-keys* and
;;; *node-hash-keys*, which reference the hash tables are also written to the
;;; file, as are *r-algerbraic-loop*, *bond-algerbraic-loop* and
;;; *deriv-causality*. A statement, informing the modules that the causality
;;; has been assigned is also included in the file.

  (setq filename (concatenate 'string "/usr/bondgraph/data/"
                              (string-downcase filename) ".data"))
  (format t (concatenate 'string "~%Data written to file " filename "~%"))
  (with-open-file (line-stream filename
                               :direction :output)
                  (print 'causality-assigned line-stream)
                  (print *line-hash-keys* line-stream)
                  (dolist (line-number *line-hash-keys*)
                    (print (gethash line-number *lines*) line-stream))
                  (print *node-hash-keys* line-stream)
                  (dolist (node *node-hash-keys*)
                    (print (gethash node *nodes*) line-stream))
                  (print *r-algerbraic-loop* line-stream)
                  (print *bond-algerbraic-loop* line-stream)
                  (print *deriv-causality* line-stream)))


(defun view-causal-implications ()

;;; This function allows the user to view the causal implications of all the
;;; elements on the screen. All the data is written to a file and the file
;;; is viewed using the unix command "more", because "more" controls the
;;; scrolling.
```

```
  (read-data-causal)
  (get-node-data-causal)
  (with-open-file (a-stream "/usr/bondgraph/data/view.output"
                            :direction :output :if-exists :supersede)
                  (call-each-node-type a-stream))
  (run-unix-program "more" :input "/usr/bondgraph/data/view.output")
  (format t "~%~%Press ENTER to continue~%")
  (read-char)
  (values))


(defun view-bond-graph-structure ()

;;; This function allows the user to view the two hash tables *lines* and
;;; *nodes*. All the data is written to a file and the file is viewed using
;;; the unix command "more", because "more" controls the scrolling.

  (read-data-causal)
  (with-open-file (a-stream "/usr/bondgraph/view.output"
                            :direction :output :if-exists :supersede)
                  (dolist (line-number *line-hash-keys*)
                    (let* ((node-pair (line-node-pair
                                        (gethash line-number *lines*)))
                           (start-node (first node-pair))
                           (end-node (second node-pair)))
                      (format a-stream "~%~%Bond ~a is ~a"
                              line-number (gethash line-number *lines*))
                      (format a-stream "~%Element ~a is ~a"
                              start-node (gethash start-node *nodes*))
                      (if (eq start-node end-node)
                          (format a-stream "~%End element is the same as ~
                                  start element")
                          (format a-stream "~%Element ~a is ~a" end-node
                                  (gethash end-node *nodes*))))))
  (run-unix-program "more" :input "/usr/bondgraph/data/view.output")
  (format t "~%~%Press ENTER to continue~%")
  (read-char)
  (values))


(defun causal-options ()

;;; This function displays the options available in this module.

  (format t "~| Do you want to~%")
  (format t "~% A  Assign causality to a bond graph.~%")
  (format t "~% B  View the causal implications.~%")
  (format t "~% C  View the bond graph data structure~%")
  (format t "~% D  Leave the causality assignment routine.~%")
  (format t "~% Press the key corresponding to your choice and hit enter  "))


(defun call-each-node-type (a-stream)

;;; This function is called by view-causal-implications and sends each element
```

```
;;; in *list-of-nodes* to get-a-particular-node.

  (let ((pos '0))
    (dolist (node *list-of-nodes*)
      (get-a-particular-node node (nth pos *list-of-node-types*)
                             (nth pos *list-of-connected-lines*) a-stream)
      (setf pos (+ 1 pos))))
  (values))


(defun get-a-particular-node (node-number node-type connected-line-data
                                          a-stream)

;;; This function is called by call-each-node-type. Depending of the node
;;; type, node-type being considered one of nine functions is called.

  (case node-type
    ('r (r-element node-number connected-line-data a-stream))
    ('c (c-element node-number connected-line-data a-stream))
    ('i (i-element node-number connected-line-data a-stream))
    ('tf (tf-element node-number connected-line-data a-stream))
    ('gy (gy-element node-number connected-line-data a-stream))
    ('se (se-element node-number connected-line-data a-stream))
    ('sf (sf-element node-number connected-line-data a-stream)))
  (values))


(defun r-element (node-number line-data a-stream)

;;; If the element currently being considered is a resistor, this function is
;;; called. The data for the element is written to the file as well as the
;;; independent and dependent variables.

  (format a-stream "~%~%RESISTOR (ELEMENT NUMBER ~a) CONNECTED TO ~%"
          node-number)
  (dolist (a-line line-data)
    (format a-stream "BOND ~a " (first a-line)))
  (format a-stream "~%~%The causality assigned implies that")
  (dolist (current-line line-data)
    (if (equal ('third current-line) 'in)
        (setf e-f-out "f")
        (setf e-f-out "e"))
    (format a-stream "~%~a~a is a function of " e-f-out (first current-line))
    (dolist (a-line line-data)
      (if (equal (third a-line) 'in)
          (setf e-f-in "e")
          (setf e-f-in "f"))
      (format a-stream "~a~a " e-f-in (first a-line)))))


(defun se-element (node-number line-data a-stream)

;;; If the element currently being considered is an effort source, this
;;; function is called. The data for the element is written to the file as
;;; well as the dependent variable.
```

```
  (let ((connected-line (first line-data)))
    (format a-stream "~%~%EFFORT SOURCE (ELEMENT NUMBER ~a) CONNECTED TO ~%"
            node-number)
    (format a-stream "BOND ~a " (first connected-line))
    (format a-stream "~%~%The causality assigned implies that e~a is the ~
            output" (first connected-line))))


(defun sf-element (node-number line-data a-stream)

;;; If the element currently being considered is a flow source, this
;;; function is called. The data for the element is written to the file as
;;; well as the dependent variable.

  (let ((connected-line (first line-data)))
    (format a-stream "~%~%FLOW SOURCE (ELEMENT NUMBER ~a) CONNECTED TO ~%"
            node-number)
    (format a-stream "BOND ~a " (first connected-line))
    (format a-stream "~%~%The causality assigned implies that f~a is the ~
            output" (first connected-line))))


(defun c-element (node-number line-data a-stream)

;;; If the element currently being considered is a capacitor, this function is
;;; called. The data for the element is written to the file as well as the
;;; independent and dependent variables.

  (format a-stream "~%~%CAPACITOR (ELEMENT NUMBER ~a) CONNECTED TO ~%"
          node-number)
  (dolist (a-line line-data)
    (format a-stream "BOND ~a " (first a-line)))
  (format a-stream "~%~%The causality assigned implies that")
  (dolist (current-line line-data)
    (if (equal (third current-line) 'out)
        (setf e-f-out "e")
        (setf e-f-out "q"))
    (format a-stream "~%~a~a is a function of " e-f-out (first current-line))
    (dolist (a-line line-data)
      (if (equal (third a-line) 'out)
          (setf e-f-in "q")
          (setf e-f-in "e"))
      (format a-stream "~a~a " e-f-in (first a-line)))))


(defun i-element (node-number line-data a-stream)

;;; If the element currently being considered is an inertia, this function is
;;; called. The data for the element is written to the file as well as the
;;; independent and dependent variables.

  (format a-stream "~%~%INERTIA (ELEMENT NUMBER ~a) CONNECTED TO ~%"
          node-number)
  (dolist (a-line line-data)
```

```
    (format a-stream "BOND ~a " (first a-line)))
  (format a-stream "~%~%The causality assigned implies that")
  (dolist (current-line line-data)
    (if (equal (third current-line) 'in)
        (setf e-f-out "f")
        (setf e-f-out "p"))
    (format a-stream "~%~a~a is a function of " e-f-out (first current-line))
    (dolist (a-line line-data)
      (if (equal (third a-line) 'in)
          (setf e-f-in "p")
          (setf e-f-in "f"))
      (format a-stream "~a~a " e-f-in (first a-line)))))


(defun tf-element (node-number line-data a-stream)

;;; If the element currently being considered is a transformer, th : function
;;; is called. The data for the element is written to the file as well as the
;;; independent and dependent variables.

  (format a-stream "~%~%TRANSFORMER (ELEMENT NUMBER ~a) CONNECTED TO ~%"
          node-number)
  (dolist (a-line line-data)
    (format a-stream "BOND ~a " (first a-line)))
  (format a-stream "~%~%The causality assigned implies that")
  (if (equal (second (first line-data)) 'out)
      (progn (setf pout '0)
             (setf pin '1))
      (progn (setf pout '1)
             (setf pin '0)))
  (if (equal (third (first line-data)) 'out)
      (progn (setf cout '0)
             (setf cin '1))
      (progn (setf cout '1)
             (setf cin '0)))
  (format a-stream "~%e~a = m e~a where m is the modulus~%"
          (first (nth pin line-data)) (first (nth pout line-data))))


(defun gy-element (node-number line-data a-stream)

;;; If the element currently being considered is a gyrator, this function is
;;; called. The data for the element is written to the file as well as the
;;; independent and dependent variables.

  (format a-stream "~%~%GYRATOR (ELEMENT NUMBER ~a) CONNECTED TO ~%"
          node-number)
  (dolist (a-line line-data)
    (format a-stream "BOND ~a " (first a-line)))
  (format a-stream "~%~%The causality assigned implies that")
  (if (equal (second (first line-data)) 'out)
      (progn (setf pout '0)
             (setf pin '1))
      (progn (setf pout '1)
             (setf pin '0)))
  (format a-stream "~%e~a = r f~a where r is the modulus~%"
```

```
        (first (nth pin line-data)) (first (nth pout line-data))))

;bott

;;; NOTE we can use the lisp function position when dealing with *list-of-nodes*
;;; or *list-of-connected-lines* because the elements in the list are never
;;; repeated, which is not the case with *list-of-node-types*
```

D.4 The module that controls the forward chaining algorithm (shell.lisp)

```
;;top
;;; The module which controls the forward chaining algorithm.
;;; Written by A.L. Stevens
;;; Modified by G.B. Grobbelaar

  (defvar *rules* 'empty-stream)
  (defvar *assertions* 'empty-stream)
  (defvar *rules-fired* nil)
  (setf file-1 "/usr/bondgraph/asserts.data"
        file-2 "/usr/bondgraph/results.data")
  (format t "~%Loading rule interpreter...")
  (load "/usr/bondgraph/fchain.3bin")
  (format t "~%Loading rules...")
  (load "/usr/bondgraph/rules.lisp")

 (defun run (fact-1 fact-2)
  ;-------------------------------------------------------------------
   "This runs Winston's rule-based forward chaining program."
  ;-------------------------------------------------------------------
  (setf *consequent-list* nil)
  (reset-rule 'a)
  (remember-assertion fact-1)
  (remember-assertion fact-2)
  (forward-chain)
  (values (first *consequent-list*)))



 (defun asserts? (&aux (counter 0))
  ;-------------------------------------------------------------------
  "This is a function which writes out the contents of a stream
   to the file asserts.data. It can be modified to process any stream."
  ;-------------------------------------------------------------------
(with-open-file (asserts-used file-1
 :direction :output)
(format asserts-used ";;top")
(format asserts-used "~%The following assertions were used in the run:~%")
(stream-transform #'(lambda (the-assertion)
 (format asserts-used "~% ~a" the-assertion)
 (incf counter)) *assertions*)
(format asserts-used "~%There were ~a assertions altogether.~%" counter)
 (format asserts-used ";;bott"))
(ed file-1))

 (defun results? ()
  ;-------------------------------------------------------------------
 (with-open-file (conseq file-2
  :direction :output)
 (format conseq ";;top")
 (format conseq "~%The following consequents occurred in the run:~%")
 (dolist (the-conseq *consequent-list*)
  (format conseq "~% ~a" the-conseq))
  (format conseq "~%;;bott")))
```

```
 (defun rules? ()
  ;-------------------------------------------------------------------------------
  (format t "
The following rules were used in the run:
~a" *rules-fired*) (values))


  (defun reset-rule (&optional switch)
  ;-------------------------------------------------------------------------------
   (if (not switch)
    (progn
     (format t "
Enter a to reset assertions, r to reset rules or b to reset both")
     (case (read)
     ('a (setf *assertions* 'empty-stream))
     ('r (setf *rules* 'empty-stream))
     ('b (setf *assertions* 'empty-stream *rules* 'empty-stream))))
     ;OTHERWISE..
     (case switch
     ('a (setf *assertions* 'empty-stream))
     ('r (setf *rules* 'empty-stream))
     ('b (setf *assertions* 'empty-stream *rules* 'empty-stream)))))

;;bott.
```

D.5 The forward chaining algorithm (fchain.lisp)

```
;;;***************fchain.lisp*****************
;;;*******************************************************************
;;; Winston's forward chaining algorithm, written by A.L. Stevens

;;;top
(defvar *consequent-list* nil)
(defvar rule-counter 0) ;FOR TESTS ONLY!
(defvar *rules-fired* nil) ;FOR TESTS ONLY!


(defun stream-endp (stream)
 (eq stream 'empty-stream))


(defun stream-first (stream)
 (first stream))


(defun stream-rest (stream)
 (second stream))


(defun stream-cons (object stream)
 (list object stream))


(defun stream-append (stream1 stream2)
 (if (stream-endp stream1) stream2
  (stream-cons (stream-first stream1)
  (stream-append (stream-rest stream1)
   stream2))))


(defun stream-member (object stream)
 (cond ((stream-endp stream) nil)
       ((equal object (stream-first stream)) ;)
       (t (stream-member object (stream-rest stream)))))

(defmacro stream-remember (object variable)
 '(unless (stream-member ,object ,variable)
   (setf ,variable
    (stream-append ,variable (stream-cons ,object 'empty-stream)))
     ,object))


(defun stream-concatenate (streams)
 (if (stream-endp streams) 'empty-stream
   (stream-append (stream-first streams)
    (stream-concatenate (stream-rest streams)))))


(defun stream-transform (procedure stream)
 (if (stream-endp stream) 'empty-stream
  (stream-cons (funcall procedure (stream-first stream))
   (stream-transform procedure (stream-rest stream)))))
```

```
(defun add-binding (pattern-variable-expression datum bindings)
 (if (eq '_ (extract-variable pattern-variable-expression))
   bindings
   (cons (make-binding
                  (extract-variable pattern-variable-expression)
    datum)
     bindings)))

(defun extract-variable (pattern-variable-expression)
 (second pattern-variable-expression))

(defun extract-key (binding)
 (first binding))

(defun extract-value (binding)
 (second binding))

(defun make-binding (variable datum)
 (list variable datum))

(defun find-binding (pattern-variable-expression binding)
 (unless (eq '_
             (extract-variable pattern-variable-expression))
  (assoc (extract-variable pattern-variable-expression)
    binding)))

(defun match-atoms (P D bindings)
 (if (eql P D) bindings 'fail))

(defun match-variable (P D bindings)
 (let ((binding (find-binding P bindings)))
   (if binding
     (match (extract-value binding) D bindings)
   (add-binding P D bindings))))

(defun match-pieces (P D bindings)
 (let ((result (match (first P) (first D) bindings)))
  (if (eq 'fail result) 'fail (match (rest P) (rest D) result))))

(defun recursive-p (P D)
 (and (listp P) (listp D)))

(defun variable-p (P)
 (and (listp P) (eq '? (first P))))

(defun elements-p (P D)
 (and (atom P) (atom D)))

(defun match (P D &optional bindings)
 (cond ((elements-p P D) (match-atoms P D bindings))
       ((variable-p P) (match-variable P D bindings))
       ((recursive-p P D) (match-pieces P D bindings))
       (t 'fail)))
```

```
(defun try-assertion (pattern assertion bindings)
 (let ((result (match pattern assertion bindings)))
  (if (eq 'fail result) 'empty-stream
   (stream-cons result 'empty-stream))))

(defun match-pattern-to-assertions (pattern bindings)
 (stream-concatenate
  (stream-transform #'(lambda (assertion)
    (try-assertion pattern assertion bindings)) *assertions*)))

(defun filter-binding-stream (pattern stream)
 (stream-concatenate
  (stream-transform #'(lambda (bindings)
   (match-pattern-to-assertions pattern bindings)) stream)))

(defun apply-filters (patterns initial-input-stream)
 (if (endp patterns) initial-input-stream
  (apply-filters (rest patterns) ;next 2 lines added by ALS.
  (if back-chain-flag
  (filter-binding-stream-back  (first patterns) initial-input-stream)
  (filter-binding-stream (first patterns) initial-input-stream)  ))))

(defun instantiate-variables (pattern a-list)
 (cond ((atom pattern) pattern)
       ((eq '? (first pattern))
        (extract-value (find-binding pattern a-list)))
       (t (cons (instantiate-variables (first pattern) a-list)
         (instantiate-variables (rest pattern) a-list)))))

(defun remember-assertion (assertion)
 (stream-remember assertion *assertions*))

(defun rule-name (rule) (first rule))

(defun rule-ifs (rule)
 (butlast (rest rule)))

(defun rule-then (rule)
 (first (last rule)))

(defun new-rule (rule) ;;;was remember-rule.
 (stream-remember rule *rules*))

(defun use-rule (rule)
 (let ((binding-stream (apply-filters (rule-ifs rule)
  (stream-cons nil 'empty-stream))))
 (do ((binding-stream binding-stream (stream-rest binding-stream))
      (success-switch nil))
     ((stream-endp binding-stream) success-switch)
  (let ((result
        (if back-chain-flag
         (instantiate-variables-back (rule-then rule)
```

```
                                        (stream-first binding-stream))
            (instantiate-variables (rule-then rule)
                                   (stream-first binding-stream)))))
      (when (remember-assertion result)
;;     (format t "~%Rule ~A has fired" (rule-name rule)) ;;ADDED BY ALS.
  ;; THE LINE ABOVE TAKEN OUT BY ALS FOR TESTS.
;;     (format t "~%Rule ~A indicates ~A." (rule-name rule) result)
      (incf rule-counter) ;FOR TESTS ONLY.
      (setq *rules-fired* (cons (rule-name rule) *rules-fired*))
           (setq *consequent-list* (cons result *consequent-list*))
  ;;*****THE ABOVE LINE ADDED BY ALS.******************
       (setf success-switch t))))))


 (defun forward-chain ()
 (setf back-chain-flag nil)
  (do ((rule-stream *rules* (stream-rest rule-stream))
        (repeat-switch nil))
       ((stream-endp rule-stream)
         (if repeat-switch
          (forward-chain)))
       (when (use-rule (stream-first rule-stream))
           (setf repeat-switch t)))
  (setf *rules-fired* (remove-duplicates *rules-fired* :test #'equal)))


;;;bot
```

D.6 The state transition table rules (rules.lisp)

```
;;******Rules to be used with "fchain.lisp"*********
;; Rules written by G.B. Grobbelaar

;;top

(new-rule '(caus-rule-1
                (start-state)
                (se)
                (away-next-effort)))

(new-rule '(caus-rule-2
                (effort-state)
                (se)
                (error-stop)))

(new-rule '(caus-rule-3
                (flow-state)
                (se)
                (reversal)))

(new-rule '(caus-rule-4
                (r-state)
                (se)
                (error-stop)))

(new-rule '(caus-rule-5
                (r-opposite)
                (se)
                (error-stop)))

(new-rule '(caus-rule-6
                (start-state)
                (sf)
                (next-next-flow)))

(new-rule '(caus-rule-7
                (effort-state)
                (sf)
                (reversal)))

(new-rule '(caus-rule-8
                (flow-state)
                (sf)
                (error-stop)))

(new-rule '(caus-rule-9
                (r-state)
                (sf)
                (error-stop)))
```

```
(new-rule '(caus-rule-10
            (r-opposite)
            (sf)
            (error-stop)))

(new-rule '(caus-rule-11
            (start-state)
            (0)
            (next-r)))

(new-rule '(caus-rule-12
            (effort-state)
            (0)
            (away-next-effort)))

(new-rule '(caus-rule-13
            (flow-state)
            (0)
            (weak-0)))

(new-rule '(caus-rule-14
            (r-state)
            (0)
            (away-previous-effort)))

(new-rule '(caus-rule-15
            (r-opposite)
            (0)
            (next-previous-flow)))

(new-rule '(caus-rule-16
            (start-state)
            (1)
            (next-r)))

(new-rule '(caus-rule-17
            (effort-state)
            (1)
            (weak-1)))

(new-rule '(caus-rule-18
            (flow-state)
            (1)
            (next-next-flow)))

(new-rule '(caus-rule-19
            (r-state)
            (1)
            (next-previous-flow)))

(new-rule '(caus-rule-20
            (r-opposite)
            (1)
            (away-previous-effort)))
```

```
(new-rule '(caus-rule-21
                (start-state)
                (i)
                (next-next-flow)))

(new-rule '(caus-rule-22
                (effort-state)
                (i)
                (reversal)))

(new-rule '(caus-rule-23
                (flow-state)
                (i)
                (reversal)))

(new-rule '(caus-rule-24
                (r-state)
                (i)
                (error-stop)))

(new-rule '(caus-rule-25
                (r-opposite)
                (i)
                (error-stop)))

(new-rule '(caus-rule-26
                (start-state)
                (c)
                (away-next-effort)))

(new-rule '(caus-rule-27
                (effort-state)
                (c)
                (reversal)))

(new-rule '(caus-rule-28
                (flow-state)
                (c)
                (reversal)))

(new-rule '(caus-rule-29
                (r-state)
                (c)
                (error-stop)))

(new-rule '(caus-rule-30
                (r-opposite)
                (c)
                (error-stop)))

(new-rule '(caus-rule-31
                (start-state)
                (r)
                (next-r)))
```

```
(new-rule '(caus-rule-32
                (effort-state)
                (r)
                (reversal)))

(new-rule '(caus-rule-33
                (flow-state)
                (r)
                (reversal)))

(new-rule '(caus-rule-34
                (r-state)
                (r)
                (error-stop)))

(new-rule '(caus-rule-35
                (r-opposite)
                (r)
                (error-stop)))

(new-rule '(caus-rule-36
                (start-state)
                (tf)
                (next-r)))

(new-rule '(caus-rule-37
                (effort-state)
                (tf)
                (away-next-effort)))

(new-rule '(caus-rule-38
                (flow-state)
                (tf)
                (next-next-flow)))

(new-rule '(caus-rule-39
                (r-state)
                (tf)
                (away-previous-effort)))

(new-rule '(caus-rule-40
                (r-opposite)
                (tf)
                (next-previous-flow)))

(new-rule '(caus-rule-41
                (start-state)
                (gy)
                (next-r)))

(new-rule '(caus-rule-42
                (effort-state)
                (gy)
                (next-next-flow)))
```

```
(new-rule '(caus-rule-43
                (flow-state)
                (gy)
                (away-next-effort)))

(new-rule '(caus-rule-44
                (r-state)
                (gy)
                (away-previous-effort)))

(new-rule '(caus-rule-45
                (r-opposite)
                (gy)
                (next-previous-flow)))

;;;tt
```

D.7 The ACSL input file creation module (bondacsl.lisp)

```
;-------------------------------------------------------------------------
;-------------------------------------------------------------------------
;                    Bond Graph Equation Formulation Module
;-------------------------------------------------------------------------
;-------------------------------------------------------------------------
;                    Written by: Grant Grobbelaar
;                    Last modified on: 07/06/91
;-------------------------------------------------------------------------
;-------------------------------------------------------------------------

;;; This function obtains the bond graph structure, gets the equations for
;;; each element in the bond graph and produces an acsl input file.

;;; This function reads the bond graph structure from a file, included in the
;;; file is a list of all the bonds attached to resistors, which have an
;;; arbitrary causality assigned to them, a list of all the bonds attached to
;;; junction structure elements, which had an arbitrary causality assigned to
;;; them and a list of all the bonds attached to storage elements which have
;;; derivative causality assigned to them. Also included in the file is a
;;; variable that is set if the causality has been assigned to the bond graph
;;; by the causality assignment routine. If the equations for an element have
;;; previously been entered, then these are displayed on the screen and the
;;; user is prompted to enter whether these equations still hold. If they do
;;; they are included in the equations contained in the acsl input file.
;;; If the equations are no longer valid, then the user is required to enter
;;; them from the keyboard. The equations can be entered in one of three
;;; forms: If the constitutive relation is linear, the user need only enter
;;; the linear coefficient (the resistance, capacitance or inertance). If the
;;; causal relation is non linear, the user has to enter the entire relation
;;; in the specified causal orientation. The user may also enter the data for
;;; each element in the form of a table. If there are any algebraic loops one
;;; or more equations are written in a form that enables them to be calculated
;;; implicitly. If there are any storage elements with derivative causality,
;;; user differentiated and possibly implicitly differentiated equations are
;;; included in the list of equations. Finally the acsl input file
;;; (filename.acsl) is written.

;top

(defvar *r-algebraic-loop* nil)          ; A global list with information
                                         ; about which bonds attached to
                                         ; resistors had the causality assigned
                                         ; arbitrarily, it is nil if there are
                                         ; no algebraic loops.
(defvar *bond-algebraic-loop* nil)       ; A global list with information
                                         ; about which bonds attached junction
                                         ; structure elements had the causality
                                         ; assigned arbitrarily, it is nil if
                                         ; there are no algebraic loops.
(defvar *deriv-causality* nil)           ; A global list with information about
                                         ; bonds attached to storage elements
                                         ; that have derivative causality
                                         ; assigned to them.
(defvar *causality-assigned?* nil)       ; This global variable is not nil if
```

```
                                        ; the causality has already been
                                        ; assigned to the bond graph
(defvar *error-flag-num* 1)             ; If there is an algebraic loop or
                                        ; derivative causality, then an ACSL
                                        ; implicit statement (IMPL) is
                                        ; required. Within this statement is
                                        ; an error flag, so that the flag
                                        ; variable is not repeated, it is
                                        ; indexed. The variable is ef and
                                        ; *error-flag-num* is the
                                        ; index.
(defvar *table-var* 1)                  ; As for the error flag, if the data
                                        ; is in the form of a table, the table
                                        ; is named tab and indexed with the
                                        ; variable *table-var*.
(defvar *list-of-nodes* nil)            ; This is a global list, which is
                                        ; initially created and remains
                                        ; unaltered. It contains the numbers of
                                        ; each bond graph element. More
                                        ; information on *list-of-nodes* can
                                        ; be found in the introduction to the
                                        ; function get-node-data-dump.
(defvar *list-of-node-types* nil)       ; Corresponding to each element number
                                        ; in *list-of-nodes*, is the element
                                        ; type in *list-of-node-types*.
                                        ; corresponding to the number. More
                                        ; information on *list-of-node-types*
                                        ; can be found in the introduction
                                        ; to the function get-node-data-dump.
(defvar *list-of-connected-lines* nil)  ; Corresponding to each element number
                                        ; in *list-of-nodes* are all the bonds
                                        ; connected to that element and are
                                        ; stored in *list-of-connected-lines*.
                                        ; More information on
                                        ; *list-of-connected-lines* can
                                        ; be found in the introduction to the
                                        ; function get-node-data-dump.
(defvar *filename* nil)                 ; A global variable containing the
                                        ; filename, without the path, of the
                                        ; most recently read bond graph data
                                        ; file.
(defvar *lines* nil)                    ; The hash table of the bonds, contains
                                        ; the element numbers at either end of
                                        ; the bond and the causality assigned
                                        ; to the bond.
(defvar *line-hash-keys* nil)           ; The list of bond numbers by which the
                                        ; hash table is referenced.
(defvar *nodes* nil)                    ; The hash table containing all the
                                        ; bond graph elements, contains the
                                        ; element type. the documentation
                                        ; string and the equations for that
                                        ; element.
(defvar *node-hash-keys* nil)           ; A list of element numbers, used to
                                        ; reference the hash table *nodes*.
```

```
(defstruct line

;;; This lisp structure is referenced by the list *line-hash-keys*. The hash
;;; table contains the two elements that the bond joins in node-pair. The
;;; first element in node-pair is the element at the end of the bond without
;;; the power half arrow and the second element in node-pair is the element
;;; that power is entering. The causality can either be nil, e or b. It is
;;; nil if causality has been assigned, e is the causal stroke is at the
;;; same end as the power half arrow or b if the causal stroke is at the
;;; opposite end to the power half arrow.

  (node-pair '(0 0))
  (causality nil))


(defstruct node

;;; This list structure contains the data for each bond graph element and is
;;; referenced by *node-hash-keys*. There hash table contain three entries,
;;; the element type, the documentation string for the bond graph element and
;;; the equations for the element. The equations slot is nil if no equations
;;; have yet been obtained.

  (type nil)
  (doc-str nil)
  (equations nil))


(defun reset-hash-dump ()
  (clrhash *lines*)
  (clrhash *nodes*))


(defun init-equ-dump ()

;;; Both the hash table's will be cleared and a new one created

  (setq *lines* (make-hash-table))
  (setq *nodes* (make-hash-table))
  (reset-hash-dump)
  (setq *line-hash-keys* nil
        *node-hash-keys* nil
        *directed-lines* nil)
  (values))


(defun read-data-from-file-dump (filename)

;;; This function reads the data stored in the file filename.data. The format
;;; of the data file is discussed at the beginning of the file enter.lisp.
;;; A portion of the function was originally written by C. Hartmann. The
;;; function was modified and added to by G. Grobbelaar

  (setq filename (concatenate 'string "/usr/bondgraph/data/"
                              (string-downcase filename) ".data"))
```

```
(with-open-file (line-stream filename :direction :input)
                (setf *causality-assigned?* (read line-stream))
                (setf *line-hash-keys* (read line-stream))
                (setf line-keys *line-hash-keys*)
                (dolist (a-line-number line-keys)
                  (setf (gethash a-line-number *lines*)
                        (read line-stream)))
                (setf *node-hash-keys* (read line-stream))
                (setf node-keys *node-hash-keys*)
                (dolist (a-node-number node-keys)
                  (setf (gethash a-node-number *nodes*)
                        (read line-stream)))
                (setf *r-algebraic-loop* (read line-stream))
                (setf *bond-algebraic-loop* (read line-stream))
                (setf *deriv-causality* (read line-stream)))
(values))


(defun get-node-data-dump ()

;;; This function takes the data stored in the two hash tables and converts
;;; it into a form that is more more suited to this program. This function
;;; creates three global lists, *list-of-nodes*, *list-of-node-types* and
;;; *list-of-connected-lines*. *list-of-nodes* is the same as *node-hash-keys*
;;; i.e. it contains a list of all the bond graph element numbers.
;;; *list-of-node-types* contains a list of all element types. The first
;;; element in *list-of-node-types* corresponds to the first element in
;;; *list-of-nodes*. The second element in *list-of-node-types* corresponds to
;;; the second element in *list-of-nodes* and so on.
;;; *list-of-connected-lines* contains a list of all the bonds attached to each
;;; bond graph element. *list-of-connected-lines* has the following format:
;;; ((..(..)..)...(...(bond-number power-direction causality-direction)...)...)
;;; *list-of-connected-lines* therefore, consists of a list of lists. The first
;;; list in *list-of-connected-lines* corresponds to the first element in
;;; *list-of-nodes*. The second list in *list-of-connected-lines*
;;; corresponds to the second element in *list-of-nodes* and so on. In each
;;; of these lists are numerous other lists. Each of these lists corresponds
;;; to a bond attached to the element, if there are two bonds attached to the
;;; bond graph element, there are two lists, if there is one bond attached,
;;; there is or list. In each of these lists corresponding to a bond there
;;; are three elements, the first is the bond number, the second is the power
;;; direction and the third is the causality direction (the  direction of the
;;; effort).
;;; Consider an example
;;;              1                2
;;;     se(1)--------\| 1(2) |--------\ r(3)
;;;                   |
;;;                   | 3
;;;                   |
;;;                   /
;;;                  ---
;;;                  i(4)
;;; The three lists are therefore
;;; *list-of-nodes* is (1 2 3 4)
;;; *list-of-node-types* is (se 1 r i)
;;; *list-of-connected-lines* is (((1 out out))((1 in in)(2 out in)(3 out out))
```

```
;;;                                ((2 in out))((3 in in)))
;;; That is the first element in *list-of-nodes* is number one, it is of type
;;; se (the first element in *list-of-node-types*) and has one bond connected
;;; (the first list in *list-of-connected-lines*), the bond is number 1, the
;;; power is flowing out of the effort source and the effort is also flowing
;;; out of the effort source etc. The second element in *list-of-nodes* is
;;; number 2, it is of type 1 and has three bonds connected, bond 1, bond 2
;;; and bond 3. The power and the effort is the input to the 1-junction on
;;; bond 1, the power is leaving and the effort entering on bond 2 and both
;;; the power and the effort are leaving on bond 3. And so on for the remaining
;;; 2 elements.

  (let ((current-node nil)
        (current-node-type nil)
        (node-lines nil))
    (setf *list-of-nodes* *node-hash-keys*)
    (dolist (node-number *node-hash-keys*)
      (setf *list-of-node-types* (cons (node-type
                                        (gethash node-number *nodes*))
                                       *list-of-node-types*)))
    (setf *list-of-node-types* (reverse *list-of-node-types*))
    (dolist (node-number *node-hash-keys*)
      (dolist (line-number *line-hash-keys*)
        (if (member node-number (line-node-pair (gethash line-number *lines*)))
            (setf node-lines
                  (cons (list line-number
                              (power-direction-dump
                               (position node-number (line-node-pair
                                                      (gethash line-number
                                                               *lines*))))
                              (causality-direction-dump
                               (position node-number (line-node-pair
                                                      (gethash line-number
                                                               *lines*)))
                               (line-causality (gethash line-number
                                                        *lines*))))
                        node-lines))))
      (setf *list-of-connected-lines*
            (cons node-lines *list-of-connected-lines*))
      (setf node-lines nil))
    (setf *list-of-connected-lines* (reverse *list-of-connected-lines*))
    (values)))


(defun power-direction-dump (num)

;;; This function is called by get-node-data-dump, where num is the position
;;; of the node in line-node-pair. If num is zero i.e. the first node is being
;;; considered then the power is flowing out of that node, otherwise (if
;;; num=1 i.e. the second node is being considered) the power is flowing into
;;; the node.

  (if (= num 0)
      (setf power-direction 'out)
      (setf power-direction 'in)))
```

```
(defun causality-direction-dump (num direc)

;;; This function is called by get-node-data-dump, where num is the position of
;;; the node in line-node-pair and direc is line-causality. If num is zero
;;; the first node is being considered and therefore power is flowing out
;;; of the node, if direc is 'e i.e. the causal stroke is at the same end
;;; as the power half arrow, then the causality is 'out, but if direc is 'b
;;; i.e. the causal stroke is at the opposite end to the power half arrow then
;;; causality is 'in. If num is 1 the power is flowing into the node, then if
;;; direc is 'e the causality is 'in otherwise if direc is 'b the causality is
;;; 'out

(cond ((and (equal num 0)
            (equal direc 'e))
       (setf causality-direction 'out))
      ((and (equal num 0)
            (equal direc 'b))
       (setf causality-direction 'in))
      ((and (equal num 1)
            (equal direc 'e))
       (setf causality-direction 'in))
      ((and (equal num 1)
            (equal direc 'b))
       (setf causality-direction 'out))))


(defun dump-acsl-equations ()

;;; This function is used to begin the equation formulation routine. Initially
;;; the hash tables are reset and the data is read from a file
;;; (init-and-read-data). The function then checks to see if the causality has
;;; been assigned, if not the user is prompted to run the causality assignment
;;; routine. If the causality has been assigned to the bond graph, this
;;; function obtains the data needed for the acsl input file (it calls
;;; functions get-init-data, get-inter-data, get-node-equations and
;;; get-end-equations). If there are any algebraic loops, the function
;;; calls either r-element-alge-loop or bond-alge-loop depending on whether
;;; the algebraic loop was caused by assigning an arbitrary causality to a
;;; resistor or a junction structure element. If there is a storage element
;;; with derivative causality, the function derivative-causality is called.
;;; All three of these functions modify and add equations to those equations
;;; entered by the user. If any of the equations entered contain the acsl
;;; statement "constant", then these equations are removed from the list and
;;; placed in a separate list so that they can be put in the "initial" section
;;; in the ACSL input file. Finally all the lists are combined into one and
;;; sent to the function dump-acsl-output. Once the ACSL input file has been
;;; written, the hash tables are re-written to the data file. This is so
;;; that the equations entered by the user are stored with the rest of the
;;; structural data.

  (let ((*list-of-nodes* nil)
        (*list-of-node-types* nil)
        (*list-of-connected-lines* nil)
        (list1 nil)
        (list2 nil)
        (list3 nil)
```

```
      (list4 nil)
      (list5 nil)
      (equ-list nil)
      (output-list nil)
      (pos '0))
    (init-and-read-data)
    (if (equal *causality-assigned?* 'causality-assigned)
        (progn (get-node-data-dump)
               (setf list1 (get-init-data))
               (setf list3 (get-inter-data))
               (dolist (node *list-of-nodes* (values list2))
                 (setf equ-list (append (get-node-equations
                                         node
                                         (nth pos *list-of-node-types*)
                                         (nth pos  *list-of-connected-lines*))
                                        equ-list))
                 (setf pos (+ 1 pos)))
               (if *r-algerbraic-loop*
                   (setf equ-list (r-element-alger-loop equ-list)))
               (if *bond-algerbraic-loop*
                   (setf equ-list (bond-alger-loop equ-list)))
               (if *deriv-causality*
                   (setf equ-list (derivative-causality equ-list)))
               (setf equ-list (reverse equ-list))
               (dolist (equ-string equ-list)
                 (if (search "constant" equ-string)
                     (setf list2 (cons equ-string list2))
                     (setf list4 (cons equ-string list4))))
               (setf list5 (get-end-data))
               (setf output-list (append list1 list2 list3 list4 list5))
               (dump-acsl-output output-list)
               (write-hash-table-to-file-dump *filename*))
        (progn (format t "~|The causality has not yet been assigned, please ~
                          run the causality assignment~%procedure from the ~
                          main menu.~%")
               (loop
                (if (y-or-n-p "~%Have you finished reading the message")
                    (return)))))
    (values)))


(defun init-and-read-data ()

;;; The function init-and-read-data calls the function init-equ-dump, which
;;; resets the hash tables and then reads in the data stored in the file
;;; *filename*.data. The user has the option of retaining or changing
;;; *filename*.

  (init-equ-dump)
  (if *filename*
      (if (not (y-or-n-p "~|Do you want to get the acsl equations of the bond~
                          graph stored in the file~%~a.data" *filename*))
          (progn (format t "~%~%Enter the file name of the file to read from, ~
                          don't include an extension~%it will be taken to be ~
                          taken to be 'data'~%")
                 (setf *filename* (read))))
```

```
    (progn (format t "~|Enter the file name of the file to read from, "
                   don't include an extension~%it will be taken to be "
                   taken to be 'data'~%")
           (setf *filename* (read))))
  (read-data-from-file-dump *filename*))


(defun get-node-equations (node-number node-type connected-line-data)

;;; This function is called by dump-acsl-equations. Depending of the node
;;; type, node-type, being considered one of nine functions is called.

  (let ((equation-list nil))
    (case node-type
      ('r (setf equation-list (r-element-equations node-number
                                                   connected-line-data)))
      ('c (setf equation-list (c-element-equations node-number
                                                   connected-line-data)))
      ('i (setf equation-list (i-element-equations node-number
                                                   connected-line-data)))
      ('1 (setf equation-list (1-element-equations node-number
                                                   connected-line-data)))
      ('0 (setf equation-list (0-element-equations node-number
                                                   connected-line-data)))
      ('tf (setf equation-list (tf-element-equations node-number
                                                     connected-line-data)))
      ('gy (setf equation-list (gy-element-equations node-number
                                                     connected-line-data)))
      ('se (setf equation-list (se-element-equations node-number
                                                     connected-line-data)))
      ('sf (setf equation-list (sf-element-equations node-number
                                                     conn   ed-line-data))))
    (values equation-list)))

;;; Because it is necessary to have parentheses in the element equations,
;;; a method must be used to enter these without causing a lisp error.
;;; The only way a parenthesis is not detected is if it is inside a string,
;;; preceded by a back-slash or if it is enclosed by vertical lines ||.
;;; The last two options are not viable as these will have to be removed
;;; before writing the acsl file, which is clumsy and slow. A string was
;;; therefore chosen. It is occasionally necessary to check if the
;;; string contains only numbers, this can be accomplished by converting
;;; the string to an object using read-from-string.


(defun r-element-equations (node-number line-data)

;;; This function is called by get-node-equations if the element currently
;;; being considered is a resistor. The function informs the user for which
;;; element the equations are being entered. If equations have previously
;;; been entered for the element, these and the current causal implications
;;; are output to the screen. The user is then required to enter whether
;;; these equations are still valid. If they are, the equations in the hash
;;; table are assigned to final-equation. If the equations stored in the hash
;;; table are no longer valid or if there were no equations stored, the user
;;; is prompted to enter the data for the resistor. If the resistor is a
```

```
;;; 1-port the data can be written in one of three forms: If the constitutive
;;; relation is linear, just the resistance need be entered. If the
;;; causal relation is non-linear, the entire non-linear equation, in the
;;; correct causal form (the user is given the required causal form) must be
;;; entered. Finally the data for the resistor can be entered in the form of
;;; a table. If a only a number is entered (the resistance) equation2 is
;;; assigned the value ("constant r" .. "=" num) where .. represents the bond
;;; number and num, the number entered by the user. equation1 is assigned the
;;; value ("e" .. "=r" .. "*f" .. ) or ("f" .. "=(1.0/r" .. ")*e" ..)
;;; depending on the causality. If an equation is entered a list is formed of
;;; the equation and assigned to equation1. If the user enters the word
;;; "table", requesting that the data be entered in the form of a table,
;;; final-equation is assigned the output of the function make-acsl-table. If
;;; the resistor is a multiport the data can be written in one of two forms:
;;; Either each equation is written out in full, when prompted or the data
;;; for each equation can be written in the form of a table. The variable
;;; assignments and function calls are as for the 1-port case. Each equation
;;; is converted to a string. The equations are entered in the hash table and
;;; all empty strings are removed.

  (let ((final-equation nil))
    (format t "~|RESISTOR (ELEMENT NUMBER ~a) CONNECTED TO ~%" node-number)
    (dolist (a-line line-data)
      (format t "BOND ~a " (first a-line)))
    (print-doc-str node-number)
    (if (node-equations (gethash node-number *nodes*))
        (progn (format t "~%~%The causality assigned implies that")
               (dolist (current-line line-data)
                 (if (equal (third current-line) 'in)
                     (setf e-f-out "f")
                     (setf e-f-out "e"))
                 (format t "~%~a~a is a function of " e-f-out
                         (first current-line))
                 (dolist (a-line line-data)
                   (if (equal (third a-line) 'in)
                       (setf e-f-in "e")
                       (setf e-f-in "f"))
                   (format t "~a~a " e-f-in (first a-line))))
               (format t "~%~%The last equations stored were")
               (dolist (an-equation (node-equations (gethash node-number
                                                             *nodes*)))
                 (format t "~%~a" an-equation))
               (if (y-or-n-p "~%~%Do these equations still hold ")
                   (progn (setf final-equation (node-equations
                                                (gethash node-number *nodes*)))
                          (setf ans 'yes))
                   (setf ans 'no))))
    (if (or (not (node-equations (gethash node-number *nodes*)))
            (equal ans 'no))
        (progn (dolist (current-line line-data)
                 (if (equal (length line-data) 1)
                     (progn (format t "~%~%Enter the data for resistance R~a."
                                    (first current-line))
                            (format t "~%If the equation is linear, enter "
                                    only the resistance coefficient.")
                            (format t "~%If the equation is non-linear enter "
```

```
                                    the entire equation with~%"))
                    (format t "~%~%Enter "))
                (if (equal (third current-line) 'in)
                    (setf e-f-out "f")
                    (setf e-f-out "e"))
                (format t "~a~a as a function of " e-f-out (first
                                                            current-line))
                (dolist (a-line line-data)
                  (if (equal (third a-line) 'in)
                      (setf e-f-in "e")
                      (setf e-f-in "f"))
                  (format t "~a~a " e-f-in (first a-line)))
                (format t "~%If the data is in the form of a table enter "
                        \"table\"~%")
                (setf   ationi (read-line))
;;; NOTE read-line reads in a string not an object, therefore
;;; a left or right or both parentheses may be read in
                      'equal equationi "table")
                      (format t "~|The table will be named tab~a~%"
                              *table-var*)
;;; We will call the tables tab1, tab2, tab3 etc, incrementing a counter
;;; every time, so that the tables are unique.
                      (format t "~%If the resistor is not modulated the "
                              independant variables are normally~%")
                      (dolist (a-line line-data)
                        (if (equal (third a-line) 'in)
                            (setf e-f-in "e")
                            (setf e-f-in "f"))
                        (format t "~a~a " e-f-in (first a-line)))
                      (setf final-equation (make-acsl-table e-f-out
                                                            node-number
                                                            (first
                                                             current-line)))
                      (setf *table-var* (+ 1 *table-var*))
                     ((numberp (read-from-string equation1))
;;; NOTE read-from-string converts a string to an object
                      (setf equation2 (list "constant r"
                                            (first current-line) "="
                                            equation1))
                      (if (equal (third current-line) 'out)
                          (setf equationi (list "e"
                                                (first current-line) "=r"
                                                (first current-line) "*f"
                                                (first current-line)))
                          (setf equationi (list "f" (first current-line)
                                                "=(1.0/r"
                                                (first
                                                current-line)
                                                ")*e"
                                                (first current-line)))
                      (setf final-equation (append (list
                                                    (form-string equation1)
                                                    (form-string equation2))
                                                   final-equation)))
                     (t (setf equationi (list equation1))
                        (setf final-equation (append (list
```

```
                                                    (form-string equation1))
                                                   final-equation)))))))
    (setf (node-equations (gethash node-number *nodes*)) final-equation)
    (setf final-equation (remove "" final-equation :test #'equal))
    (values final-equation)))


(defun se-element-equations (node-number line-data)

;;; This function is called by get-node-equations if the element currently
;;; being considered is an effort source. The function informs the user for
;;; which element the equations are being entered. If equations have previously
;;; been entered for the element, these and the current causal implications
;;; are output to the screen. The user is then required to enter whether
;;; these equations are still valid, if they are, the equations in the hash
;;; table are assigned to final-equation. If the equations stored in the hash
;;; table are no longer valid or if there were no equations stored, the user
;;; is prompted to enter the data for the effort source. The data can be
;;; written in one of three forms: If the input is a step at time t=0, only
;;; the magnitude of the step need be entered. If the input is not a step at
;;; time t=0, the user is required to enter the entire equation. In addition,
;;; the data for the source may be written in the form of a table. If a number
;;; is entered, equation is set equal to ("constant e" .. "=" num) where ..
;;; represents the bond number and num the number entered by the user. If an
;;; equation is entered, a list is formed of the input and assigned to
;;; equation. If the user enters the word "table", requesting that the data be
;;; entered in the form of a table, final-equation is assigned the output of
;;; the function make-acsl-table. Finally the equations are entered in the
;;; hash table.

  (let ((connected-line (first line-data)))
    (format t "~|EFFORT SOURCE (ELEMENT NUMBER ~a) CONNECTED TO "%"
            node-number)
    (format t "BOND ~a " (first connected-line))
    (print-doc-str node-number)
    (if (node-equations (gethash node-number *nodes*))
        (progn (format t "~%~%The causality assigned implies that e~a is the "
                       output" (first connected-line))
               (format t "~%~%The last equations stored were")
               (dolist (an-equation (node-equations (gethash node-number
                                                             *nodes*)))
                 (format t "~%~a" an-equation))
               (if (y-or-n-p "~%~%Do these equations still hold ")
                   (progn (setf final-equation (node-equations
                                                (gethash node-number *nodes*)))
                          (setf ans 'yes))
                   (setf ans 'no))))
    (if (or (not (node-equations (gethash node-number *nodes*)))
            (equal ans 'no))
        (progn (format t "~%~%Enter the data for effort source SE~a."
                       (first connected-line))
               (format t "~%If the effort input is a step at t=0 then "
                       enter only the height of the step.")
               (format t "~%If the effort input is anything but a step "
                       at t-0 then enter the entire")
               (format t "~%equation, using the form e~a=.... "
```

```
                (first connected-line))
          (format t "~%If the data is in the form of a table enter ~
                \"table\"~%")
          (setf equation (read-line))
          ;;; NOTE read-line reads in a string not an object, therefore a
          ;;; left or right or both parentheses may be read in
          (cond ((equal equation "table")
                (format t "~|The table will be named tab~a~%"
                      *table-var*)
;;; We will call the tables tab1, tab2, tab3 etc, incrementing a counter
;;; every time, so that the tables are unique.
                (format t "~%If the effort source is not modulated the ~
                      independant variable is normally time~%(t)")
                (setf final-equation (make-acsl-table "e"
                                                      node-number
                                                      (first
                                                       connected-line)))
                (setf *table-var* (+ 1 *table-var*)))
               ((numberp (read-from-string equation))
;;; NOTE read-from-string converts a string to an object
                (setf equation (list "constant e" (first connected-line)
                                     "=" equation))
                (setf final-equation (list (form-string equation))))
               (t (setf equation (list equation))
                  (setf final-equation (list (form-string equation)))))))
    (setf (node-equations (gethash node-number *nodes*)) final-equation)
    (values final-equation)))


(defun sf-element-equations (node-number line-data)

;;; This function is called by get-node-equations if the element currently
;;; being considered is an flow source. The function informs the user for
;;; which element the equations are being entered. If equations have previously
;;; been entered for the element, these and the current causal implications
;;; are output to the screen. The user is then required to enter whether
;;; these equations are still valid, if they are, the equations in the hash
;;; table are assigned to final-equation. If the equations stored in the hash
;;; table are no longer valid or if there were no equations stored, the user
;;; is prompted to enter the data for the flow source. The data can be
;;; written in one of three forms: If the input is a step at time t=0, only
;;; the magnitude of the step need be entered. If the input is not a step at
;;; time t=0, the user is required to enter the entire equation. In addition,
;;; the data for the source may be written in the form of a table. If a number
;;; is entered, equation is set equal to ("constant f" .. "=" num) where ..
;;; represents the bond number and num the number entered by the user. If an
;;; equation is entered, a list is formed of the input and assigned to
;;; equation. If the user enters the word "table", requesting that the data
;;; be entered in the form of a table, final-equation is assigned the output
;;; of the function make-acsl-table. Finally the equations are entered in the
;;; hash table.

  (let ((connected-line (first line-data)))
    (format t "~|FLOW SOURCE (ELEMENT NUMBER ~a) CONNECTED TO ~%" node-number)
    (format t "BOND ~a " (first connected-line))
    (print-doc-str node-number)
```

```
    (if (node-equations (gethash node-number *nodes*))
        (progn (format t "~%~%The causality assigned implies that f~a is the "
                       output" (first connected-line))
               (format t "~%~%The last equations stored were")
               (dolist (an-equation (node-equations (gethash node-number
                                                             *nodes*)))
                 (format t "~%~a" an-equation))
               (if (y-or-n-p "~%~%Do these equations still hold ")
                   (progn (setf final-equation (node-equations
                                                (gethash node-number *nodes*)))
                          (setf ans 'yes))
                   (setf ans 'no))))
    (if (or (not (node-equations (gethash node-number *nodes*)))
            (equal ans 'no))
        (progn (format t "~%~%Enter the data for flow source SF~a."
                       (first connected-line))
               (format t "~%If the flow input is a step at t=0 then enter "
                       only the height of the step.")
               (format t "~%If the flow input is anything but a step at t=0 "
                       then enter the entire")
               (format t "~%equation, using the form f~a=....."
                       (first connected-line))
               (format t "~%If the data is in the form of a table, enter "
                       \"table\"~%")
               (setf equation (read-line))
               ;;; NOTE read-line reads in a string not an object, therefore a
               ;;; left or right or both parentheses may be read in
               (cond ((equal equation "table")
                      (format t "~|The table will be named tab~a~%"
                              *table-var*)
;;; We will call the tables tab1, tab2, tab3 etc, incrementing a counter
;;; every time, so that the tables are unique.
                      (format t "~%If the flow source is not modulated the "
                              independant variable is normally time~%(t)~%")
                      (setf final-equation (make-acsl-table "f"
                                                            node-number
                                                            (first
                                                             connected-line)))
                      (setf *table-var* (+ 1 *table-var*)))
                     ((numberp (read-from-string equation))
                      ;;; NOTE read-from-string converts a string to an object
                      (setf equation (list "constant f" (first connected-line)
                                           "=" equation))
                      (setf final-equation (list (form-string equation))))
                     (t (setf equation (list equation))
                        (setf final-equation (list (form-string equation)))))))
    (setf (node-equations (gethash node-number *nodes*)) final-equation)
    (values final-equation)))


(defun c-element-equations (node-number line-data)

;;; This function is called by get-node-equations if the element currently
;;; being considered is a capacitor. The function informs the user for which
;;; element the equations are being entered. If equations have previously
;;; been entered for the element, these and the current causal implications
```

```
;;; are output to the screen. The user is then required to enter whether
;;; these equations are still valid. If they are, the equations in the hash
;;; table are assigned to final-equation. If the equations stored in the hash
;;; table are no longer valid or if there were no equations stored, the user
;;; is prompted to enter the data for the capacitor. If the capacitor is a
;;; 1-port the data can be written in one of three forms: If the constitutive
;;; relation is linear, just the capacitance need be entered. If the
;;; causal relation is non-linear, the entire non-linear equation, in the
;;; correct causal form (the user is given the required causal form) must be
;;; entered. Finally the data for the capacitor can be entered in the form of
;;; a table. If a only a number is entered (the capacitance) equation2 is
;;; assigned the value ("constant c" .. "=" num) where .. represents the bond
;;; number and num the number entered by the user. equation1 is assigned the
;;; value ("e" .. "=(1.0/c" .. ")*q" .. ) or ("q" .. "=c" .. "*e" ..)
;;; depending on the causality. If an equation is entered a list is formed of
;;; the equation and assigned to equation1. If the user enters the word
;;; "table", requesting that the data be entered in the form of a table,
;;; final-equation is assigned the output of the function make-acsl-table. If
;;; the capacitor has integral causality assigned to it, the user is required
;;; to enter the initial value of the displacement (at time t=0). If the user
;;; enters only a number, equation4 is set equal to ("constant q" .. "i=" num)
;;; and equation3 is set equal to ("q" .. "=integ(f" .. ",q" .. "i)"), where
;;; .. represents the bond number and num the number entered by the user. If
;;; the user enters an expression for the initial displacement equation3 is
;;; set equal to ("q" .. "=integ(f" .. "," expr ")"), where .. represents the
;;; bond number and expr the expression entered by the user, equation4 is set
;;; equal to the empty string. If the capacitor has derivative causality
;;; assigned to it, equation3 is set equal to ("f" .. "=q" .. "d"), where ..
;;; represents the bond number and d the derivative (note the acsl derivative
;;; function is not used, instead the equation is differentiated implicitly),
;;; equation4 is set equal to the empty string. If the capacitor is a
;;; multiport the data can be written in one of two forms: Either each
;;; equation is written out in full, when prompted or the data for each
;;; equation can be written in the form of a table. The variable assignments,
;;; initial displacement assignment and function calls are as for the 1-port
;;; case. Each equation is converted to a string. The equations are entered
;;; in the hash table and all empty strings are removed.

(let ((final-equation nil))
  (format t "~|CAPACITOR (ELEMENT NUMBER ~a) CONNECTED TO ~%" node-number)
  (dolist (a-line line-data)
    (format t "BOND ~a " (first a-line)))
  (print-doc-str node-number)
  (if (node-equations (gethash node-number *nodes*))
      (progn (format t "~%~%The causality assigned implies that")
             (dolist (current-line line-data)
               (if (equal (third current-line) 'out)
                   (setf e-f-out "e")
                   (setf e-f-out "q"))
               (format t "~%~a~a is a function of " e-f-out
                       (first current-line))
               (dolist (a-line line-data)
                 (if (equal (third a-line) 'out)
                     (setf e-f-in "q")
                     (setf e-f-in "e"))
                 (format t "~a~a " e-f-in (first a-line))))
```

```
                (format t "~%~%The last equations stored were")
                (dolist (an-equation (node-equations (gethash node-number
                                                              *nodes*)))
                  (format t "~%~a" an-equation))
                (if (y-or-n-p "~%~%Do these equations still hold ")
                    (progn (setf final-equation (node-equations
                                                 (gethash node-number *nodes*)))
                           (setf ans 'yes))
                    (setf ans 'no))))
    (if (or (not (node-equations (gethash node-number *nodes*)))
            (equal ans 'no))
        (progn (dolist (current-line line-data)
                 (if (equal (length line-data) 1)
                     (progn (format t "~%~%Enter the data for capacitance C~a."
                                    (first current-line))
                            (format t "~%If the equation is linear, enter "
                                    only the capacitance coefficient.")
                            (format t "~%If the equation is non-linear enter "
                                    the entire equation with~%"))
                     (format t "~%~%Enter "))
                 (if (equal (third current-line) 'out)
                     (setf e-f-out "e")
                     (setf e-f-out "q"))
                 (format t "~a~a " e-f-out " a function of " e-f-out
                         (first current-line))
                 (dolist (a-line line-data)
                   (if (equal (third a-line) 'out)
                       (setf e-f-in "q")
                       (setf e-f-in "e"))
                   (format t "~a~a " e-f-in (first a-line)))
                 (format t "~%If the data is in the form of a table, enter "
                         \"table\"~%")
                 (setf equation1 (read-line))
                 ;;; NOTE read-line reads in a string not an object, therefore
                 ;;; a left or right or both parentheses may be read in
                 (cond ((equal equation1 "table")
                        (format t "~%The table will be named tab~a~%"
                                *table-var*)
;;; We will call the tables tab1, tab2, tab3 etc, incrementing a counter
;;; every time, so that the tables are unique.
                        (format t "~%If the capacitor is not modulated the "
                                independant variables are normally~%")
                        (dolist (a-line line-data)
                          (if (equal (third a-line) 'out)
                              (setf e-f-in "q")
                              (setf e-f-in "e"))
                          (format t "~a~a " e-f-in (first a-line)))
                        (setf equation1 (make-acsl-table e-f-out
                                                         node-number
                                                         (first
                                                          current-line)))
                        (setf equation2 (list""))
                        (setf *table-var* (+ 1 *table-var*)))
                       ((numberp (read-from-string equation1))
                        ;;; NOTE read-from-string converts a string to an object
                        (setf equation2 (list "constant c"
```

```
                                        (first current-line)
                                        "=" equation1))
                        (if (equal (third current-line) 'out)
                            (setf equation1 (list (form-string (list
                                        "e" (first current-line)
                                        "=(1.0/c"
                                        (first current-line) ")*q"
                                        (first current-line)))))
                            (setf equation1 (list (form-string (list
                                        "q" (first current-line)
                                        "=c"
                                        (first current-line) "*e"
                                        (first current-line)))))))
                    (t (setf equation1 (list equation1))
                       (setf equation2 (list "")))))
                (if (equal (third current-line) 'out)
                    (progn (format t "~%Enter the value of q~a at time t=0~%"
                                   (first current-line))
                           (setf initial-value (read-line))
                           (if (numberp (read-from-string initial-value))
                               (progn (setf equation4 (list "constant q"
                                                            (first
                                                             current-line)
                                                            "i="
                                                            initial-value))
                                      (setf equation3 (list "q"
                                                            (first
                                                             current-line)
                                                            "=integ(f"
                                                            (first
                                                             current-line)
                                                            ",q"
                                                            (first
                                                             current-line)
                                                            "i)")))
                                   (progn (setf equation4 (list ""))
                                          (setf equation3 (list "q"
                                                            (first
                                                             current-line)
                                                            "=integ(f"
                                                            (first
                                                             current-line)
                                                            "," initial-value
                                                            ")")))))
                    (progn (setf equation4 (list ""))
                           (setf equation3 (list "f" (first current-line) "=q"
                                                 (first current-line) "d"))))
                (setf final-equation (append equation1
                                             (list (form-string equation2))
                                             (list (form-string equation3))
                                             (list (form-string equation4))
                                             final-equation)))))
        (setf (node-equations (gethash node-number *nodes*)) final-equation)
        (setf final-equation (remove "" final-equation :test #'equal))
        (values final-equation)))
```

```
(defun i-element-equations (node-number line-data)

;;; This function is called by get-node-equations if the element currently
;;; being considered is an inertia. The function informs the user for which
;;; element the equations are being entered. If equations have previously
;;; been entered for the element, these and the current causal implications
;;; are output to the screen. The user is then required to enter whether
;;; these equations are still valid. If they are, the equations in the hash
;;; table are assigned to final-equation. If the equations stored in the hash
;;; table are no longer valid or if there were no equations stored, the user
;;; is prompted to enter the data for the inertia. If the inertia is a
;;; 1-port the data can be written in one of three forms: If the constitutive
;;; relation is linear, just the inertance need be entered. If the
;;; causal relation is non-linear, the entire non-linear equation, in the
;;; correct causal form (the user is given the required causal form) must be
;;; entered. Finally the data for the inertia may be entered in the form of
;;; a table. If a only a number is entered (the inertance) equation2 is
;;; assigned the value ("constant i" .. "=" num) where .. represents the bond
;;; number and num the number entered by the user. equation1 is assigned the
;;; value ("f" .. "=(1.0/i" .. ")*p" .. ) or ("p" .. "=i" .. "*f" ..)
;;; depending on the causality. If an equation is entered a list is formed of
;;; the equation and assigned to equation1. If the user enters the word
;;; "table", requesting that the data be entered in the form of a table,
;;; final-equation is assigned the output of the function make-acsl-table. If
;;; the inertia has integral causality assigned to it, the user is required
;;; to enter the initial value of the momentum (at time t=0). If the user
;;; enters only a number, equation4 is set equal to ("constant p" .. "i=" num)
;;; and equation3 is set equal to ("p" .. "=integ(e" .. ",p" .. "i)"), where
;;; .. represents the bond number and num the number entered by the user. If
;;; the user enters an expression for the initial momentum equation3 is
;;; set equal to ("p" .. "=integ(e" .. "," expr ")"), where .. represents the
;;; bond number and expr the expression entered by the user, equation4 is set
;;; equal to the empty string. If the inertia has derivative causality
;;; assigned to it, equation3 is set equal to ("e" .. "=p" .. "d"), where ..
;;; represents the bond number and d the derivative (note the acsl derivative
;;; function is not used, instead the equation is differentiated implicitly),
;;; equation4 is set equal to the empty string. If the inertia is a
;;; multiport the data can be written in one of two forms: Either each
;;; equation is written out in full, when prompted or the data for each
;;; equation can be written in the form of a table. The variable assignments,
;;; initial momentum assignment and function calls are as for the 1-port
;;; case. Each equation is converted to a string. The equations are entered
;;; in the hash table and all empty strings are removed.

  (let ((final-equation nil))
    (format t "~|~%INERTIA (ELEMENT NUMBER ~a) CONNECTED TO ~%" node-number)
    (dolist (a-line line-data)
      (format t "BOND ~a " (first a-line)))
    (print-doc-str node-number)
    (if (node-equations (gethash node-number *nodes*))
        (progn (format t "~%~%The causality assigned implies that")
               (dolist (current-line line-data)
                 (if (equal (third current-line) 'in)
                     (setf e-f-out "f")
                     (setf e-f-out "p"))
                 (format t "~%~a~a is a function of " e-f-out
```

```
                    (first current-line))
              (dolist (a-line line-data)
                (if (equal (third a-line) 'in)
                    (setf e-f-in "p")
                    (setf e-f-in "f"))
                (format t "~a~a " e-f-in (first a-line))))
            (format t "~%~%The last equation entered were")
            (dolist (an-equation (node-equations (gethash node-number
                                                          *nodes*)))
              (format t "~%~a" an-equation))
            (if (y-or-n-p "~%~%Do those equations still hold ")
                (progn (setf final-equation (node-equations
                                             (gethash node-number *nodes*)))
                       (setf ans 'yes))
                (setf ans 'no))))
    (if (or (not (node-equations (gethash node-number *nodes*)))
            (equal ans 'no))
        (progn (dolist (current-line line-data)
                 (if (equal (length line-data) 1)
                     (progn (format t "~%~%Enter the data for inertiance I~a."
                                    (first current-line))
                            (format t "~%If the equation is linear, enter ~
                                       only the inertiance coefficient.")
                            (format t "~%If the equation is non-linear enter ~
                                       the entire equation with~%"))
                     (format t "~%~%Enter "))
                 (if (equal (third current-line) 'in)
                     (setf e-f-out "f")
                     (setf e-f-out "p"))
                 (format t "~a~a as a function of " e-f-out
                         (first current-line))
                 (dolist (a-line line-data)
                   (if (equal (third a-line) 'in)
                       (setf e-f-in "p")
                       (setf e-f-in "f"))
                   (format t "~a~a " e-f-in (first a-line)))
                 (format t "~%If the data is in the form of a table, enter ~
                            \"table\"~%")
                 (setf equation1 (read-line))
                 ;;; NOTE read-line reads in a string not an object, therefore
                 ;;; a left or right or both parentheses may be read in
                 (cond ((equal equation1 "table")
                        (format t "~%The table will be named tab~a~%"
                                *table-var*)
;;; We will call the tables tab1, tab2, tab3 etc, incrementing a counter
;;; every time, so that the tables are unique.
                        (format t "~%If the inertia is not modulated the ~
                                   independant variables are normally~%")
                        (dolist (a-line line-data)
                          (if (equal (third a-line) 'in)
                              (setf e-f-in "p")
                              (setf e-f-in "f"))
                          (format t "~a~a " e-f-in (first a-line)))
                        (setf equation1 (make-acsl-table e-f-out
                                                         node-number
                                                         (first
```

```
                                        current-line)))
      (setf equation2 (list ""))
      (setf *table-var* (+ 1 *table-var*)))
     ((numberp (read-from-string equation1))
    ;;; NOTE read-from-string converts a string to an object
      (setf equation2 (list "constant i"
                            (first current-line) "="
                            equation1))
      (if (equal (third current-line) 'in)
          (setf equation1 (list (form-string (list
                                "f" (first current-line)
                                "=(1.0/i"
                                (first current-line) ")*p"
                                (first current-line)))))
          (setf equation1 (list (form-string (list
                                "p" (first current-line)
                                "=i" (first current-line)
                                "*f"
                                (first current-line)))))))
     (t (setf equation2 (list ""))
        (setf equation1 (list equation1))))
(if (equal (third current-line) 'in)
    (progn (format t "~%Enter the value of p at time t=0~%"
                   (first current-line))
           (setf initial-value (read-line))
           (if (numberp (read-from-string initial-value))
               (progn (setf equation4 (list "constant p"
                                            (first
                                             current-line) "i="
                                            initial-value))
                      (setf equation3 (list "p" (first
                                           current-line)
                                            "=integ(e"
                                            (first
                                             current-line)
                                            ",p"
                                            (first
                                             current-line)
                                            "i)")))
               (progn (setf equation4 (list ""))
                      (setf equation3 (list "p" (first
                                           current-line)
                                            "=integ(e"
                                            (first
                                             current-line)
                                            "," initial-value
                                            ")")))))
    (progn (setf equation4 (list ""))
           (setf equation3 (list "e" (first current-line)
                                 "=p" (first current-line)
                                 "d"))))
(setf final-equation (append equation1
                             (list (form-string equation2))
                             (list (form-string equation3))
                             (list (form-string equation4))
                             final-equation)))))
```

```
    (setf (node-equations (gethash node-number *nodes*)) final-equation)
    (setf final-equation (remove "" final-equation :test #'equal))
    (values final-equation)))


(defun tf-element-equations (node-number line-data)

;;; This function is called by get-node-equations if the element currently
;;; being considered is a transformer. The function informs the user for which
;;; element the equations are being entered. If equations have previously been
;;; entered for the element, these and the current causal implications
;;; are output to the screen. The user is then required to enter whether
;;; these equations are still valid. If they are, the equations in the hash
;;; table are assigned to final-equation. If the equations stored in the hash
;;; table are no longer valid or if there were no equations stored, the user
;;; is prompted to enter the data for the transformer modulus. The user is
;;; informed, based on which port the power is entering, of the definition of
;;; the modulus in terms of the two efforts. The data for the transformer
;;; modulus can be written in one of three forms: If the transformer is not
;;; modulated the user needs only to enter the values of the modulus. If the
;;; transformer is modulated, the user has to enter the expression for to
;;; modulus. The data for the modulus can also be written in the form of a
;;; t.   . If only a number is entered equation3 is assigned the value
;;; ("..constant m" .. "=" num) where .. represents the node number and num
;;; the number entered by the user. equation1 and equation2 are the two
;;; constitutive relations, the form they take, being dependent on the power
;;; and causality direction. If an expression is entered equation3 is set
;;; equal to the empty string, equation1 and equation2 are as before. If the
;;; user enters the word "table", equation3 is set equal to the output from
;;; the function make-acsl-table, equation1 and equation2 are again the same.
;;; Each equation is converted to a string. The equations are entered in the
;;; hash table and all empty strings are removed.

  (format t "~|TRANSFORMER (ELEMENT NUMBER ~a) CONNECTED TO ~%" node-number)
  (dolist (a-line line-data)
    (format t "BOND ~a " (first a-line)))
  (print-doc-str node-number)
  (if (node-equations (gethash node-number *nodes*))
      (progn (format t "~%~%The causality assigned implies that")
             (if (equal (second (first line-data)) 'out)
                 (progn (setf pout '0)
                        (setf pin '1))
                 (progn (setf pout '1)
                        (setf pin '0)))
             (if (equal (third (first line-data)) 'out)
                 (progn (setf cout '0)
                        (setf cin '1))
                 (progn (setf cout '1)
                        (setf cin '0)))
             (format t "~%e~a = m e~a where m is the modulus~%"
                     (first (nth pin line-data)) (first (nth pout line-data)))
             (format t "~%~%The last equations stored were")
             (dolist (an-equation (node-equations (gethash node-number
                                                           *nodes*)))
               (format t "~%~a" an-equation))
             (if (y-or-n-p "~%~%Do these equations still hold ")
```

```
          (progn (setf final-equation (node-equations
                                        (gethash node-number *nodes*)))
                 (setf ans 'yes))
                 (setf ans 'no))))
  (if (or (not (node-equations (gethash node-number *nodes*)))
          (equal ans 'no))
      (progn (if (equal (second (first line-data)) 'out)
                 (progn (setf pout '0)
                        (setf pin '1))
                 (progn (setf pout '1)
                        (setf pin '0)))
             (if (equal (third (first line-data)) 'out)
                 (progn (setf cout '0)
                        (setf cin '1))
                 (progn (setf cout '1)
                        (setf cin '0)))
             (format t "~%~%Enter the modulus m of the transformer between "
                     lines ~a and ~a" (first (first line-data))
                     (first (second line-data)))
             (format t "~%The modulus m is defined such that e~a = m e~a~%"
                     (first (nth pin line-data)) (first (nth pout line-data)))
             (format t "~%If the data for the modulus is in the form of a "
                     table, enter \"table\"~%")
             (setf modulus (read-line))
             ;;; NOTE read-line reads in a string not an object, therefore a
             ;;; left or right or both parentheses may be read in
             (cond ((equal modulus "table")
                    (format t "~%It the transformer is modulated, only the "
                            modulus can be entered in the form "
                            of a table, this is to ensure that the transformer "
                            remains energy conserving.~%")
                    (format t "~%The table will be named tab~a~%"
                            *table-var*)
;;; We will call the tables tab1, tab2, tab3 etc, incrementing a counter
;;; every time, so that the tables are unique.
                    (setf equation3 (make-acsl-table "m" node-number
                                                     node-number))
                    (setf *table-var* (+ 1 *table-var*))
                    (if (equal (second (first line-data))
                               (third (first line-data)))
                        (progn (setf equation1 (list "f"
                                                     (first (nth cin line-data))
                                                     "=(1.0/m" node-number
                                                     ")*f"
                                                     (first (nth cout
                                                                 line-data))))
                               (setf equation2 (list "e"
                                                     (first (nth cout
                                                                 line-data))
                                                     "=(1.0/m" node-number
                                                     ")*e"
                                                     (first (nth cin
                                                                 line-data)))))
                        (progn (setf equation1 (list "f"
                                                     (first (nth cin
                                                                 line-data))
```

```
                                        "=m" node-number "*f"
                                        (first
                                         (nth cout line-data))))
                    (setf equation2 (list "e"
                                        (first (nth cout
                                                    line-data))
                                        "=m" node-number "*e"
                                        (first
                                         (nth cin
                                              line-data))))))
          ((numberp (read-from-string modulus))
           (setf equation3 (list (form-string
                                  (list "constant m" node-number
                                        "=" modulus))))
           (if (equal (second (first line-data))
                      (third (first line-data)))
               (progn (setf equation1 (list "f"
                                        (first (nth cin line-data))
                                        "=(1.0/m" node-number
                                        ")*f"
                                        (first (nth cout
                                                    line-data))))
                      (setf equation2 (list "e"
                                        (first (nth cout
                                                    line-data))
                                        "=(1.0/m" node-number
                                        ")*e"
                                        (first (nth cin
                                                    line-data)))))
               (progn (setf equation1 (list "f"
                                        (first (nth cin
                                                    line-data))
                                        "=m" node-number "*f"
                                        (first
                                         (nth cout line-data))))
                      (setf equation2 (list "e"
                                        (first (nth cout
                                                    line-data))
                                        "=m" node-number "*e"
                                        (first
                                         (nth cin
                                              line-data))))))))
          (t (setf equation3 (list ""))
             (if (equal (second (first line-data))
                        (third (first line-data)))
                 (progn (setf equation1 (list "f"
                                          (first (nth cin
                                                      line-data))
                                          "=(1.0/(" modulus
                                          "))*f"
                                          (first (nth cout
                                                      line-data))))
                        (setf equation2 (list "e"
                                          (first
                                           (nth cout line-data))
                                          "=(1.0/(" modulus
```

```
                                                    "))*e"
                                                    (first
                                                     (nth cin line-data)))))
                                (progn (setf equation1 (list "f"
                                                    (first (nth cin
                                                                line-data))
                                                    "=(" modulus ")*f"
                                                    (first
                                                     (nth cout line-data))))
                                       (setf equation2 (list "e"
                                                    (first
                                                     (nth cout line-data))
                                                    "=(" modulus ")*e"
                                                    (first
                                                     (nth cin
                                                          line-data)))))))))
                    (setf final-equation (append (list (form-string equation1))
                                                 (list (form-string equation2))
                                                 equation3))))
          (setf (node-equations (gethash node-number *nodes*)) final-equation)
          (setf final-equation (remove "" final-equation :test #'equal))
          (values final-equation))


(defun gy-element-equations (node-number line-data)

;;; This function is called by get-node-equations if the element currently
;;; being considered is a gyrator. The function informs the user for which
;;; element the equations are being entered. If equations have previously been
;;; entered for the element, these and the current causal implications
;;; are output to the screen. The user is then required to enter whether
;;; these equations are still valid. If they are, the equations in the hash
;;; table are assigned to final-equation. If the equations stored in the hash
;;; table are no longer valid or if there were no equations stored, the user
;;; is prompted to enter the data for the gyrator modulus. The user is
;;; informed, based on which port the power is entering, of the definition of the
;;; modulus in terms of the two efforts. The data for the gyrator modulus
;;; can be written in one of three forms: If the gyrator is not modulated
;;; the user needs only to enter the values of the modulus. If the gyrator
;;; is modulated, the user has to enter the expression for to modulus. The
;;; data for the modulus can also be written in the form of a table. If a
;;; only a number is entered equation3 is assigned the value
;;; ("constant r" .. "=" num) where .. represents the node number and num
;;; the number entered by the user. equation1 and equation2 are the two
;;; constitutive relations, the form they take, being dependent on the power
;;; and causality direction. If an expression is entered equation2 is set
;;; equal to the empty string, equation1 and equation2 are as before. If the
;;; user enters the word "table", equation3 is set equal to the output from
;;; the function make-acsl-table, equation1 and equation2 are again the same.
;;; Each equation is converted to a string. The equations are entered in the
;;; hash table and all empty strings are removed.

  (format t "~|GYRATOR (ELEMENT NUMBER ~a) CONNECTED TO ~%" node-number)
  (dolist (a-line line-data)
    (format t "BOND ~a " (first a-line)))
  (print-doc-str node-number)
```

```
(if (node-equations (gethash node-number *nodes*))
    (progn (format t "~%~%The causality assigned implies that")
           (if (equal (second (first line-data)) 'out)
               (progn (setf pout '0)
                      (setf pin '1))
               (progn (setf pout '1)
                      (setf pin '0)))
           (format t "~%e~a = r f~a where r is the modulus~%"
                   (first (nth pin line-data)) (first (nth pout line-data)))
           (format t "~%~%The last equations stored were")
           (dolist (an-equation (node-equations (gethash node-number
                                                         *nodes*)))
             (format t "~%~a" an-equation))
           (if (y-or-n-p "~%~%Do these equations still hold ")
               (progn (setf final-equation (node-equations
                                            (gethash node-number *nodes*)))
                      (setf ans 'yes))
               (setf ans 'no))))
(if (or (not (node-equations (gethash node-number *nodes*)))
        (equal ans 'no))
    (progn (if (equal (second (first line-data)) 'out)
               (progn (setf pout '0)
                      (setf pin '1))
               (progn (setf pout '1)
                      (setf pin '0)))
           (format t "~%~%Enter the modulus r of the gyrator between lines ~
                  ~a and ~a" (first (first line-data))
                  (first (second line-data)))
           (format t "~%The modulus r is defined such that e~a = r f~a~%"
                   (first (nth pin line-data)) (first (nth pout line-data)))
           (format t "~%If the data for the modulus is in the form of a ~
                  table, enter \"table\"~%")
           (setf modulus (read-line))
           ;;; NOTE read-line reads in a string not an object, therefore a
           ;;; left or right or both parentheses may be read in
           (cond ((equal modulus "table")
                  (format t "~|It the gyrator is modulated, only the ~
                         modulus can be entered in the form ~
                         of a table, this is to ensure that the gyrator ~
                         remains energy conserving.~%")
                  (format t "~|The table will be named tab~a~%"
                          *table-var*)
;;; We will call the tables tab1, tab2, tab3 etc, incrementing a counter
;;; every time, so that the tables are unique.
                  (setf equation3 (make-accl-table "r"
                                                   node-number
                                                   node-number))
                  (setf *table-var* (+ 1 *table-var*))
                  (if (equal (third (first line-data)) 'in)
                      (progn (setf equation1 (list "f"
                                                   (first (nth pin line-data))
                                                   "=(1.0/r" node-number
                                                   ")*e"
                                                   (first
                                                    (nth pout line-data))))
                             (setf equation2 (list "f"
```

```
                                    (first (nth pout
                                                line-data))
                                    "=(1.0/r" node-number
                                    ")*e"
                                    (first
                                     (nth pin line-data))))))
            (progn (setf equation1 (list "e"
                                    (first (nth pin
                                                line-data))
                                    "=r" node-number "*f"
                                    (first
                                     (nth pout line-data))))
                   (setf equation2 (list "e"
                                    (first (nth pout
                                                line-data))
                                    "=r" node-number "*i"
                                    (first
                                     (nth pin line-data))))))))
      ((numberp (read-from-string modulus))
       (setf equation3 (list (form-string
                              (list "constant r" node-number
                                    "=" modulus))))
       (if (equal (third (first line-data)) 'in)
           (progn (setf equation1 (list "f"
                                    (first (nth pin line-data))
                                    "=(1.0/r" node-number
                                    ")*e"
                                    (first
                                     (nth pout line-data))))
                  (setf equation2 (list "f"
                                    (first (nth pout
                                                line-data))
                                    "=(1.0/r" node-number
                                    ")*e"
                                    (first
                                     (nth pin line-data)))))
           (progn (setf equation1 (list "e"
                                    (first (nth pin
                                                line-data))
                                    "=r" node-number "*f"
                                    (first
                                     (nth pout line-data))))
                  (setf equation2 (list "e"
                                    (first (nth pout
                                                line-data))
                                    "=r" node-number "*f"
                                    (first
                                     (nth pin line-data)))))))
      (t (setf equation3 (list ""))
         (if (equal (third (first line-data)) 'in)
             (progn (setf equation1 (list "f"
                                     (first (nth pin
                                                 line-data))
                                     "=(1.0/(" modulus
                                     "))*e"
                                     (first
```

```
                                                    (nth pout line-data))))
                                    (setf equation2 (list "f"
                                                          (first (nth pout
                                                                      line-data))
                                                          "=(1.0/(" modulus
                                                          "))*c"
                                                          (first
                                                           (nth pin line-data)))))
                                (progn (setf equation1 (list "e"
                                                          (first (nth pin
                                                                      line-data))
                                                          "=(" modulus  ")*f"
                                                          (first
                                                           (nth pout line-data))))
                                       (setf equation2 (list "e"
                                                          (first (nth pout
                                                                      line-data))
                                                          "=(" modulus ")*f"
                                                          (first
                                                           (nth pin
                                                                line-data)))))))))
                    (setf final-equation (append (list (form-string equation1))
                                                 (list (form-string equation2))
                                                 equation3))))
  (setf (node-equations (gethash node number *nodes*)) final-equation)
  (setf final-equation (remove "" final-equation :test #'equal))
  (values final-equation))


(defun 1-element-equations (node-number line-data)

;;; If the element being considered is of type 1 this function is called by
;;; get-node-equations. The function takes the list passed to it and finds
;;; which list in it has the effort as the output. This inner list is removed
;;; and consed onto the front of the list. All the lists but the first in
;;; line-data are considered in turn. If the power direction of the list
;;; being considered is the same as the power direction of the first list then
;;; the sign is set to "-" otherwise the sign is set to "+". The sign, "e"
;;; and the bond number corresponding to each list but the first in line-data
;;; is consed onto the list equation. Finally "=", the bond number of the
;;; first list in line-data and "e" are consed onto equation. equation is
;;; converted into a string and consed onto final-equation. Each list but
;;; the first in line-data is again considered. equation is assigned the
;;; value ("f" .a. "=f" .b.) where .a. is the bond number in list being
;;; considered and .b. of the bond in the first list. A string is formed of
;;; equation and it is consed onto final-equation. The equations are then
;;; added into the hash table.

  (let ((equation nil)
        (final-equation nil))
    (dolist (single-line line-data)
      (if (equal (third single-line) 'out)
          (setf pos (position single-line line-data))))
    (setf line-data (cons (nth pos line-data)
                          (remove (nth pos line-data) line-data)))
    (dolist (a-line (rest line-data))
```

```
      (if (equal (second (first line-data)) (second a-line))
          (setf sign "-")
          (setf sign "+"))
      (setf equation (cons sign (cons "e" (cons (first a-line) equation)))))
    (setf equation (cons "e" (cons (first (first line-data))
                                   (cons "=" equation))))
    (setf final-equation (cons (form-string equation) final-equation))
    (dolist (a-line (rest line-data) (values final-equation))
      (setf equation (list "f" (first a-line) "=f" (first (first line-data))))
      (setf final-equation (cons (form-string equation) final-equation)))
    (setf (node-equations (gethash node-number *nodes*)) final-equation)
    (values final-equation)))


(defun 0-element-equations (node-number line-data)

;;; If the element being considered is of type 0 this function is called by
;;; get-node-equations. The function takes the list passed to it and finds
;;; which list in it has the effort as the input. This inner list is removed
;;; and consed onto the front of the list. All the lists but the first in
;;; line-data are considered in turn. If the power direction of the list
;;; being considered is the same as the power direction of the first list then
;;; the sign is set to "-" otherwise the sign is set to "+". The sign, "f"
;;; and the bond number corresponding to each list but the first in line-data
;;; is consed onto the list equation. Finally "=", the bond number of the
;;; first list in line-data and "f" are consed onto equation. equation is
;;; converted into a string and consed onto final-equation. Each list but
;;; the first in line-data is again considered. equation is assigned the
;;; value ("e" .a. "=e" .b.) where .a. is the bond number in list being
;;; considered and .b. of the bond in the first list. A string is formed of
;;; equation and it is consed onto final-equation. The equations are then
;;; added to the hash table.

  (let ((equation nil)
        (final-equation nil))
    (dolist (single-line line-data)
      (if (equal (third single-line) 'in)
          (setf pos (position single-line line-data))))
    (setf line-data (cons (nth pos line-data)
                          (remove (nth pos line-data) line-data)))
    (dolist (a-line (rest line-data))
      (if (equal (second (first line-data)) (second a-line))
          (setf sign "-")
          (setf sign "+"))
      (setf equation (cons sign (cons "f" (cons (first a-line) equation)))))
    (setf equation (cons "f" (cons (first (first line-data))
                                   (cons "=" equation))))
    (setf final-equation (cons (form-string equation) final-equation))
    (dolist (a-line (rest line-data) (values final-equation))
      (setf equation (list "e" (first a-line) "=e" (first (first line-data))))
      (setf final-equation (cons (form-string equation) final-equation)))
    (setf (node-equations (gethash node-number *nodes*)) final-equation)
    (values final-equation)))


(defun make-table-the-right-size (acsl-table)
```

```
;;; This function is called by the function make-acsl-table, the aim is
;;; divide acsl-table into a number of portions so that each portion fits
;;; onto one fortran line. The lines are terminated after the tenth
;;; occurrence of a comma. Each portion is placed in a separate list within
;;; a l  t called sized-table. Each list in sized-table, with exception of
;;; the last is terminated with three dots indicating that the acsl statement
;;; continues on the following line.

  (let (( zed-table nil)
        .tart-count 0)
       (start-count2 0)
       (halt-var 'go)
       (posi 0)
       (final-table nil))
    (loop
     (if (equal halt-var 'stop)
         (return))
     (setf start-count 0)
     (dotimes (a-count 10)
       (if (not (> start-count (length acsl-table)))
           (setf posi (position #\, acsl-table :start start-count)))
       (if posi
           (setf start-count (+ posi 1))
           (setf halt-var 'stop)))
     (if posi
         (setf end-count2 posi)
         (setf end-count2 (length acsl-table)))
     (setf sized-table (cons (subseq acsl-table start-count2 end-count2)
                             sized-table))
     (setf acsl-table (subseq acsl-table end-count2)))
    (setf sized-table (reverse sized-table))
    (dolist (table-part sized-table)
      (if (not (equal (list table-part) (last sized-table)))
          (setf final-table (cons (concatenate 'string table-part "...")
                                  final-table))
          (setf final-table (cons table-part final-table))))
    (setf final-table (reverse final-table))))


(defun make-acsl-table (e-f-out node-number bond-number)

;;; This function is called when the user wants to enter the data for a
;;; particular element in the form of a table. Three variables are passed to
;;; the function, the output variable name, the number of the element and the
;;; bond number. The user is prompted to enter the independent variables in
;;; the form of a list. Next the user is requ.te. to enter the number of data
;;; points for each independent variable. The user then enters all the data
;;; points for each independent variable. The value of the dependent
;;; variable (e-f-out nodenumber) for each combination of independent variable
;;; data is then entered. The function then forms a list, called table-list
;;; which contains the acsl statement "table", the table name, the number of
;;; independent variables, the number of data points for each independent
;;; variable, the data points of each independent variable and the values
;;; of the dependent variable corresponding to each combination of
;;; independent variable. Another list called equation1 is then formed, it
```

```
;;; has the form (dependent variable = table name ( independent variables
;;; separated by commas )). The function make-table-the-right-size is then
;;; called, this function divides table-list into portions that will fit
;;; onto one fortran line. The two lists are then appended together.

  (let ((dimlist nil)
        (tablelist nil)
        (one-ind-list nil)
        (ind-var nil)
        (ind-dim nil)
        (dims nil)
        (ind-list nil)
        (data-list nil)
        (ind-var-comma nil)
        (dimcount 0))
;;; dimlist is a list of the dimensions of the variables and tablelist is a
;;; list containing all the table data
    (format t "~%~%Enter the independant variables in the form of a list "
            eg (f3 s5 f7)~%")
    (setf ind-var (read))
    (dolist (an-ind-var ind-var)
      (format t "~%Enter the number of data points for ~a~%"
              (string-downcase an-ind-var))
      (setf ind-dim (cons (read) ind-dim)))
    (setf ind-dim (reverse ind-dim))
    (dolist (an-ind-dim ind-dim)
      (setf ind-list-sep nil)
      (do ((a-count 1 (+ 1 a-count))) ((= a-count (+ 1 an-ind-dim)))
        (format t "~%Enter value ~a for ~a~%" a-count (string-downcase
                                                       (nth dimcount ind-var)))
        (setf ind-list-sep (cons (read) ind-list-sep)))
      (setf ind-list (cons (reverse ind-list-sep) ind-list))
      (setf dimcount (+ 1 dimcount)))
    (setf ind-list (reverse ind-list))
    (case (length ind-var)
      (1 (dolist (f-var (first ind-list))
           (format t "~%Enter the value of ~a~a when ~a is ~a~%"
                   e-f-out bond-number
                   (string-downcase (first ind-var)) f-var)
           (setf data-list (append (list (read)) '(",") data-list))))
      (2 (dolist (s-var (second ind-list))
           (dolist (f-var (first ind-list))
             (format t "~%Enter the value of ~a~a when ~a is ~a and ~a is ~a~%"
                     e-f-out bond-number
                     (string-downcase (first ind-var)) f-var
                     (string-downcase (second ind-var)) s-var)
             (setf data-list (append (list (read)) '(",") data-list)))))
      (3 (dolist (t-var (third ind-list))
           (dolist (s-var (second ind-list))
             (dolist (f-var (first ind-list))
               (format t "~%Enter the value of ~a~a when ~a is ~a, ~a is ~a
                       and ~a is ~a~%" e-f-out bond-number
                       (string-downcase (first ind-var)) f-var
                       (string-downcase (second ind-var)) s-var
                       (string-downcase (third ind-var)) t-var)
               (setf data-list (append (list (read)) '(",") data-list)))))))
```

```
    (dolist (f-var (first ind-list))
      (setf one-ind-list (append (list f-var) '(",") one-ind-li.t)))
    (dolist (s-var (second ind-list))
      (setf one-ind-list (append (list s-var) '(",") one-ind-list)))
    (dolist (t-var (third ind-list))
      (setf one-ind-list (append (list t-var) '(",") one-ind-list)))
    (setf data-list (list (string-left-trim ","
                                            (form-string
                                             (append (reverse one-ind-list)
                                                     (reverse data-list))))))
    (setf table-name (concatenate 'string "table tab" (princ-to-string
                                                        *table-var*)))
    (dolist (an-ind-dim ind-dim)
      (setf dims (append (list an-ind-dim) '(",") dims)))
    (setf dims (reverse dims))
    (setf table-list (form-string (append (list table-name) '(",")
                                          (list (length ind-var))
                                          dims '("/") data-list '("/"))))
    (dolist (an-ind-var ind-var)
      (setf ind-var-comma (append (list (string-downcase an-ind-var)) '(",")
                                  ind-var-comma)))
    (setf equation1 (form-string (list s-f-out bond-number "=tab" *table-var*
                                       "(" (string-left-trim ","
                                                             (form-string
                                                              (reverse
                                                               ind-var-comma
                                                               ))) ")")))
    (setf table-list (make-table-the-right-size table-list))
    (setf final-equation (append table-list (list equation1)))))


(defun print-doc-str (node-number)

;;; This function prints the documenta.ion string

  (format t "~%~%The documentation for this element is:")
  (print (node-doc-str (gethash node-number *nodes*))))


(defun get-init-data ()

;;; This function is called by dump-acsl-equations. This program sets up the
;;; first part of the acsl file, prompts for the program name, the integration
;;; step length and the simulation time and sets up the necessary equations
;;; involving these.

  (let ((resulting-list nil)
        (a-list nil))
    (format t "~%~%Enter the acsl program name.~%")
    (setf a-list (cons "program " (cons (read-line) a-list)))
    (setf resulting-list (cons (form-string a-list) resulting-list))
    (setf resulting-list (cons "initial" resulting-list))
    (setf resulting-list (cons "constant points=1000" resulting-list))
    (setf a-list nil)
    (format t "~%~%Enter the integration step size, ccalc~%")
    (setf a-list (cons "constant ccalc=" (cons (read-line) a-list)))
```

```
    (setf resulting-list (cons (form-string a-list) resulting-list))
    (setf a-list nil)
    (format t "~%~%Enter the duration of the simulation, simtim.~%")
    (setf a-list (cons "constant simtim=" (cons (read-line) a-list)))
    (setf resulting-list (cons (form-string a-list) resulting-list))
    (setf resulting-list (cons "simend=simtim-ccalc" resulting-list))
    (setf resulting-list (cons "cint=simtim/points" resulting-list))
    (setf resulting-list (cons "nstp=cint/ccalc" resulting-list))
    (setf resulting-list (reverse resulting-list))))


(defun get-inter-data ()

;;; This function is called by dump-acsl-equations. This function forms a list
;;; containing the end of the initial section and the start of the dynamic
;;; and derivative sections.

  (let ((resulting-list nil))
    (setf resulting-list (cons "end $\"initial\"" resulting-list))
    (setf resulting-list (cons "dynamic" resulting-list))
    (setf resulting-list (cons "derivative" resulting-list))
    (setf resulting-list (reverse resulting-list))))


(defun get-end-data ()

;;; This function is called by dump-acsl-equations. This function forms a
;;; list of the statements to terminate the simulation and end the sections.

  (let ((resulting-list nil))
    (setf resulting-list (cons "termt(t.gt.simend)" resulting-list))
    (setf resulting-list (cons "end $\"derivative\"" resulting-list))
    (setf resulting-list (cons "end $\"dynamic\"" resulting-list))
    (setf resulting-list (cons "terminal" resulting-list))
    (setf resulting-list (cons "end $\"terminal\"" resulting-list))
    (setf resulting-list (cons "end $\"program\"" resulting-list))
    (setf resulting-list (reverse resulting-list))))


(defun form-string (any-list)

;;; This function is called by numerous other functions. It takes as its input
;;; any list. The function forms a string of each element in the list and
;;; concatenates them all together to form one string i.e. If the input is
;;; ("f" 5 "=(1.0/r" 5 ")*e" 5) the output will be "f5=(1.0/r5)*e5".

  (let ((resulting-string ""))
    (dolist (ele (reverse any-list) (values resulting-string))
      (setf resulting-string (concatenate 'string (princ-to-string ele)
                                          (princ-to string
                                           resulting-string))))))

;;; princ-to-string takes any object and converts it to string, note the
;;; function string only converts strings and symbols to strings not other
;;; data types eg numbers.
```

```
(defun dump-acsl-output (output-list)

;;; This function is called by dump-acsl-equations. It opens a file
;;; *filename*.acsl and dumps each element (a string) in the list
;;; output-list to the file.

  (setf filename (concatenate 'string "/usr/bondgraph/acslfile/"
                              (string-downcase *filename*) ".acsl"))
  (with-open-file (output-stream filename :direction :output)
                  (dolist (a-string output-list)
                    (format output-stream "~%~a" a-string)))
  (format t "~|Acsl input file dumped to the file ~a~%" filename)
  (values))


(defun get-lhs (a-string)

;;; This function takes a string input for example "f2 = sqrt (1 - 5*e2)"
;;; and returns all the character to the left of the equal sign. It also
;;; removes all the spaces. Consider the string above the function would
;;; return "f2".

  (remove #\space (subseq a-string 0 (position #\= a-string :test #'equal))
          :test #'equal))


(defun get-rhs (a-string)

;;; This function takes a string input for example "f2 = sqrt (1 - 5*e2)"
;;; and returns all the charact  to the right of the equal sign. Consider the
;;; string above the function would return " sqrt (1 - 5*e2)".

  (subseq a-string (+ 1 (position #\= a-string :test #'equal))))


(defun r-element-alger-loop (list-of-equations)

;;; If the causality had to be completed by assigning an arbitrary causality
;;; to a resistor then this function is called, this occurs when the global
;;; variable *r-algerbraic-loop* has a value. This function gets the data to
;;; the left of the equal sign (calls function get-lhs) of each string in
;;; list-of-equations. If it is equal to a string of "e" .. where .. is the
;;; bond to which the causality was assigned arbitrarily then this string is
;;; assigned to rem-str and a new string new-str is formed. new-str consists
;;; of an acsl implicit function replacing rem-str. rem-str is removed from
;;; list-of-equations and new-str inserted into it. *error-flag-num* is the
;;; index of the error flag variable in the acsl implicit statement, so that
;;; each flag variable is unique. *error-flag-num* is incremented by one
;;; every time it is used  Constant default values for each variable found in
;;; the acsl implicit statement are given, but the user has the option of
;;; specifying the value of each variable or the expression used to
;;; calculate it.

  (dolist (an-r-alger-loop *r-algerbraic-loop*)
    (dolist (equ-str list-of-equations)
```

```
      (if (equal (get-lhs equ-str)
                 (concatenate 'string "e" (princ-to-string
                                            (third an-r-alger-loop))))
;;; If an R element is used to complete the causality it doesn't
;;; matter if the effort or the flow is made implicit, in this program
;;; the effort is always made implicit for convenience only
          (progn (format t "~|The equation ~a has to be calculated "
                         implicitly~%" equ-str)
                 (format t "~%That is it will be written in the form~%")
                 (format t "~%~a = impl(yz, e, m, ef~a, ~a, ydl)~%"
                         (get-lhs equ-str) *error-flag-num* (get-rhs equ-str))
                 (format t "~%where yz is the initial guess")
                 (format t "~%      e is the error bound")
                 (format t "~%      m is the maximum number of iterations "
                         to try")
                 (format t "~%      ef~a is the error flag variable"
                        *error-flag-num*)
                 (format t "~%      ydl is the value used to increment the "
                         initial value to estimate the              "
                         derivative~%")
                 (format t "~%~%Please enter yz (a constant or expression) "
                         default 0.0  ")
                 (setf yz (read-line))
                 (if (equal yz "")
                     (setf yz "0.0  "))
                 (format t "~%Please enter e (a constant or expression) "
                         default 0.0001  ")
                 (setf e (read-line))
                 (if (equal e "")
                     (setf e "0.0001  "))
                 (format t "~%Please enter m (an integer constant) "
                         default 50  ")
                 (setf m (read-line))
                 (if (equal m "")
                     (setf m "50  "))
                 (format t "~%Please enter ydl (a constant or expression) "
                         default 0.001  ")
                 (setf ydl (read-line))
                 (if (equal ydl "")
                     (setf ydl "0.001"))
                 (setf rem-str equ-str)
                 (setf new-str (list (get-lhs equ-str)
                                     "=impl(" yz "," e "," m ",ef"
                                     *error-flag-num* "," (get-rhs equ-str)
                                     "," ydl ")")))))
    (setf new-str (form-string new-str))
    (setf list-of-equations (remove rem-str list-of-equations :test #'equal))
    (setf list-of-equations (cons new-str list-of-equations))
    (setf *error-flag-num* (+ 1 *error-flag-num*)))
  (values list-of-equations))


(defun bond-alger-loop (list-of-equations)

;;; If the causality had to be completed by assigning an arbitrary causality
;;; to a bond attached only to junction structure elements, then this function
```

```
;;; is called, this occurs when the global variable *bond-algerbraic-loop*
;;; has a value assigned to it. This function gets the data to the left of the
;;; equals sign (calls function get-lhs) on each string in list-of-equations.
;;; If the string is equal to a string of "e" .. where .. is the bond to which
;;; the causality was assigned arbitrarily then this string is assigned to
;;; rem-str1 and a new string new-str1 is formed. new-str1 consists of an acsl
;;; implicit function, replacing rem-str1. If the left hand side of the string
;;; is equal to "f" .. where .. is the bond to which the causality was
;;; assigned arbitrarily then this string is assigned to rem-str2 and a new
;;; string new-str2 is formed. new-str2 consists of an acsl implicit function.
;;; rem-str1 and rem-str2 are removed from list-of-equations and new-str1 and
;;; new-str2 are inserted into it. Because there are two implicit functions
;;; *error-flag-num* is incremented twice. Constant default values for each
;;; variable found in the acsl implicit statement are given, but the user has
;;; the option of specifying the value of each variable or an expression used
;;; to calculate it.

(dolist (a-bond-alger-loop *bond-algerbraic-loop*)
  (dolist (equ-str list-of-equations)
    (if (equal (get-lhs equ-str) (concatenate 'string "e"
                                              (princ-to-string
                                               (third a-bond-alger-loop))))
        (progn (format t "~|The equation ~a has to be calculated "
                       implicitly~%" equ-str)
               (format t "~%That is it will be written in the form~%")
               (format t "~%~a = impl(yz, e, m, ef~a, ~a, ydl)~%"
                       (get-lhs equ-str) *error-flag-num* (get-rhs equ-str))
               (format t "~%where yz is the initial guess")
               (format t "~%      e is the error bound")
               (format t "~%      m is the maximum number of iterations "
                       to try")
               (format t "~%      ef~a is the error flag variable"
                       *error-flag-num*)
               (format t "~%      ydl is the value used to increment the "
                       initial value to estimate the "
                       derivative~%")
               (format t "~%~%Please enter yz (a constant or expression) "
                       default 0.0  ")
               (setf yz (read-line))
               (if (equal yz "")
                   (setf yz "0.0"))
               (format t "~%Please enter e (a constant or expression) "
                       default 0.0001  ")
               (setf e (read-line))
               (if (equal e "")
                   (setf e "0.0001"))
               (format t "~%Please enter m (an integer constant) "
                       default 50  ")
               (setf m (read-line))
               (if (equal m "")
                   (setf m "50"))
               (format t "~%Please enter ydl (a constant or expression) "
                       default 0.001  ")
               (setf ydl (read-line))
               (if (equal ydl "")
                   (setf ydl "0.001"))
```

```
                (setf rem-str1 equ-str)
                (setf new-str1 (list (get-lhs equ-str)
                                     "=impl(" yz "," e "," m ",ef"
                                     *error-flag-num* "," (get-rhs equ-str)
                                     "," ydl ")"))))
    (if (equal (get-lhs equ-str) (concatenate 'string "f"
                                              (princ-to-string
                                               (third a-bond-alger-loop))))
        (progn (format t "~|The equation ~a has to be calculated "
                       implicitly~%" equ-str)
               (format t "~%That is it will be written in the form~%")
               (format t "~%~a = impl(yz, e, m, ef~a, ~a, ydl)~%"
                       (get-lhs equ-str) (+ 1 *error-flag-num*)
                       (get-rhs equ-str))
               (format t "~%where yz is the initial guess")
               (format t "~%      e is the error bound")
               (format t "~%      m is the maximum number of iterations "
                       to try")
               (format t "~%      ef~a is the error flag variable"
                       (+ 1 *error-flag-num*))
               (format t "~%      ydl is the value used to increment the "
                       initial value to estimate the
                       derivative~%")
               (format t "~%~%Please enter yz (a constant or expression) "
                       default 0.0  ")
               (setf yz (read-line))
               (if (equal yz "")
                   (setf yz "0.0"))
               (format t "~%Please enter e (a constant or expression) "
                       default 0.0001  ")
               (setf e (read-line))
               (if (equal e "")
                   (setf e "0.0001"))
               (format t "~%Please enter m (an integer constant) "
                       default 50  ")
               (setf m (read-line))
               (if (equal m "")
                   (setf m "50"))
               (format t "~%Please enter ydl (a constant or expression) "
                       default 0.001  ")
               (setf ydl (read-line))
               (if (equal ydl "")
                   (setf ydl "0.001"))
               (setf rem-str2 equ-str)
               (setf new-str2 (list (get-lhs equ-str)
                                    "=impl(" yz "," e "," m ",ef"
                                    (+ 1 *error-flag-num*) ","
                                    (get-rhs equ-str) "," ydl ")")))))
  (setf new-str1 (form-string new-str1))
  (setf new-str2 (form-string new-str2))
  (setf list-of-equations (remove rem-str1 list-of-equations :test #'equal))
  (setf list-of-equations (remove rem-str2 list-of-equations :test #'equal))
  (setf list-of-equations (cons new-str1 list-of-equations))
  (setf list-of-equations (cons new-str2 list-of-equations))
  (setf *error-flag-num* (+ 2 *error-flag-num*)))
(values list-of-equations))
```

```
(defun rhs-search (rhs-str)

;;; This function given the right hand side of an equation gets a list of
;;; all the bond graph variables in that equation, for example if
;;; rhs-str is "e7+e9-f8+q23*(p34+e7)"
;;; it would return (e7 e9 f8 q23 p34).
;;; The function finds the position of each bond graph variable name (e, f,
;;; p, q) where it is followed by a number in rhs-str. The function get-var
;;; is called to get each variable name.

  (let ((pos-list nil)
        (var-list nil)
        (final-list nil))
    (setf rhs-str (substitute #\[ #\( rhs-str))
          (setf rhs-str (substitute #\] #\) rhs-str))
    (setf rhs-str (substitute #\* #\, rhs-str))
;;; This substitution is done so that a parenthesis is never read on its own
;;; to determine if it is a number.
    (do ((num 0 (+ num 1))) ((= num (- (length rhs-str) 1)))
      (if (and (or (equal (princ-to-string (elt rhs-str num)) "f")
                   (equal (princ-to-string (elt rhs-str num)) "e")
                   (equal (princ-to-string (elt rhs-str num)) "q")
                   (equal (princ-to-string (elt rhs-str num)) "p"))
               (numberp (read-from-string (princ-to-string
                                           (elt rhs-str (+ 1 num))))))
          (setf pos-list (cons num pos-list))))
    (dolist (pos-n pos-list)
      (setf var-list (cons (get-var rhs-str pos-n) var-list)))
    (setf var-list (reverse var-list))
    (dolist (each-ele var-list)
      (if (not (member each-ele final-list :test #'equal))
          (setf final-list (cons each-ele final-list))))
;;; Makes sure that there is only one of each variable in the list
    (values final-list)))


(defun get-var (r-string pos)

;;; This function is called by rhs-search and is passed as parameters, the
;;; string and the starting position of a bond graph variable name. The
;;; function first removes the bond graph variable letter and increments the
;;; position by one. The function then checks to see whether the end of the
;;; string has been reached, if so, all the variables that were removed are
;;; placed into one string and this is the resulting bond graph variable. If
;;; the end of the string has not been reached the function checks to see
;;; whether the next element is a number, if so it is removed, the position
;;; counter is incremented by one and the process repeats itself. If the
;;; element is not a number, the loop is exited.

  (setf a-var (list (princ-to-string (elt r-string pos))))
  (setf pos (+ 1 pos))
  (loop
   (if (>= pos (length r-string))
       (return))
```

```
  (if (numberp (read-from-string (princ-to-string (elt r-string pos))))
      (setf a-var (cons (princ-to-string (elt r-string pos)) a-var))
      (return))
  (setf pos (+ 1 pos)))
 (setf a-var (reverse (form-string a-var))))


(defun derivative-causality (list-of-equations)

;;; If one or more storage elements have derivative causality assigned to them
;;; the this function is called, this occurs when the global variable
;;; *deriv-causality* has a value assigned to it. This function takes the
;;; displacement or momentum corresponding to the storage element with
;;; derivative causality and searches through the list of all the equations
;;; to get the equation where the momentum or displacement is the output.
;;; The function then takes the right hand side of the equation and if it
;;; contains the word "integ", informs the user that the derivative of the
;;; equation has to be calculated implicitly. Constant default values for each
;;; variable found in the acsl implicit statement are given, but the user has
;;; the option of specifying the value of each variable or an expression used
;;; to calculate it. For example, if the equation was "q3=integ(f3,q3i)", then
;;; the new equation to be added to the list of equations would be
;;; "q3d=impl( yz, e, m, efx, f3, ydl)", where yz, e, m and ydl are the
;;; variables the user has the option of assigning a value to, x is the
;;; error flag index. Where q3d is the derivative of q3. If the word "integ"
;;; is not in the equation for which the momentum or displacement is the
;;; output, then the user is prompted to enter the time derivative of the
;;; right hand side of the equation. For example, if the equation was "q3=5.0*e3+e5",
;;; the user would be prompted to enter the time derivative of 5.0*e3+e5.
;;; The users response would be 5.0*e3d+e5d where e3d and e5d is the
;;; terminology used to represent the derivative of e3 and e5 respectively.
;;; The new equation included in the list of equations would be
;;; "q3d=5.0*e3d+e5d". The user differentiated right hand side is sent to
;;; the function rhs-search, which removes the bond graph variables from the
;;; string. In this example the function rhs-search would return (e3 e5).
;;; The process is now repeated with each of the variables returned by
;;; rhs-search. This process is complete when there are no more bond graph
;;; variables left in the list lhs-batch. This list contains all the
;;; bond graph elements returned by rhs-search.

 (let ((s-var "go"))
   (dolist (a-deriv-caus *deriv-causality*)
     (if (equal (first a-deriv-caus) 'c)
         (setf lhs-batch (list (form-string (list "q" (third a-deriv-caus)))))
         (setf lhs-batch (list (form-string (list "p" (third a-deriv-caus))))))
     (loop
      (if (not lhs-batch)
          (return))
      (setf lhs-strings lhs-batch)
      (setf lhs-batch nil)
      (dolist (lhs-str lhs-strings)
        (dolist (an-equa list-of-equations)
          (if (equal (form-string (list lhs-str "d")) (get-lhs an-equa))
              (setf s-var "do-nothing")))
        (if (not (equal s-var "do-nothing"))
            (progn (dolist (an-equa list-of-equations)
```

```
  (if (equal lhs-str (get-lhs an-equa))
      (setf rhs-str (get-rhs an-equa))))
(if (search "integ" rhs-str)
    (progn (format t "~|The equation ~ad = ~a has to be ~
                   calculated implicitly~%" lhs-str
                   (first (rhs-search rhs-str)))
           (format t "~%That is it will be written in ~
                   the form~%")
           (format t "~%~ad = impl(yz, e, m, ef~a, ~
                   ~a, ydl)~%" lhs-str
                   *error-flag-num*
                   (first (rhs-search rhs-str)))
           (format t "~%where yz is the initial guess")
           (format t "~%      e is the error bound")
           (format t "~%      m is the maximum number of ~
                   iterations to try")
           (format t "~%      ef~a is the error flag ~
                   variable" *error-flag-num*)
           (format t "~%      ydl is the value used to ~
                   increment the initial value to ~
                   estimate the                   ~
                   derivative~%")
           (format t "~%~%Please enter yz (a constant or ~
                   expression)  default 0.0  ")
           (setf yz (read-line))
           (if (equal yz "")
               (setf yz "0.0"))
           (format t "~%Please enter e (a constant or ~
                   expression) default 0.0001  ")
           (setf e (read-line))
           (if (equal e "")
               (setf e "0.0001"))
           (format t "~%Please enter m (an integer ~
                   constant) default 50  ")
           (setf m (read-line))
           (if (equal m "")
               (setf m "50"))
           (format t "~%Please enter ydl (a constant or ~
                   expression) default 0.001  ")
           (setf ydl (read-line))
           (if (equal ydl "")
               (setf ydl "0.001"))
           (setf new-str (list lhs-str
                               "d=impl(" yz "," e "," m
                               ",ef" *error-flag-num*
                               "," (first (rhs-search
                                           rhs-str))
                               "," ydl ")"))
           (setf *error-flag-num* (+ 1 *error-flag-num*))
           (setf new-equation (form-string new-str))
           (setf list-of-equations (cons
                                    new-equation
                                    list-of-equations)))
    (progn (format t "~|Enter the time derivative of the ~
                   function ~a~%" rhs-str)
           (setf new-rhs-str (read-line))
```

```
                                (setf new-equation (form-string
                                                    (list lhs-str "d="
                                                          new-rhs-str)))
                                (setf list-of-equations
                                      (cons new-equation list-of-equations))
                                (setf lhs-batch
                                      (append (rhs-search new-rhs-str)
                                              lhs-batch)))))))))
      (values list-of-equations)))


(defun write-hash-table-to-file-dump (filename)

;;; This function writes the line and the node hash tables to a file
;;; *filename*.data. In addition to the hash tables, *line-hash-keys* and
;;; *node-hash-keys*, which reference the hash tables are also written to the
;;; file, as are *r-algorbraic-loop* , *bond-algerbraic-loop* and
;;; *deriv-causality*. A statement, informing the modules that the causality
;;; has been assigned is also included in the file.

  (setq filename (concatenate 'string "/usr/bondgraph/data/"
                              (string-downcase filename) ".data"))
  (format t (concatenate 'string "~%Hash table written to file " filename "~%"))
  (with-open-file (line-stream filename
                               :direction :output)
                  (print 'causality-assigned line-stream)
                  (print *line-hash-keys* line-stream)
                  (dolist (line-number *line-hash-keys*)
                    (print (gethash line-number *lines*) line-stream))
                  (print *node-hash-keys* line-stream)
                  (dolist (node *node-hash-keys*)
                    (print (gethash node *nodes*) line-stream))
                  (print *r-algerbraic-loop* line-stream)
                  (print *bond-algerbraic-loop* line-stream)
                  (print *deriv-causality* line-stream)))


;bot
```

D.8 The ACSL execution module (run-acsl.lisp)

```
(defvar *filename* nil)

(defun run-acsl ()
  (if *filename*
      (if (not (y-or-n-p "~|Do you want to simulate the system stored in the "
                         "file~%~a.acsl" *filename*))
          (progn (format t "~%~%Enter the file name of the file to simulate, "
                         "don't include an extension~%it will be taken to be "
                         "taken to be 'acsl'~%")
                 (setf *filename* (read))))
      (progn (format t "~|Enter the file name of the file to simulate, "
                     "don't include an extension~%it will be taken to be "
                     "taken to be 'acsl'~%")
             (setf *filename* (read))))
  (setf filename1 (concatenate 'string "/usr/bondgraph/acslfile/"
                               (string-downcase *filename*) ".acsl"))
  (setf filename2 (concatenate 'string "/usr/bondgraph/acslfile/"
                               (string-downcase *filename*) ".csl"))
  (run-unix-program "addcr" :input filename1 :output filename2
                    :if-output-exists :supersede :wait nil)
  (write-run-acsl.bat)
  (run-unix-program "vpix")
  (remove-contents-of-run-acsl.bat)
  (values))

(defun write-run-acsl.bat ()
  (with-open-file (line-stream "/usr/bondgraph/run-acsl.bat"
                               :direction :output)
                  (format line-stream "key-fake \"acslexec\"+chr(13)+"
                          \"acslclg "a\"+chr(13)" (string-downcase *filename*))))

(defun remove-contents-of-run-acsl.bat ()
  (with-open-file (line-stream "/usr/bondgraph/run-acsl.bat"
                               :direction :output)))
```