1.2.3.1 Modeling from Physical Principles

François E. Cellier
Department of Electrical and Computer Engineering, The University of Arizona, Tucson, Arizona 85721, U.S.A., e-mail: Cellier@ece.arizona.edu

Hilding Elmqvist
Dynasim AB, Research Park Ideon, S-223 70 Lund, Sweden, e-mail: Elmqvist@Dynasim.se

Martin Otter
Institute for Robotics and System Dynamics, German Aerospace Research Establishment Oberpfaffenhofen (DLR), Postfach 1116, D-82230 Wessling, Germany, e-mail: Martin.Otter@DLR.de

1 Introduction

Modeling and simulation are central to the design of control systems, since, for complex large–scale nonlinear industrial plants, there usually don't exist, or at least are not known, any suitable analytical design approaches. Simulation may often be the only resort short of building the physical plant itself and experimenting with the real system.

Another article in this handbook by Otter and Cellier entitled "Software for Modeling and Simulating Control Systems" deals with a wide palette of issues relating to the simulation of control systems. This article, on the other hand, shall concentrate on issues relating to modeling the physical plant to be controlled.

Modeling physical systems seems to be a straightforward task. Since physical systems and experiments are often reproducible in a reliable fashion, since measurements from physical systems are frequently available in abundance and of high quality, since the meta–laws of physics are mostly well understood, it seems to be a particularly easy task to come up with accurate mathematical descriptions of most physical plants.

Yet, there are some typical pitfalls and frequent misconceptions about the modeling of physical systems, especially among control engineers. These shall be illustrated, and a sound methodological basis for modeling from physical principles shall then be created.

2 Common Misconceptions

2.1 1st Misconception: State–Space Models Form the Basis of Physics

Control engineers are used to dealing with state–space models of the form:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \\ \mathbf{y} &= \mathbf{g}(\mathbf{x}, \mathbf{u}, t) \end{aligned} \tag{1}$$

Consequently, they look at the physical world as if it consisted of these types of equations, and, when modeling, try to start out with system descriptions that come as close as possible to the familiar state–space form.

This concept shall be explained by means of a simple lunar lander module as shown in Fig. 1. Only the vertical movement of the space craft shall be considered. The modeler starts

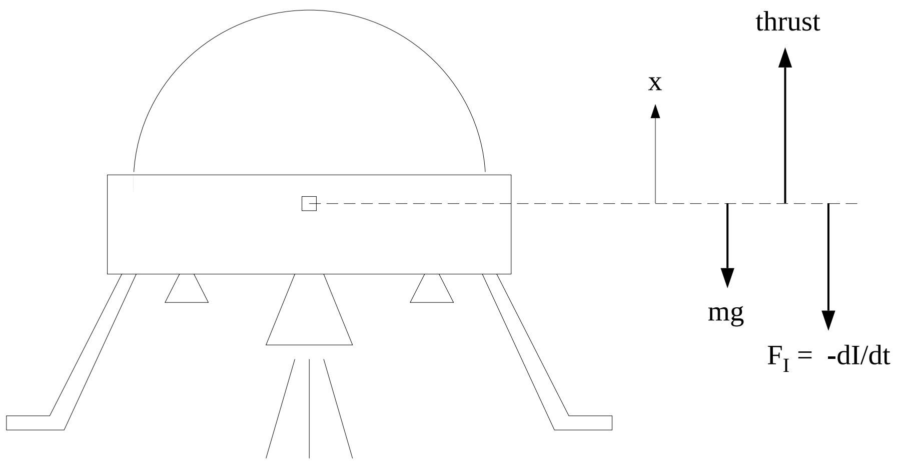

Figure 1: Lunar lander module.

out with Newton's laws (translational motion) or the d'Alembert principle, since Newton's laws come very close, in description, to the desired state–space formulation. Since the mass of the space craft is time–varying (due to fuel consumption), the resulting equation takes the following form:

$$\frac{d(m \cdot v)}{dt} = \frac{dm}{dt} \cdot v + m \cdot \frac{dv}{dt} = thrust - m \cdot g \tag{2}$$

Making the reasonable assumption that the change in mass is proportional to the magnitude of the thrust:

$$\frac{dm}{dt} = -k \cdot |thrust| \tag{3}$$

and assuming that the thrust is always positive, Equation (2) can be rewritten as:

$$\frac{dv}{dt} = \frac{1}{m} \cdot ((1 + k \cdot v) \cdot thrust - m \cdot g) \tag{4}$$

which is already very close to the desired state–space model.

There is only one major drawback of this model — it is unfortunately incorrect. This can be seen easily. For simplicity, it shall be assumed that the space craft is far away from any source of gravity. Thus, $g \approx 0$, and Equation (4) can be simplified to:

$$\frac{dv}{dt} = \frac{1}{m} \cdot (1 + k \cdot v) \cdot thrust \tag{5}$$

Making the additional assumption that the initial velocity of the spacecraft is:

$$v_0 = -\frac{1}{k} \tag{6}$$

it follows that the acceleration is exactly equal to zero irrespective of the thrust, and the doomed crew will not be able to either accelerate or decelerate their space craft ever again. Luckily for them, physics doesn't work that way.

Where is the bug in the model? Had the modeler, instead of blindly and unthinkingly applying Newton's laws to the space craft, formulated either the principle of energy conservation or alternatively that of momentum conservation (both concepts work equally well in this simple example) to a piece of space surrounding the rocket, he or she would have noticed that the plume of exhaust of the space craft also carries mass and momentum and energy, and the disaster could have been prevented. More precisely:

$$\mathcal{I}(t + \Delta t) = \mathcal{I}(t) + \Delta\mathcal{I}(t \rightarrow t + \Delta t) \tag{7}$$

i.e., the total momentum $\mathcal{I}$ of a system at time $t + \Delta t$ equals the total momentum at time t plus the (positive or negative) momentum added to (subtracted from) the system between time t and time $t + \Delta t$. Applied to the space craft:

$$(m - \Delta m) \cdot (v + \Delta v) + \Delta m \cdot v = m \cdot v + thrust \cdot \Delta t \tag{8}$$

The first term on the left–hand side of Equation (8) denotes the momentum of the space craft at time $t + \Delta t$. The second term denotes the momentum of the cloud of exhaust at the same time. The first term on the right–hand side denotes the momentum of the space craft at time t, and the second term denotes the added momentum due to the drive of the space craft. Notice that the exhaust must be included. Either the cloud of exhaust is considered a part of the system by adding it to the left–hand side of Equation (8), as done above, or the cloud must be considered as leaving the system between time t and time $t + \Delta t$, and then the same term must be subtracted from the right–hand side of Equation (8).

Neglecting terms in Equation (8) that are of second order small leads to:

$$m \cdot \Delta v = thrust \cdot \Delta t \tag{9}$$

or by dividing through Δt and letting Δt go to zero:

$$m \cdot \frac{dv}{dt} = thrust \tag{10}$$

Thus, although the mass of the space craft is undeniably changing with time, the more familiar form of Newton's laws must be used in this case.

This is of course a very simple example, and few control engineers would fall into this trap. However, such errors indeed do happen, and often, it is because of the control engineers' infatuation with state–space models.

2.2 2nd Misconception: Signals Capture Physics

The second misconception has to do with treating individual signals as portraying physical principles. This idea shall be illustrated by means of a simple thermal model describing, as a function of time, the temperature distribution along a perfectly insulated rod.

Most physics textbooks (e.g. [Holman, 1992]) can be consulted to provide the following model:

$$\frac{\partial T}{\partial t} = \frac{\lambda}{c \cdot \rho} \cdot \frac{\partial^2 T}{\partial x^2} \tag{11}$$

where t denotes time, x denotes the location along the rod, T is the temperature measured in Kelvin, λ is the specific thermal conductance of the material, c represents the specific thermal capacitance of the material, and ρ stands for its density.

Control engineers don't usually like partial differential equations, so they conceptually cut the rod into slices of length Δx, approximate the spacial derivative by third–order accurate finite differences, and, introducing the abbreviation:

$$\sigma = \frac{\lambda}{c \cdot \rho} \tag{12}$$

end up with the state–space model:

$$\frac{dT_i}{dt} = \sigma \cdot \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} \tag{13}$$

Is there anything wrong with this model? Indeed, this model describes rather accurately what happens to the temperature at each point along the rod and at each moment in time, if, starting out from an initial spatial temperature distribution, the system is allowed to settle into steady state without external influences, which is a fine physical experiment, but doesn't represent what most engineers might like to do with most rods most of the time.

The problem is that the state–space model of Equation (13) is autonomous. There is no input at all to this model. If an engineer, for example, decides to let a current flow through the rod and observe how the rod heats up as a consequence, there is no way that the above model will be able to help him or her describe the desired effect.

The problem with the above model is that it is expressed in terms of *temperature*. Temperature is not what drives the physical phenomenon. It is only an observational quantity. What really goes on in the rod is a phenomenon of *heat flow*. Thus, it would have been much more sensible to describe the heat flow through the rod, rather than focusing on the temperature distribution.

In essence, the two misconceptions mentioned so far boil down to the same thing. By focusing on heat flow, the modeler is thinking in terms of energy. By focusing on temperature, he or she is modeling in terms of a phenomenological signal.

Every first–year undergraduate student of electrical engineering knows that the relationship between voltage and current in an electrical capacitor is described by the equation:

$$i_C = C \cdot \frac{du_C}{dt} \tag{14}$$

and the students are told early on in their careers that the behavior of electrical circuits is governed by two fundamental variables: *voltage* and *current*. Equation (14) doesn't tell why current is flowing through the capacitor. Yet students will rarely ask that question. To them, the correct circuit behavior follows directly from a systematic application of Kirchhoff's laws. But what precisely do these laws express in terms of physical rather than mathematical, knowledge?

The product of voltage and current comes into the picture only as an afterthought, if at all. It rarely sinks in with students that what the circuit is really doing is, starting out from an initial state, try to balance energy at all times.

Most state–space models are expressed in terms of phenomenological variables that are related by equations representing remote consequences of the basic physical laws, not the laws themselves. It is then the modeler's obligation to check whether all the conditions, all the silent assumptions, that went into these equations are indeed satisfied in the situation at hand, and, in order to make this assertion, the modeler often needs to revert to the physical laws that form the basis of these equations. Yet, engineers often don't do this. In fact, they may have forgotten the basic laws altogether and only remember the derived laws that they use frequently in their everyday's work. And this is where the problems start.

Might it not make more sense to provide a modeling tool that can start out from the basic physical principles directly and derive the final state–space model on its own?

2.3 3rd Misconception: Causality Forms the Basis of Physics

The third misconception has to do with the notion of causality in physical systems. Many engineers believe that physics is essentially causal in nature. Someone takes a conscious decision to affect the world in a particular way, thereby *causing* the world to react to his or her actions.

Sir Isaak Newton followed the same line of reasoning when he formulated his famous law about *actio* being equal to *reactio*. If someone or something exerts a force (the *actio*) on a body that cannot move, then the system has to react by creating a counterforce (the *reactio*) of equal magnitude and opposite direction, such that the two forces annihilate each other.

Yet, the distinction between *actio* and *reactio* is a deeply human and moral concept, not a physical one. There is no physical experiment in the world that can distinguish between *actio* and *reactio*. It is clear that, if a drunk driver smashes his or her car into a tree, this is the fault of the driver and not of the tree. Yet, this is a purely human concept. Physics can't tell the difference.

The relationship between voltage across and current through an electrical resistor can be described by Ohm's law:

$$u_R = R \cdot i_R \tag{15}$$

yet, whether it is the current flowing through the resistor that causes a voltage drop, or whether it is the difference between the electrical potentials on the two wires that causes current to flow is, from a physical perspective, a meaningless question.

Physics is essentially acausal. Even where time is involved, physics locally only obeys conservation laws that, by nature, again are acausal. In spite of the recent efforts of science philosophers such as Roger Penrose, physics has still not discovered the true nature of the conscious mind. Maybe, consciousness is in fact a mere illusion, the true nature of which is, however, impossible to determine due to Gödel's theorem?

Luckily, being engineers, we don't have to concern ourselves with these philosophical questions. Yet, the acausal nature of physics is indeed of much concern. State–space models are written in assignment statement form. There is always exactly one variable to the left of the equal sign, and the model implies that the expression to the right of the equal sign is evaluated, and the result of this evaluation is assigned as a new value to the variable to the left.

Consequently, the modeler needs two different models to describe an electrical resistor: a *voltage–drop–causer* model and a *current–flow–causer* model. As mentioned earlier, from a physical perspective, this makes no sense whatsoever.

Would it not be more meaningful to have a modeling tool available that allows the formulation of models in a declarative (acausal) form, and let the software worry about establishing the appropriate *computational causality* of each equation before generating simulation run–time code?

3 Energy Modeling and Bond Graphs

For a long time, mechanical engineers have used the Lagrangian or Hamiltonian equations to model mechanical systems, i.e., the modeling is based on the total *energy* contained in a mechanical system. However, the approach has two major drawbacks: (i) If non–conservative forces are present, the Lagrangian and Hamiltonian equations are no longer solely based on the energy of the system but additionally on the virtual work of the non–conservative forces. (ii) The equations are based on the total energy of the complete system, i.e., it is not possible to use the energy of subsystems, and then connect these subsystems together in a *modular* fashion. Similar problems occur in other engineering fields if modeling is based on energy.

To overcome these two deficiencies, Henry Paynter invented in the early 60s the *bond graph* [Paynter, 1961]. By computing the time derivative of the energy:

$$P(t) = \frac{dE(t)}{dt} \tag{16}$$

and using the power P instead of the energy E for modeling, both of the aforementioned difficulties vanish. In this new approach, *power continuity* equations are formulated instead of *energy conservation* laws. It turns out that, in any physical system, the power balance is a local property, i.e., the modeler can express power balance equations for each subsystem separately, and then connect all the subsystems, as long as he or she makes sure that the power is also balanced at all the interfaces between submodels [Karnopp, *et al.*, 1990].

This very same property also makes it possible to use power balance equations to describe dissipative systems. Resistors simply become two–port elements where free energy “opts out” and decides to henceforth be called heat.

As a bonus, power in any physical system can be written as the product of two adjugate variables, called the *effort* and the *flow* in bond graph terminology. In a bond graph, energy flow from one point of a system to another is denoted by a harpoon (a semi–arrow), as shown in Fig. 2.

Power is the product of effort and flow:

$$P = e \cdot f \tag{17}$$

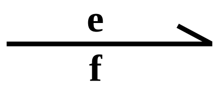

Figure 2: The bond.

In an electrical system, it is customary to select the voltage (or electrical potential), u, as the effort variable, and the current, i, as the flow variable. In a translational mechanical system, the force, f, will be treated as effort, and the velocity, v, as flow. In a rotational system, the torque, τ, assumes the role of the effort, and the angular velocity, ω, that of the flow. However, the assignment is arbitrary. Effort and flow are dual variables, and the assignment could just as well be done the other way around.

When power splits, or is combined, in a so–called *junction*, the power continuity equation dictates that the sum of incoming power streams must equal the sum of outgoing power streams. This requirement can be satisfied in many different ways, but the two easiest ones are certainly to keep one of the two adjugate variables constant around the junction, and formulate the balance equation for the other. In bond graph terminology, these two junction types are called the *0–junction* and the *1–junction*. Their constitutive equations are shown in Fig. 3.

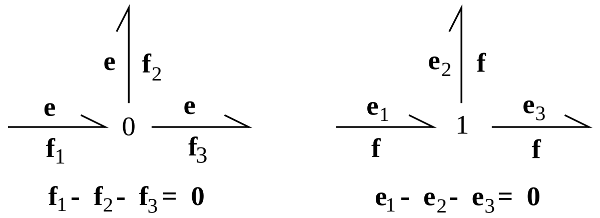

Figure 3: The two basic junction types.

Luckily for us engineers, physics seems to have a preference for simple solutions, and therefore, these two basic junction types are very commonly found in all kinds of physical systems. For example, one recognizes at once Kirchhoff's current law formulated for any node or cutset (the 0–junction), and Kirchhoff's voltage law formulated for any mesh or loop (the 1–junction).

The concept and benefits of bond–graph modeling shall be demonstrated by means of a model describing an armature–controlled DC–motor. A schematic diagram of the motor is shown in Fig. 4: and the corresponding bond graph is shown in Fig. 5: The 0–junction to the left of the bond graph depicts the electrical power port. Through it, the power:

$$P_{\text{elect}} = u_a \cdot i_a \tag{18}$$

enters the subsystem containing the DC–motor. The 0–junction is the interface to the next subsystem to the left. Only a portion of the incoming power is available for conversion to mechanical energy. Some of the power gets stored in the armature inductance L_a, and some gets irreversibly converted to heat in the armature resistance R_a. Since the armature circuit is a mesh, the distribution of energy is described by a 1–junction. The remaining power is available for conversion to mechanical energy, thus:

$$P_{\text{mot}} = u_i \cdot i_a = \tau_m \cdot \omega_m \tag{19}$$

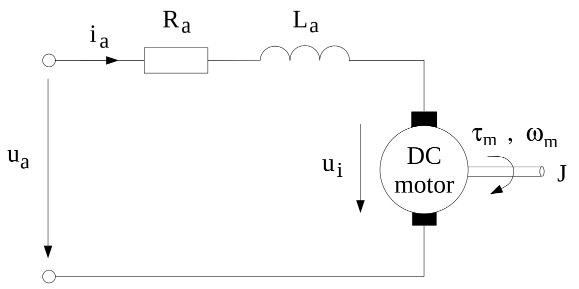

Figure 4: Model of a DC motor.

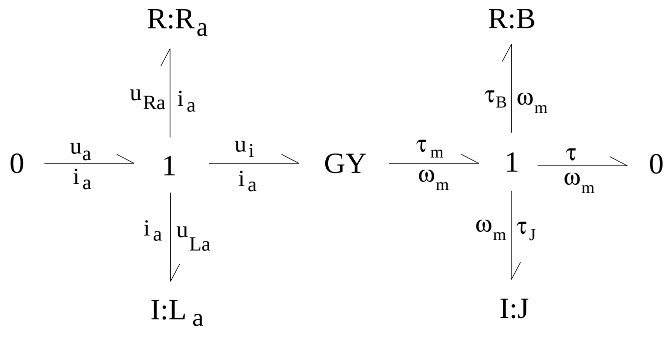

Figure 5: Bond graph of a DC-motor.

Again, power continuity across the converter can be satisfied in many different ways, yet, there are only two simple solutions, as shown in Fig. 6: namely the *transformer (TF)* and

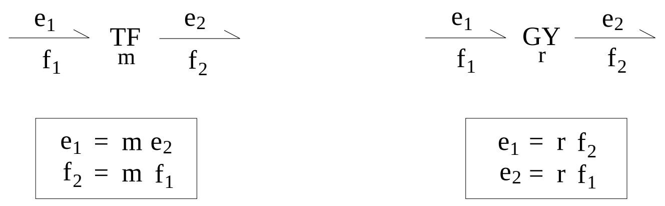

Figure 6: The two basic converter types.

the *gyrator (GY)*. In fact, these two converters are dual to each other. Had the convention for efforts and flows been chosen dually on either the electrical or the mechanical side, the gyrator of Fig. 5 would have become a transformer.

The right–hand part of Fig. 5 shows the mechanical side of the DC–motor. Some of the converted power gets stored in the rotor J (another inductance), and some of it gets dissipated in the bearings B (another resistor). Since the angular velocity is the same for all these elements, the power distribution is described by another 1–junction. The remaining power:

$$P_{\text{mech}} = \tau \cdot \omega_m \tag{20}$$

is available to drive a rotational load. It leaves the subsystem through the second power port, which is represented in the bond graph as a second 0–junction.

The reader may notice that the flux constant, ψ_{Flux}:

$$\tau_m = \psi_{\text{Flux}} \cdot i_a \tag{21}$$

expressed in Nm/A, and the electromotive force constant, ψ_{EMF}:

$$u_i = \psi_{\text{EMF}} \cdot \omega_m \tag{22}$$

expressed in Vs/rad, are physically two different aspects of one and the same phenomenon, and the numerical values of these two constants must hence be the same:

$$\psi_{\text{EMF}} = \psi_{\text{Flux}} \tag{23}$$

otherwise the gyrator will either lose or miraculously generate energy. The consequent application of energy–flow modeling (here expressed in terms of a bond graph) makes it possible to discover such types of modeling errors. Signal–flow modeling (e.g. expressed in terms of a block diagram) will not reveal this limitation.

Does the bond graph approach to power modeling help resolve some of the previously mentioned riddles? In fact, it does. To clarify this, the problem of temperature distribution along an ideally insulated rod shall be revisited.

It has been known since the time of Gabriel Kron [Kron, 1962] that the finite difference approximation of Equation (13) can be interpreted as the electrical circuit shown in Fig. 7. A bond graph representation of this circuit is given in Fig. 8: In bond graph notation, it is

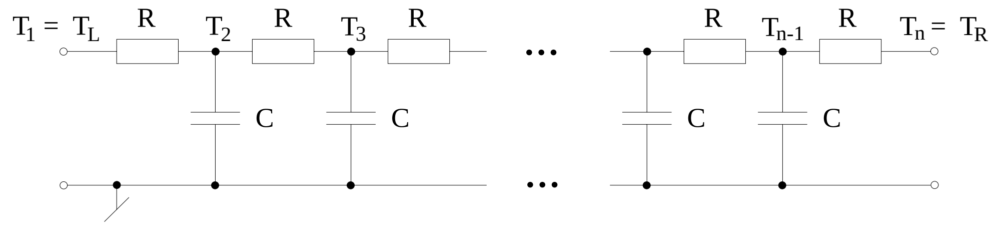

Figure 7: Electrical circuit representation of a diffusion chain.

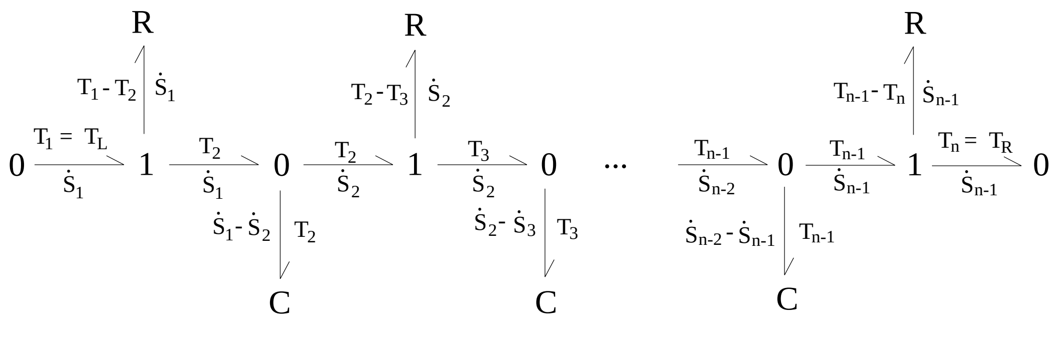

Figure 8: Bond graph representation of a diffusion chain.

common to write heat flow as the product of temperature and entropy flow:

$$\frac{dQ}{dt} = T \cdot \frac{dS}{dt} \tag{24}$$

where the temperature, T, is used as the effort variable, and the entropy flow, dS/dt, is interpreted as the flow variable. Thus, it seems that the bond graph model is providing more than just an implementation of the circuit diagram, since it not only computes temperature values, but also provides a heat flow interpretation.

Unfortunately, the bond graph of Fig. 8 is certainly incorrect. The problem with this bond graph has to do with the dissipated heat in the resistive elements. It was okay to ignore the generated heat in the armature resistance and bearings of the DC–motor model. It simply means that the modeler chose not to include the thermal variables and equations in his or her model. However, the temperature dissipation model is already in the thermal domain. There is no way that the modeler can ignore what happens to the generated entropy and get away with it. Since the rod is thermally insulated, the generated entropy cannot escape and has to stay in the rod.

Jean Thoma proposed to represent resistors with entropy generation as *resistive source (RS)* elements [Thoma, 1990]. The modified bond graph takes the form of Fig. 9. The bond

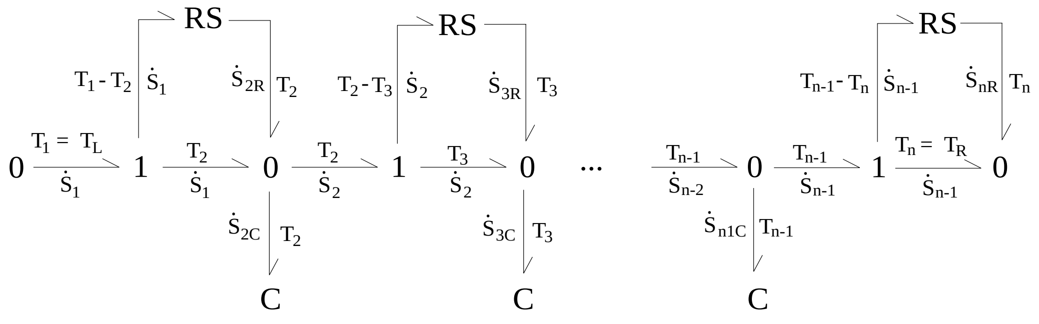

Figure 9: Corrected bond graph representation of a diffusion chain.

graph of Fig. 9 represents correctly both the entropy generation and flow in the rod as well as the temperature distribution. Now, it has become easy to model also the electrically heated rod. Each resistor element is represented in the bond graph through an additional small source of entropy, as shown in Fig. 10.

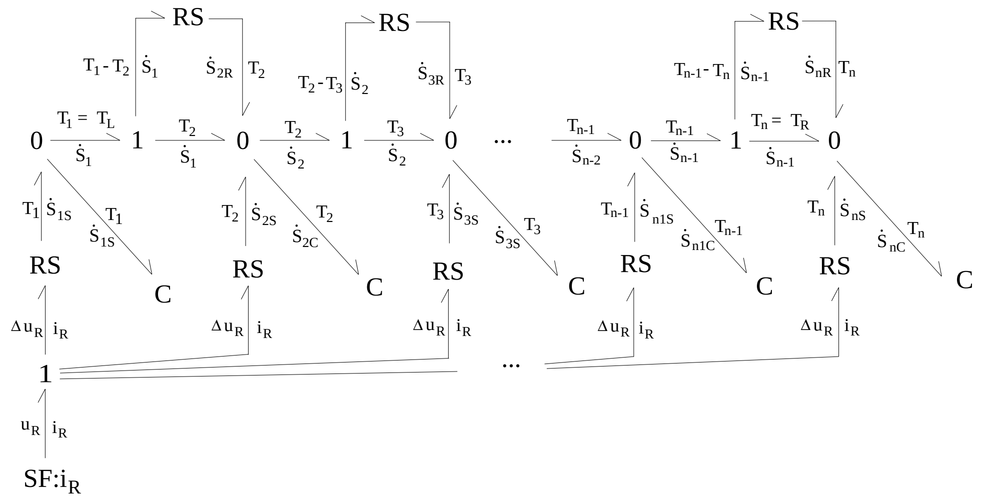

Figure 10: Bond graph representation of an electrically heated rod.

4 Object–Oriented Modeling

It was shown that using bond graphs for representing physical systems constitutes a safe way of modeling. However, in some cases, bond graphs may be unnecessarily constraining. What is important is not the bond graph syntax *per se*, but some of the properties inherent in bond graphs.

The most important feature of a bond graph is that it operates on energy flows rather than on individual signals. Thereby, it is guaranteed that no component and no interface will ever generate or lose energy in an uncontrolled fashion. Yet, other modeling formalisms may offer this same feature.

Another important facet of bond graphs is the fact that they are inherently acausal. Looking at the bond graphs presented in the previous section of this article, nothing indicates whether the resistive elements are of the voltage–drop–causer or of the current–flow–causer variety. The equations that are derived from a bond graph are declarative in nature. It is correct that some researchers have introduced so–called "causality strokes" into the bond graph methodology, turning the formerly acausal bond graphs into a causal variety. However, as shown in [Cellier *et al.*, 1995], causality strokes are not necessarily helpful, and, in the case of switching circuits, they are even harmful. Hence it was decided not to augment the bond graphs shown in this article by causality strokes.

A third very important feature of bond graphs is that they are modular and hierarchical [Cellier, 1992]. This makes it possible to construct bond graphs of submodels and connect them topologically to more complex bond graph elements that can then be connected further in a hierarchical manner. Since the interface points between bond graph submodels can be restricted to be always 0–junctions, power continuity at the interface between submodels is automatically guaranteed.

However, a bond graph is a fairly low–level modeling interface. It may not always be convenient or efficient to model down to that interface. Other modeling methodologies share some of the benefits of bond graphs but offer an either more convenient or higher–level interface. For example, if a modeler wants to describe an electrical circuit, there is nothing wrong with using the circuit diagram as a modeling tool. Circuit diagrams are modular and hierarchical, and power continuity at the interface between submodels is guaranteed by systematically applying Kirchhoff's laws to nodes and/or meshes [Huelsman, 1984]. The bond graph has the advantage of being domain independent, but a circuit diagram may be a more natural and equally powerful tool as far as electrical circuits are concerned.

The case of three–dimensional mechanical devices, so–called multibody systems (MBS) [Nikravesh, 1988], is considerably worse. An important subclass of MBS devices are tree–structured robots. The use of bond graphs to describe the motion of such robots will, within the constraints of today's software technology, almost invariably lead to very inefficient simulation code. The problem is that bond graphers always express the motion of bodies in terms of absolute velocities [Bos, 1986]. This leads, in the generated equations, necessarily to large algebraically–coupled higher index equation systems that are very hard to break. The trick in generating efficient simulation code is always to express the motion of each joint relative to the motion of the previous one [Otter *et al.*, 1993]. It is evidently true that the bond graph contains complete information about the system, and a smart preprocessor could convert the model to a form that might be used to generate efficient run–time code, but this would necessarily call for specialized translation algorithms that only work in the case of MBS problems. It is not useful to develop such a preprocessor, because it is perfectly feasible to come up with a modular hierarchical description of MBS components that can be connected in a safe fashion and which generates efficient simulation code directly.

The object–oriented modeling paradigm, introduced by Hilding Elmqvist in the late 70s [Elmqvist,1978], provides for a platform that allows the modeler to implement all of the above modeling formalisms including electrical circuit diagrams [Cellier and Elmqvist, 1993], bond graphs [Cellier, 1992] and MBS [Otter *et al.*, 1993]. Use of this methodology is a little less safe than using a more restricted modeling tool, such as a circuit diagram editor or a bond graph editor, because it provides more flexibility. It is the user's and/or model library developer's responsibility to ensure that the power continuity equation is satisfied at the interfaces between models. Modelers are thus advised, not to define the *cuts* (the interface points) of their models arbitrarily, but to restrain themselves, and only use proven connection mechanisms. The question that the modeler should *always* ask him– or herself when designing these model interfaces is whether connecting such models in an arbitrary fashion will always ensure that power, momentum, and mass are balanced at the interfaces.

It is the authors' experience that many modeling errors are introduced while interconnecting models that violate balance equations at their interfaces. Hence it is important to systematize the approach to interfacing models, and the large–scale example shown in the sequel of this article explains one way of accomplishing this.

An object–oriented modeling environment for physical system modeling should offer at least the following features:

- *Encapsulation of knowledge*: The modeler must be able to encode all knowledge related to a particular object in a compact fashion in one place with well–defined interface points to the outside.

- *Topological interconnection capability*: The modeler should be able to interconnect objects in a topological fashion, plugging together component models in the same way as an experimenter would plug together real equipment in a laboratory. This requirement entails that the equations describing the models must be declarative in nature, i.e., they must be acausal.

- *Hierarchical modeling*: The modeler should be able to declare interconnected models as new objects, making them indistinguishable from the outside from the basic equation models. Models can then be built up in a hierarchical fashion.

- *Object instantiation*: The modeler should have the possibility to describe generic object classes, and instantiate actual objects from these class definitions by a mechanism of model invocation.

- *Class inheritance*: A useful feature is class inheritance, since it allows the encapsulation of knowledge even below the level of a physical object. The so encapsulated knowledge can then be distributed through the model by an inheritance mechanism, which ensures that the same knowledge will not have to be encoded several times in different places of the model separately.

- *Generalized Networking Capability*: A useful feature of a modeling environment is the capability to interconnect models through *nodes*. Nodes are different from regular models (objects) in that they offer a variable number of connections to them. This feature mandates the availability of *across* and *through* variables, so that power continuity across the nodes can be guaranteed.

In the sequel, one such object–oriented modeling environment, Dymola [Elmqvist, 1995] shall be described. The language Omola with its environment Omsim [Andersson, 1994] represents another research effort with similar aims and properties.

5 Dymola: An Object–Oriented Modeling Tool

Dymola offers all of the above features plus a few more. In Dymola, an electrical resistor (a simple object) can be described as:

```
model class Resistor
    cut WireA(vA/iA), WireB(vB/iB)
    main cut TwoWires [ WireA , WireB ]
    main path Orientation < WireA , WireB >
    local u, i
    parameter R
        u = vA − vB
        i = iA
        iA + iB = 0
        u = R ∗ i
end
```

A *cut* is a mechanism used for interconnection of models. It describes an interface point of the model. The resistor has two such interface points, namely its two pins, called *WireA* and *WireB* respectively. Each pin carries two variables, an electrical potential (an across variable), and the current *into* the device (a through variable).

In Dymola, both the across variables and the through variables are lists of variables separated by a comma; the two lists themselves are separated by a slash. The difference between across and through variables becomes apparent in a connection only: *across variables* are set equal at a connection point, whereas *through variables* are summed up to zero. Note that, in the above example, the product of the two *cut* variables (e.g. $v_A \cdot i_A$) is the power flowing into the element. In this way, it is guaranteed that the energy is balanced at the interfaces of the resistor. In other words, across and through variables can be interpreted as the effort and flow variables of the bond graphs, whereby connection points are always 0–junctions.

The third and *main cut* is a hierarchical cut. It consists of the two previously defined basic cuts. This declaration enables the modeler to bend the legs of the resistor and plug the resistor as a whole with a single motion into a socket attached to a circuit board. The *path* declaration provides a logical orientation of the device. If resistors are connected in parallel or in series, Dymola uses the *path* declaration to determine, which way they are connected into the circuit. In the case of a resistor, this may not really matter, but in other cases, it might. The four equations formulating the relationships between the terminal (interconnect) variables, the local variables, the parameters, and the constants are declarative in nature. Dymola will solve each of them for the appropriate variable, once it has access to all the equations.

The reader may notice that many of the equations described in the above model are shared by all one–port devices. Thus, it makes sense to encapsulate this knowledge in a separate model class, and then migrate the knowledge to the resistor class by means of inheritance. In Dymola, this is done in the following way:

```
model class OnePort
    cut WireA(vA/i), WireB(vB/ − i)
    main cut TwoWires [ WireA , WireB ]
    main path Orientation < WireA , WireB >
    local u
        u = vA − vB
end

model class Resistor
    inherit OnePort
    parameter R
        u = R ∗ i
end
```

The *inherit* statement tells Dymola to copy all the declarations and equations from the parent class to the inheriting class. The current equations were slightly simplified to avoid having to carry around unnecessary aliases.

An interesting one–port is the electrical switch:

```
model class Switch
    inherit OnePort
    terminal open
        0 = if open then i else u
end
```

If the switch is open, then $i = 0$, else $u = 0$. The reader may remember that equations are declarative in nature, and that the computational causality will only be determined later. It turns out that Dymola can turn around *if* expressions as easily as algebraic expressions. A *terminal* is another interface point. It is a simplified *cut* that contains only a single across variable. Whereas *WireA* and *WireB* are power interfaces, *open* is a signal interface. The variable *open* is a control input to the model. It could have been declared as *input*, but there were good reasons for not doing so and using the direction–neutral *terminal* declaration instead, as shall soon become clear. By using a signal interface, the modeler acknowledges that he or she considers the energy flow needed to turn the switch negligible in comparison with other power paths in the system. It is also acknowledged that both the time needed to deliver the energy to the switch as well as the time needed for the switch to react can be ignored. Models never reflect all facets of reality, and simplifications are okay, as long as the modeler makes them consciously, and knows what he or she is doing.

An ideal diode can be described in the following way:

```
model class Diode
    inherit Switch
        open = u <= 0 and not i > 0
end
```

An ideal diode is a switch. A logical equation is added that determines the value of the boolean variable *open*. The diode goes into its *on* state (switch closed), when the voltage becomes positive, and it goes into its *off* state when the current becomes negative.

Evidently, Dymola's inheritance option is hierarchical, since diodes are switches which in turn are one–ports.

It becomes now evident, why *open* had not been explicitly declared as an *input*. In the diode model, the computational causality of the signal port is turned around. Now, *open* is no longer a control input, but merely a measurement variable that is reported to the outside world through the interface. It is hardly ever justified to declare a variable explicitly as *input* or *output*, except at the top level, i.e., in the main model.

Given the simple electrical circuit shown in Fig. 11:

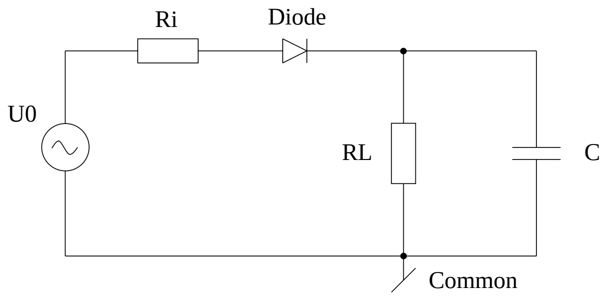

Figure 11: Rectifier with load.

This circuit can be modeled in Dymola as follows:

```
model Rectifier
    submodel (VSource)   U0
    submodel (Resistor)  Ri(R = 10), RL(R = 50)
    submodel (Capacitor) C(C = 0.001)
    submodel (Diode)     D
    submodel Common

    parameter f = 50

    connect Common − ((U0 − Ri − D)//C//RL)

    U0.u0 = sin(2 ∗ 3.14159 ∗ f ∗ Time)
end
```

Instantiations of models from model classes are obtained using the *submodel* statement. The *connect* statement plugs the circuit together. Since this circuit was sufficiently simple, no *nodes* were explicitly introduced, and the interconnections between the models were simply accomplished through series (−) and parallel (//) connections[1]. The dot–notation is used to access individual variables of submodels directly rather than through port connections. $C.u$ is the variable u of the model C, which in turn is of model class *Capacitor*. This is a case of back–door programming, and the feature should be used sparingly.

Object–oriented modeling can be used as an alternative to bond graph modeling, provided the model classes are designed in such a way that the power balance is satisfied at

[1]For larger models it may be more convenient to make the connections using Dymodraw, a graphical object–diagram editor that generates Dymola code.

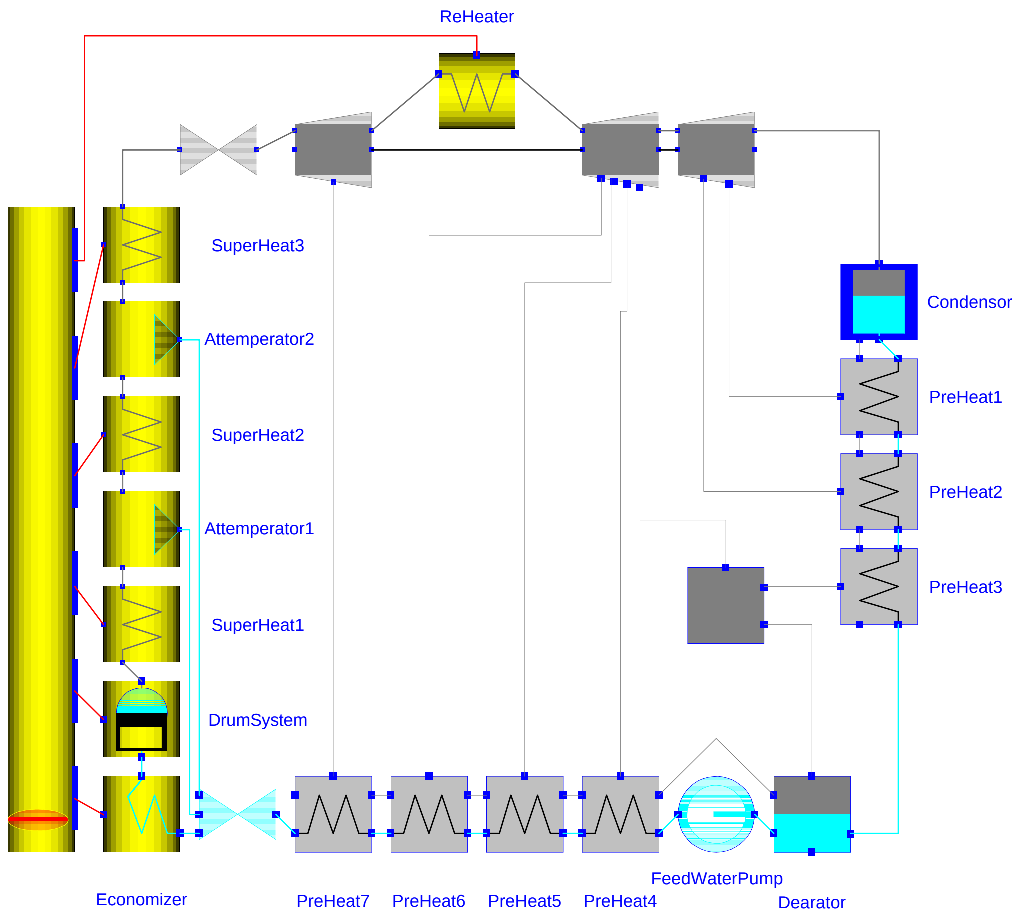

Figure 12: Steam power plant modeled in Dymodraw.

the model interfaces. Object–oriented modeling has the additional benefit that it resembles more closely the physical reality (e.g. electrical components are represented in their familiar form rather than through more abstract domain–independent bond graph elements). Object–oriented models are more general than bond graphs, since they can just as easily be used to represent block diagrams or signal flow graphs when the need arises.

6 Object–Oriented Model of a Steam Power Plant

Fig. 12 shows a top–level object–diagram of a steam power plant originally modeled by [Lindahl, 1976] consisting of a drum system, super–heaters, pre–heaters, several turbines operating at different pressure levels, attemperators, dearators, condensors, a feedwater pump, a combustion chamber, and a few other components.

The drum can, for example, be described in the following manner. The model given here has been derived from first principles by [Åström and Bell, 1988]. Total energy and mass balances are stated as well as energy and mass balances for the risers. The variation of the steam–water mass ratio along the risers is complex. A linear variation is assumed, which, after integration along the risers, gives the average steam–water volume ratio, a_m, as shown below.

```
model class Drum
    { Including risers and downcomers }
    parameter Adrum = 20 { Drum wet area [m^2] } ,
              Vdrum = 40 { Drum volume [m^3] } ,
              Vr = 37    { Riser Volume [m^3] } ,
              Vdc = 19   { Downcomer volume [m^3] } ,
              k = 0.01   { Friction coefficient } ,
              cp = 360   { Specific heat of metal [J kg^-1 K^-1] } ,
              m = 0      { Mass of metal [kg] } ,

    cut InWater  (p, Hfw/qfw)      { Pressure, enthalpy and flow rate of feed water. }
    cut InPower  ( /Pow)           { Power from fuel. }
    cut OutSteam (p, Hs/ - qs)     { Pressure, enthalpy and flow rate of steam. }

    local dl     { Drum level } ,
          am     { Steam quality volume ratio } ,
          Vw     { Drum water volume } ,
          Vwt    { Total water volume } ,
          xr     { Steam quality at riser outlet } ,
          Vst    { Total steam volume } ,
          Tm     { Metal temperature } ,
          qdc    { Downcomer flow } ,
          qr     { Riser flow } ,
          rs     { Steam density } ,
          Hw     { Water enthalpy } ,
          rw     { Water density } ,
          Ts     { Steam temperature }
    local Ed, Md, Er, Mr { Energies and masses }

          { Total energy balance. }
          Ed = rs * Vst * Hs + rw * Vwt * Hw + m * cp * Tm
          der(Ed) = Pow + qfw * Hfw - qs * Hs

          { Total mass balance. }
          Md = rs * Vst + rw * Vwt
          der(Md) = qfw - qs

          { Energy balance in risers. }
          Er = rs * am * Vr * Hs + rw * (1 - am) * Vr * Hw
          der(Er) = Pow + qdc * Hw - xr * qr * Hs - (1 - xr) * qr * Hw

          { Mass balance in risers. }
          Mr = am * rs * Vr + (1 - am) * rw * Vr
          der(Mr) = qdc - qr

          { Momentum balance for downcomers. }
          am * Vr * (rw - rs) = k * qdc * *2/2

          { Average steam-water volume ratio in the risers. }
          am = rw/(rw - rs) * (1 - rs/(rw - rs)/xr * ln(1 + (rw/rs - 1) * xr))

          { Total steam volume. }
          Vst = Vdrum - Vw + am * Vr

          { Total water volume. }
          Vwt = Vw + Vdc + (1 - am) * Vr

          { Drum water level. }
          dl = (Vw + am * Vr)/Adrum

          { Steam and water properties. }
          Hs = H2P(p)
          rs = 1/V2P(p)
```

```
        $Hw = H1P(p)$
        $rw = 1/V1P(p)$
        $Ts = TP(p)$
        $Tm = Ts$
    end
```

The model of the drum is in itself interesting. It is of index two because of the constraint that the sum of the water volume and steam volume must be equal to the drum volume including risers and downcomers. This means that water and steam volumes cannot both be chosen as state variables. The index can be reduced by differentiating certain equations [Pantelides, 1988]. The index reduction technique also makes it possible to choose more appropriate state variables p, V_w, and x_r than the variables appearing differentiated E_d, M_d, E_r, and M_r. When solving for the corresponding derivatives, a system of 13 linear simultaneous equations is detected. It can be reduced to a 3×3 system by use of tearing, which can then be solved either symbolically or numerically [Elmqvist and Otter, 1994]. In [Åström and Bell, 1988], the determination of which equations to differentiate, the differentiation, elimination of variables, and the solution of the linear system of equations were done manually. Dymola is capable of performing all these steps automatically.

7 Conclusions

This article has shown energy flow modeling to be a corner stone in safe descriptions of physical systems. Many potential problems can be avoided by systematically applying this approach to modeling. However, energy flow modeling puts heavy demands on the modeling software. Only a full–fledged object–oriented modeling tool, such as Dymola, can satisfy these demands in all circumstances.

A very important problem has not been dwelled upon in this article. Evidently, it is never possible to consider all aspects of a system in a model. Any model must always represent an idealization, a simplification of reality. How this is done, which facets of reality are left out from the model, is a question that needs to be addressed by the modeler. Finding the answer to this question is not something that can be automated in general. It requires intuition into how the system works, and a lot of experience on the side of the *human* modeler.

Yet, energy balances can assist the user also in this respect. It is a sound proposition to request that even after the model simplifications have become effective, all energy flows internal to the model are still balanced, and also that energy can enter or leave the model only through a limited number of well–defined and carefully monitored ports. Although it may not always be possible to decide on the basis of energy considerations alone, which facets of reality to keep in the model and which other facets to throw out, energy modeling *will* support the user in ensuring that simplifications, once decided upon, are implemented in a consistent fashion.

References

Andersson, M. 1994. *Object–Oriented Modeling and Simulation of Hybrid Systems*, Ph.D. thesis TFRT–1043, Dept. of Automatic Control, Lund Inst. of Technology, Lund, Sweden.

Åström, K.J. and Bell, R.D. 1988. Simple drum–boiler models. *Proc. IFAC Symp. Power Systems Modelling and Control Applications*, Brussels, Belgium.

Bos, A.M. 1986. *Modelling Multibody Systems in Terms of Multibond Graphs with application to a motorcycle*, Ph.D. dissertation, Technical University Twente, Enschede, The Netherlands.

Cellier, F.E. 1992. Hierarchical non–linear bond graphs: a unified methodology for modeling complex physical systems. *Simulation*, 58(4):230–248.

Cellier, F.E. and Elmqvist, H. 1993. Automated formula manipulation supports object–oriented continuous–system modeling. *IEEE Control Systems*, 13(2):28–38.

Cellier, F.E., Otter, M., and Elmqvist H. 1995. Bond graph modeling of variable structure systems. *Proc. ICBGM'95, 2nd Intl. Conf. on Bond Graph Modeling and Simulation*, Las Vegas, NV, pp. 49–55.

Elmqvist, H. 1978. *A Structured Model Language for Large Continuous Systems*, Ph.D. dissertation. Report CODEN:LUTFD2/(TFRT–1015), Dept. of Automatic Control, Lund Inst. of Technology, Lund, Sweden.

Elmqvist, H., and Otter, M. 1994. Methods for Tearing Systems of Equations in Object–Oriented Modeling. *Proc. ESM'94, European Simulation Multiconference*, Barcelona, Spain, pp.326–332.

Elmqvist, H. 1995. *Dymola — User's Manual*, Dynasim AB, Research Park Ideon, Lund, Sweden.

Holman, J.P. 1992. *Heat Transfer*, 7th ed., McGraw–Hill, New York.

Huelsman, L.P. 1984. *Basic Circuit Theory* 2nd ed., Prentice–Hall, Englewood Cliffs, N.J.

Karnopp, D.C., Margolis, D.L., and Rosenberg, R.C. 1990. *System Dynamics: A Unified Approach*, 2nd ed., John Wiley & Sons, New York.

Kron, G. 1962. *Diakoptics — The Piecewise Solution to Large–Scale Systems*, MacDonald & Co., London.

Lindahl, S. 1976. *A Non–linear Drum Boiler – Turbine Model*, Report TFRT–3132, Dept. of Automatic Control, Lund Inst. of Technology, Lund, Sweden.

Nikravesh, P.E. 1988. *Computer–Aided Analysis of Mechanical Systems*, Prentice–Hall, Englewood Cliffs, N.J.

Otter, M., Elmqvist, H., and Cellier, F.E. 1993. Modeling of multibody systems with the object–oriented modeling language Dymola. *Proc. NATO/ASI, Computer–Aided Analysis of Rigid and Flexible Mechanical Systems*, Troia, Portugal, Vol. 2, pp. 91–110.

Pantelides, C.C. 1988. The consistent initialization of differential–algebraic systems. *SIAM J. Scientific and Statistical Computing*, 9:213–231.

Paynter, H.M. 1961. *Analysis and Design of Engineering Systems*, MIT Press, Cambridge, MA.

Thoma, J.U. 1990. *Simulation by Bondgraphs*, Springer–Verlag, Berlin.