A Composable Simulation Environment to Support the Design of Mechatronic Systems

Antonio Diaz-Calderon

Submitted in Partial Fulfillment
of the Requirements
for the degree of
Doctor of Philosophy
in Electrical and Computer Engineering

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

June 2000.

This research was supported in part by DARPA under contract ONR # N00014-96-1-0854, by the National Institute of Standards and Technology, by the NSF under grant # CISE/IIS/KDI 9873005, by the Pennsylvania Infrastructure Technology Alliance, by the National Council of Science and Technology of Mexico (CONACyT), and by the Institute for Complex Engineered Systems at Carnegie Mellon University. The views and conslusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the funding agencies.

Abstract

A Composable Simulation Environment to Support the Design of Mechatronic Systems

by

Antonio Diaz-Calderon

Doctor of Philosophy in Electrical and Computer Engineering

Carnegie Mellon University

Professor Pradeep K. Khosla, Chair

Multi-disciplinary simulation is an important tool in the design of mechatronic systems. The demand for less expensive products that can be introduced quickly to respond to market demands requires that these products be designed with minimal prototyping, relying on simulation instead to verify design requirements.

The use of physical prototypes for design verification is a very costly and time-consuming process. As a result, there is an important trend towards design verification and analysis in virtual, simulated environments. However, creating simulations for complex mechatronic systems can be quite a challenging task itself. This issue becomes more complicated when the design process is considered. In this case, simulation models must allow the designer to work with high-level concepts that can be specialized at later stages in the design process. To be able to support a simulation-based design paradigm, new simulation tools are required. Such simulation tools should allow designers and analysts to combine models from different disciplines into integrated system-level models, allow models of sub-systems to evolve throughout the design process (from conceptual design to detailed design), and allow designers and analysts with expertise in different disciplines to collaborate in an open design environment.

Multi-disciplinary and evolutionary simulation models are based on our *port-based modeling* paradigm. In our modeling paradigm, systems are described from a systems engineering point of view where subsystems interact with their environment through energy exchange. We describe systems as self-contained entities, whose interactions with the environment are independent of the internal behavior of the system. The port-based modeling paradigm is based on two concepts: *ports* and *connections*. Ports represent localized points on the boundary of the system where energy exchange between the system and the environment takes place. A connection between two ports represents the energy exchange between two subsystems.

Based on our port-based modeling paradigm, we build system simulations through composition of individual modeling entities; we call this approach *composable simulation*. In composable simulation, CAD models of system components are augmented with simulation models describing the component's dynamic behavior in different energy domains. By composable simulation we mean the ability to generate system-level simulations automatically by simply organizing the system components in a CAD system. A system component can be either a physical component (electrical motor, gearbox, etc.) or an information technology component (embedded controller or other software component). Each of these system components has one or more simulation models associated with it describing its dynamics in multiple energy domains, across energy domains, and possibly at multiple levels of accuracy (with varying computational requirements). When these system components are combined into a complete system, our framework automatically combines a selection of the associated component models into a system-level simulation.

To support the evolutionary aspect of the design process, we introduce a new modeling paradigm called *reconfigurable models*. Reconfigurable models are an extension to our port-based models. In a reconfigurable model, the interface of the model and the implementation of its behavior are considered to be two separate, but dependent concepts. By considering these two concepts independently, it is possible to associate different implementations to a single interface, achieving structural modification of models in addition to the traditional changes in parameter values.

A reconfigurable model is based on two principles: *composition* and *instantiation*. The composition principle denotes the mechanism by which the formal behavior of the component is described in terms of interfaces of subcomponents and their interactions. The second principle—the principle of instantiation—describes the mechanism by which the interface of a model is bound to its implementation

A reconfigurable model represents the modeling space of a system component. Elements of this space are models that vary from abstract (conceptual) to concrete (fully determined), thereby supporting the evolutionary nature of the design process. They allow the designer to work with high-level concepts that can be specialized at later stages in the design process.

Table of contents

List of figures

Chapter 1 Introduction

1.1 Motivation

Due to the fierce competition in the current global economy, it is critical for successful companies to react quickly to changing trends in the marketplace: new technologies, changes in customer demands, fluctuations in the cost of basic materials and commodities, etc. Because design is such an important component in the development of new products [151], reduced design cycle time will provide a distinct competitive advantage.

Until recently, design verification required building a physical prototype of the design artifact—a costly and time-consuming process even when rapid prototyping equipment is used. With the advent of inexpensive high-speed computing, it has become feasible to verify a design in a virtual environment, based on functional simulations of the design artifact. Such virtual prototyping has the potential to provide significant reductions in the design cycle time, under the assumption, of course, that it is less expensive and time-consuming to create a simulation model than a physical mock-up. However, creating high

fidelity simulations for complex mechatronic systems can be quite a challenging task itself. To maximize the benefits of virtual prototyping, it is important that simulations can be created effortlessly and at any stage of the design cycle. This thesis proposes a simulation framework based on the concepts of port-based objects and composition that simplifies the creation of virtual prototypes.

The research described here focuses on mechatronic systems. Mechatronics can be defined as “a technology which combines mechanics, electronics and information technology to form both functional interaction and spatial integration in components, modules, products and systems” [19]. Examples of mechatronic systems are anti-lock braking systems, automatic guided vehicles, and consumer products like CD-players.

The design approach for today's complex multi-disciplinary systems has changed dramatically. Traditionally, multi-disciplinary system design has employed a sequential design-by-discipline approach [123]. For example, the design of an electromechanical system is often accomplished in three steps beginning with the mechanical design, followed by the power and microelectronics, and finally the design and implementation of the control algorithm. The main drawback of the design-by-discipline approach is that fixing the design at various points in the sequence imposes artificial constraints that needlessly restrict the design space: inter-domain coupling is neglected. A mechatronic approach (one where the inter-domain coupling is considerably large), on the other hand, is based on a concurrent, instead of sequential, approach to discipline design, resulting in products with more synergy between sub-systems. The mechatronic design approach results in a tight integration of subsystems with significant functional interaction as well as spatial integration between the different disciplines.

Typical benefits from the mechatronic design approach appear when the domain boundaries in a design are not fixed. By exploiting inter-domain coupling and considering trade-offs between solutions in different domains it may be possible to achieve a performance-to-cost ratio that cannot be obtained using classical approaches (i.e., design-by-discipline).

Since the mechatronic design approach is multidisciplinary, we need a multidisciplinary simulation paradigm to support virtual prototyping. To approach this need, in this thesis,

we propose a modeling paradigm based on port-based objects. Simulation tools based on this modeling paradigm should combine port-based models from different disciplines into integrated system-level models.

To provide simulation support throughout the design process, models in this modeling paradigm should be evolvable and shareable. An evolvable port-based model is one that captures the behavior of a physical component at different levels of detail, and provides mechanisms for its reconfiguration. As a result, the designer can create simulations of the design at different stages in the design process, while changing the behavior of the models as the design process evolves. Port-based models need to be shareable to support the multidisciplinary aspects of the design process. A shared port-based model is reused in different contexts by different design teams. In this way, designers can collaborate in a design environment by working in different subsystems and sharing their designs.

We believe that the framework developed in this thesis for composable simulation addresses these requirements by integrating simulation tightly with the design environment (i.e. CAD software) and allowing the designer to create the simulations directly with minimal intervention of simulation experts.

1.2 Objectives and Approach

The goals of this thesis are the following:

1. Develop a simulation framework in which system-level simulation models can be composed from sub-system models in different disciplines.
2. Formalize a multi-domain modeling paradigm for mechatronic systems: a representation of the models that allows the system model to evolve with the design process, increasing in detail as the design process progresses.
3. Develop the infrastructure to integrate simulation models with the design environment so that consistency between the artifacts and their corresponding models is automatically maintained at all times.

4. Evaluate these technological concepts by developing a prototype computer aided engineering design environment.

In this thesis we approach the design of mechatronic systems as follows:

- We regard artifacts from a systems point of view, i.e., as a structure of interrelated elements that are embedded in an environment.
- We concentrate on aspects that relate to the energetic behavior of the systems. It is through energy exchange that functional interactions between subsystems take place.
- We study systems that realize their functionality using mainly electronic and mechanical parts, and information technology components (such as controllers). However, the methodology presented in this thesis is by no means limited to these three energy domains.

To attain our goals we have developed a multi-domain modeling paradigm based on the composition of port-based objects. These objects represent system models that have well-defined interaction points (i.e., ports). Ports represent points on the system where energy flow is present. Consequently, a port-based object that has ports in different energy domains models multi-domain systems.

Defining interactions between port-based objects creates system-level representations. These interactions are defined by connecting ports of different objects having the same energy domain. Once a complete system-level representation is given, it is translated into a low-level system representation based on a linear graph [143], from which a set of differential algebraic equations is derived.

In addition to a modeling paradigm, other issues need to be addressed to complete the goals of this thesis, including:

Composability of port-based simulation models. We regard artifacts from a systems point of view. This means that in our framework for composable simulation, system components (including information technology components) have well-defined interaction points (i.e., ports) and well-defined behavior (i.e., differential equations). When these com-

ponents are combined into a complete system, their behaviors should also be combined into a system-level behavior that includes the behavior derived from the interaction between system components. This approach is different from the approach taken in traditional simulation environments such as SimuLink (Matlab) in that we support composition of system-level components as opposed to composition of simulation models. Composition of system components does not map directly into composition of the simulation models; the system-level simulation model is not simply a concatenation of individual component models. Sometimes, additional simulation components need to be introduced in order to describe the interactions between components (e.g. friction between gears). These additional models vary with the physical layout of the components and the type of interactions between them. Other times, multiple physical models are combined into one simulation component.

The port-based modeling paradigm captures the multi-domain characteristics of mechatronic systems. That is, it captures their functional interactions and spatial integration into a complete system-level simulation model.

Model reconfiguration. Reconfigurability of simulation models provides the flexibility to allow different representations for the same modeling concept. That is, a modeling concept can be represented in the form of an analytical expression relating inputs and outputs, a software method, or as a structural arrangement of submodels (Figure 1-1). We extended the port-based modeling paradigm to support reconfigurable models.

Low-level system representation. The underlying mechanism to represent mathematical models of port-based objects is based on linear graphs [143]. We call these graphs *system graphs*. A system graph is a linear graph that represents the energy flow through the system. It is based on two types of measurements: across and through measurements. The across and through variables for each energy domain are chosen such that the power of the corresponding component is equal to their product. The mathematical relations between these variables are called terminal equations; they define the component's physical characteristics.

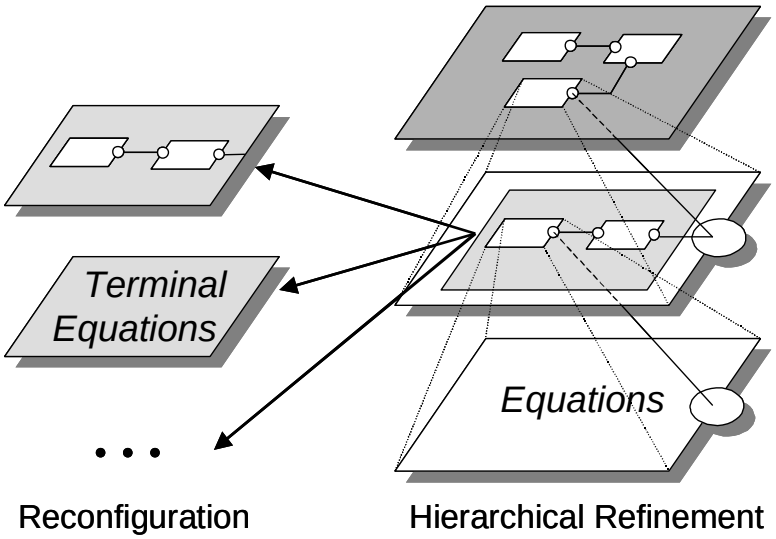


Figure 1-1. Hierarchical encapsulation and reconfigurability of models.

Model refinement. Ideally, at any time during the design cycle, a simulation would be available that requires minimal computational resources, but still is sufficiently rich to verify whether certain design criteria are met. This requires that simulation models be refined as one progresses through the design task. At the conceptual design stage, high-level functional models can be used to evaluate some initial design trade-offs. As physical components are selected to implement the required function, the high-level simulation model can be refined by replacing functional models with component models. At the same time, model refinement is achieved by increasing the level of detail of individual component models and their interactions (Figure 1-1). For example, one could start with a simple kinematic simulation of a mechanism, then add dynamics, and include control algorithms. One can refine the model further by considering interactions in different energy domains, such as electromagnetic and thermal domains. Finally, a model can be refined by increasing the number of degrees of freedom; this can be accomplished by modeling mechanical components as flexible rather than rigid, or including parasitic capacitance in an electrical model. Each level of refinement requires significant changes to the overall simulation model. Particularly cumbersome are the refinements that result in a modified simulation topology.

We approach the problem of model refinement through reconfigurable models. Reconfigurable models provide the mechanisms to perform parameter configuration as well as structural configuration of models. The structural configuration mechanism is useful when the refinement involves changes in the basic topology of the system model.

Model description. The concept of composable simulation becomes even more useful as component models become more readily available (e.g., manufacturers may one day include simulation models with the components they sell), and when they can be reused, shared, and exchanged between users. To facilitate such exchange, we capture the complete component model in XML (extensible markup language), a web-compatible, computer-interpretable format.

1.3 Related research

In this section, we will consider the most relevant branches of research that have contributed to the work presented in this thesis. These include software architectures, software engineering, port-based objects, and mathematical modeling. Additional references will be given in subsequent chapters.

1.3.1 Software architectures and object-oriented design

Software architecture is a specification of a class of systems. It consists of a set of specifications called *interfaces* [6, 74], a set of *connection rules* [7, 8] that define valid communication channels between the interfaces, and a set of *formal constraints* that define legal or illegal patterns of communication. Each element in the architecture is divided into two major parts:

Interface: a description of the component's features including its input/output relationship with the environment.

Implementation: a procedural description of the component's interface.

This separation promotes re-usability and assemblability of architectural components. The set of connection rules defines the *topology* of the *connection graph* of the architecture. The connection graph is a graph in which the nodes represent the interfaces to the elements in

the configuration, and the edges represent the communication channels defined by the communication rules. The interfaces specify the components of the system, and the connections and constraints define how the components may interact.

A significant amount of work aimed at formalizing the properties and interactions between elements in the architecture has been performed in this area [2, 6, 7, 8, 72, 73, 74, 75, 122]. In our context, the concepts defined in this area of research are relevant to the definition of our modeling paradigm: the power of reconfigurable models is achieved through the expression of the model by its interface and related implementations.

A technique to encapsulate the interface and implementation of a software architecture is object-oriented design. The use of objects is a popular method for designing reusable software. An object is defined as a software entity which encapsulates data and behavior.

Object-oriented design defines the interrelation and interaction between objects. The interrelation of objects is defined (through inheritance) using the concepts of class, superclass and subclass [1, 115]. A class is intended to describe the structure of all the objects generated from the class. Like any class, a subclass describes the structure of a set of objects. However, it does so incrementally by describing extensions and changes to its direct superclass. Data from a superclass is implicitly replicated in a subclass, and new data may be added. Methods from a superclass may be either replicated in a subclass, by default, or explicitly overridden by similarly named and typed methods.

Inheritance is the sharing of data and methods between a class and its subclasses. However, it is not always the case that inheritance equates to subclassing [1]. This observation is used in our work of reconfigurable models and forms the basis for the organization of components presented in Chapter 6.

1.3.2 Port-based objects

A port-based object is a modeling abstraction that combines the object-based design with port-automaton design [132, 133, 134]. Stewart [133] defines object-based design as a technology that only defines the encapsulation of data and access to that data. A port-automaton

[133] is a concurrent process where an output response is computed as a function of an input response.

A port-based object is defined as an object that includes ports for communication with its environment [133]. As with any standard object, a port-based object has a state and the object is characterized by its own behavior. The internal details of the object are hidden from other objects in the environment; only the ports of the object are visible to other objects. In this model, each port-based object has zero or more *input ports* and zero or more *output ports*. Input and output ports are used for communication between objects in the environment. Communication between objects is established by connecting an output port of an object to an input port of another object.

The port-based modeling paradigm captures signal flow in the system. However, mechatronic systems include, in addition to signal flow, energy flow; i.e., energy-based systems. Therefore we extended the port-based modeling paradigm to model the energy flow of the system.

1.3.3 Mathematical modeling

1.3.3.1 Qualitative and quantitative modeling

In the mathematical modeling of physical systems, two different approaches exist: qualitative modeling and quantitative modeling [21]. Qualitative modeling [14, 48, 54, 55, 56, 65, 66, 102, 137, 152] is an artificial intelligence-based approach in which domain knowledge provides the necessary information for generative or selective modeling. In contrast, quantitative modeling is based on differential algebraic equations that must be analytically or numerically solved to determine the behavior of the system.

Qualitative simulation and quantitative simulation are both abstractions of actual behavior (Figure 1-2). In quantitative simulation, differential equations describe a physical system in terms of a set of state variables and constraints on those state variables. The solution to the equations may be a function representing the behavior of the system over time. Qualitative simulation describes a physical system in terms of qualitative constraints between qualitative states. Kuipers [65] defines the *qualitative state* of a function f at t as a pair

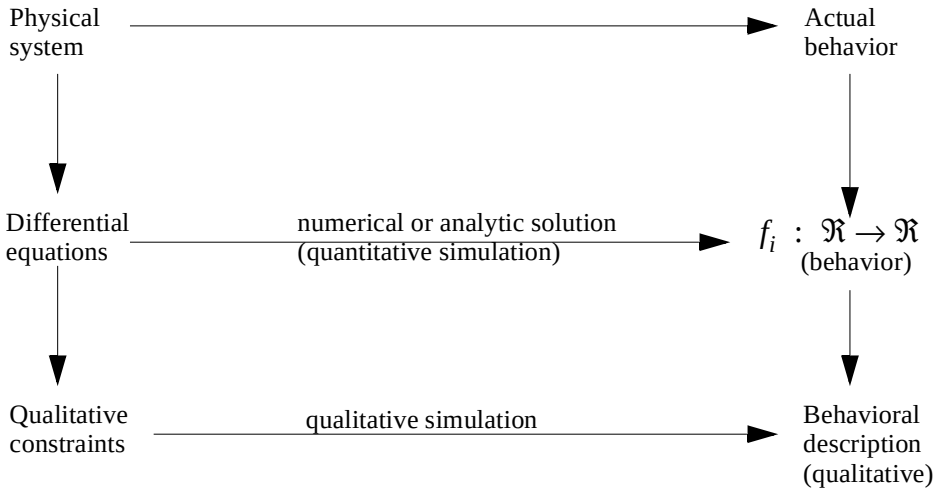


Figure 1-2. Abstractions of actual behavior [65].

$\langle qval, qdir \rangle$ where *qval* is either a point l_j, which is called landmark value, if $f(t) = l_j$; or an interval (l_j, l_{j+1}) if $f(t) \in (l_j, l_{j+1})$, and *qdir* is a label that indicates the direction of the derivative of *f* at *t*. The qualitative behavior of the function *f* in the closed interval [*a*,*b*] is the sequence of qualitative states of *f*.

Qualitative simulation has been applied to the problem of automatic model synthesis [3, 4, 15, 49, 50, 58, 77, 88, 89]. In this context, simulations are composed based on the knowledge of the state of the system which is maintained on Truth Maintenance Systems [28, 29, 30]. These systems maintain a knowledge-base of the current state of the world. Their goal is to maintain a consistent set of assumptions and facts that are relevant to the problem.

1.3.3.2 Graph-based and equation-based modeling

Within quantitative modeling, we can identify two broad modeling paradigms, namely, graph-based and equation-based. Equation-based approaches are generally based in a modeling language that describes the structure of the system. Based on the type of equations that describe the system, we can classify the systems being modeled as discrete or continuous. Modeling languages that provide constructs to describe both types of systems are called hybrid modeling languages [13, 21].

Within the graph-based approach there are two ways to specify a continuous time system, the *conservative law* model and the *signal-flow* model. The conservative law model defines

a system by specifying relations between two complementary variables, and by specifying algebraic constraints between them, which correspond to the *Kirchhoffian network* laws. In contrast, the signal-flow model (or non-conservative) signals represent system variables that flow along lines connecting elements. Elements represent mappings (linear or non-linear) between a specified set of input signals and a specified set of output signals. This type of modeling is also called block diagram modeling.

There are two main approaches to describe a conservative law model, linear graphs and bond graphs. The relationship between physical systems and linear graphs was first recognized by Trent [143] and by Brannin [17]. Roe [109], Koenig [63] and Seshu [121] apply the theory of linear graphs to the systems theory and provide important results that can be directly related to the two basic laws in circuit theory: Kirchhoff's voltage and current laws. Linear graph theory has been used in the analysis of rigid body dynamics [10, 69, 80, 81, 82, 83, 94,106, 107, 108, 124] and in the analysis of other engineering systems that include interaction between different energy domains [40, 86].

Besides linear graphs, bond graphs [16, 36, 61, 76, 97, 112, 113] have also been used for system modeling. Bond graphs are energy-based system descriptions in which energy elements are connected by energy-conserving junction structures. Similar to our approach, bond graphs define a minimal set of generalized elements that can be used to model system behavior across energy domains. Connections between elements are made through power bonds which represent the power flow in the system.

Bond graphs have been used in design of mechatronic systems. In this context the bond graph is used to define a language to describe designs [52, 53, 104, 129]. The bond graph helps to define design rules that can be applied by an expert system. These rules create and modify the design by manipulating the topology of the graph.

Although bond graphs (with appropriate extensions [67, 110, 144]) can be used to represent mechatronic systems, we have chosen linear graphs because they can be more easily adapted to model 3D rigid body mechanics. Furthermore, linear system graphs reflect the topology of the physical system directly, making it easier for non-specialists to create system descriptions.

1.3.3.3 Commercial simulation packages

In the commercial world, there are a number of simulation packages specifically designed for particular application areas. In the area of CAE there exist several packages for rigid body dynamics [120] including ADAMS [84, 91, 92, 93], DADS [70, 90], and Mesa Verde [155, 156, 157]. Within the same area of CAE, although not in the commercial world, we can also include the work by Baraff in the simulation of rigid bodies [11, 12]. The main characteristic of these systems is that the equations of motion are generated numerically from the geometric description of the mechanism. Some (i.e., ADAMS and DADS) can be integrated with CAD packages such that the geometry of the mechanism is directly derived from the CAD model.

A second group of commercial simulation packages provides support for general systems modeling and simulation. We can identify three main approaches: (1) block-diagrams, (2) object-oriented modeling and (3) bond graphs. Systems using the first approach include EASY5 [139] and Matlab/Simulink [140]. Both systems are based on an interactive environment for modeling where the user defines the system as a network of interconnected blocks. EASY5, however, takes the modeling approach a step further in which the system is modeled by defining the interactions between components instead of between simulation blocks as with the block-diagram approach. The main characteristic that distinguishes these systems from the rest is that they use procedural rather than equation-based modeling. In procedural modeling, the model is represented by a collection of functions that, given a set of inputs, compute the respective outputs. In other words, this is a causal modeling paradigm. Equation-based modeling, on the other hand, represents the models by non-causal equations.

Recently, a large number of modeling languages has emerged that provide reuse through object orientation [9, 19, 41, 47, 60, 100, 118]. These languages are all derived from the original simulation language called “the continuous system simulation language (CSSL) [135]”. The object-oriented approach facilitates model reuse and simplifies maintenance. Using these modeling languages, software executables can be generated automatically from individual sub-models and the interactions between them. Although our port-based modeling paradigm bears some resemblance to the object-oriented modeling languages

described so far, it presents an important characteristic that sets it apart from these languages: the modularization of the object into interface and implementation allows our port-based models to span the model space (i.e., the space of alternative models) for a particular model. Consequently, we can select models at different levels of detail and augment the configuration mechanisms to admit changes in system topology.

Object oriented multi-domain modeling languages use the second approach, and we can include Dymola [21, 22, 23, 24, 41, 43, 45, 95], Omola [9], Sidops+ [19], ASCEND [100, 101], NMF [87, 117, 118, 119], Modelica [46, 47, 57, 78], and VHDL-AMS [25, 60]. Dymola is an object-oriented language and a program for modeling large systems. Reuse of modeling knowledge is supported by use of libraries containing model classes and through inheritance. Dymola also supports the new object-oriented modeling language Modelica. All modeling languages represent the model dynamics in non-causal form which provides greater modeling flexibility by not forcing the modeler to use predefined input/ output relationships when defining model dynamics.

Finally, systems using the third approach—bond graph modeling—include ENPORT [114], CAMP-G [20], and 20-sim [26]. These systems provide component libraries of bond graph components, and graphical user interfaces optimized to the creation of bond graph models.

1.3.3.4 Model refinement and model abstraction

Some preliminary results also exist in the areas of model refinement and model abstraction. For instance, there exist two different approaches to the generation of bond graph models. A first approach uses the system's bandwidth as a measure for the level of refinement [51, 71, 111, 129, 148, 149, 153, 154]. Starting from a simple model, additional degrees-of-freedom are introduced, increasing the frequency contents of the model, until the required bandwidth has been achieved; this corresponds to model refinement. Model abstraction, on the other hand, is used in energy-based methods. These methods analyze the power [111] and energy [71, 129, 148, 149, 153, 154] profiles of all the bonds in the model. If the energy or power level of a bond drops below a certain threshold, it is assumed that the contribution

of that bond can be neglected. By selectively removing bonds in this fashion, complex models can be abstracted to their basic dynamic characteristics.

1.3.3.5 Automated support for mathematical modeling

The last area of research directly related to the work presented in this thesis is the work in the area of automated support for the design of mechatronic systems. This area includes the work presented in [18, 19, 141, 142, 145, 146, 147]. In these works, the authors present how modeling paradigms affect the support for simulation during conceptual design. All the approaches use bond graphs as an intermediate model representation. Coming from different directions, the work presented in this thesis and the work presented in [145, 146] share several characteristics from the point of view of reconfigurable models. In [146] the author defines a similar partitioning of the models into interface and implementation and elaborates on those concepts to create a framework to support hierarchical refinement and reconfiguration. However, our work on reconfigurable models and the work presented in [146] differ in the following aspects. The work presented in [146] uses a bond graph formalism to represent the dynamics of the systems. This limits its applicability to physical systems that can be represented by a set of scalar values. As we mentioned before, bond graphs, with appropriate extensions, can be used to model systems that have vector-valued state variables; however, these extensions are cumbersome and not provided in their software environment that synthesizes the dynamic equations. On the other hand, we use a linear graph formalism, which may include vector-valued states, to represent the dynamics of a system.

Another difference between our work and the work in [146] is that although their approach modularizes the model into type and specification (interface and implementation), the specification of the model includes its type. This kind of modularization forces us to define the instance of a model to be its specification. Consequently, if multiple instances of a model need to be included in a model library, such as for instance when they may differ in parameter values or in structure, new types must be defined for each choice of specification, and each separate instance must be stored in a library of models. In contrast, we do not require the interface to be part of the implementation. This modularization allowed us to define the concept of *binding* of an implementation to an interface such that an interface describes a family of component models. In this way, only one interface needs to be stored in the

library and multiple implementations can be bound to the interface. Instances of a reconfigurable model are given by the binding of an implementation to an interface, which reduces the type redundancy presented in [146].

Finally, parameter handling is not well defined in [146]. For each new set of parameter definitions, we are required to define a new type, and the propagation of the parameters to inner submodels is not explicit. Our reconfigurable models provide an explicit mechanism for parameter propagation and do not require the definition of a new interface for each set of parameters.

1.3.4 Heterogeneous modeling

Heterogeneous modeling incorporates different modeling paradigms—such as discrete event, continuous time, finite state machines, and others—into a single simulation model. In this respect, Ptolemy II, a software environment for heterogeneous concurrent modeling developed at the University of California at Berkeley, provides the machinery to represent and combine different modeling paradigms [68]. Ptolemy II provides several modeling paradigms, including continuous time, discrete events, finite state machines, and others. The fundamental assumption taken in Ptolemy II is that all these modeling paradigms can be expressed using the block-and-arrow diagram. Under this premise, models are represented by directed graphs where the nodes are entities and the arcs are relations.

Of all modeling paradigms provided in Ptolemy II, the three modeling paradigms mentioned are the most relevant to our work. The others are targeted to modeling system behavior at different levels of abstraction. For example, the communicating sequential processes (CSP) modeling paradigm can model problems involving resource management, and the process networks modeling paradigm is targeted to modeling signal processing systems where infinite streams of data samples are incrementally transformed by a collection of processes executing in parallel.

The continuous time modeling paradigm is based on the signal-flow model. Thus, models of physical systems capture signal flow between simulation components (such as Simulink). This is in contrast to the approach to modeling physical systems in this thesis, which

is based on the observation that any two subsystems interact through energy exchange. Our port-based modeling paradigm, which is based on the conservative laws to represent energy-based systems, includes an extension to interact with signal-flow models.

The simulation engine in the continuous time modeling paradigm in Ptolemy II has many commonalities with the simulation engine developed in the early stages of this research. The reason is that the continuous time model and the modeling paradigm used in the software-based simulation environment (Chapter 2) are described by the signal-flow model. Since the signal-flow model considers every element as a function that maps its inputs to its outputs, the simulation engine provides ways of obtaining an evaluation order of the elements in a signal path. The scheduler used in Ptolemy II to achieve that ordering is similar to the scheduler developed for our simulation environment for software components (see Chapter 2).

In addition to specify the system as a signal-flow model, the continuous time modeling paradigm is designed to interoperate with other Ptolemy modeling paradigms. These are the discrete event modeling paradigm, to achieve mixed signal modeling, and the finite state machine modeling paradigm. The latter is used to describe models that are valid on well defined operating regions.

The port-based modeling paradigm also supports the mixed signal modeling paradigm. However, we rely on the target language (to which our port-based reconfigurable models are translated) for the syntactical constructs to define the events in a model (Chapter 6). In addition, our port-based models provide modeling constructs to define operating regions for which different models are valid. We call this *meta knowledge* and introduce the concept in Chapter 6.

1.3.5 Structural knowledge representation

Structural knowledge representation deals with the problem of defining appropriate representations to describe the structure of models. In Chapter 6, we define a reconfigurable model, which is a modeling paradigm that allows changes in structure as well as parameter configuration. To describe the structural knowledge embedded in a reconfigurable model

we defined a representation based on an AND-OR tree. A similar tree-based representation, called *system entity structure* (SES) [159], is a structural knowledge representation scheme that systematically organizes a family of possible structures of a system. Such a family characterizes decomposition, coupling, and taxonomic relationships among entities.

The difference between our AND-OR tree representation and the SES is that the SES captures system architecture alternatives, while the component structure describes modeling alternatives for a single component in the system. To capture system architecture alternatives, the SES defines a set of labels that specify both coupling information and selection constraints imposed on the elements of the system.

1.4 Contributions

Due to the nature of the work developed in this dissertation, we have categorized the contributions into two major areas: intellectual and implementational contributions. Intellectual contributions include new ideas, and new algorithms, while implementational contributions include new framework and representational structures. The intellectual contributions of this work include composable simulation, port-based multi-domain modeling of mechatronic systems, and reconfigurable models. The main implementational contribution is the multidisciplinary modeling and simulation environment.

1.4.1 Intellectual contributions

1.4.1.1 Composable simulation

In this thesis, we developed the idea of *composable simulation*. By composable simulation we mean the ability to generate system-level simulations automatically by simply organizing the system components in a CAD system.

A system component can be either a physical component (electrical motor, gearbox, etc.) or an information technology component (embedded controller or other software component). Each of these system components has one or more simulation models associated with it describing its dynamics in multiple energy domains, across energy domains, and possibly at multiple levels of accuracy (with varying computational requirements). When these

system components are combined into a complete system, our framework automatically combines a selection of the associated component models into a system-level simulation.

The user interaction occurs thus at the level of composition of *system components* rather than *simulation components* as in most traditional simulation environments (Matlab/Simulink, Easy5, etc.). These traditional simulation environments do not consider the mapping from system components to simulation models. This mapping is not one-to-one. The system-level simulation model is not simply a concatenation of individual component models, but may require combining multiple system components into one simulation model (to avoid algebraic loops or index problems [98, 99]). Or, conversely, it may require multiple simulation components for a single physical component (describing its behavior in multiple energy domains, for instance). Raising the level of user interaction to composition of system components rather than composition of simulation models will result in a significant reduction of effort in creating and modifying system-level simulations and will reduce the simulation and modeling expertise required of the user.

Our framework for composable simulation will therefore enable designers and control engineers to verify their physical designs and control software with much less effort and time than is required in current simulation environments, as described in the following sections.

Our concept of composable simulation is implemented using port-based modeling and reconfigurable models.

1.4.1.2 Port-based multi-domain modeling of mechatronic systems

We developed a novel modeling paradigm based on port-based objects [133]. The port-based object approach allows us to model system components by describing their behavior and their interaction with the environment. Interaction paths capture energy flow (for energy-based systems) or signal flow (for non-energy based systems). In this way, we can describe a system as a graph where nodes represent high-level system components and edges represent their interactions.

Port-based objects can be compound or primitive. Compound port-based objects define the behavior of a system as a structural arrangement of subsystems (also modeled as port-based objects), while primitive port-based objects are defined by the constitutive equations describing the behavior of the object.

The port-based modeling paradigm is the basis for our multidisciplinary modeling and simulation environment, as well as for our concept of reconfigurable models. A port-based object is transformed into a hybrid mathematical representation based on linear graphs and block diagrams. This underlying representation provides the tools required to synthesize the set of differential algebraic equations that describe the behavior of the system (Section 1.4.2.1). A reconfigurable model (Section 1.4.1.3) is an extension of the port-based objects in that the interface is separated from the implementation of the behavior of the component.

1.4.1.3 Reconfigurable models

We formalized the concept of a reconfigurable system model. Reconfigurable models are a powerful abstraction that allows the designer to change the simulation models dynamically. The modeling paradigm of reconfigurable models is based on the separation of the properties that are necessary to classify the subsystem and those that are not. Necessary and non-necessary properties are collected into two groups, called *interface* and *implementation* respectively.

Using the concept of subtyping, we organize the component interfaces into a semantic network. An important virtue of this network is that by traversing it (upward or downward) we define two operations: *refinement* and *generalization*. Reconfigurability is achieved when an implementation is bound to an interface. Therefore, this network completely defines the basic operations that are required to support reconfigurable models, namely, specialization, generalization, and reconfiguration.

We developed a component model structure to describe the modeling space of a reconfigurable component model. The structure is based on an AND-OR tree [105]: modeling alternatives are captured in the OR arcs, while individual alternative models are captured by

means of the AND arcs. Based on this structure we developed models of concrete components. These models are characterized by an induced tree on the AND-OR tree. The collection of reconfigurable models represented by this component structure are stored in a library of components.

1.4.2 Implementational contributions

1.4.2.1 Multidisciplinary modeling and simulation environment

We developed a novel multidisciplinary modeling paradigm that combines energy-based and non-energy based systems into a single modeling representation. The formalism used to represent a multidomain system is based on linear graphs [143]. We extended this formalism and created a hybrid representation for mechatronic systems. In this representation, energy-based systems are modeled using the linear graph formalism, and non-energy-based systems are modeled using block diagrams. We have combined the two formalisms into a hybrid representation that allows the description of both types of systems. New elements were defined to seamlessly interface the two formalisms.

We developed algorithms to automatically synthesize the linear graphs for all energy domains involved, including the mechanical energy domain (geometry). In this respect, the algorithms that synthesize the linear graph for the mechanical energy domain take care to simplify the graph in order to minimize the possibility of obtaining both high-index algebraic differential equations and fully constrained mechanisms by removing redundant kinematic joints (i.e., joints that have been identified to have coincident joint axes).

We formalized the causality problem as that of finding a minimum cost spanning tree on the linear graph. This provided a convenient way for finding causal directions for all the equations in the system. To incorporate the equations derived from the non-conservative system, we defined an extension of the classic *Block Lower Triangular* algorithm to find a feasible order of evaluation of the DAEs.

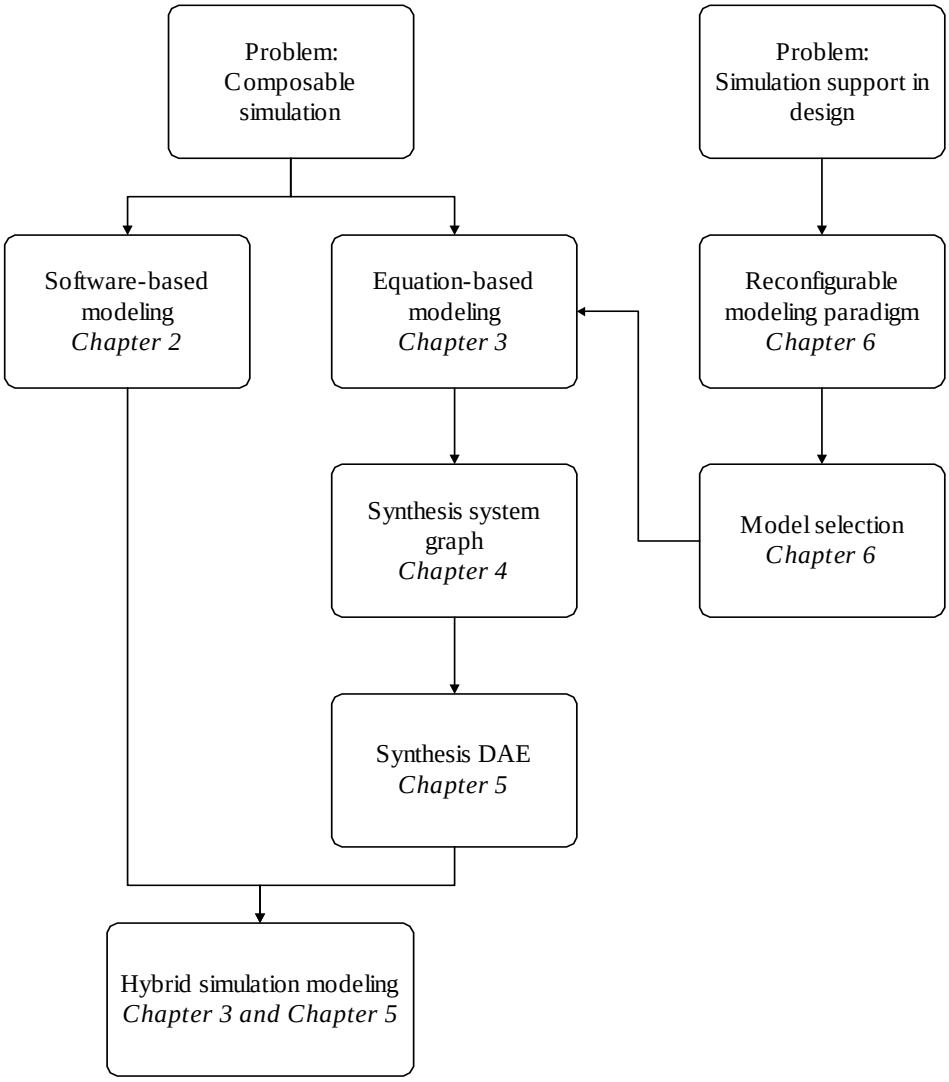


Figure 1-3. Research road map.

1.5 Research road map

Two problems are addressed in this research: composable simulation and simulation support for the design process. Although we addressed each problem independently, in the end the two approaches merge at a common point, as is illustrated in Figure 1-3. Initially, we implemented a purely software-based composition environment. Due to the nature of the underlying equations, this approach rapidly reached its limits. Nevertheless, the lessons learned from this attempt are valuable. The most valuable lesson learned was the under-

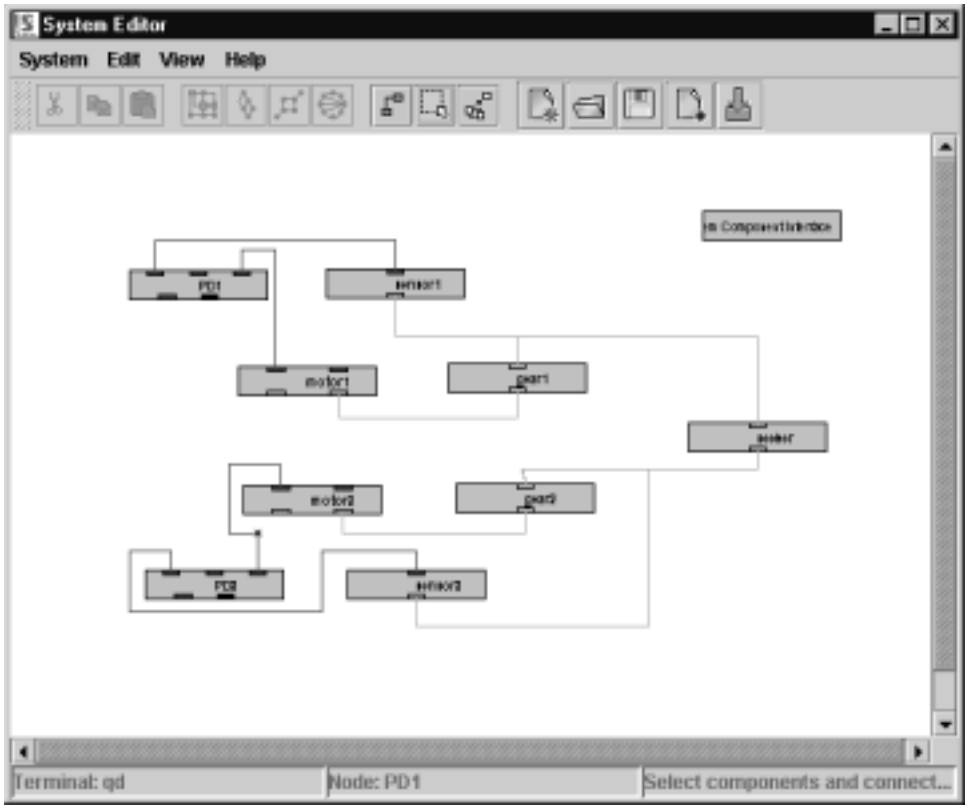


Figure 1-4. High-level description of a mechatronic system.

standing of the interactions between components and, specifically, how these components may interact in a hybrid simulation environment, one which includes a mixture of equation-based models and software components.

Based on the limitations we encountered with the previous approach, we focused on equation-based modeling. In this modeling paradigm, the models are given as equations rather than as procedures as is the case with a software component. A new modeling paradigm, namely port-based modeling, was developed. This modeling paradigm does not define a fixed causality for the equations involved but leaves the task of determining it to the simulation environment once the complete topology of the system is known. We adopted the method of the system graph to represent the underlying topology of the system based on energy flow. This system graph is derived from a high-level component graph composed of port-based object models as illustrated in Figure 1-4. In this graph, the high-level system topology is defined in terms of high-level system components and interactions. This graph is used to synthesize the graph of the system.

Once the system graph was readily available, we used it to derive the differential-algebraic equations that describe the behavior of the system, including references to software components.

To address the issue of providing simulation support to design tasks, we focused our attention on extending the port-based modeling paradigm to provide model selection capabilities as well as to provide the ability to dynamically reconfigure the model by means of reconfigurable models. Reconfigurable models are translated into the linear graph representation. At this point, the two paths leading from the two general problems shown in Figure 1-3 are merged. In this case, once a reconfigurable model is translated into a linear graph, the methods derived in this thesis are applied to synthesize the set of differential-algebraic equations that define the behavior of the system. This approach to modeling allows the designer to test different candidate designs.

We illustrate the use of our framework with a design scenario presented in Chapter 7.

Chapter 2 Composition of Simulation Software

2.1 Introduction

In this chapter, we describe a simulation framework aimed at providing simulation support for non-energy-based systems. In this framework, simulations can be composed by defining interactions between component models that represent signal flow between models. The framework proposes a new modeling approach based on the theory of software architectures [6, 74] and on port-based objects [133], in which a component is viewed as an entity with an interface and an implementation. The interface defines the interaction points of the component with its environment, while the implementation defines the behavior of the component. This approach to modeling forms the basic ideas for our port-based modeling paradigm and the reconfigurable models presented later in this thesis.

One of the goals of composable simulation is to support the rapid assembly of simulation programs. In this respect, we developed a software framework based on a description of a software component called *subsystem interface* (sbs). The subsystem interface provides a

low-level uniform definition of a simulation software module; that is, it provides the application programming interface (API) required to embed software components in the simulation environment. Software modules are described in the environment by a model description language that captures the hierarchical definition of a module in terms of ports and composition of other submodules.

From the experimental results, we observed two problems. The first is the occurrence of algebraic loops. In this case, we have opted for their detection; however, no effort is made to try to solve them. Algebraic loops in two or more software components imply tight coupling between the underlying equations. To solve this kind of system, we would need to use a nonlinear solver or an iterative algorithm to break the algebraic loop (i.e., tearing) [42, 44]. Non-linear solvers require knowledge of the derivatives of the software component, which in general are not available. These methods are very costly and they do not always converge to a solution.

The second problem we observed was related to the granularity of the software components. We found that the granularity of the software components directly affects the performance of computing a solution for the system. This problem is closely related to the previous one because we can improve the efficiency of the computation of a solution by including the equations of the software components that create the algebraic loop into a single component. In this way, we are reducing the granularity of the components, and hence we can implement an efficient nonlinear solver that solves the set of simultaneous equations. This can be done if the equations of software components involved in the algebraic loop are combined, reducing them to a single software component.

2.2 System architecture

In this architecture, the representation of a component model consists of two parts: the interface and the implementation. The interface represents a portion of a design that has well-defined interaction points and performs a well-defined function. The implementation of the model represents the behavior of the component and is described by a software module. The module can be described as the structural arrangement of interfaces or as a single

procedural module. The structural arrangement of submodules defines the configuration of the component.

To support composability of simulation software modules, we propose a software architecture based on three abstraction levels [31]: *conceptual level*, *component level*, and *process level*. The highest level of abstraction in the architecture (*conceptual level*) defines the system using a *conceptual graph*. A conceptual graph is a directed graph in which nodes represent components of the system and edges represent their interactions. Edges leaving from a node represent output ports of the subsystem; incoming edges to a node represent input ports.

The second level of abstraction of the architecture is the *component level*. This level provides an abstract representation of the behavior of the component, which is described by an *interface* that includes the input and output ports of the component. At this level, the conceptual graph is mapped into a simulation software architecture. The resulting architecture is the main representation of the semantics of the system; i.e., a network of interfaces and interactions among components. The third level of abstraction in the architecture, the *process level*, provides the algorithmic representation of the behavior of the component. At this level, the software architecture is instantiated and executed. Instantiating an architecture requires finding processes that match the features listed in the interface of each component. Execution of the architecture may require calls to external libraries such as Matlab, ACIS or Odepack/Linpack. The process level deals with the communication and synchronization of processes included in the architecture and ensures data consistency. Based on these abstraction levels, component models can be hierarchically composed. Hierarchical composition allows the composition of component models into a compound model; this compound model captures the structural arrangement of the models.

To illustrate these concepts, consider the system shown in Figure 2-1. A control system is defined by a conceptual graph. The model of this system is hierachically built. This is shown by expanding *actuator* component, which is composed by three subcomponents: *amplifier*, *motor*, and *coupling*. The description of these subsystems are given at the conceptual level. The component level defines the interfaces of each of the components that

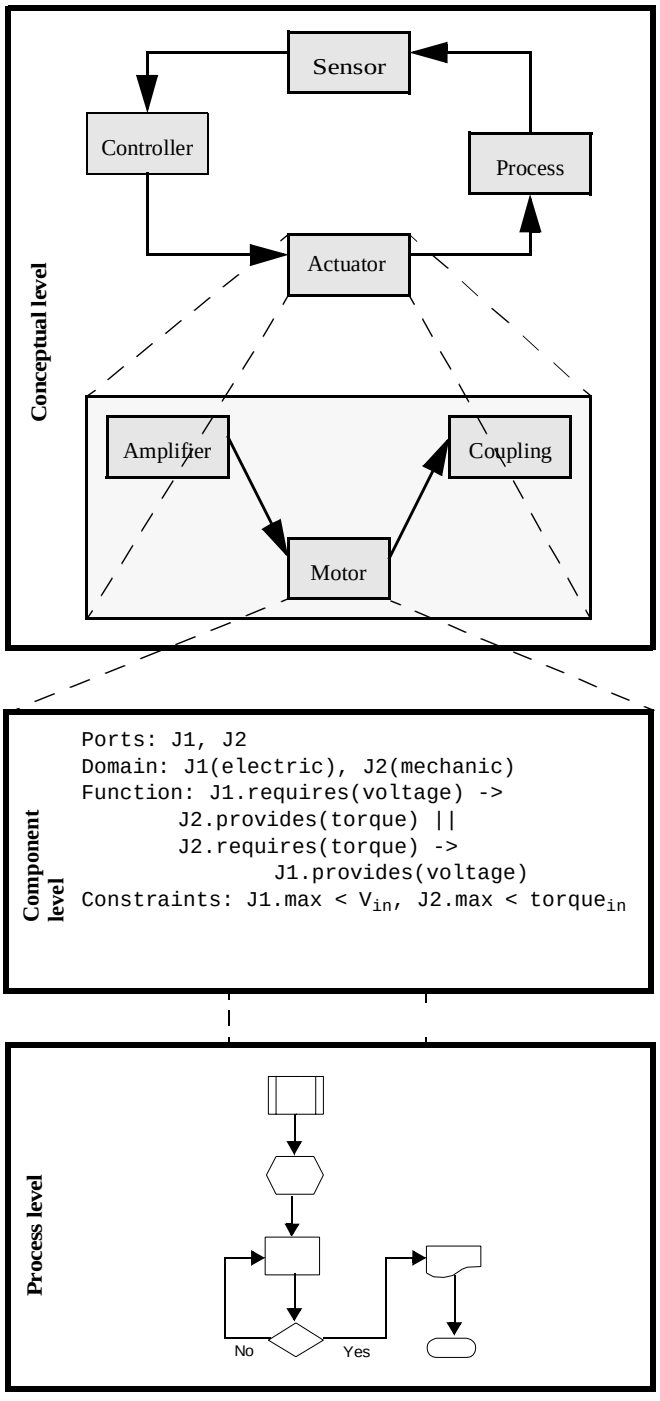


Figure 2-1. Model abstraction levels

compose the design. In this case, the interface of the DC motor is shown in the component level. Every interface in the component level needs to be instantiated. This process is achieved by associating an implementation of the behavior of the component to the inter-

Figure 2-2. Component

face. The behavior of the component is described in the process level, which gives an algorithmic interpretation of the desired behavior.

2.2.1 Component model

A *component* is the generalization of a port-based object [132, 133, 134] and it consists of two separate parts: an interface through which it interacts with other design entities, as illustrated in Figure 2-2, and an implementation that either encapsulates an executable prototype of the behavior of the component, or hierarchically defines it as a configuration of other components.

The interface defines the information that completely characterizes the software component, such as input/output ports and their corresponding energy domains; the functions the interface provides to other design entities or the functions it requires from other design entities; and constraints on behavior of the component.

For instance, the interface definition for a DC motor/generator component might look like this:

```
Ports       : J1, J2
Domain      : J1(electric), J2(rotational)
Functions   : J1.requires(current) -> J2.provides(torque) ||
              J2.requires(torque) -> J1.provides(current)
Constraints : J1.max < Im, J2.max < Torquem
```

The interface of a component may be satisfied by more than one implementation. For example, the interface definition for the mechanical system of a missile seeker may specify as input ports the torques applied to the system, and as output ports the angular accelerations of the system produced by the given torques (Figure 2-3). There are two alternatives to choose from in selecting the implementation for this interface: one using the Newton-Euler iterations, the other using the Lagrange-Euler method. Both implementations con-

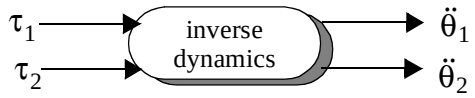


Figure 2-3. Inverse dynamics component

form to the interface but provide different mechanisms to compute the desired results. This feature can be used in many ways; for example, to dynamically reconfigure the simulation, to produce finer simulation results, or to test different implementation approaches.

A configuration is recursively defined to be composed of either sub-configurations or components or both. This definition supports the hierarchical nature of a mechatronic system. A configuration, like a component, consists of two parts: interface and implementation. The interface of a configuration will export only those features visible at the sub-system level. The implementation of the configuration will be defined by the network of sub-configurations and design entities described in the definition of the configuration. Since the structural arrangement of components defines a configuration, a configuration is legal only if every input port is connected to one output port and all the communication constraints are satisfied. An output port may be connected to multiple input ports, but an input port may only be connected to a single output port.

An instance of a configuration of a system is created when all the interfaces in the configuration are assigned conforming implementations [73]. An implementation conforms to an interface if it contains all features specified by the interface. In [73], the authors define three conformance criteria that we adopt to define semantically correct systems:

Interface conformance: Each implementation in the system must conform to its interface. This means that the implementation has to match the interface definition semantically; otherwise, the implementation cannot be used in the given context.

Decomposition: Each particular instance of a configuration is decomposed in a number of implementations; these implementations must conform to the interfaces of the configuration. This means that for each interface there must be at least one conform implementation.

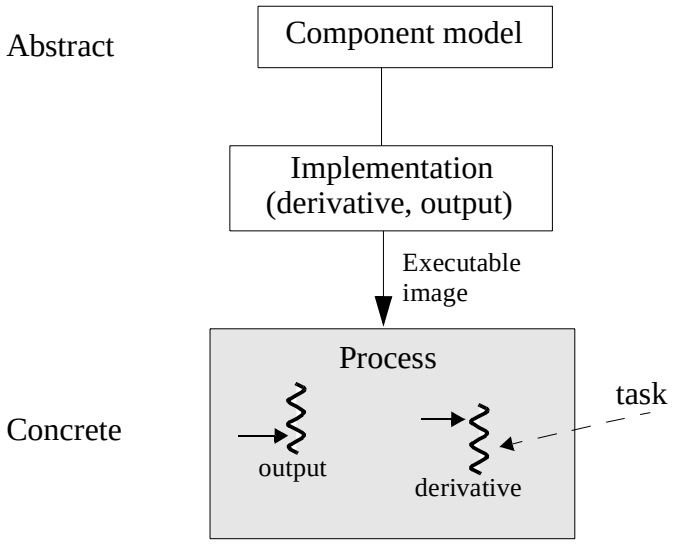


Figure 2-4. Executable abstraction of a component

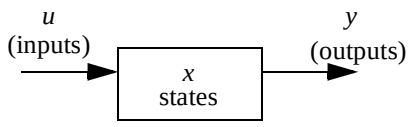
Communication integrity: The system's components interact only as specified by the configuration.

Components in the implementation of a configuration have a well-defined scope. This means that messages sent locally in a configuration cannot reach components outside the boundaries of that configuration; only those features indicated in the interface are exported and therefore can be used by other configurations.

Since an interface may have more than one conforming implementation, it is valid to replace the complete network attached to the implementation of a configuration with a different network or with a single component. The new implementation may capture the behavior of the component at any level of detail, as long as it maintains the basic functionality specified by its interface.

2.3 Scheduling the execution of component models

Component models are executed when their implementation is loaded into the system. However, implementations cannot run at just any time; rather, they must follow a predefined pattern of execution. This results in a scheduling problem. To devise a solution to



$$\dot{x} = f_d(t, x, u)$$

$$y = f_o(t, x, u)$$

Figure 2-5. Generic template for an implementation

this problem, we propose a conceptual organization of the component that reflects transformation from an abstract concept to an actual executable as illustrated in Figure 2-4.

In this model, the implementation of a component provides two operators, the *derivative operator* and the *output operator* as illustrated in Figure 2-5. The derivative operator computes the derivatives of the state variables, while the output operator computes the output variables of the component. Once an implementation is loaded into the system, a *process* which represents the executable image of the implementation is created. Since each process has two operators, we associate the execution of each operator within a process to a *task*, or a thread of execution of an implementation's operator (represented by the curly line in Figure 2-4).

By explicitly specifying the derivative and output operators, the scheduling problem is reduced to finding the schedule for the output tasks without considering the derivative tasks. Once the output tasks have been evaluated, the input variables to each component are ready and the derivative tasks can be evaluated in any order.

To obtain a correct schedule of the tasks spawned by an implementation, we first classify the component according to its output operator, and then present a scheduling algorithm based on a constraint graph that captures the interconnections between components.

2.3.1 Classification of component models

The implementation of a component is an abstract representation of its behavior. In general, the models are described by a set of differential algebraic equations that represent the dynamics of the system. These equations contain the following information:

- A set of continuous variables denoted x.
- A set of equations depending on x.
- A set of assignments where variables in x are assigned to continuous functions: $x\colon\ x_i = g_i(w)$.

In traditional simulation environments or even in object-oriented simulation environments, the model descriptions are transformed into a single block of equations. However, before the equations can be solved, they undergo a series of transformations that put them in suitable form to be used in numerical algorithms. The goal here is to find an ordering of the set of equations for which a solution can be found. This ordering produces a correct model with the following elements:

- dynamic state variables x and their first derivatives $\dot{x}$
- a set of algebraic state variables z
- a set of auxiliary variables v
- a set of dynamic equations sorted in computational order.

In general, we can write the equations that result from the above transformations as follows:

$$\begin{aligned} \dot{x}_i &= f_i^d(x, \dot{x}, z, v, u) \\ v_i &= g_i(x, \dot{x}, z, v) \\ y_i &= f_i^o(x, \dot{x}, z, v, u) \end{aligned}$$

Equation 2-1

where u is the vector of inputs to the component model and y is the vector of outputs of the component.

In Equation 2-1 some equations are explicit assignments to state derivatives, while other are assignments to algebraic variables. Based on the form of the output equations y_i we classify a component model as follows: if the output variables y_i are explicit functions of the input vector u, we call it a *direct feed-through* component. On the other hand, if the output variables y_i are not an explicit function of the input vector u, we call it an *integral* component. If at least one output variable is an explicit function of the inputs, the compo-

nent would belong to the direct feed-through class. This classification influences the computation order established for a configuration since that is dependent on the execution order of the internal components, which is in turn dependent on their classification.

2.3.2 Task scheduling

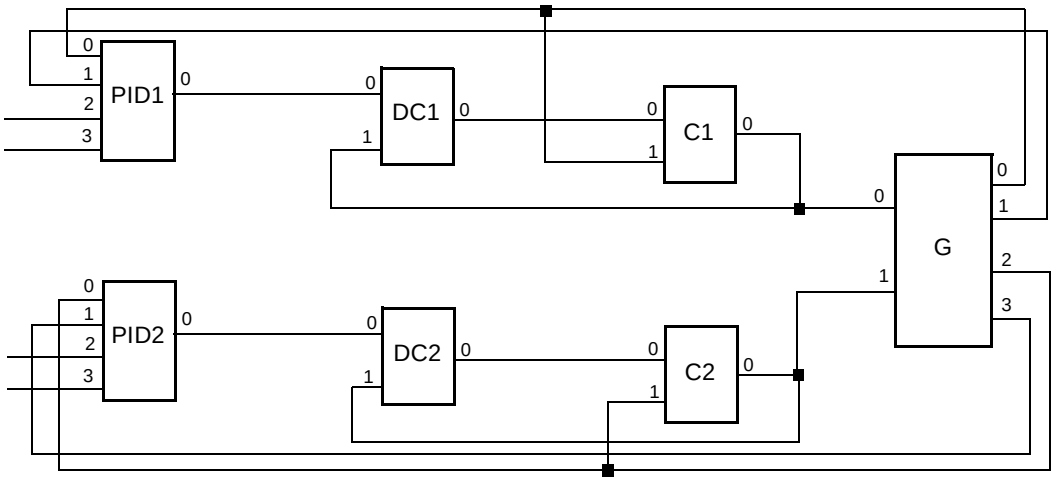
Tasks of a configuration are organized in a constraint graph. A constraint graph is a bipartite graph in which the nodes are divided into two sets: one representing port variables and the other representing the tasks to be scheduled. Edges in the constraint graph connect nodes of opposite sets, and there are no edges connecting nodes within the same set. Edges in the constraint graph are directed according to the precedence relationships derived from the connections given in the configuration.

For a configuration with n design entities, the bipartite graph will include $2n$ task nodes; i.e., one node for each task spawned by the implementation. Once the constraint graph is defined, the schedule is found by topological sorting [62] the nodes in the graph.

To illustrate these concepts, consider the description of a missile seeker shown in Figure 2-6. This configuration consists of the following software components:

1. a gimbal mechanism,
2. coupling elements for pitch and yaw
3. actuators (DC motors) for pitch and yaw, and
4. PID controllers

The connections in the configuration have a defined directionality. That is, the connection of output port-0 of module C1 with input port-0 of module G defines a flow of information from C1 to G.



G: gimbal
C_i: coupling elements
DC_i: actuators
PID_i: PID controllers

Figure 2-6. Configuration of the missile seeker

Each module in the configuration is classified as follows:

Table 2-1. Classification of modules in the missile seeker

Module	**Class**
Gimbal (G)	integral
Couplers (C1, C2)	feed-through
DC motors (DC1, DC2)	integral
PID controllers (PID1, PID2)	feed-through

Given the information provided by the configuration and following the classification of the software modules given in Table 2-1, the constraint graph is constructed as illustrated in Figure 2-7. A special characteristic of this graph is that every task has a port variable as a successor and as its predecessor. This means that if a variable is the predecessor of a task, the task's outputs are explicit functions of its inputs. Successor port variables are the result of the output operation of the component.

The order in which the output and derivative operations are executed is given by a partial ordering on the constraint graph. This partial order, as we already pointed out, is obtained

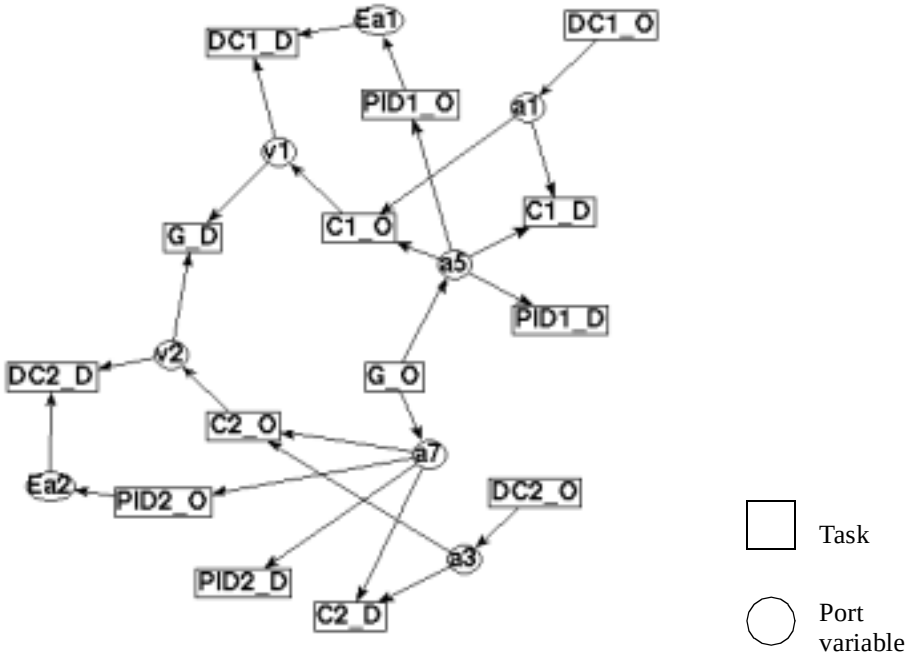


Figure 2-7. Constraint graph of the missile seeker

by topological sorting of the constraint graph. The result is shown in Figure 2-8. Since the gimbal and DC motor modules are classified as integral, their outputs do not depend on their inputs. Thus we can schedule the output operations to be executed immediately since all the information they need is already available. The execution of the output operators produces the port variables a_5, a_7, a_1, a_3. These variables are inputs to the PID controllers and to the couplers. Since we have classified them as feed-through modules, their output operators make explicit use of their inputs. In addition, from the constraint graph, we see that the derivative operators also use the same variables. Since all the information required by these modules is available at their inputs, they are assigned the next place in the schedule. The execution of these modules generates the port variables E_{a_1}, v_1, v_2. The inputs to the derivative operators of the gimbal and DC motors are ready; therefore, the algorithm schedules them next, resulting in a correct execution schedule.

The schedule generated by the algorithm has an interesting property. We note that the levels in the tree in Figure 2-8 are grouped by node classes; i.e., even-numbered levels include tasks only while odd-numbered levels include port variables only. This property can be exploited to implement a parallel execution of all tasks in even-numbered levels. For exam-

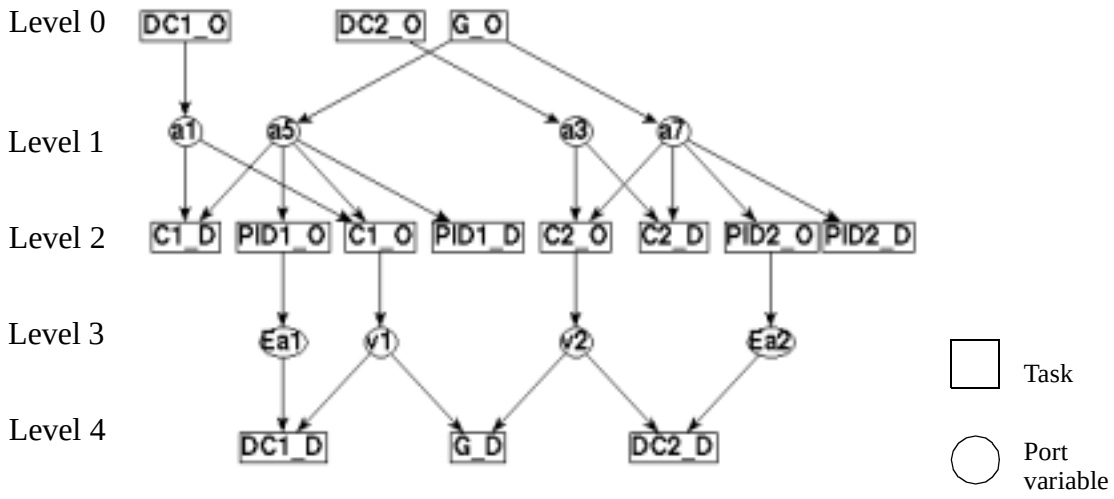


Figure 2-8. Topological order of the software components for the seeker

ple, all tasks in level 0 (DC1_O, DC2_O, G_O) can be executed in parallel. Once the tasks in this level are completed, tasks at level 2 can be executed in parallel and so on. Another execution schema that can be obtained by exploiting this property is that instead of waiting for all tasks to finish in an even numbered level, as soon as one task finishes we may start executing tasks at higher levels provided all the port variables are defined. These methods of execution can greatly improve the speed of a simulation by taking advantage of the computing power of networked computers.

2.4 Simulation kernel

The simulation kernel controls the simulation by executing the schedule computed for a given configuration. It implements the solver of ordinary differential equations that keeps track of the simulation time and integrates the system of equations given implicitly in the software components. Two different approaches were taken to implement the simulation kernel.

The first approach, based on CORBA, was a distributed simulation kernel. The ability to run simulations in a distributed environment allows one to take advantage of the computational power of networked computers. However, to reduce communications overhead, care

should be taken to avoid unnecessary fragmentation of the simulation. Only large simulation components warrant execution on a separate workstation.

Drawing from the experience obtained in the development of the CORBA-based simulation kernel, and to virtually eliminate the problem of communications overhead, we developed a second simulation kernel. This kernel utilizes the multithreading capabilities of the host operating system to run the simulation components in separate threads. If in addition the simulation kernel is run in a multi-processor computer, each thread can be run in a separate processor, which minimizes the computation time of the overall simulation.

Several issues must be resolved to efficiently execute the composition of simulation components in the framework; namely, development of inter-process communication mechanisms, and integration of numerical integration algorithms in the simulation kernel. We will address these issues in the following sections.

2.4.1 Kernel object model

The first step in defining the simulation kernel is to define the underlying abstractions that will capture the description of configurations in the framework. For this purpose, we present an object model that describes the object-oriented architecture of the simulation kernel.

The object model [115, 116], shown in Figure 2-9, defines the *module* as the fundamental abstraction. A module can be a primitive module or a compound module (configuration). Primitive modules cannot be decomposed into submodules, and for that reason they form the basic building blocks of the system. Compound modules (configurations) are aggregations of primitive modules and/or other compound models.

Configurations and primitive objects have two objects in common: the *ovar* object and the *interface* object. The *ovar* object implements the inter-process communication mechanism between objects in the system. The *interface* object contains the port definitions as well as properties relevant for the software component such as the direct feed-through and number of internal states.

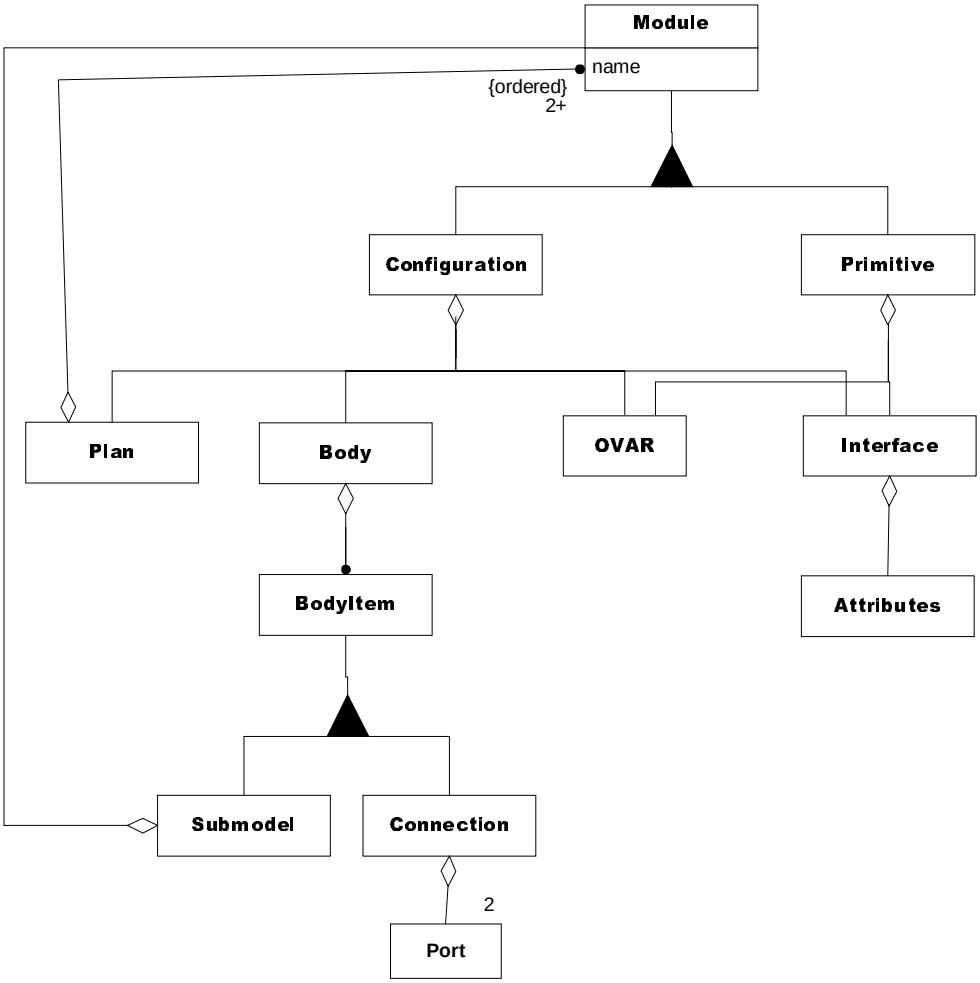


Figure 2-9. Object model.

In the object model, the *interface* object is shared between a configuration and a primitive because when a configuration is used in a larger configuration it is considered as if it were a single software component. The simulation kernel takes care of the scheduling of all internal software components so that when the output operator of the module is executed in the parent configuration, all internal operators are executed before the outputs of this operator are presented to the environment. That is, the parent configuration will expect to see the component's outputs after its execution elapses. The number of internal states is used to determine the number of elements in the *ovar* object.

Two more objects compose a *configuration* object: *plan* and *body* objects. The *plan* object represents the evaluation schedule of the components in the configuration. It represents the

topological sort of the components in the *configuration* based on the *direct feed-through* field and the directionality of the ports. The *body* object allows the hierarchical definition of a *configuration* by means of the *submodel* object, and defines the topology of the *configuration* by means of the *connection* object. In Figure 2-9 a submodel object is an aggregation of modules. This allows a configuration to be hierarchically defined in terms of primitives and their configurations. Finally, the *connection* object is an aggregation of exactly two ports.

2.4.2 Inter-process communication

Inter-process communication is achieved through the *ovar* object. This object implements *shared memory* schema where *modules* are assigned different memory segments from which they can access their data. The shared memory segment is divided up into virtual segments each having a well-defined lexical scope. The lexical scope of a memory segment spans the module in which it is defined. In other words, a *module* cannot write out values outside its lexical scope.

Conceptually this is shown in Figure 2-10. The *ovar* object contains two segments, namely, the *input segment* and the *output segment*. The output segment is shared among all modules in the configuration. This results in a single memory segment used in the topmost configuration, which is the one that implements the shared memory. The input segment in each *ovar* object defines a mapping between inputs and output ports of different components. This can be seen from the observation that for each input to a module there will be a corresponding output of another module. Storing the indices of the output array in the input segment (for those outputs connected to the inputs) completely defines the inputs to the module.

To illustrate this, consider the configuration shown in Figure 2-11. In this configuration, modules B and C are the outputs of the top level configuration which has no inputs while modules A and D have no inputs.To access the second input to module C, we access its input segment to find that it contains the index 8 in the entry number 2. This index points to the global shared output memory segment which in the address 8 contains the input to

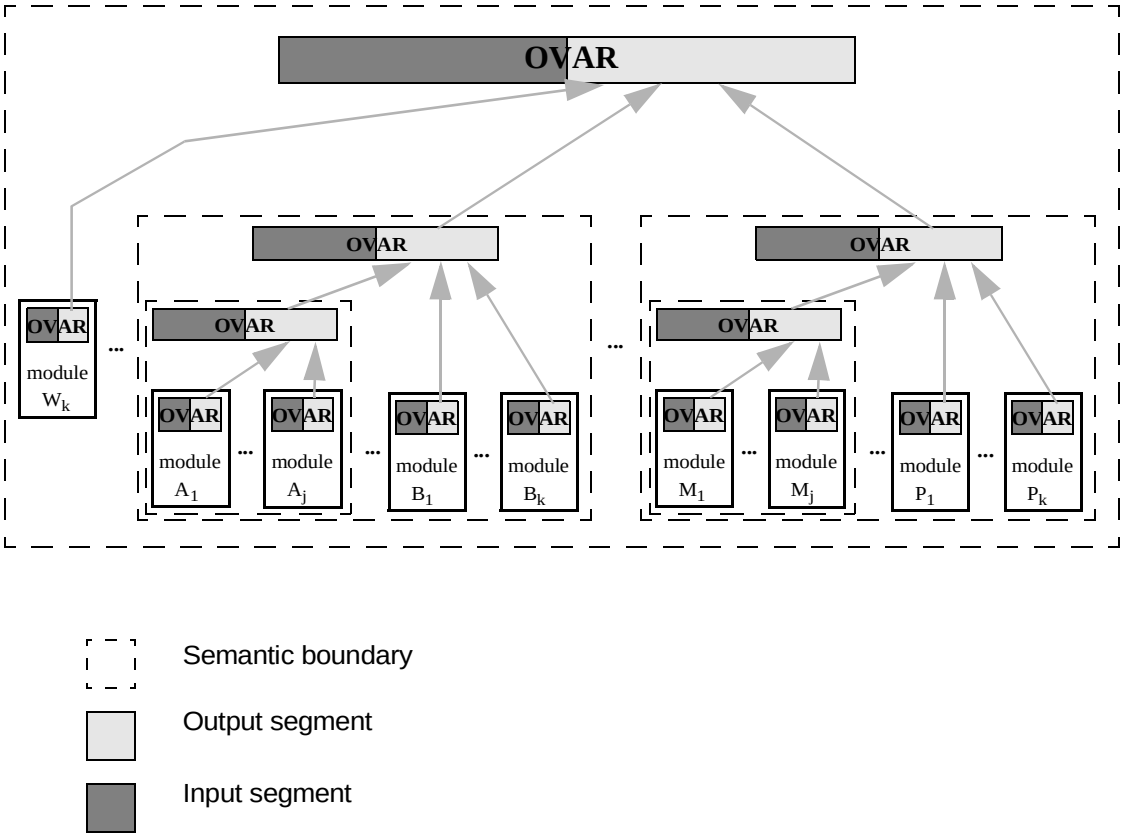
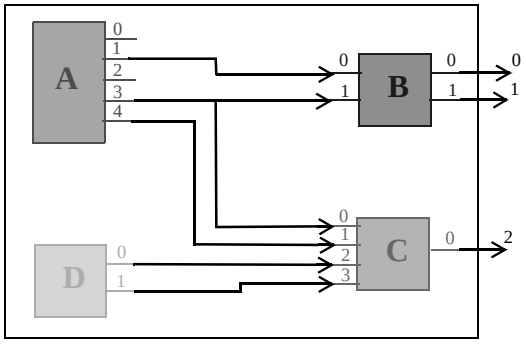


Figure 2-10. Inter-process communication architecture.

module C coming from module D. In this communication model, the hierarchical structure of the configuration is flattened. This, however, does not prevent us from hierarchically defining the configuration. It is only the communication model which is required to be flat to improve performance.

The final issue to address is data consistency. Recall that the schedule first computes the outputs of all non-algebraic modules. This produces a partial set of updated output variables at the current time. Since the schedule guarantees that no algebraic module will be executed before its inputs are ready, it is not possible to compute an output from an outdated input. Therefore, we can conclude that the schedule guarantees that the modules will work with consistent information throughout the simulation run.



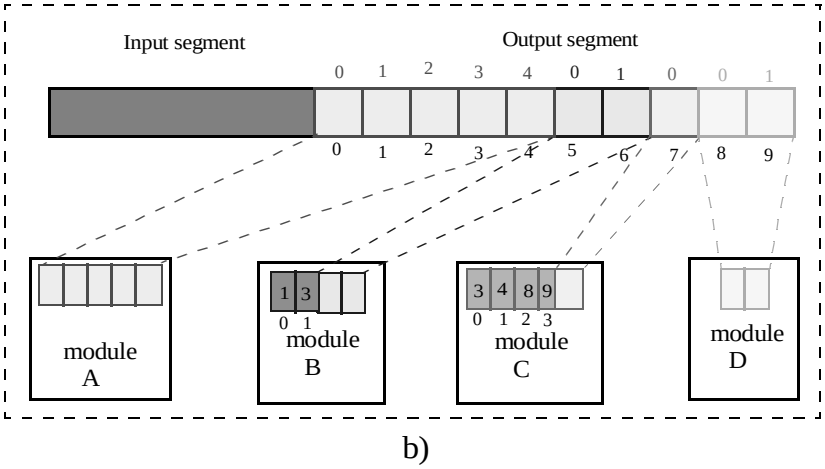


Figure 2-11. Shared memory schema.

2.4.3 Task execution model

Before the kernel can execute a task, it has to be properly initialized. In order to accomplish this, every task has an associated state which indicates its status in the simulation process.

2.4.3.1 Tasks states

We associate each task with a context. The context of a task specifies its current state, which can be one of the following: *running*, *ready, suspended*, *dormant* (Figure 2-12). A running task is the one that is actually executing. A task can enter the running state from a *ready* state upon receiving a *cycle* signal. The task will run until it finishes its computation, after which it returns to the *ready* state. Tasks in the *ready* state are those which are ready to run but not running. A task enters the *ready* state if it was executing and its cycle ended.

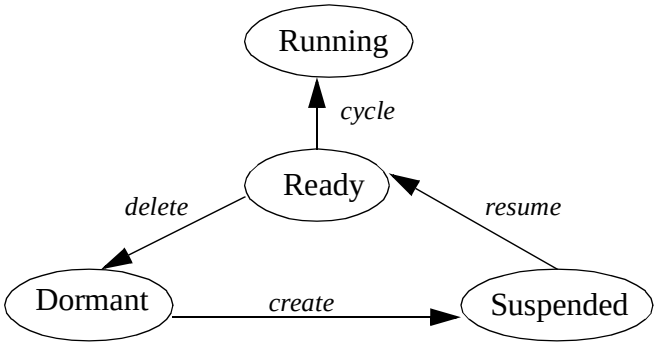


Figure 2-12. Task states.

If it was in the *suspended* state, then it can enter the *ready* state if an event that initializes it occurs. Tasks that are created but not scheduled to run are put in the *suspended* state. Finally, the *dormant* state describes a task that exists but is unavailable to the simulation kernel.

As indicated in Figure 2-12, a task can change states as a result of any of the signals *create, cycle, delete,* and *resume*. When a task receives a signal, it reacts by executing one of its services, which are part of the low-level implementation of the task (Figure 2-13).

The signals *create* and *delete* are special in that they are handled by the host operating system and not by the module. When a task receives a *create* signal, it is the operating system of the host machine that provides the functionality to load the task. When the simulation has ended, the tasks loaded into the kernel are released along with all the resources allocated to them (i.e., memory). This occurs when the tasks receive a *delete* signal from the operating system. After the delete signal is received, the task goes to the *dormant* state.

The two other signals, *cycle* and *resume,* generate events that the module has to handle. The handlers for these events are implemented as two services: *init* and *cycle*. The *init* service initializes the module allocating the resources it will use as indicated in the *interface* object. The *cycle* service performs the evaluation of the output and derivative tasks of the process, which communicate with other processes in the kernel through the *ovar* object. The numerical integration algorithm running in the kernel, as indicated in the following section, controls this service.

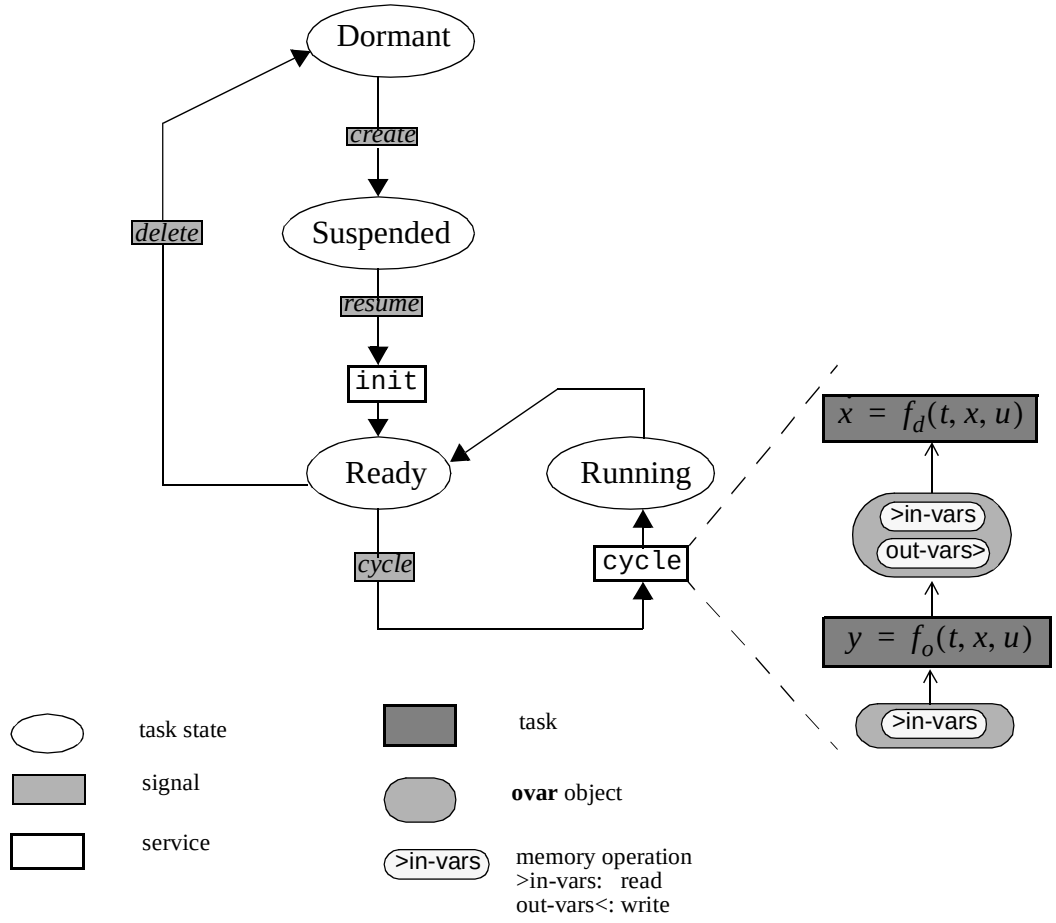


Figure 2-13. Signal diagram.

2.4.3.2 Numerical integration

This section describes the interaction between the numerical integration algorithms and the task-scheduling algorithm presented in Section 2.3.

Consider the following system of equations:

$$\dot{y} = f(t, y) \qquad \textbf{Equation 2-2}$$

Assume y is a vector where y_i is the *i-th* state variable. Furthermore, consider also the second order Runge-Kutta formulas:

$$k_1 = hf(t_n, y_n)$$

$$k_2 = hf\left(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1\right) \qquad \textbf{Equation 2-3}$$

$$y_{n+1} = y_n + k_2 + O(h^3)$$

where the constant vectors k_1 and k_2 are of same dimension as vector y.

The integration formulas evaluate the derivative functions at different points as indicated in Equation 2-3. Every evaluation of f must correspond to the evaluation of the schedule for the configuration since the integrator sees the configuration as a single function. Based on this, the numerical integration method implemented in the kernel first evaluates the output functions and then the derivative functions, as prescribed in the schedule, for every major and minor integration step (Figure 2-14).

2.5 Module definition language

We defined a high-level language to describe component models and configurations. In the design of our language, we need to choose a set of high-level concepts from the domains we are dealing with, give them a suitable syntactic form, and give them precise meaning in terms of some underlying mechanism. Such a language must provide notations and tools to describe component models and their interactions to form complete configurations. It must handle large-scale, high-level interfaces, and it must support the adaptation of these interfaces to specific implementations. Three properties characterize what an ideal module definition language (MDL) should provide: composition, reusability, and analysis [122].

Composition: Dictates that it should be possible to describe a system as an aggregation of design entities and their connections. This allows design entities to be combined into larger systems.

Reusability: It should be possible to reuse design entities in different system descriptions.

Analysis: It should be possible to perform a rich analysis of system descriptions. This is equivalent to verifying the correctness of the connections in the system.

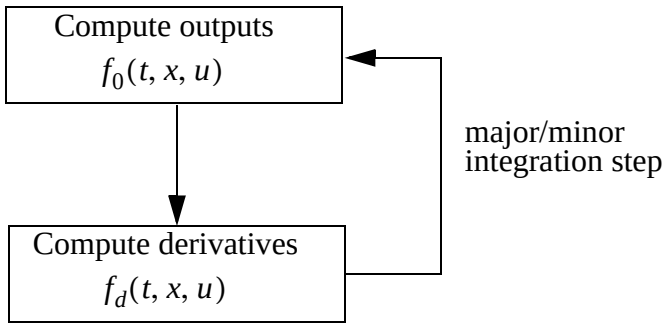


Figure 2-14. Flowchart of a single integration step

Even though the language was intended to implement the complete functionality of the component, it presents some limitations on the descriptive power of the component. In this language, the types of ports are all assumed real valued quantities. No effort was made to associate energy domains to the ports as indicated before. In addition, an interface has only one implementation. Binding the binary code that executes the implementation to the interface description creates the default match between the implementation and the interface. The language was not developed further because in our experiments we observed that the approach taken was not the best suited for the kind of problems we were interested in solving. In the rest of this thesis, we present a modeling paradigm that will overcome these difficulties.

2.5.1 Module

A module is an abstract definition of the behavior of a family of components. It defines the logical points of interconnection between the component and its environment. Any number of module instances can be obtained from a single module description. A module can have any number of attributes of the following types:

- other module definitions called component modules
- connections
- ports
- initial values on the port variables

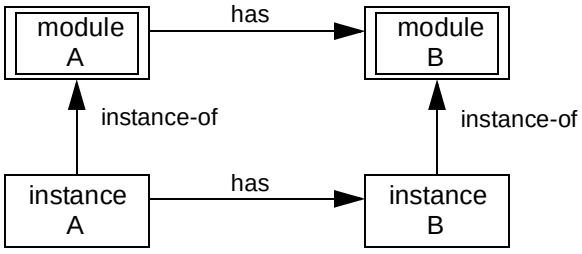


Figure 2-15. Module instantiation

A module is defined by the following grammatical construct:

```
module_def    ::=  module identifier module_qualifier
                   module_body endmodule ;

module_body   ::=  interface_def |
                   body_def |
                   interface_def body_def

body_def      ::=  submodules body_decl ;
                   connections body_connections ;
                   initialization body_initializations ;
```

The module's body has two parts: the definition of the module interface and the definition of the module's body which includes submodules, connections and initializations. The non-terminal symbol *module_qualifier* in the module definition rule serves the purpose of identifying the module as defining a configuration or a primitive component. In this respect, it is semantically incorrect to define a module body for a primitive module because it is assumed that a primitive has no other information except its interface. The primitive module must be initialized in the configuration that is instantiating the module.

Hierarchical structures are constructed by local module definitions through the *submodule* construct. The embedding of a module into a configuration carries the following semantics: a module *A* with an embedded module *B* implies that every instance of *A* has an instance of *B* (Figure 2-15).

This instantiation model meets the reusability property required in the definition of the MDL. Since we can instantiate a module a number of times, given different module param-

eters, the instantiation process will take care of the configuration of the module according to the definition given in the configuration containing the module.

2.5.2 Interface

The interface of a module defines the communication channels between the module and its environment. It is defined by the following grammatical rule:

```
interface_def ::=  interface interface_constituent ;
```

An interface block contains a number of declarations that specify the number of input and output ports the length of the state vector. In addition, it identifies the module as integral or direct feed-through.

Ports are the logical point of interaction between the component and the environment. A port has a defined causality identified by two keywords: *inputs* and *outputs*. An input causality value means that the environment in which the module is embedded computes the value of the port. An output causality value means that the module owning the port computes the value of the port. To illustrate these modeling constructs, consider the following fragment.

Listing 2-6. The definition of a primitive module

```
module gimbal is
     interface
          declare(inputs, 2);
          declare(outputs, 4);
          declare(states, 4);
          declare(dft, false);
endmodule;
```

In the example, the module gimbal is defined. This definition declares that the module gimbal has two inputs, four output ports and four states. Since the output operator of the gimbal module does not depend on its inputs, the module is classified as integral (i.e., has a direct feed-through value of false).

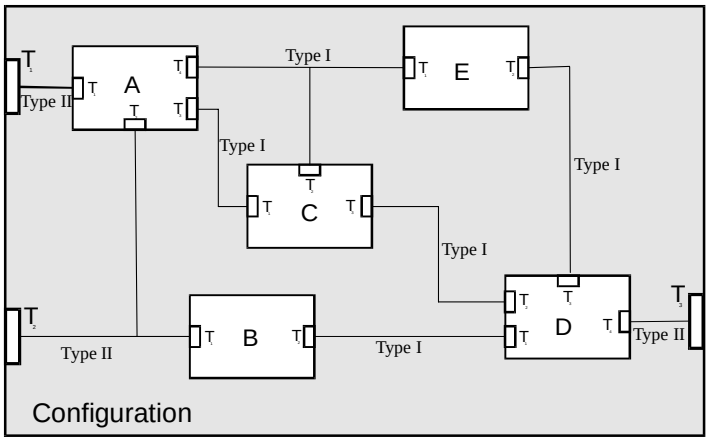


Figure 2-16. Valid connection schemes

2.6.3 Connections

A connection defines a symmetric relation between two ports. A connection is semantically correct only if the involved ports have identical structures and if they are connectable: two ports are connectable if their causality is consistent.

Not all connection schemes are valid. We can classify connections in two types: type I connections, which involve ports owned by modules that belong to the same configuration; type II connections, which involve a port owned by a module embedded in a configuration and a port owned by the configuration (Figure 2-16).

Given this classification, the semantic meaning of a connection can be interpreted as follows:

Table 2-2. Semantic meaning of connections in a configuration

Ports		**Type I**	**Type II**
x<in>	y<in>	—	y := x
x<in>	y<out>	x := y	—
x<out>	y<in>	y := x	—
x<out>	y<out>	—	x := y

A connection implicitly defines a constraint equation between the variables associated with the two ports.

2.6.4 Initializations

The initialization block of a module serves the purpose of providing the parameters relevant to the execution of the submodules. This results in code reusability; only one instance of the executable code is loaded into the kernel, but it is used with different parameters. An initialization is defined using the following rule:

```
body_init_decl::=  setstate ( init_statement_args ) |
                   setparam ( init_statement_args )

init_statement_args ::=identifier , array_def
```

An example of a complete module specification is given in Listing 2-7 on page 50.

Listing 2-7. Example of the definition of a configuration

```
module seeker isa configuration with
      interface
            declare(inputs, 4);
            declare(outputs, 2);
            declare(dft, true);

      submodules
            g isa module gimbal;
            pid_y, pid_p isa module pid;
            c1, c2 isa module coupling;
            dc_p, dc_y isa module dcmotor;

      connections
            g.in[0] @ c1.out[0]; g.in[1] @ c2.out[0];
            g.out[0] @ c1.in[1]; g.out[2] @ c2.in[1];
            g.out[0] @ pid_p.in[0]; g.out[2] @ pid_y.in[0];
            g.out[1] @ pid_p.in[1]; g.out[3] @ pid_y.in[1];
            g.out[0] @ out[0]; g.out[2] @ out[1];

            dc_p.in[1] @ c1.out[0]; dc_p.out[0] @ c1.in[0];
            dc_y.in[1] @ c2.out[0]; dc_y.out[0] @ c2.in[0];

            pid_p.out[0] @ dc_p.in[0]; pid_y.out[0] @ dc_y.in[0];
            pid_p.in[2] @ in[0]; pid_y.in[2] @ in[2];
            pid_p.in[3] @ in[1]; pid_y.in[3] @ in[3];

      initialization
            setState(g, [0.0, 0.0, 0.0, 0.0]);
            setParam(g, {[1.041e-6, 1.118e-6, 1.660e-6],
                  [2.647e-7, 2.634e-7, 1.975e-7],
                  [2.079e-3, -1.266e-3, -8.206e-3, 54.21e-3],
                  [1.573, 0.81108]});
            setState(pid_y, [-0.14]);
            setParam(pid_y, [45, 0.01, 0.5]);
            setState(pid_p, [0.14]);
            setParam(pid_p, [30, 0.01, 0.6]);
            setState(dc_p, [0.0, 0.0]);
            setParam(dc_p, [4.05e-7, 7.9e-3, 8.13e-4, 100, 10]);
            setState(dc_y, [0.0, 0.0]);
            setParam(dc_y, [4.05e-7, 7.9e-3, 8e-5, 400, 10]);
            setParam(c1, [1000, 10]);
            setParam(c2, [500, 10]);
endmodule;
```

2.8 Summary

In this chapter, we described the software engineering abstractions used to develop composable simulations of software components. Based on these software abstractions, we described the system architecture of a computational tool for rapidly creating simulations for mechatronic systems.

The abstractions are based on a three-level hierarchy: *conceptual level, component level* and *process level*. Each abstraction level represents a different aspect of the model, and the three collectively support *composability* of simulation software modules. Similar to a physical design where subcomponents come together to form large more complex components, the simulation software components can be made up of smaller, simpler components by combining them into a *configuration.*

We identified properties that are shared by configurations and component models: *interface* and *implementation*. The fact that they share these properties allows the use of configurations in larger configurations as if the configuration were an individual software component. The advantages provided by this characteristic are twofold: first, we have a mechanism to hierarchically compose simulation models. This is useful when dynamic reconfiguration of simulation software is required to achieve refinement, and therefore different levels of granularity, in the simulation. Second, we can exchange back and forth between the use of a component or the use of a complete configuration both having the same interface, thereby possibly reducing simulation time.

Composability of software components is a powerful mechanism that provides hierarchical composition of simulation models. Furthermore, it allows one to build the simulation of a complex system that integrates mechanics with electronics and information technology modules.

Chapter 3 Linear Graph-Based Modeling of Mechatronic Systems

3.1 Introduction

Mathematical models of components identified in a physical system serve as building blocks in the analysis and design of such system. These mathematical models in general should cross energy domain boundaries to capture the complex interactions between different energy processes taking place in the system.

A modeling paradigm to model the behavior of mechatronic systems needs to be able to capture the complex inter-domain interactions that occur in the system. There exist different methodologies that can be used to model such systems, including, object-oriented modeling and graph-based modeling methods. In this thesis, we have taken the graph-based approach. More specifically, we model mechatronic systems by means of a linear graph that captures the energy flow of the system.

The linear graph approach is based on the fact that for any physical system we can find two variables, namely, across and through variables that capture the energy flow through the system. These two variables were first proposed by Trent [143] as a way of relating two measurements, taken between the terminals of a physical component, to a mathematical representation of the system. This representation captures the energy flow in the system since the selection of across and through variables is made such that their product gives the power flowing through the component. We call these kinds of systems, conservative systems [60] because the flow of energy is constrained by two sets of equations, one that specifies that the sum of through variables entering a node is zero, and the other that specifies that the sum of across variables around a loop of edges is zero. These equations are Kirchhoff's network equations of conservative systems.

In addition to conservative systems, mechatronic systems include non-conservative systems. These systems do not satisfy Kirchhoff's networks laws. The signal domain of a mechatronic system is a non-conservative system.

This chapter defines the two complementary variables for a wide range of physical components and presents the fundamental building blocks used to represent mechatronic systems, including conservative and non-conservative systems. We propose a hybrid representation based on a combination of linear graphs and block diagrams to capture the different aspects of the system.

It is important to emphasize here that the modeling concepts presented in this chapter will only be applied to energy domains other than the mechanical energy domain. Although the concepts presented here apply to mechanical models in which the dynamic variables are scalar variables, the process is quite different when planar and spatial mechanisms are included in the analysis. Since the topic of rigid body dynamics is out of the scope of this work, we build on the work by Dr. McPhee in the Motion Research Group at the Department of Systems Engineering at the University of Waterloo to deal with the mechanical domain for complete spatial (3D) rigid and flexible body dynamics [69, 80, 81, 82, 83, 124].

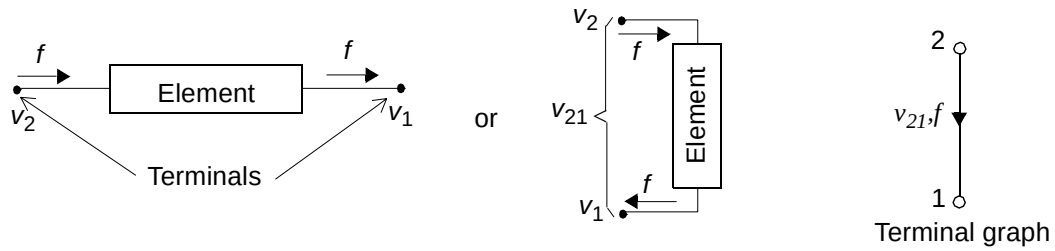


Figure 3-1. Two-terminal element.

The material presented in Section 3.2 is a review of the work by Roe [109] and Trent [143] in Systems Theory. In Section 3.3, we extend this approach to model non-energetic systems by using a hybrid representation based on linear graphs and block diagrams.

3.2 Dynamic system elements

In this section, we will summarize system elements and generalize their properties in terms of energy storage, dissipation and transformation. A uniform terminology and symbolism applicable to all the physical systems involved in a mechatronic system will be developed. In addition, we will introduce the concepts of energy sources, which can supply energy to dynamic systems, transformers, and gyrators. Proper use of these idealized lumped elements permits any physical system to be modeled.

3.2.1 Generalized variables, power and energy

A variable is a measurable characteristic of a system that may change with time. In this modeling approach, a system element is described by a relationship between two variables, a *through* variable, which has the same value at the two terminals or ends of the element, and an *across* variable, which is specified in terms of a relative value between the terminals (Figure 3-1). These variables are called *terminal variables*. We will use the symbols *f*, and *v* to indicate any physical through and across variables, respectively.

These two variables may be expressed as the time derivative of the integrated through variable *h*, and the integrated across variable *x*, respectively.

$$f = \frac{dh}{dt}$$

$$v = \frac{dx}{dt}$$

Equation 3-1

Table 3-1 lists the through and across variables f and v and their respective integrals h and x for the various physical systems.

Table 3-1. Through and across variables for physical systems

Type of system	Through variable, f	Integrated through variable, h	Across variable, v	Integrated across variable, x
Mechanical translational	Force, F	Translational momentum, P	Velocity difference, v_{21}	Displacement difference, x_{21}
Mechanical rotational	Torque, T	Angular momentum, h	Angular velocity difference, Ω_{21}	Angular displacement difference, Θ_{21}
Electrical	Current, i	Charge, q	Voltage difference, v_{21}	Flux linkage, λ_{21}
Hydraulic	Fluid flow, Q	Volume, V	Pressure difference, P_{21}	Pressure momentum, Γ_{21}
Thermal	Heat flow, q	Heat energy, E	Temperature difference, θ_{21}	Not generally used

The power flow $\wp$ into an element or system through two points (1) and (2) which have a common through variable f and an across variable difference v_{21} is generally

$$\wp = fv_{21}$$

Equation 3-2

and the energy W transferred is the time integral of the power. Thus, during time interval $t_a \rightarrow t_b$,

$$W_{ab} = \int_{t_a}^{t_b} \wp \, dt = \int_{t_a}^{t_b} fv_{21} dt$$

Equation 3-3

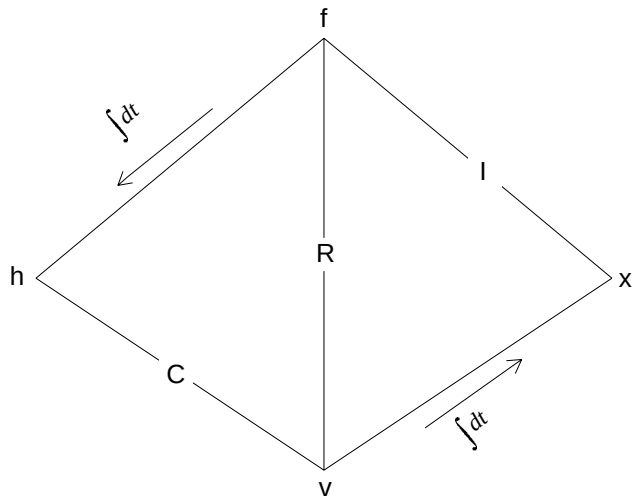


Figure 3-2. Tetrahedron of state for two-terminal elements.

The only exception to the relations given in Equation 3-2 and Equation 3-3 is that in a thermal system power is the through variable itself and energy is the integrated through variable or the amount of heat transferred.

3.2.2 Two-terminal elements

An element or a system composed internally of many elements, which is described as shown in Figure 3-1 by the relation between a single through variable *f* and a single across variable difference v_{21}, is called a *two-terminal* element or system. Since energy can flow into or out of this system only by virtue of *f* and v_{21} (their product in most cases), the system is often called a single energy-port. Since the behavior of the element is given by a relationship between the terminal variables *f* and v_{21}, the energy flow to or from the two-terminal element is determined by either terminal variable. Three types of elements can be identified: energy storage (delay and accumulator), dissipative, and source elements [143].

The generalized relations between the through and across variables for the delay, accumulator and dissipative elements may be summarized with of a tetrahedron of state [61, 112], where the across and through storage variables determine the state of the element (Figure 3-2).

3.2.2.1 Delay energy storage elements: generalized inductance

Generalized inductances are described by a single-valued relationship between the through variable (f) and the integrated across variable (x). For such an element we can write this relationship as follows:

$$x_{21} = g(f) \qquad \textbf{Equation 3-4}$$

where $x_{21} = 0$ when $f = 0$. If the element is ideal (i.e., linear), we can write Equation 3-4 as $x_{21} = Lf$, or if L is constant,

$$v_{21} = \frac{dx_{21}}{dt} = L\frac{df}{dt} \qquad \textbf{Equation 3-5}$$

where L is called the generalized inductance. For ideal springs, L is the reciprocal stiffness or compliance; for ideal inductances, L is the inductance; and for ideal hydraulic elements, L is the inertance.

The energy W supplied to a delay element defined by Equation 3-4 is

$$W = \int_{t_a}^{t_b} f v_{21} dt = \int_0^f f dx_{21} \qquad \textbf{Equation 3-6}$$

The energy function is a function of the terminal equation and the final value of the through variable. Energy is thus stored by virtue of the through variable, and these elements are called *delay energy storage elements*. Table 3-2 summarizes the delay energy storage elements.

Table 3-2. Summary of delay energy storage system elements.

Physical element	Terminal equation	Energy function	Ideal terminal equation	Ideal energy
Translational spring	$x_{21} = g(F)$	$W = \int_0^F F dx_{21}$	$v_{21} = \frac{1}{k}\frac{dF}{dt}$	$W = \frac{1}{2}\frac{F^2}{k}$
Rotational spring	$\Theta_{21} = g(T)$	$W = \int_0^T T d\Theta_{21}$	$\Omega_{21} = \frac{1}{K}\frac{dT}{dt}$	$W = \frac{1}{2}\frac{T^2}{K}$

Table 3-2. Summary of delay energy storage system elements.

Physical element	Terminal equation	Energy function	Ideal terminal equation	Ideal energy
Electrical inductance	$\lambda_{21} = g(i)$	$W = \int_0^i i d\lambda_{21}$	$v_{21} = L\frac{di}{dt}$	$W = \frac{1}{2}Li^2$
Hydraulic inertance	$\Gamma_{21} = g(Q)$	$W = \int_0^Q Q d\Gamma_{21}$	$P_{21} = I\frac{dQ}{dt}$	$W = \frac{1}{2}IQ^2$

3.2.2.2 Accumulator energy storage elements: generalized capacitance

Translational and rotary masses, as well as electrical, hydraulic, and thermal capacitances are defined by a single-valued function of the form:

$$h = g(v_{21}) \qquad \textbf{Equation 3-7}$$

where h is the integrated through variable and is defined as zero when v_{21} is zero. In all cases except the electrical capacitance, the elements described by Equation 3-7 must have one terminal attached to a constant across variable so that $dv_1/dt = 0$; i.e., a reference point.

For any ideal capacitance,

$$h = Cv_{21} \qquad \textbf{Equation 3-8}$$

and, if C is constant,

$$f = \frac{dh}{dt} = C\frac{dv_{21}}{dt} \qquad \textbf{Equation 3-9}$$

where C is the generalized capacitance. For ideal masses, C is the mass or moment of inertia. For ideal electrical, hydraulic and thermal capacitance, C is the electrical, hydraulic or thermal capacitance, respectively.

The energy supplied to a generalized capacitance is

$$W = \int_{t_a}^{t_b} v_{21} f dt = \int_0^{v_{21}} v_{21} dh \qquad \textbf{Equation 3-10}$$

Energy is, therefore, stored as a function of v_{21} and this type of element is called *accumulator energy storage element*. Table 3-3 summarizes the accumulator elements.

Table 3-3. Summary of accumulator energy storage system elements.

Physical element	Terminal equation	Energy function	Ideal terminal equation	Ideal energy
Translational mass	$p = g(v_2)$	$W = \int_0^{v_2} v_2 dp$	$F = m\frac{dv_2}{dt}$	$W = \frac{1}{2}m_2^2$
Inertia	$h = g(\Omega_2)$	$W = \int_0^{\Omega_2} \Omega_2 dh$	$T = J\frac{d\Omega_2}{dt}$	$W = \frac{1}{2}J\Omega_2^2$
Electrical capacitance	$q = g(v_{21})$	$W = \int_0^{v_{21}} v_{21} dq$	$i = C\frac{dv_{21}}{dt}$	$W = \frac{1}{2}Cv_{21}^2$
Hydraulic capacitance	$V = g(P_2)$	$W = \int_0^{P_2} P_2 dV$	$Q = C_f\frac{dP_2}{dt}$	$W = \frac{1}{2}C_f P_2^2$

3.2.2.3 Energy dissipator element: generalized resistance

Translational and rotational dampers, and electrical, hydraulic and thermal resistances are defined by the single-valued function:

$$f = g(v_{21}) \qquad \textbf{Equation 3-11}$$

where the function g is such that $f = 0$ when $v_{21} = 0$ and the signs of f and v_{21} are always the same. An element that meets the requirements of Equation 3-11 is called a *generalized resistance*. If the resistance is ideal,

$$f = \frac{1}{R}v_{21} \qquad \textbf{Equation 3-12}$$

where R is the generalized resistance of the element. The resistance of ideal translational and rotational dampers is the reciprocal of the damping coefficients. the electrical, hydraulic, and thermal elements have their resistances equal to R.

The *power* $\wp$ supplied to a generalized resistance is:

$$\wp = fv_{21} = v_{21}g(v_{21}) \qquad \textbf{Equation 3-13}$$

Since the signs of f and v_{21} are always alike, $\wp$ is always positive and power always flows into the resistance. Hence, the generalized resistance dissipates energy, and it is called *energy dissipator element*. Table 3-4 shows the energy dissipator elements for the mechanical, electrical, hydraulic, and thermal systems.

Table 3-4. Summary of energy dissipator system elements: $\wp \geq 0$.

Physical element	Terminal equation	Power function	Ideal terminal equation	Ideal power
Translational damper	$F = g(v_{21})$	$\wp = Fv_{21}$	$F = bv_{21}$	$\wp = bv_{21}^2$
Rotational damper	$T = g(\Omega_{21})$	$\wp = T\Omega_{21}$	$T = B\Omega_{21}$	$\wp = B\Omega_{21}^2$
Electrical resistance	$i = g(v_{21})$	$\wp = iv_{21}$	$i = \frac{1}{R}v_{21}$	$\wp = \frac{1}{R}v_{21}^2$
Hydraulic resistance	$Q = f(P_{21})$	$\wp = QP_{21}$	$Q = \frac{1}{R_f}P_{21}$	$\wp = \frac{1}{R_f}P_{21}$

3.2.2.4 Energy sources

By source we mean a device capable of delivering energy continuously to a system. Two types of idealized sources are considered, one in which the across variable is a prescribed function of time, and one in which the through variable is a prescribed function of time. The first type of idealized source is called an *across source* and the second is called a *through source*:

$$v_{21} = g_v(t)$$
$$f = g_f(t)$$

Equation 3-14

Although sources are usually used to supply energy to a system, the source may also absorb energy. When the source is absorbing energy, the sign of the conjugate variable for a source (f for an across source or v_{21} for a through source) is the same as that of the variable defining the source. For any source, the flow of power *into* the source is $\wp = fv_{21} > 0$. If f and v_{21} are of opposite sign $\wp < 0$ (the source supplies energy). Although $\wp$ can also be negative for energy storage elements (energy is being removed from the element), only sources can supply power and energy continuously over an extended period of time.

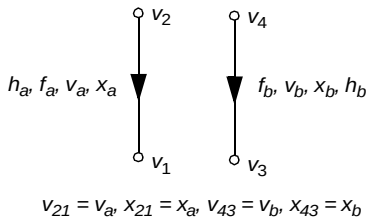


Figure 3-3. Terminal graph for couplers

3.2.3 Multi-terminal components

Multi-terminal components, called couplers, arise in system modeling to capture interactions between two energy domains, or to provide transformations between terminal variables of the same energy domain.

The terminal equation for a coupler specifies whether it is obtained from a hybrid parameter model or not. Hybrid parameter couplers are called *direct couplers* because in their ideal form they impose constraints on dynamic variables of the same type. That is, constraints between across variables alone or constraints between through variables alone. Non-hybrid parameter couplers are called *inverse couplers* because in their ideal form they impose constraints between variables of different types [143]. Figure 3-3 shows the terminal graph for direct or inverse couplers.

3.2.3.1 Direct couplers (transformer)

Terminal equations for direct couplers are of the form:

$$\begin{bmatrix} v_a \\ f_b \end{bmatrix} = \begin{bmatrix} h_{aa} & h_{ab} \\ h_{ba} & h_{bb} \end{bmatrix} \begin{bmatrix} f_a \\ v_b \end{bmatrix} \qquad \textbf{Equation 3-15}$$

Matrix **H** is assumed to be invertible so that one can find a new model where the roles of the across and through variables are exchanged. If we let $h_{aa} = h_{bb} = 0$ and $h_{ba} = -h_{ab}$ we obtain the model of an ideal direct coupler. That is, the coefficient matrix for an ideal coupler is skew-symmetric so that no power is stored in the element:

$$\wp = \begin{bmatrix} f_a & v_b \end{bmatrix} \begin{bmatrix} v_a \\ f_b \end{bmatrix} = \begin{bmatrix} f_a & v_b \end{bmatrix} \begin{bmatrix} 0 & h_{ab} \\ -h_{ab} & 0 \end{bmatrix} \begin{bmatrix} f_a \\ v_b \end{bmatrix} \equiv 0 \qquad \textbf{Equation 3-16}$$

Examples of direct couplers are shown in Table 3-5. With the exception of the electrical transformer, transformers in other energy domains must have the terminals 1 and 3 (show in Figure 3-3) common (this requirement does not apply to direct couplers that model energy transducers).

Table 3-5. Example of physical elements represented by generalized direct couplers. These models are based on those presented by Roe [109].

Name	**Terminal equations**
Gear train (transformer)	$\begin{bmatrix} \Omega_{41} \\ T_a \end{bmatrix} = \begin{bmatrix} 0 & \frac{N_b}{N_a} \\ -\frac{N_b}{N_a} & 0 \end{bmatrix} \begin{bmatrix} T_b \\ \Omega_{21} \end{bmatrix}$ where N_b/N_a is the gear ratio.
Lever (transformer)	$\begin{bmatrix} v_{41} \\ F_a \end{bmatrix} = \begin{bmatrix} 0 & \frac{r_b}{r_a} \\ -\frac{r_b}{r_a} & 0 \end{bmatrix} \begin{bmatrix} F_b \\ v_{21} \end{bmatrix}$ where r_b/r_a is the lever ratio.
Electric transformer	$\begin{bmatrix} v_{43} \\ i_a \end{bmatrix} = \begin{bmatrix} 0 & \frac{N_b}{N_a} \\ -\frac{N_b}{N_a} & 0 \end{bmatrix} \begin{bmatrix} i_b \\ v_{21} \end{bmatrix}$ where N_b/N_a is the turns ratio.
DC motor (electrome-chanical transducer)	$\begin{bmatrix} v_{21} \\ T_b \end{bmatrix} = \begin{bmatrix} R_a + L_a\frac{d}{dt} & k_e \\ -k_m & B_b + J_b\frac{d}{dt} \end{bmatrix} \begin{bmatrix} i_a \\ \Omega_{43} \end{bmatrix}$ where k_e and k_m are the electrical and motor constants respectively.

3.2.3.2 Inverse couplers (gyrator)

Terminal equations for inverse couplers are of the form:

$$\begin{bmatrix} v_a \\ v_b \end{bmatrix} = \begin{bmatrix} q_{aa} & q_{ab} \\ q_{ba} & q_{bb} \end{bmatrix} \begin{bmatrix} f_a \\ f_b \end{bmatrix} \qquad \textbf{Equation 3-17}$$

This is a generalization of a two terminal dissipative component. This form of the inverse coupler terminal equations is called *impedance form* since it represents a dissipative component where the independent variables are through variables. If the model is inverted, we obtain the second form of an inverse coupler: the *conductance* form.

$$\begin{bmatrix} f_a \\ f_b \end{bmatrix} = \begin{bmatrix} g_{aa} & g_{ab} \\ g_{ba} & g_{bb} \end{bmatrix} \begin{bmatrix} v_a \\ v_b \end{bmatrix} \qquad \textbf{Equation 3-18}$$

Where $\mathbf{G} = \mathbf{Q}^{-1}$. If the coefficient matrix of an inverse coupler is also skew-symmetric, we have the ideal inverse coupler form and no power is stored in the element:

$$\wp = \begin{bmatrix} f_a & f_b \end{bmatrix} \begin{bmatrix} v_a \\ v_b \end{bmatrix} = \begin{bmatrix} f_a & f_b \end{bmatrix} \begin{bmatrix} 0 & q_{ab} \\ -q_{ab} & 0 \end{bmatrix} \begin{bmatrix} f_a \\ f_b \end{bmatrix} \equiv 0 \qquad \textbf{Equation 3-19}$$

These models represent non-ideal characteristics of their corresponding system components. However, we can make some simplifications to find the idealized model, which will be of one of the previous two classes: direct or inverse ideal coupler. For example, if in the model of the gear train we neglect the damping coefficient and the inertia effects we obtain the ideal direct coupler. Similarly, if we neglect the leakage, damping and mass in the model for the hydraulic piston we obtain the ideal inverse coupler.

An example of an inverse coupler is the hydraulic piston:

$$\begin{bmatrix} F_a \\ Q_b \end{bmatrix} = \begin{bmatrix} 0 & A \\ -A & 0 \end{bmatrix} \begin{bmatrix} v_{21} \\ P_{43} \end{bmatrix} \qquad \textbf{Equation 3-20}$$

where A is the area of the piston.

3.2.4 Linear graph representation of n-terminal elements

A two-terminal component that is connected to two terminals, *A* and *B,* in a given system, can be represented as a directed edge between two vertices, *a* and *b,* in a graph. In general, the graph representation of the component is a directed edge that joins two terminal points. This graph representation is called *terminal graph* of the component, and associated with this terminal graph are the component's terminal variables. This is illustrated in Figure 3-1.

Often, there will be a one-to-one correspondence between the terminal graph of a two-terminal component and the physical object; however, this does not always have to be the case. Consider the position of the center of mass of a rigid body. To obtain meaningful measurements, we require a reference point with respect to which we will measure the position of the center of mass. If we consider the position of the body in 3-dimensional space, we need three measurements to determine its position unambiguously, for instance, the *x*-position, the *y*-position, and the *z*-position, relative to the reference point. Each measurement is given by an instrument located between two terminals; one terminal is associated with the rigid body and the other terminal with a fixed reference. If we also include the orientation of the rigid body, three more measurements are needed. Therefore, the rigid body should not be treated as a single component; rather it should be treated as if it were six distinct components, each having its own terminal graph. This means that there is a clear distinction between simulation component and system components: a simulation component is a *modeling abstraction* used to characterize a dynamic property of a system component and thus it is related to a terminal graph. If we admit the variables associated with the terminal graphs to be elements of $\Re^6$ we can represent the rigid body as a single two-terminal component having vector-valued across and through variables (Figure 3-4), however we must not forget the fact that we need six measurements to determine the position and orientation of the object.

Another aspect of interest in modeling physical systems is that of non-ideal properties. For instance, one might be interested in modeling the passive (resistive) effects occurring in a transformer while it is coupling two electric networks. In such situations, one treats the

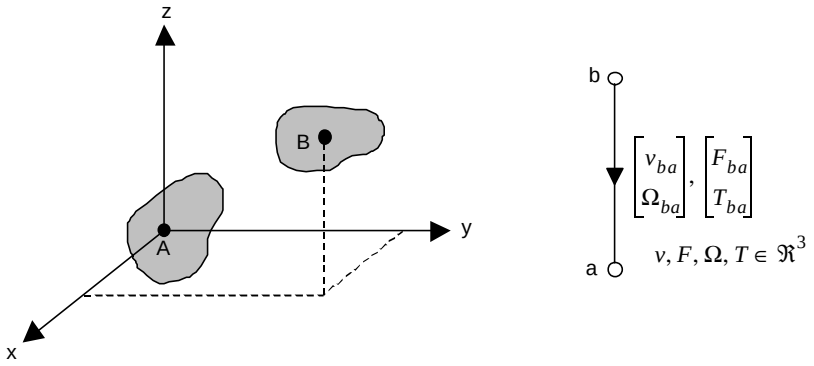


Figure 3-4. Displacement measurement of a rigid body in space with respect to a reference frame.

device as being two components, separating in this way the coupling function from the passive function. A similar decomposition arises when a physical inductor is treated as two components, an ideal inductor with a resistor in series. In this case, the physical device is not described by a single terminal graph but rather by a collection of terminal graphs each modeling a particular aspect of the device.

Interactions between components in different energy domains cannot be described with a two-terminal element. It is necessary to introduce elements that have more than two terminals; i.e., *n-terminal* elements. Within this category, we find the transducer elements previously defined. The system graph associated with an n-terminal element will be derived from measurements taken between pairs of terminals. However, as is shown by Roe [109], we only need $n-1$ across measurements to completely determine the across variables between any pair of terminals. This number corresponds to the number of branches in a tree selected in the graph of the n-terminal element. The graph of an *n*-terminal element is derived from connecting every terminal in the component to every other terminal in the component. The *terminal graph* of an *n*-terminal element is the tree $\mathbf{T}$ of $n-1$ edges connecting the *n* vertices corresponding to the *n* terminals of the system component. To illustrate this case consider the electric transformer (a 3-terminal system component) shown in Figure 3-5. The graph of the component includes an edge from node 2 to node 4. However, only two across measurements will completely determine the device giving a terminal graph with two edges.

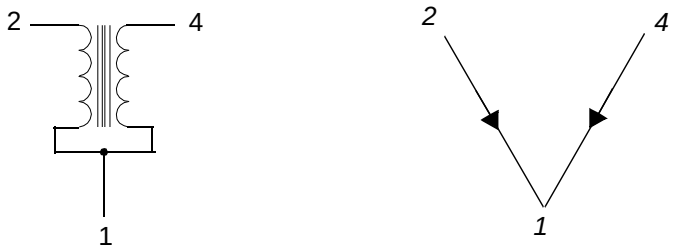


Figure 3-5. n-terminal component

As is the case with two-terminal components, the edges in a terminal graph of an *n*-terminal component will be associated with measurements taken between terminal pairs in the physical system. The number *m* of independent across and through measurements required to completely characterize the component has an upper bound:

$$m \leq n - 1 \qquad \textbf{Equation 3-21}$$

The inequality in Equation 3-21 holds if the component has any of the following two properties [143]:

1. For all *t,* an across instrument placed between a pair of terminals gives a null reading, in which case the two terminals can be treated as one.
2. An instrument attached to two points shows that these are dynamically independent; in which case, an instrument needs not be attached at these two points.

In the case where multiple energy domains are associated with a given physical object, a set of terminals is associated with each energy domain. Once the sets of terminals are chosen, the complementary variables (across and through variables) of each energy domain are identified on the terminal graph associated with each energy domain (Figure 3-6)

In summary, there exists an isomorphism between linear graphs and physical systems. For a system composed of *m* subsystems, the *system graph* is the union of all terminal graphs for all the components of the system in one-to-one correspondence with their interconnection.

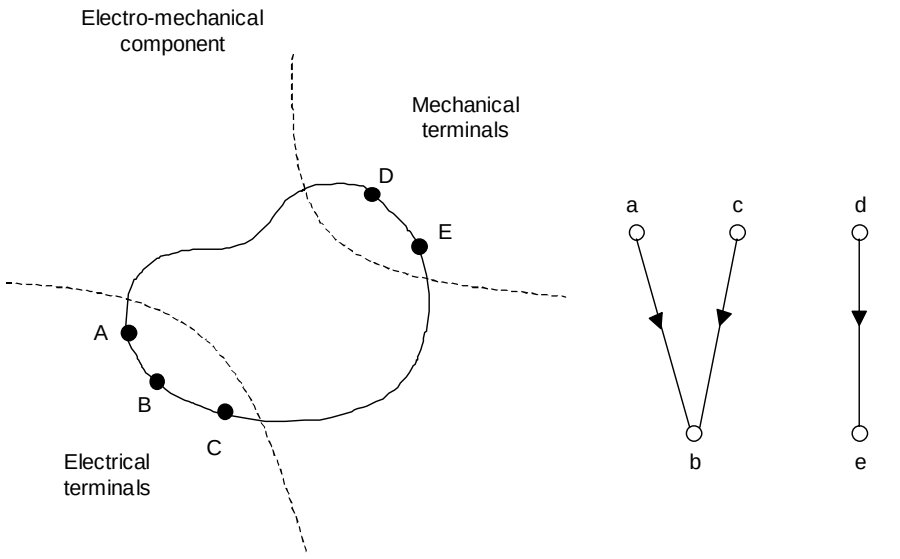


Figure 3-6. Terminal graph identifying the variables in a multi-domain component.

The topological properties of any graph are captured in two matrices, namely, the incidence and the circuit matrices. For a system graph, these matrices provide the basis to define two theorems of systems theory that define the conservative properties of a system [109].

Theorem I. The *oriented sum* of through variables associated with the edges incident on a given vertex is zero at any instant of time:

$$\mathbf{Ay} = \mathbf{0} \qquad \textbf{Equation 3-22}$$

where matrix **A** is the incidence matrix of the system graph.

Theorem II. The oriented sum of the across variables associated with the edges in a given circuit is zero at any instant of time.

$$\mathbf{Bx} = \mathbf{0} \qquad \textbf{Equation 3-23}$$

where matrix **B** is the circuit matrix of the system graph.

The proofs of these two theorems can be found in [109].

Given these two theorems, we can restate Theorem A-I to account for our definition of across and through variables as follows:

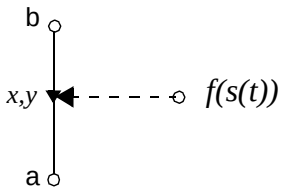


Figure 3-7. Terminal graph of signal-controlled driver.

Theorem III. If **T** is an arbitrarily selected tree of a system graph, the across variables of the chords of **T** can be expressed as linear combinations of the across variables of the branches of **T**, and the through variables of the branches of **T** can be expressed as linear combinations of the through variables of the chords of **T**.

The proof of this theorem is similar to that of Theorem A-I and it follows from Equation A-6 and Equation A-7 in Appendix A.

3.3 Low-power component modeling

In order to include information technology components as well as other types of low-power devices in the system graph, it is necessary to extend the system graph representation to include signals. A *signal* represents the value of some system variable as a function of time.

To introduce signals in the system graph we define the concept of *variable elements*. A variable element is an element that can have one or more input signals that modify its response. The simplest variable element is the signal-controlled across or through driver. In this case, either the across or through variable associated with the terminal graph will be completely defined by the signal; i.e., $x(t) = f(s(t))$ or $y(t) = h(s(t))$. Where x and y represent across and through variables, respectively.

Similarly, a variable passive element is also signal-controlled, but here, the input signal is modulating one of the element parameters (Figure 3-7). Output signals are obtained from the system graph as “measurements” of system variables (Figure 3-8).

In the context of mechatronics, it is important to have a system representation that is capable of handling signal elements. Mechatronic systems include information technology com-

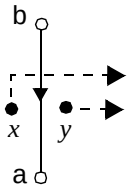


Figure 3-8. Reading values from a terminal graph

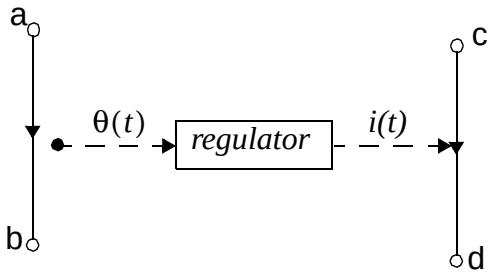


Figure 3-9. A positioning system. The system graph shows the interaction between the signal block and the terminal graph.

ponents for which there is no energy flow and that therefore cannot be represented by a terminal graph. As an example, consider an embedded controller. The control algorithms are provided as algorithmic components that must interact with the rest of the system but do not generate or transfer any measurable power.

To illustrate this, consider a portion of a positioning system composed of an angular position sensor, a regulator, and a current source (Figure 3-9). The regulator obtains the signal input from the position sensor to provide an output signal used to modulate the current source.

In summary, signals can only arrive at an edge of a terminal graph and they can only be read from abstract nodes associated with that edge. In this way, no ambiguities can arise when augmenting the system graph with a block diagram describing the interaction of low power components and we have a better representation to derive a set of system equations.

The synthesis of system equations proceeds by considering only the system-graph portion of the entire model. This way the algorithms presented in Chapter 5 can determine a suitable causality assignment. Once the system equations are derived, the equations derived

from the block-diagram are incorporated in the set of equations and a sorting algorithm is performed to find a correct computational order of evaluation of the system equations derived from the system graph and the computational blocks specified in the block diagram.

3.4 Port-based multi-domain modeling of mechatronic systems

In this section, we present the modeling paradigm used to describe mechatronic systems for which the behavior is given by a linear graph. In this approach, which is based on object-oriented modeling concepts, system models are defined by interfaces, and interactions between components are modeled by connections between components' interaction points. The objective of this modeling layer is to encapsulate all behavioral information (the linear graph) into a single entity that can be used to build larger systems.

Simulation models of mechatronic systems must be able to capture interactions between components in different energy domains. In this respect, we regard components from a systems point of view, i.e., as a structure of interrelated elements that are embedded in an environment. Taking a systems approach to modeling mechatronic systems fits well with our concept of composability—namely, as the synthesis of simulations through the definition of the constituent components and their interactions.

In our modeling paradigm, subsystems interact with each other through ports [31, 35]. Ports represent localized points on the boundary of the system where energy exchange between the system and the environment takes place. At a port, energy flows in and out of the system. Consequently, there is a port for each interaction point, and each port will belong to an energy domain. The energy flow through a port is represented by means of the generalized across and a through variables (see Equation 3-1 on page 55). For example, consider an electric transformer with four terminals. Each terminal represents a port through which electrical energy flows in and out of the transformer. In this example, the ports belong to the electrical energy domain; this captures the flow of power in terms of the voltages and currents on the two sides of the transformer.

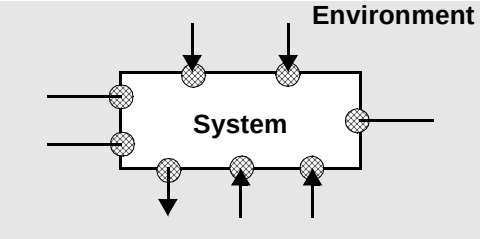


Figure 3-10. Model of an engineering system. Energy ports are represented by non-directed lines while signal ports are represented by arrows.

Connections between ports represent the interactions between different components. A connection between two ports represents the energy exchange between two subsystems and imposes algebraic constraints on the port variables involved in the connection. In general, these constraints take two forms: one form enforces the equality of the across variables, and the second enforces the sum of the through variables to be zero;these are the Kirchhoff network constraints.

Physical interactions that represent energy exchange have no predefined direction. Therefore, we capture a physical interaction with undirected connections representing non-causal interactions. This approach to modeling reflects the physical interactions more accurately and relieves the modeler of specifying the input/output relations, as would be required in a modeling environment such as Simulink.

Besides ports and connections that model energy flow, we also consider signals and signal ports. No energy flows through the signal ports, and the interaction between signal ports is causal. That is, signal ports have a predefined input-output direction that constrains the signal flow between components. Signal and signal ports capture a system based on a block diagram description similar to Simulink.

The system's ports are collectively grouped into an interface, which defines the interaction points of the system with the environment. In this way, we can describe systems as self-contained entities, whose interactions with the environment can be described independently of the internal behavior of the system, as illustrated in Figure 3-10.

The port-based modeling paradigm can describe component interactions in any energy domain as long as the interaction is not distributed but lumped. Consider for example a flexible beam. A finite element model may describe the behavior of the beam. However, presuming that the interaction points of the beam are localized at the two ends, we can describe its interaction with the environment with two ports located at the two ends. Thus, our modeling paradigm is limited to interactions that are localized at a finite number of points on the boundary of the system.

As illustrated in Figure 3-11, the port-based modeling paradigm also supports a hierarchical model structure. The hierarchy can have any number of levels; however, in order to transform it into an adequate simulation model, the hierarchy must be terminated with *primitive* systems, or systems that cannot be divided into smaller subsystems. *Compound*

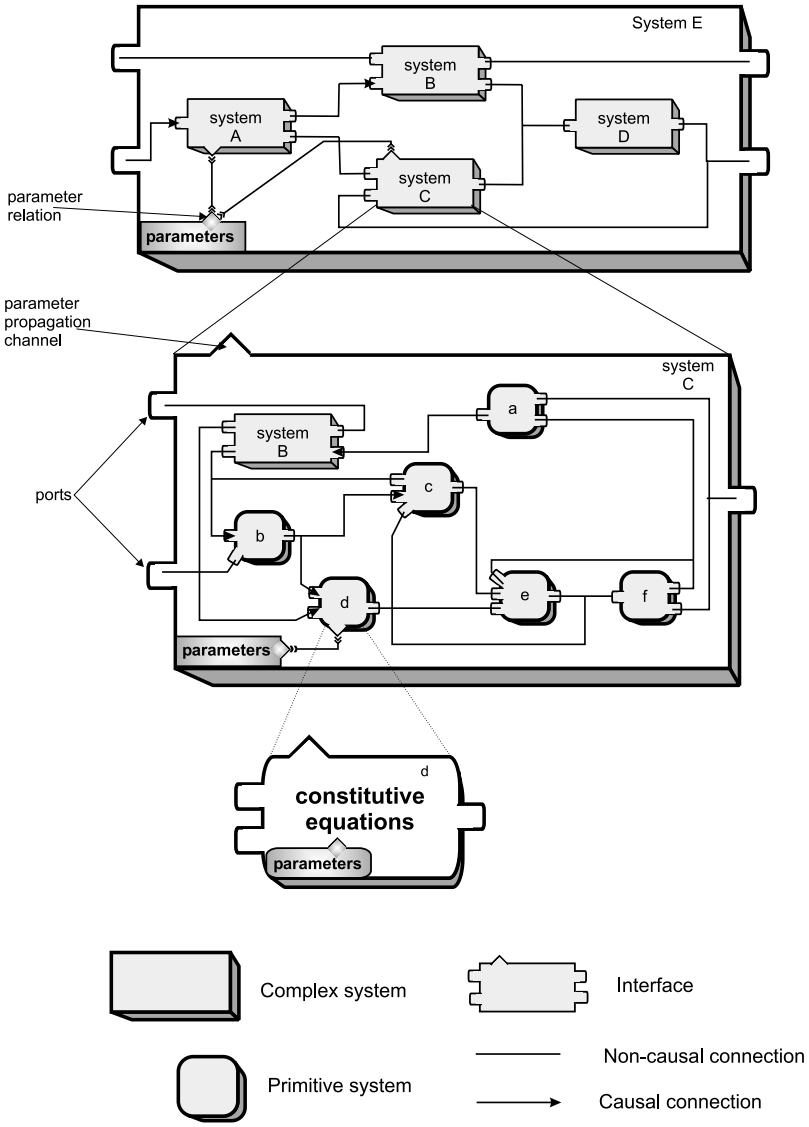


Figure 3-11. Hierarchical model structure

systems on the other hand are composed by connecting primitive models or compound models at a lower level in the hierarchy.

3.5 Summary

In this chapter, we defined the fundamental building blocks for modeling mechatronic systems. The approach is based on graph-theoretic concepts and is applicable to any physical system for which we can find two types of variables: across variables and through variables. The graph-theoretic modeling paradigm was extended to include signal domain com-

ponents. As a result, we defined a hybrid modeling approach combining linear graphs and block diagrams with signal-controlled elements.

The system graph defines the topology of the energy flow of the system and in order to have a complete model we include the terminal equations of the components. Terminal equations model physical characteristics of the device and it was argued that a physical device may have more than one terminal equation which in turn is associated with an edge of the terminal graph of the component.

A last note on the terminal equations, they are non-causal relationships between terminal variables. This fact provides greater modeling flexibility since the task of assigning the causal form is not part of the modeler responsibilities. The causal form is derived from the topological properties of the system graph, which improves on model reuse and reduces possible modeling errors.

The mathematical model described by the linear-graph approach is encapsulated into a port-based object to provide a self contained entity that allows its composition into a larger system.

Chapter 4 Synthesis of the system graph for mechatronic systems

4.1 Introduction

In this chapter, we define the approach to synthesize the system graph of a mechatronic system. The approach takes two steps, which can be performed concurrently. On one side, the system graph for the non-mechanical part of the system is built, and on the other side, the mechanical system graph is built.

The system graph for a mechatronic system is constructed with the help of a *system editor* that is tightly integrated with a CAD system. The approach to building a system in the system editor is called *schematic-diagraming.* In this approach, the modeling is done at the component level and the interactions between components are defined by connections between *terminals*.

As is shown in Figure 4-1, the system editor is based on the concept of *modeling layers* each of which represents a different energy domain of the system. The modeling layer for the

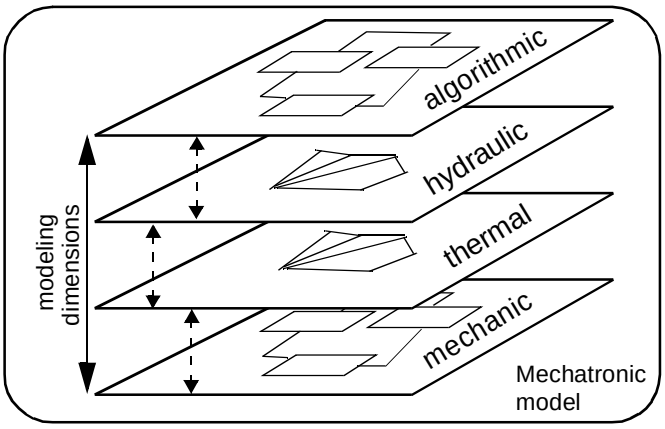


Figure 4-1. Modeling layers of a mechatronic system.

mechanical energy domain is implemented in a CAD system. When a component is brought into the system editor, the models that make up its entire description are included in their respective modeling layers.

The user identifies the interactions between components by defining the connections between the terminals of the components. Interactions are classified as: 1) mechanical interactions, 2) terminal connections, and 3) edge associations. Terminal connections and edge associations arise from the interconnection of elements in non-mechanical modeling layers. On the other hand, mechanical interactions such as rigid connections, prismatic joints or revolute joints arise from the interconnection of two rigid bodies.

The diagram illustrated in Figure 4-2 shows the different stages and the flow of information required in order to arrive at a set of algebraic differential equations that define the behavior of the system. This chapter covers the topmost nodes in the diagram, namely, the synthesis of the system graph. The rest of the flow diagram will be covered in Chapter 5.

After the design is defined in the system editor, the topology and geometry of the system is derived from the high-level description given in the system editor, which provides as outputs the system topology and its geometry. The analysis occurs in two parts: the analysis of the mechanical domain, and the analysis of the non-mechanical energy domains. In the mechanical domain, the analysis starts by extracting the kinematic and geometric information from the CAD model. This information is used in the generation of the mechanical

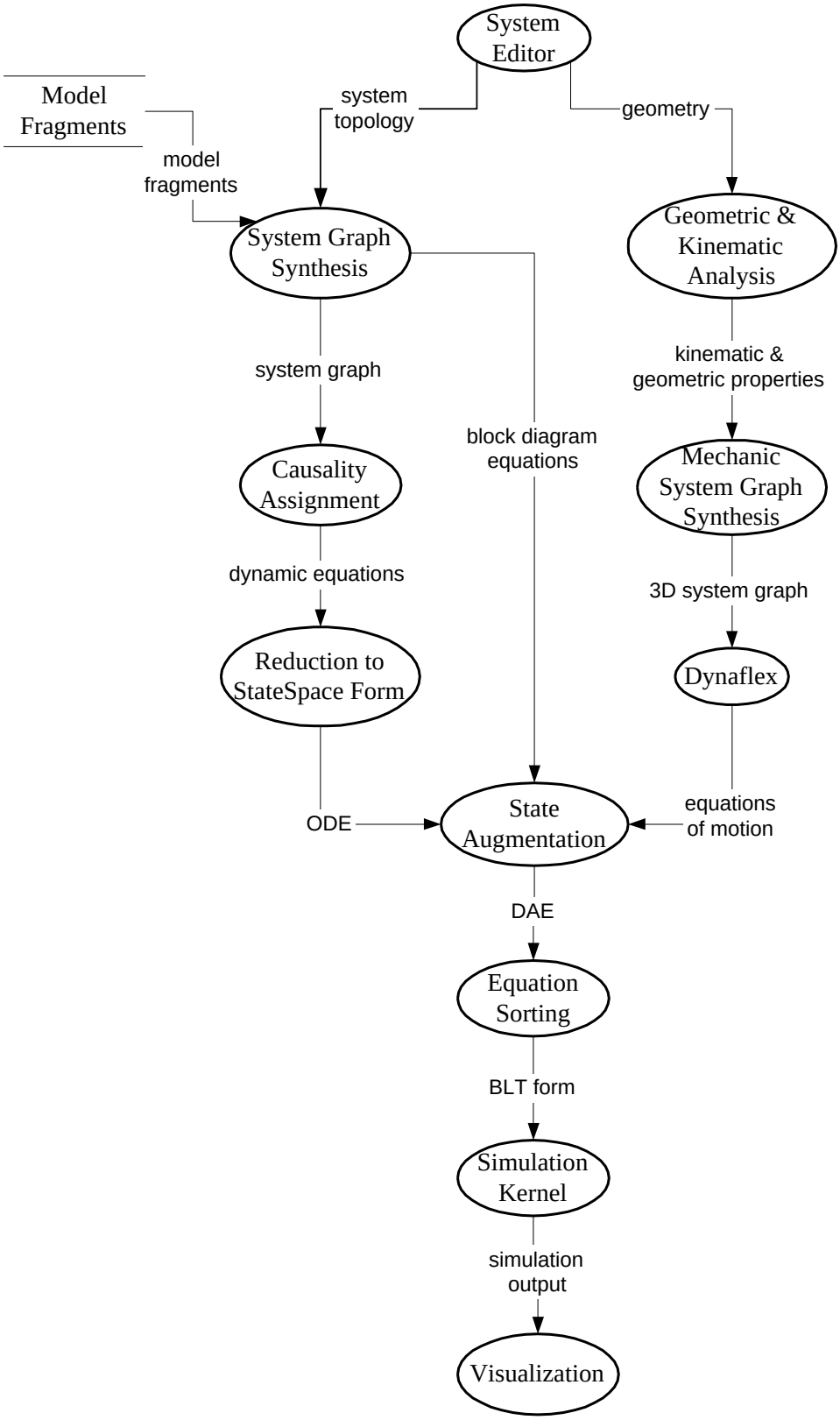


Figure 4-2. System data flow diagram.

system graph. The output of this process is passed on to Dynaflex [124] to generate the equations of motion of the mechanism. The analysis of the non-mechanical energy domains starts by deriving the system graph from the topological information provided by the system editor. The system graph is used to write the terminal equations in causal form, which are reduced to a state space form. This set of equations is combined with the equations derived from Dynaflex and with the equations derived from the signal domain. This new set of equations is transformed to a set of differential algebraic equations, which is sorted to find a computationally correct order of evaluation [34].

The output of the sorting process is a system of equations in *Block Lower Triangular* (BLT) form [39]. In this implementation the equations in BLT form are translated into ASCEND language [100, 101], which is the target simulation language.

The output of the ASCEND simulation is sent to the visualization process. In this process two different approaches are provided: 1) an animated view of the motion of the mechanism or 2) a graph of the state variables versus time.

4.2 Synthesis of the system graph for non-mechanical energy domains

Terminal connections represent the interaction between components within the non-mechanical energy domains. Interactions are non-causal, which means that the terminals involved in the connection do not have a predefined direction. A terminal connection between two terminals indicates that both terminals are mapped to a single node in the system graph.

As defined in Chapter 3, the mathematical model of a component consists of two parts: the terminal equations (behavior) and the terminal graph (topology). The process of generating the system graph is a two-step process [32, 33]. First, the terminal graphs of the individual components are instantiated to create a disconnected graph with n_c components, where n_c is the number of terminal graphs in the system. Second, the information provided by the terminal connections is used to reduce the graph to a non-connected graph with $n_E < n_c$ components, where n_E is the number of energy domains involved in the design.

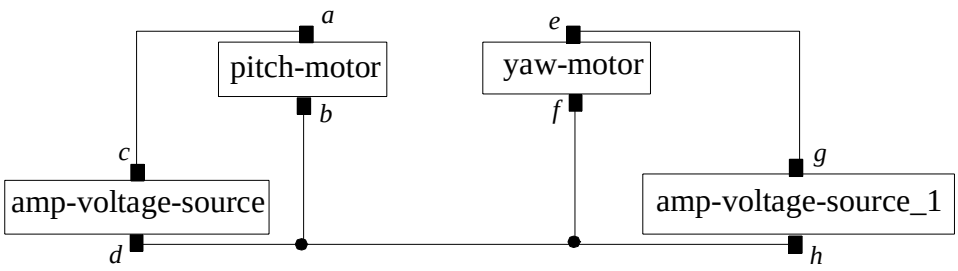


Figure 4-3. Schematic diagram of the electrical components of the missile seeker

We define a topological operator *merge(u, v)*—given two vertices *u* and *v*, *merge* combines the two vertices into a single vertex such that the edges adjacent to *u* and *v* now share the same vertex—as follows. Let $\mathbf{G}$ be the system graph defined by $\mathbf{G} = (V, E)$ where V is the set of vertices and E is the set of edges represented as ordered sets $\{u, v\}$. Then,

$$merge(u, v) \sim \begin{array}{l} \{u, v\} \in V \Rightarrow V \leftarrow V - \{v\}, \\ \qquad \forall (e \in E), v \in e \Rightarrow e \leftarrow e \oplus_v u \end{array} \qquad \textbf{Equation 4-1}$$

where the operator $\oplus_v$ replaces the occurrence of vertex *v* by *u* in the ordered pair *e*.

The generation of the system graph is a transformation operation that reduces a non-connected graph with n_c components to a non-connected graph with n_E components by a successive application of the merge operator on the terminal graphs of the components.

To illustrate this, consider the design of the electric system of a missile seeker shown in Figure 4-3. The components included in this design are signal amplifiers and actuators (yaw and pitch). The initial system graph consists of $n_c = 4$ terminal graphs. Terminal connections derived from the system description are used to perform successive applications of the merge operation to reduce the electrical system graph to a connected graph with $n_E < n_c = 1$. For instance, the electrical system of the *pitch-motor* and the *amp-voltage-source* are connected through connections *(a, c)* and *(b, d)* which result in two merge operations as indicated in Figure 4-4.

A similar process occurs with the other two electric components leaving an electric system graph with two components. However, the terminal connection between the *amp-voltage-*

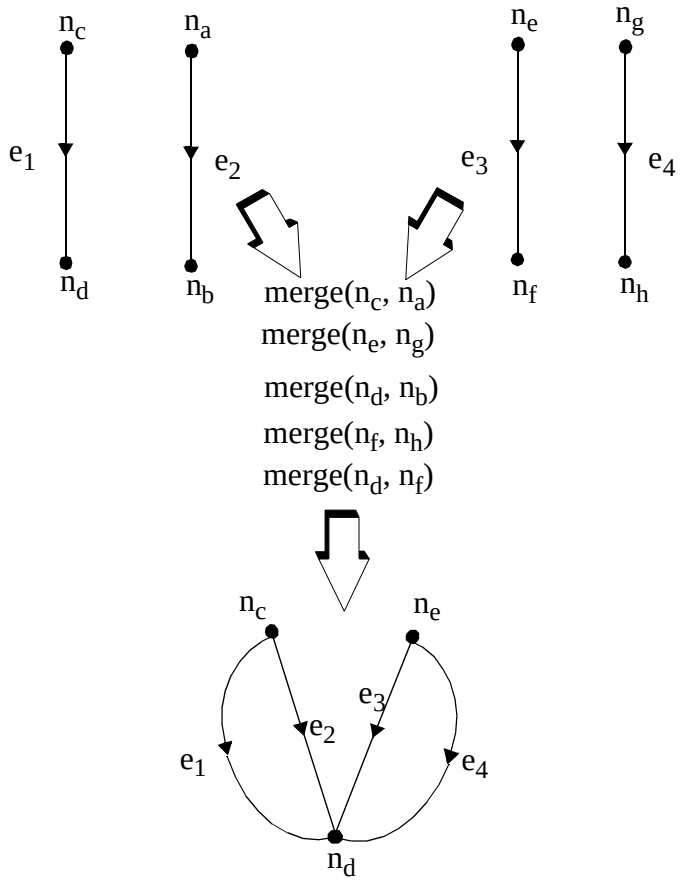


Figure 4-4. Topological operations to a connected electrical system graph

sources defines the common ground, which merges the ground nodes of the two components reducing the electrical system graph to a connected graph with a single component.

Edge associations arise from the energy exchange between different energy domains. They occur when system variables in the terminal equations of a component are associated with other edges in the terminal graph. For example, consider the terminal equation of the electrical edge of a DC motor.

$$v(t) = K_m\dot{\theta}(t) + R + L\frac{d}{dt}i(t) \qquad \textbf{Equation 4-2}$$

Variable $\dot{\theta}(t)$ is a system variable that is associated with an edge (in the mechanical system graph) that is not part of the electrical domain. These types of variables are called *exogenous* since they are assumed to be known within that portion of the model. The definition

of exogenous variables within a terminal equation establishes an association between edges in the system graph.

4.3 Synthesis of the system graph for 3D Mechanics.

The difference between the system graph for non-mechanical energy domains and the system graph for the mechanical domain lies in the dimensionality of the terminal variables. Variables in the mechanical domain are elements of $\Re^3 \times SO(3)$ whereas variables in the system graph are elements of $\Re$ (i.e., scalars).

Once the system graph for the mechanical domain is generated, the dynamic equations of the 3D mechanics of the system are derived using a sub-module (Dynaflex), which is specifically designed for the analysis of three-dimensional constrained mechanical systems [124]. Dynaflex is a research system developed at the University of Waterloo based on a graph-theoretic approach in which the connectivity of the bodies in the mechanism and the forces acting on them are represented by a linear graph (mechanical system graph). Dynaflex is based on the same principles as those used in the derivation of the system equations for non-mechanical energy domains, and therefore it can be seamlessly integrated with our approach.

Our approach is general enough to accept different mechanism analysis tools; however, the only restriction is that it must provide dynamic equations in symbolic form. This is because these equations are combined with the remaining system equations, which are derived from the other non-mechanical energy domains (see Figure 4-2).

Similar to the basic modeling elements we defined in Chapter 3, Dynaflex provides a set of modeling elements for mechanical systems, including, body elements, arm elements (position vectors), motion and force drivers, spring-damper-actuator elements, and joint elements [124]. This classification of elements is used in a way analogous to that presented in Chapter 5 to find the normal tree of the system graph. Once the mechanical system graph has been defined—as indicated later in Section 4.3.2—penalty costs are assigned to the edges of the system graph based on the type of element they model. This weighted graph is used to find a tree that will define the causality of the terminal equations associated with

the edges in the mechanical system graph. This tree is found by applying the minimum cost spanning tree algorithm to the weighted graph.

4.3.1 3D mechanisms

In this section we present a summary of the features that are available in Dynaflex, and that we use to describe our mechanical models.

A 3D mechanism is defined as a finite number of bodies (rigid or non-rigid) connected in an arbitrary fashion by mechanical joints that limit the relative motion between pairs of bodies. Multibody systems contain a number of fundamental elements, which can be classified as follows:

- Rigid bodies.
- Joints that provide kinematic constraints.
- Forces: springs, dampers, actuators.

In the following subsections, we describe the types of kinematic constraints and external forces that can be included in the mechanism (i.e., elements defined by Dynaflex) as well as their linear graph representation. The material presented herein is for completeness only. Its presentation will make it easier to understand Dynaflex' representation of 3D mechanisms and the algorithms to automatically derive this representation from the CAD model (Section 4.3.2).

4.3.1.1 Kinematic constraints

To define the kinematic joints in the system uniquely, we assume that the mechanism does not have fully constrained joints (i.e., they are not rigidly connected). If this were not the case, then the bodies that are involved in the joint should be combined into a single rigid body to avoid structural singularities or index problems in the resulting equations of motion.

For a 3D mechanism with an open kinematic chain, if n is the number of bodies we will find that there will be exactly $n-1$ articulated joints. With this in mind, we must make a number of choices to appropriately specify the kinematics of a joint.

1. *Choice of body fixed reference frame*—for each body i $(i = 0 \ldots n)$ a body fixed reference frame must be chosen. On bodies $i = 1 \ldots n$ the origin of the reference frame must coincide with the center of mass of the body, while for the ground body $(n = 0)$ any point can be chosen as origin.

2. *Choice of reference bodies*—in general, the choice is arbitrary. However, we must decide for each joint, which of the two coupled bodies is considered as the reference body and which one is the body in motion relative to the reference body in order to find an orientation for the edge that represents the joint in the system graph (see vector $\mathbf{c}_i$ in Figure 4-5).

3. *Choice of articulation points*—for each joint j $(j = 1 \ldots n)$ an articulation point on each of the corresponding bodies is defined. To isolate the effects of reaction forces on each connected body, the joint is broken up as illustrated in Figure 4-5, where $\mathbf{r}_i$ and $\mathbf{r}_j$ are two edges in the terminal graph of the joint. Their terminal equations define position vectors in a local reference frame that identify the position of the articulation points A_k. Edge $\mathbf{c}_i$ represents the joint. This edge is directed towards the body whose motion is relative to the reference body. The terminal equations associated with edge $\mathbf{c}_i$ are grouped into the (r_c, F_c) set of equations. The set $r_c = (x, \omega)$ contains the terminal equations for relative displacement and angular velocity of the joint. The set $F_c = (F, T)$ contains constraint forrce/torque terminal equations.

Every kinematic joint imposes forces and/or torques as well as kinematic limits on the relative motion of the connected bodies. The nature of these forces and torques provides a classification scheme of different kinematic joints [69] among which we include the following.

Revolute joint. The revolute joint allows no relative displacement between the connection points but imposes a rotational constraint such that rotation can occur only about one axis:

$$\begin{aligned} x &= 0 \qquad T \cdot \hat{u} = 0 \\ \omega &= \dot{\phi}\hat{u} \end{aligned}$$

Equation 4-3

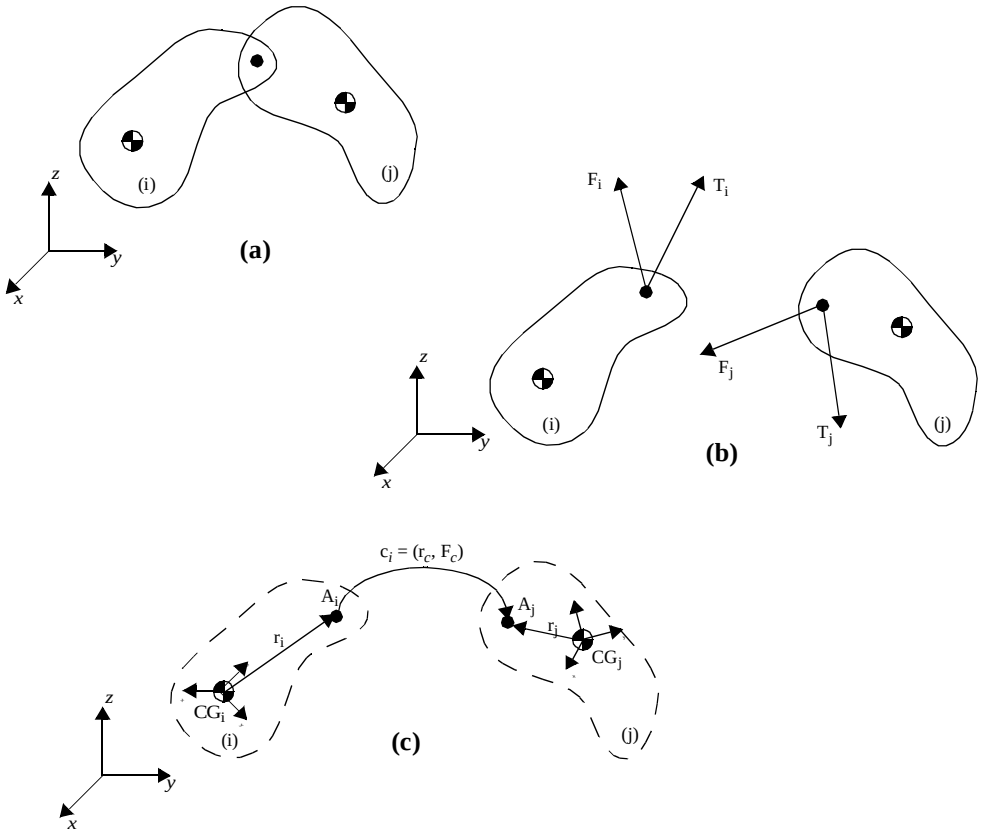


Figure 4-5. Joint description

Where $\hat{u}$ is the unit vector along the axis of rotation, and ϕ is the angular displacement about that axis.

Prismatic joint. The prismatic joint allows no relative rotation between connected bodies but permits relative translation along a fixed vector on one of the bodies. The terminal equations are the following:

$$r = s\hat{u} \qquad F \cdot \hat{u} = 0$$
$$\omega = 0$$

Equation 4-4

where $\hat{u}$ is a unit vector along the sliding axis, ω is the angular velocity associated with the joint and *s* is the translational displacement of the moving body relative to the reference body.

Fixed joint. The fixed joint allows neither relative rotation nor relative translation of the connection points. This joint specifies a rigid connection between two bodies:

$$x = 0$$
$$\omega = 0$$

Equation 4-5

4.3.1.2 Forces

In Dynaflex, two types of forces are identified: those acting on a joint or on a body from an inertial reference frame (called *force driver element*) and those acting between bodies (represented by *spring-damper-actuator elements*).

The first type of force driver, called *position driver*, is represented by a vector departing from the origin of the inertial reference frame and directed towards the point of application of the force. The second type of force driver, or *joint driver*, is represented by spring-damper-actuator elements located parallel to the joint's degree of freedom. The definition of a joint force driver element acting parallel to a joint specifies three spring-damper-actuator elements, one for each degree of freedom the joint might have.

Forces acting between bodies are represented by spring-damper-actuator elements. Generally these elements appear together as shown in Figure 4-6, thus they are incorporated in a single spring-damper-actuator element having the following equations:

Let $\boldsymbol{r}_k$, $k \in \{i, j\}$ in Figure 4-6 be two edges of the terminal graph of the element. The terminal equations associated with edges $\mathbf{r}_k$ represent the positions of the attachment points $\boldsymbol{P}_k$ in the inertial reference frame. The length of the spring-damper-actuator element can be written as:

$$l = \left\| \boldsymbol{r}_j - \boldsymbol{r}_i \right\|$$

Equation 4-6

and the time rate of change of length is then

$$\dot{l} = \frac{(\boldsymbol{r}_j - \boldsymbol{r}_i) \cdot (\dot{\boldsymbol{r}}_j - \dot{\boldsymbol{r}}_i)}{\left\| \boldsymbol{r}_j - \boldsymbol{r}_i \right\|}$$

Equation 4-7

The force vector of the spring-damper-actuator is given by

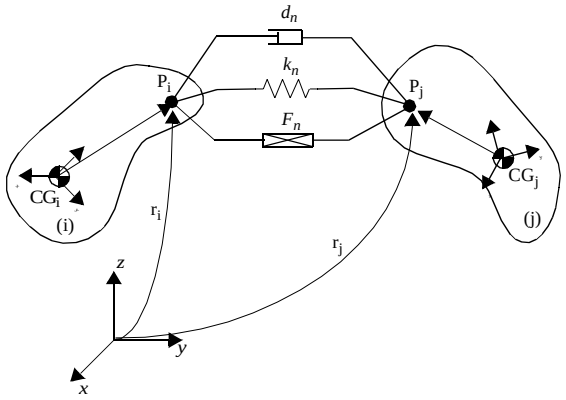


Figure 4-6. Spring-damper-actuator element

$$\mathbf{F} = [k(l - l_0) + d\dot{l} + F(t)]\boldsymbol{u}_{ij} \qquad \textbf{Equation 4-8}$$

where k, d, F are the spring constant, damping coefficient and actuator force respectively, l_0 is the undeformed spring length and $\boldsymbol{u}_{ij}$ is the unit vector that defines the direction of the element in space.

Similarly to the translational spring-damper-actuator element, torsional spring-damper-actuator elements may be defined between adjacent bodies that are connected by a revolute joint. For such element the net torque can be written as

$$\tau = [k_r(\theta - \theta_0) + d_r\dot{\theta} + T(t)]\boldsymbol{u}_{ij} \qquad \textbf{Equation 4-9}$$

Where k_r, d_r, T are the spring constant, damping coefficient and actuator torque respectively, θ_0 is the undeformed spring angular displacement and $\boldsymbol{u}_{ij}$ is the unit vector that defines the direction of the element in space.

4.3.1.3 Kinematic analysis of the 3D mechanism

The system graph of the mechanical system captures the topology of the mechanism. However, to have a complete model, geometric and inertial information must be added to the topology. Work related to kinematic and geometric analysis [126] allows us to automatically determine the instantaneous kinematic relationships between components in the mechanism. This geometric analysis further provides information about the origin of the

inertial frame, the center of mass of each body, the location of articulation points in each body, the type of joint, and the points of application of internal forces.

Using a geometric kernel (ACIS) and the kinematic analysis tools developed in our group [126], we are able to update the kinematic information in the mechanical system graph from changes made in the CAD model. In the future, the geometric kernel can be replaced by a commercial CAD environment. As a result, modifications to the geometry of a component in a CAD system could be automatically updated in the corresponding simulation model.

4.3.2 Synthesis of the mechanical system graph

The process for obtaining the graph representation suitable for Dynaflex consists of three steps. First, an *extended* system graph is generated. This step maps the geometry of the mechanism directly into a linear graph representing its topology. The second step identifies composite bodies consisting of rigidly connected subcomponents. In a final step, single bodies replace composite bodies reducing the system graph to a minimal graph with the same topological properties.

The generation of the extended system graph involves a direct translation of the kinematic information into the linear graph representation. The result of this translation is a system graph that includes all kinematic information including fixed joints and redundant joints. However, to avoid structural singularities or index problems, and to improve the efficiency of the symbolic computations in Dynaflex, we simplify this initial system graph by lumping all rigidly connected bodies into a single composite body.

Algorithm A. *(Synthesis of the mechanical system graph)*. This algorithm takes a kinematic description of a 3D mechanism and generates an extended system graph describing the topology of the system. Bodies in the system are identified by integer numbers from 0 to n where the 0*th* body is assumed to be the ground body. Modeling elements are represented by single edge terminal graphs.

A1. Define an inertial frame of reference. Assign a node in the system graph to the origin of the inertial frame. Call this node n_d.

A2. For each body i $(i = 0 \ldots n)$ in the system, assign a node n_i to the center of gravity CG_i.

A3. For each body i $(i = 0 \ldots n)$, add a *rigid body* element and a *force driver* element modeling the weight of the body from n_d to n_i.

A4. For each joint j $(j = 1 \ldots n)$ connecting bodies k and l, assign a node n_{ip} to an articulation point in body i $(i = k, l)$. Assign a *position vector* element to each of the connected bodies such that the element is oriented from n_i $(i = k, l)$ to n_{ip}. If joint j is a prismatic joint, a *sliding vector* element is assigned instead.

A5. For each joint j $(j = 1 \ldots n)$ assign a *joint* element oriented from n_{kp} to n_{lp} where body k is assumed to be the reference body. A restriction imposed by Dynaflex is that a prismatic joint be represented by two elements: a sliding vector element and a fixed joint. Thus if joint j is a prismatic joint, in addition to replacing the position vector element by a sliding vector element we also assign a fixed joint element properly oriented.

A6. Assign a position vector element oriented from n_d to n_0 to specify the position of the ground body with respect to the inertial frame.

Performing a depth-first traversal on the extended system graph identifies composite bodies. The algorithm explores all paths created by rigid connections and collects all bodies along the path into a single composite body. We can formally state the algorithm as follows:

Algorithm B. *(Composite body identification)*. The algorithm takes as an input the system graph and generates as output the set of composite bodies. The algorithm uses the following sets to keep track of all nodes in the graph: the set CLOSED, contains all nodes already visited. The set LUMPS contains all the composite bodies in the system. OPEN is a set containing all the nodes to-be-visited. ξ contains the bodies to be combined into the current composite, and SYSTEM is the set of CG nodes of the system graph.

B1. Set $CLOSED \leftarrow \varnothing$, $LUMPS \leftarrow \varnothing$

B2. While $SYSTEM \neq \varnothing$ do

B3. Set $cg \leftarrow first(SYSTEM)$, $\xi \leftarrow \{cg\}$,

B4. $OPEN \leftarrow sucessors(cg) \setminus CLOSED$,

$SYSTEM \leftarrow SYSTEM \setminus \{cg\}$

$CLOSED \leftarrow CLOSED \cup \{cg\}$

B5. While $OPEN \neq \varnothing$ do

B6. $cg \leftarrow first(OPEN)$,

B7. $\xi \leftarrow \xi \cup \{cg\}$,

$OPEN \leftarrow (OPEN \setminus \{cg\}) \cup (sucessors(cg) \setminus CLOSED)$

$SYSTEM \leftarrow SYSTEM \setminus \{cg\}$

$CLOSED \leftarrow CLOSED \cup \{cg\}$

B8. Continue B5.

B9. Set $LUMPS \leftarrow LUMPS \cup \{\xi\}$

B10. Continue B2.

B11. The algorithm terminates. We have checked all bodies in the system and have defined the composite bodies that must be created.

Algorithm B uses the predicate *successors*. Given a node in the system graph, *successors* returns the adjacent nodes if the path to these nodes is through a rigid connection. However, since we want to find the path of all fixed joints we require that adjacent nodes satisfy the condition that a node n_s is a successor of node n_p if and only if n_s is adjacent to an edge e of type *position vector* or type *fixed*, and the edge e is incident to node n_p.

The last stage in the synthesis of the system graph is to perform the reduction process that will combine the identified bodies into single composite bodies and remove redundant joints. To find redundant joints we first need to determine those bodies that are constrained

by more than one joint. We do this by detecting loops in the system graph where the bodies in the loop appear more than once. Once we have found such bodies, the next step is to decide wether the joints that constraint these bodies are redundant or not. Revolute joints are redundant if and only if their axes are colinear. This would result in an over constrained mechanism for which one of the two joints can be discarded for analysis purposes. If their axes are not colinear, either because of numerical inaccuracies or because of design intent, the configuration represents an overconstrained body for which we cannot discard any of the joints for analysis purposes. In this event, the algorithm reports the problem to the user stating that the system is fully constrained. The reduction algorithm can be stated as follows:

Algorithm C. *(Graph reduction)*. All bodies $i \in \xi$ are lumped together and all their corresponding elements are replaced by one single *rigid body* element. This new element has an attribute that specifies what individual bodies are composing this compound body. This attribute is used to compute geometric and physical properties of the compound object. The input to the algorithm is the system graph and the set LUMPS computed by Algorithm B.

C1. Set $n \leftarrow |LUMPS|$. For $1 \leq k \leq n$ replace the edges in the system graph which are associated with the bodies in $LUMPS_k$ for single edges that represent the element $LUMPS_k$.

C2. Set $joint \leftarrow articulatedbodies(LUMPS)$. In this step we obtain a set of equivalent joints formed by the new bodies in the new graph. This process takes the old joint definitions and maps the constrained bodies to the lumped bodies, thus creating new joint definitions in terms of lumped bodies.

C3. Step C2 may reveal that any two bodies are connected by more than one joint. Since this is not allowed we must remove redundant joints. We decide which joints to remove as follows: if any two bodies are connected by more than one joint and at most one joint has an associated motor, we keep the actuated joint and remove the rest. If no motors are associated with any redundant joint, we arbitrarily pick one. If

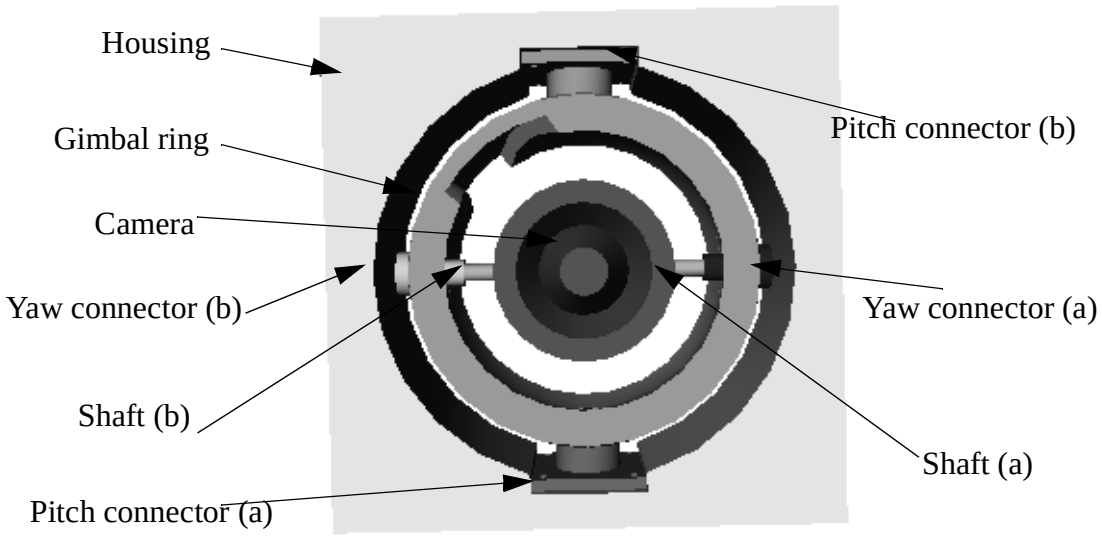


Figure 4-7. Missile seeker

there are more than one actuated joints this is an error condition and the system is overconstrained and cannot be solved. Set $m \leftarrow |joint|$.

C4. For each $joint_j$ $(j = 1 \ldots m)$ connecting bodies k and l, assign a node n_{ip} to an articulation point in body i $(i = k, l)$. Assign a position vector element to each of the connected bodies such that the element is oriented from n_i $(i = k, l)$ to n_{ip}. If $joint_j$ is a prismatic joint, the sliding vector element is assigned instead.

C5. For each $joint_j$ $(j = 1 \ldots m)$ assign a joint element oriented from n_{kp} to n_{lp} where body k is assumed to be the reference body. If $joint_j$ is a prismatic joint, in addition to replacing the position vector element by a sliding vector element we also assign a fixed joint element properly oriented.

As an example of how these steps are followed consider the design of a missile seeker shown in Figure 4-7.

This design contains 9 bodies: housing, gimbal ring, camera, pitch connector (2) yaw connector (2), shaft (2). A kinematic description of the system reveals that there are a number of bodies that may be combined to form composites (Table 1).

Table 1. Kinematic description for the seeker system

Type of Joint	Reference body	Secondary body
FIXED	housing	pitch connector (a)
FIXED	housing	pitch connector (b)
REVOLUTE*	pitch connector (a)	gimbal ring
REVOLUTE	pitch connector (b)	gimbal ring
FIXED	gimbal ring	yaw connector (a)
REVOLUTE*	yaw connector (a)	shaft (a)
FIXED	gimbal ring	yaw connector (b)
REVOUTE	yaw connector (b)	shaft (b)
FIXED	shaft (a)	camera
FIXED	shaft (b)	camera

From the kinematic description shown in Table 1, the first stage of our derivation generates an extended system graph shown in Figure 4-8. Secondly, Algorithm B identifies the composites listed in Table 2.

Table 2. Composite bodies found by Algorithm B

BODY_1	shaft (b)	camera	shaft (a)
BODY_2	housing	pitch connector (b)	pitch connector (a)
BODY_3	gimbal ring	yaw connector (b)	yaw connector (a)

Finally, the reduction stage yields the following kinematic relations:

Table 3. Kinematic description for the composite bodies in the seeker

Type of Joint	Reference body	Secondary body
REVOLUTE	BODY_2	BODY_3
REVOLUTE*	BODY_2	BODY_3
REVOLUTE	BODY_3	BODY_1
REVOLUTE*	BODY_3	BODY_1

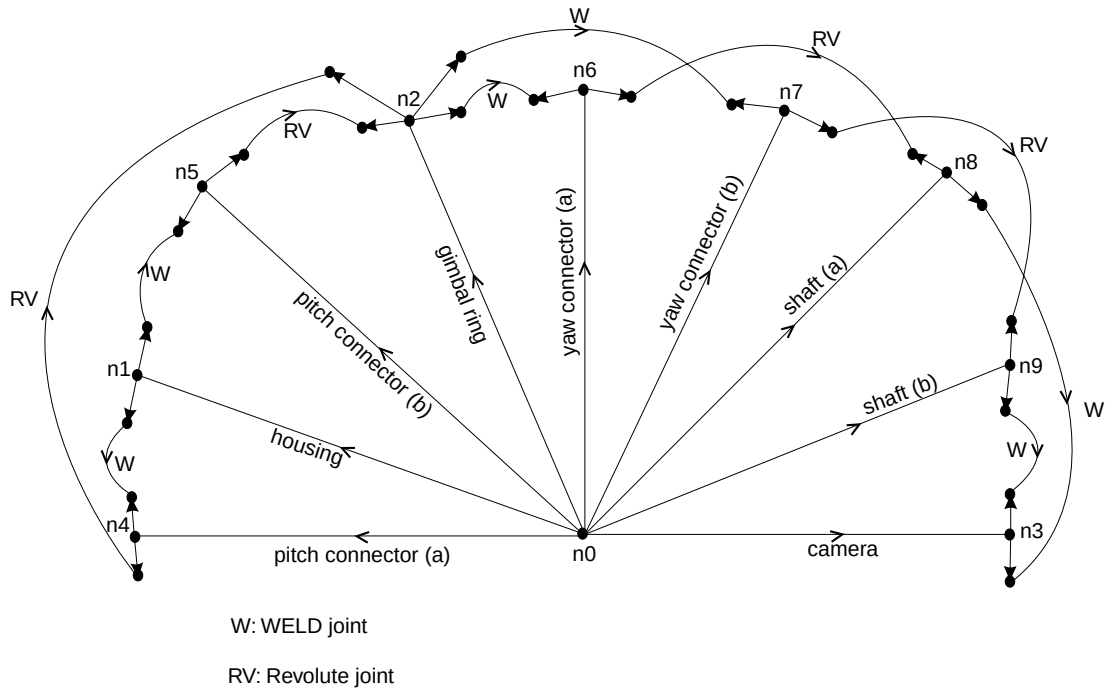


Figure 4-8. Extended mechanical system graph. Only joint and body elements are shown for clarity

Notice that there are two revolute joints per pair of composite bodies. Kinematic analysis reveals that the rotation axes of each pair of joints coincide. For the Dynaflex analysis, one of the two points is removed to avoid concluding over constrained kinematics; only the joints marked with an asterisk are considered. At the end of the reduction process, we obtain the reduced mechanical system graph shown in Figure 4-9.

In addition to the inertial parameters and the kinematic properties, dynamic elements such as external forces, and forces acting between any two bodies are included in Dynaflex representation. For this example, only two force elements are introduced: e8 and e12, which are the result of the motors built into the corresponding joints. Furthermore, we introduce gravity forces acting on the bodies at their center of mass (e1, e3, e5) representing the weight of BODY_2, BODY_3, and BODY_1, respectively.

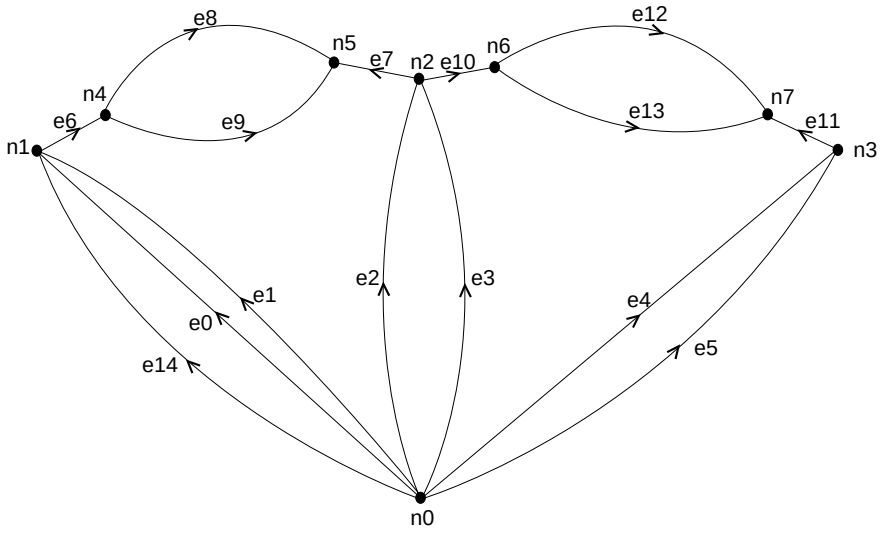


Figure 4-9. Reduced mechanical system graph

4.4 Summary

In this chapter, we defined the algorithms to automatically synthesize the system graph of a mechatronic system. In general, the system graph is a non-connected graph in which each connected component corresponds to an energy domain involved in the system.

It was shown that the process of generating the non-mechanical system graph involves the topological modification of the system graph formed as a non-connected graph where each connected component corresponds to a terminal graph. For the mechanical system graph, it was shown that the extended graph is a direct translation of the kinematic information into the graph representation. However, to avoid structural singularities or index problems, this graph is further reduced to combine rigid bodies connected by fixed joints. As a last step, the articulated redundant joints are removed from the graph to prevent Dynaflex from considering the system to be overconstrained.

Chapter 5 Automatic Generation of System-level Dynamic Equations

5.1 Introduction

A modeling environment capable of generating the dynamic equations of the system requires the ability to determine the causality of the equations. Recall that the basic condition for composability is that the equations in the model be non-causal. Causality is defined once the complete topology of the system is known.

Converting the system of non-causal equations into a system of causal assignments can be divided in two major steps: first, finding the causal directions for all the equations and second, sorting the equations into a computationally feasible order of evaluation.

Care must be taken when the system of equations includes software components; software components have fixed causality which imposes restrictions when mixing symbolic equations and software components.

Finding a normal tree in the system graph causally orients the system of equations. Sorting of the equations is performed by a modification of the classic *Block Lower Triangular* algorithm to account for software components in the system of equations.

5.2 Algebraic properties of linear graphs

The terminal equations are insufficient to describe the mechatronic system completely. An additional *e* equations are required to make the problem well-posed: *2e* equations in *2e* unknowns. These additional equations are derived from the connectivity of the components given by the topology of the system graph.

In any connected graph $\mathbf{G}$, a tree $\mathbf{T}$ is a connected subgraph that contains all the nodes of $\mathbf{G}$ but no loops. The edges that are not part of the tree form a subgraph $\bar{\mathbf{T}}$ called cotree. For a graph with *e* edges and *v* vertices, there are exactly $v - 1$ branches (the edges of T). Consequently, the number of chords (the edges of $\bar{\mathbf{T}}$) in the cotree equals $e - v + 1$.

If we add any chord between any two vertices in the tree, we establish a circuit. Since in a connected graph there are $e - v + 1$ chords for a given tree $\mathbf{T}$, there are as many unique circuits defined by the chords of $\mathbf{T}$. These are called *f-circuits* (fundamental circuits) of the graph and an element of this set is called *f-circuit*.

The branches of $\mathbf{T}$ provide the dual of the f-circuits: the *f-cutsets* (fundamental system of cut-sets). A cut-set of a connected graph is the set of edges such that the removal of these edges from the graph leaves the graph partitioned in exactly two connected components. The f-cutsets with respect to a tree $\mathbf{T}$ of a connected graph $\mathbf{G}$ is the set of $v - 1$ cut-sets in which each cut-set includes exactly one branch of $\mathbf{T}$. An element of the f-cutsets is called a *f-cutset*.

We now regard the system graph as two subgraphs, a *spanning tree*[1] $\mathbf{T}$ and a *cotree*. We can identify $v - 1$ pairs of terminal variables (v_T, f_T) with the *branches* of the spanning tree and $e - v + 1$ pairs of terminal variables (v_C, f_C) with the *chords* of the cotree. If the system graph is divided in two non-connected subgraphs by a cut including exactly one

1. A spanning tree for $\mathbf{G}$ is a free tree that connects all vertices in $\mathbf{G}$.

branch of $\mathbf{T}$ and some chords, the cut is unique (i.e., an f-cutset). It is clear that for a tree $\mathbf{T}$ with $v-1$ branches, there will be as many unique cuts; i.e., the f-cutsets of the system graph with respect to $\mathbf{T}$.

The connectivity relations of the system graph can be completely specified by means of the augmented incidence matrix, denoted $\overline{\mathbf{A}}$. The incidence matrix is a square matrix of dimensions $v \times e$, where *v* are the vertices and *e* are the edges of the system graph, and it contains information both about the orientation of edges in the graph and how they are joined to form nodes (See "Matrix representations of linear graphs" on page 181).

In general, for a graph with *p* connected components, the incidence matrix is a *direct sum*. A matrix $\mathbf{M}$ is said to be a direct sum of rectangular submatrices $\mathbf{M}_1, \mathbf{M}_2, \ldots \mathbf{M}_p$ if for any $\mathbf{M}_k$ in $\mathbf{M}$ no nonzero element lies in a row or column of $\mathbf{M}$ associated with any of the other submatrices [143]. The existence of a direct sum in a matrix always indicates the existence of subgraphs; therefore, the $\mathbf{M}_k$ matrices can be regarded as the incidence matrices of each of the *p* connected components.

Consider the incidence matrix $\overline{\mathbf{A}}$ of a connected graph $(p = 1)$ $\mathbf{G}$. Since the sum of all rows of $\overline{\mathbf{A}}$ equals zero, its rows are linearly dependent. Removing any row from $\overline{\mathbf{A}}$ will leave $v-1$ linearly independent rows. We call this new matrix the reduced incidence matrix, denoted $\mathbf{A}$. From graph theory [121], we know that if $\mathbf{T}$ is a tree of a connected graph $\mathbf{G}$, the $v-1$ columns of $\mathbf{A}$ that correspond to the branches of the tree $\mathbf{T}$ constitute a nonsingular matrix. Thus if a tree is chosen and the columns of $\mathbf{A}$ are properly arranged, the matrix $\mathbf{A}$ can be partitioned into the $(v-1)\times(v-1)$ submatrix $\mathbf{A}_T$, referring to the branches of the tree only, and the $(v-1)\times(e-v+1)$ submatrix $\mathbf{A}_C$, referring to the chords or to the cotree.

$$\mathbf{A} = \left[\mathbf{A}_T\ \mathbf{A}_C\right] \qquad \textbf{Equation 5-1}$$

Two new matrices can be defined to describe the topology of the graph. The *fundamental circuit matrix* (designated $\mathbf{B}$) captures the connectivity relations between circuits and edges, and the *fundamental cut-set matrix* designated $\mathbf{Q}$. Matrix $\mathbf{Q}$ captures the connectivity between cut-sets and edges.

If the columns of matrix $\mathbf{B}$ are properly arranged matrix $\mathbf{B}$ can be partitioned into the $(e-v+1)\times(v-1)$ submatrix $\mathbf{B}_T$ referring to the branches of the tree and the $(e-v+1)\times(e-v+1)$ submatrix $\mathbf{B}_C$ referring to the chords of the cotree. However, since each chord appears exactly once in any given f-circuit in the positive sense, the matrix $\mathbf{B}_C = \mathbf{U}_C$; i.e., a unit matrix. Then we can write

$$\mathbf{B} = \left[\mathbf{B}_T\ \mathbf{U}_C\right] \qquad \textbf{Equation 5-2}$$

When the columns of matrix $\mathbf{A}$ are properly arranged such that the first $v-1$ columns of $\mathbf{A}$ are in direct correspondence with the branches of some tree $\mathbf{T}$ of a graph $\mathbf{G}$, an equivalent matrix $\mathbf{Q}$ can be derived from $\mathbf{A}$ by applying row operations to $\mathbf{A}$.

Matrix $\mathbf{Q}$ represents the *fundamental system of cut-sets* with respect to the tree $\mathbf{T}$, and includes the $v-1$ cut-sets of $\mathbf{G}$ in which each cut-set includes only one branch of $\mathbf{T}$. Then

$$\mathbf{Q} = \left[\mathbf{U}_T\ \mathbf{Q}_C\right] \qquad \textbf{Equation 5-3}$$

In Appendix A we show that the incidence matrix and the circuit matrix capture the algebraic properties of the associated linear graph. Specifically, Theorem A-I shows that the across variables associated with the chords of a tree $\mathbf{T}$ of the system graph can be expressed as linear combinations of the across variables associated with the branches; this equations are referred to as *fundamental circuit equations*. Similarly, the through variables associated with the branches of the tree $\mathbf{T}$ of the system graph can be expressed as linear combinations of the through variables associated with the chords. This equations are referred to as *fundamental cut-set equations*.

The $e-v+p$ fundamental circuit equations and the $v-p$ cut-set equations of a system graph $\mathbf{G}$ with e edges, v vertices and p connected components are referred to as the *constraint equations* of the system. These equations are, given by:

$$\mathbf{v}_c(t) = -\mathbf{B}_T\mathbf{v}_b(t) \qquad \textbf{Equation 5-4}$$

$$\mathbf{f}_b(t) = -\mathbf{Q}_C\mathbf{f}_c(t) \qquad \textbf{Equation 5-5}$$

Terminal equations together with the constraint equations of the system constitute the mathematical model of the system. Terminal equations represent a model of the physical

characteristics of the component while constraint equations describe the topological relationships between components. It must be emphasized here that the model of each component in a system includes both the terminal equations and the associated terminal graph: the terminal graph only provides the topological structure of the system component while the terminal equations provide the mathematical model of the basic operation of the component. Together the two create a complete model that can be used in larger systems that are more complex.

Structural analysis of the system equations is accomplished in two stages [32]. First, the causality of each element in the system graph is determined. This procedure is called *selection of the normal tree*. Once a normal tree has been selected, the system equations can be derived directly from the tree. The second stage deals with the ordering of the equations and software components into a correct sequence of evaluation.

5.3 Selection of the normal tree

Once a mechatronic system has been described as a system graph, the dynamic equations can be derived from the graph without the need to consider the underlying physics in each of the energy domains. The system equations can be derived by simultaneously considering the *e* terminal equations and the *e* independent topological constraints (cut-set and circuit equations). The remaining questions that we will address in this section are, which topological constraints need to be considered, and which of the two system variables (across or through) should be the independent variable in each of the *e* terminal equations? Both of these questions are answered in the normal tree selection algorithm presented in this section.

The terminal equations plus any independent set of *e* constraint equations unambiguously define the dynamics of the system. However, before these equations can be numerically solved they must be expressed in state space form in which the derivatives of a state *x* are expressed as explicit functions of the states and time:

$$\dot{x} = f(x, t) \qquad \textbf{Equation 5-6}$$

Expressing the equations of the system in this form implies using the smallest possible number of equations (equal to the order of the system) and requires expressing the high order derivatives as a function of low order derivatives of state variables in each equation.

This can be accomplished in the following way. Let us divide the system variables into two groups: primary variables and secondary variables—one of each for every edge. Assume now that in the terminal equation of an edge, the highest order derivative of the primary variable *p* is expressed as a function of the secondary variable, *s*:

$$p^{(n)} = f(s) \qquad \textbf{Equation 5-7}$$

On the other hand, assume that in the constraint equations the secondary variables are expressed as a function of the primary variables:

$$s = g(p) \qquad \textbf{Equation 5-8}$$

Then, by substituting the constraint equations (5-8) into the terminal equations (5-7), we get a minimal set of dynamic equations of the form:

$$p^{(n)} = f(g(p)) \qquad \textbf{Equation 5-9}$$

which is exactly the desired state-space representation.

The final step in the derivation of our approach is the selection of the primary and secondary variables. According to the fundamental circuit equations (5-4) and the fundamental cut-set equations (5-5) the dependent variables in the constraint equations are the through variables in the branches of the tree and the across variables in the chords of the cotree:

$$\begin{aligned} \mathbf{f}_T &= -\mathbf{Q}_C\mathbf{f}_C \\ \mathbf{v}_C &= -\mathbf{B}_T\mathbf{v}_T \end{aligned} \qquad \textbf{Equation 5-10}$$

From equations (5-8) and (5-10), we can identify primary variables with the set of $v - p$ across variables associated with the branches of a forest and the set of $e - v + p$ through variables associated with the chords of a coforest. Similarly, the dependent variables in equation (5-10) are identified as s*econdary variables* of the system graph.

Based on the selection of primary and secondary variables, we can obtain dynamic equations of the form (5-9). This is achieved by selecting a tree on the system graph such that

the following two conditions are satisfied: (1) the highest order derivatives of as many primary variables as possible appear in the terminal equations as functions of secondary variables and low order derivatives of primary variables, and (2) the terminal equations contain as few derivatives of secondary variables as possible. The tree that satisfies these two conditions is called a *normal tree* of the system graph.

The form of the terminal equation is used to guide the selection process as follows. If an algebraic terminal equation can be inverted, one may classify the corresponding edge as either a branch or a chord. This implies no preferred causality on that element. If the component is a driver, the corresponding edge will be classified depending of the type of driver. Across drivers are confined always to the tree thus the corresponding edge will be a branch. Through drivers are always confined to the cotree and the corresponding edge will be a chord. That is, drivers have a predefined causality that cannot be changed: recall that the terminal equation for a driver is an explicit function of time. Then if an across driver has a defining function $x(t) = f(t)$ the primary variable x is completely specified for all t and we cannot solve otherwise.

If all through drivers cannot be included in the cotree then they form a cut-set, similarly if all across drivers cannot be included in the tree, they form a circuit. Cut-sets of through drivers or circuits of across drivers in general violate the vertex and circuit theorems (See Appendix A). A system graph without cut-sets of through drivers or circuits of across drivers is a consistent system.

For terminal equations containing derivative terms of the terminal variables, we are interested in writing them as explicit functions of low-order derivatives and time. To achieve this we examine which time derivatives of the terminal variables occur in the terminal equation. If the highest order derivative of an across terminal variable appears in the equation, the corresponding edge is assigned to the tree if the topology of the system allows it. If on the other hand the highest order derivative of a through terminal variable appears, the edge is assigned to the cotree. This preferred assignment of causality results in a set of ordinary differential equations written in canonical form.

Preferred causality, however, is not always achievable. This may be because of the violation of the fundamental property of a tree of a system graph; if introducing a new edge into the tree creates a cycle the edge cannot become a branch and therefore it must be confined to the cotree. This implies that the associated terminal equation cannot be written in canonical form, which means that the system of differential equations is structurally singular; i.e., it has lost one degree of freedom. A common example that reflects this is that of two capacitors in parallel. In that system, we have two differential equations but only one degree of freedom.

Elements in the system graph are classified as follows. If in the terminal equation of the element the term $d^n v_i(t)/dt$ occurs, element *i* is called *nth-order accumulator element*. Similarly if $d^n f_i(t)/dt$ occurs in the terminal equation, element *i* is called *nth-order delay element*. If element *i* does not correspond to either an across driver or a through driver, and neither $d^n v_i(t)/dt$ nor $d^n f_i(t)/dt$ occur, we call the element *i* an *algebraic element.*

This classification of elements is useful only for two-terminal elements. Multiterminal components that have terminal equations, which can be written in the form given in Equation 5-11 (i.e., ideal transducers) present topological restrictions that must be taken into account [109].

$$\begin{bmatrix} h_{111}\frac{d}{dt} + h_{110} & 0 \\ 0 & h_{221}\frac{d}{dt} + h_{220} \end{bmatrix} \begin{bmatrix} v_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 0 & k_{121}\frac{d}{dt} + k_{120} \\ k_{211}\frac{d}{dt} + k_{210} & 0 \end{bmatrix} \begin{bmatrix} f_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} g_1(t) \\ g_2(t) \end{bmatrix}$$

Equation 5-11

The issue here is to decide whether the element that corresponds to v_1 and f_1 or the element that corresponds to v_2 and f_2 should be confined to the forest. Both elements cannot be on the forest because Equation 5-11 expresses v_1 in terms of v_2 and f_1 in terms of f_2, and we must avoid doubly specifying across or through variables. Since the topological restrictions on elements whose terminal equations have this form are very similar to those regarding

the locations of across and through drivers, we classify the corresponding elements as *generalized drivers*.

The choice of what element is confined to the forest is arbitrary since in principle both can be confined to the forest. However, a decision is made by looking at the global topology of the system to consider the interactions between the multiterminal component and other elements in the system graph. To illustrate this, consider element e_1 and element e_2 composing a multiterminal component with terminal equations of the form of Equation 5-11 (Figure 5-1). Assume that element e_1 is connected to an across driver. Expanding Equation 5-11 we observe that the highest derivative of element e_1 is the derivative of its across variable (v_1); Therefore element e_1 can be considered an accumulator element.

$$h_{111}\frac{dv_1}{dt} + h_{110}v_1 = k_{121}\frac{dv_2}{dt} + k_{120}v_2 + g_1(t)$$
$$h_{221}\frac{df_2}{dt} + h_{220}f_2 = k_{211}\frac{df_1}{dt} + k_{210}f_1 + g_2(t)$$

Equation 5-12

Consequently, element e_1 should go into the tree such that its terminal equation expresses the derivative of the across variable (v_1) as a function of the exogenous variables. However, since the across driver must go to the tree, adding element e_1 to the tree creates a loop, as is illustrated in Figure 5-1, which violates the tree property rendering the system graph inconsistent. Therefore, element e_1 must go to the cotree. Once we decided that element e_1 must go into the cotree, its complementary element (i.e., element e_2) must go into the forest. The global topological structure of the system graph constrained element e_1 to be confined into the coforest. Had we only considered the local topology, element e_1 could have been assigned to the tree. However, from the global perspective that choice is not valid.

Multiterminal components for which the terminal equations cannot be written in the form of Equation 5-11 (i.e., non-ideal transducers) need to be classified following the rules presented in this section to classify two-terminal elements. That is, we must consider the form of each terminal equation based only on the terminal variables associated with the element.

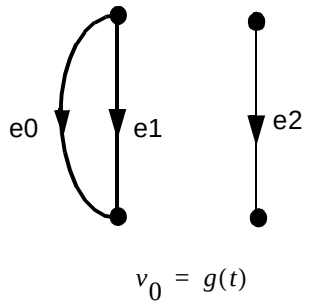


Figure 5-1. Assignment of elements in a multiterminal component.

Each element has two variables associated with it, namely $v_i(t)$, and $f_i(t)$. For each terminal equation, we look for the highest order derivative of the terminal variables and classify the element accordingly. If there are no derivative terms of the terminal variables, we call this element an algebraic element. If on the other hand, the terminal equation associated with the element can be written such that a terminal variable (either $v_i(t)$ or $f_i(t)$) can be written as an explicit function of the terminal variables $v_j(t), f_j(t) \therefore j \neq i$, we call this component a generalized driver. Therefore, an element *i* will be considered a generalized driver if either its across or through variable is completely defined by the complementary variables of the elements $j \neq i$ of the multiterminal component. In the case of multiterminal components coupling different energy domains, the terminal variable of element *i* is completely defined outside its energy domain (i.e., exogenous variables).

For example, consider the following terminal equations for a DC motor where the inductance *L* and the inertia of the rotor *J* have been omitted to create a simpler model:

$$\begin{bmatrix} v_1(t) \\ T_2(t) \end{bmatrix} = \begin{bmatrix} R_1 i_1(t) + k_b \dot{\theta}_2(t) \\ -\kappa_m i_1(t) + B_2 \dot{\theta}_2(t) \end{bmatrix} \qquad \textbf{Equation 5-13}$$

We classify the element associated with the electrical domain (first row in Equation 5-13) as algebraic since both terminal variables *v* and *i* appear in algebraic form. Similarly, the element associated with the mechanical domain (second row in Equation 5-13) is classified as a first order accumulator element since the angular velocity appears in the equation. If on the other hand, we had considered $\dot{\theta}_2$ in the equation for the electrical domain, we would have concluded that the element is a first order accumulator. However, in this case we

would be saying that the equation of the electrical domain could be used to compute the angular velocity of the rotor, but we also have a way of computing the angular velocity of the rotor by using the terminal equation of the mechanical domain. This results in a double specification of the angular velocity of the rotor:

$$k_b(T_2(t) + \kappa_m i_1(t)) = B_2(v_1(t) - R_1 i_1(t)) \qquad \textbf{Equation 5-14}$$

Giving a single equation in three unknowns. Furthermore, if the damping factor B_2 is neglected we have

$$\begin{bmatrix} v_1(t) \\ T_2(t) \end{bmatrix} = \begin{bmatrix} R_1 i_1(t) + k_b \dot{\theta}_2(t) \\ -\kappa_m i_1(t) \end{bmatrix} \qquad \textbf{Equation 5-15}$$

which forces us to classify the element assigned to the mechanical domain as a generalized through driver since the torque is completely specified outside the mechanical domain.

This taxonomy of elements is used in guiding the selection of the normal tree. The following section presents an extension to the method presented by Roe to find a normal tree. This suffers from the problem of finding and comparing subgraphs in larger graphs (i.e., subgraph isomorphism) making it excessively complex. The subgraph isomorphism problem is an important problem in complexity theory and it is known to be NP-complete [128]. In Section 5.3.2, we present an algorithm to find the normal tree based on the minimum cost spanning tree (MCT) algorithm. The total running time of MCT algorithm is $O(n_e \log_2 n_e)$ where n_e is the number of edges in the system graph [27].

5.3.1 Extension to Roe's method

The approach that follows to find a normal tree is an extension of Roe's method to find a normal tree. The original algorithm can only accept first-order derivative elements and algebraic elements. We have extended this algorithm to work with higher-order derivative elements as well as with multiterminal components.

1. We start by selecting a subgraph $\mathbf{G}_1$ of $\mathbf{G}$ consisting of all across drivers and generalized across drivers. Select a tree $\mathbf{T}_1$ in $\mathbf{G}_1$. Note that for a solution to exist, $\mathbf{T}_1 = \mathbf{G}_1$ since it is assumed $\mathbf{G}_1$ contains no circuits.

2. Consider a subgraph $\mathbf{G}_2$ of $\mathbf{G}$ containing all elements of $\mathbf{G}_1$ and all *nth* order accumulator elements. Select a tree $\mathbf{T}_2$ of $\mathbf{G}_2$ containing $\mathbf{T}_1$. Cotree $\mathbf{T}_2$' contains at most *nth* order across elements, i.e., those that cannot be included in the tree.

3. Consider subgraphs $\mathbf{G}_3, \ldots, \mathbf{G}_{n_\alpha}$ where $\mathbf{G}_3$ includes *(n-1)-th* order accumulator elements, $\mathbf{G}_4$ includes *(n-2)-th* order accumulator elements and so on until $\mathbf{G}_{n_\alpha}$ includes first order accumulator elements. Subgraphs $\mathbf{G}_3, \ldots, \mathbf{G}_{n_\alpha}$ are selected such that $\mathbf{G}_1 \subset \mathbf{G}_2 \subset \mathbf{G}_3 \subset \ldots \subset \mathbf{G}_{n_\alpha}$. Select trees such that $\mathbf{T}_1 \subset \mathbf{T}_2 \subset \mathbf{T}_3 \subset \ldots \subset \mathbf{T}_{n_\alpha}$

4. Consider a subgraph $\mathbf{G}_{(n_\alpha + 1)}$ containing all elements of $\mathbf{G}_{n_\alpha}$ and all dissipative elements. Select a tree $\mathbf{T}_{(n_\alpha + 1)}$ such that $\mathbf{T}_{n_\alpha} \subset \mathbf{T}_{(n_\alpha + 1)}$.

5. Consider subgraphs $\mathbf{G}_{(n_\alpha + 2)}, \ldots, \mathbf{G}_{(n_\alpha + n_\delta)}$ such that $\mathbf{G}_{(n_\alpha + 1)} \subset \mathbf{G}_{(n_\alpha + 2)}$ and also includes all *nth* order delay elements, $\mathbf{G}_{(n_\alpha + 2)} \subset \mathbf{G}_{(n_\alpha + 3)}$ and also includes all *(n-1)-th* order delay elements, and $\mathbf{G}_{(n_\alpha + n_\delta - 1)} \subset \mathbf{G}_{(n_\alpha + n_\delta)}$ which also includes all first order delay elements. Select trees in $\mathbf{G}_i$ such that $\mathbf{T}_{n_\alpha} \subset \mathbf{T}_{(n_\alpha + 1)} \subset \mathbf{T}_{(n_\alpha + 2)} \subset \ldots \subset \mathbf{T}_{(n_\alpha + n_\delta)}$. Since we prefer that all delay elements be in the coforest we have $\mathbf{T}_{(n_\alpha + 1)} = \mathbf{T}_{(n_\alpha + 2)} = \ldots = \mathbf{T}_{(n_\alpha + n_\delta)}$.

6. Consider a subgraph $\mathbf{G}_N$ that includes all elements of $\mathbf{G}_{(n_\alpha + n_\delta)}$ and all through drivers and generalized through drivers, i.e., the entire graph $\mathbf{G}$. Select a tree $\mathbf{T}_N$ which includes $\mathbf{T}_{(n_\alpha + n_\delta)}$. For a solution to exists, this graph must not contain cut-sets of through drivers. Also we require that the through drivers be in the coforest so we have $\mathbf{T}_{(n_\alpha + 1)} = \mathbf{T}_{(n_\alpha + 2)} = \ldots = \mathbf{T}_{(n_\alpha + n_\delta)} = \mathbf{T}_N$

This systematic selection process will produce a normal tree where a maximum number of accumulator elements are assigned to the tree and a maximum number of delay elements are assigned to the cotree. However, there may be the case that the topology of the system requires accumulator elements to be assigned to the cotree or delay elements to be assigned to the tree. This poses no problem in the selection of the normal tree. However in the derivation of the state equations this would mean that the system presents a structural singularity and that the system is of lower order. The effect that this has on the formulation of state equations as we will see in the next section is that not all the dynamic elements contribute to the state variable vector.

Although the algorithm presented here is suitable for automatization, it has some drawbacks. Finding a subgraph that contains a set of specified elements is a complex process that requires subgraph isomorphism. A simpler algorithm based on the classification criteria presented earlier, can be devised.

5.3.2 Selection of the normal tree: minimum cost spanning tree

The algorithm, based on the minimum-cost spanning tree [5, 27] presents an improvement over Roe's algorithm since there is no need to find subgraphs of the system graph. This algorithm is based on the fact that it is always possible to find a *spanning tree* in a weighted graph having minimum total cost [5].

Algorithm D. *(Normal tree)*. The normal tree of a system graph $\mathbf{G}$ is found by defining a real function w: $e \rightarrow \Re^+$ on the edges of $\mathbf{G}$ that computes the weight of the edges as follows:

Let κ_α and κ_τ represent the highest derivative order of all accumulator elements and all delay elements respectively, and $O{:}e \rightarrow \Re^+$ be a real function defined on the edges that computes the highest derivative order of the element associated with edge *e*. Next, classify the edges of $\mathbf{G}$ as follows. Let all across drivers and generalized across drivers belong to the class c^Δ, accumulator elements to the classes

$$c_i^\alpha = \{e_\alpha \,|\, O(e_\alpha) = \kappa_\alpha - i\} \qquad i = 0, \ldots, \kappa_\alpha - 1 \ , \qquad \textbf{Equation 5-16}$$

dissipative elements to class $c^\delta = \{e_\delta \,|\, O(e_\delta) = 0\}$, and delay elements to the classes

$$c_i^\tau = \{e_\tau \,|\, O(e_\tau) = \kappa_\tau - i\} \qquad i = 0, \ldots, \kappa_\tau - 1 \ . \qquad \textbf{Equation 5-17}$$

Finally, all through drivers and generalized through drivers will belong to class c^Φ. The weight functions *w* defined on the edges of $\mathbf{G}$ are chosen for each class such that

$$w^\Delta < w_0^\alpha < w_1^\alpha < \ldots < w_{\kappa_\alpha - 1}^\alpha < w^\delta < w_0^\tau < w_1^\tau \ldots < w_{\kappa_\tau - 1}^\tau < w^\Phi \qquad \textbf{Equation 5-18}$$

where w^Δ is the weight function associated with class c^Δ, w_i^α is the weight function associated with class c^α, w_i^δ is the weight function associated with class c^δ, w_i^τ is the weight function associated with class c^τ, and w_i^Φ is the weight function associated with class c^Φ.

In other words, the weight function *w* ranks the edges of $\mathbf{G}$ based on their respective classes. Any weight function that satisfies the ranking in equation (5-18) is admissible.

The next step in our approach is to find a *minimum cost spanning tree* that minimizes the total cost (weight) of the weights assigned to the branches of the tree. Aho *et. al.* [5] shows that it is always possible to find such a tree based on the following property. Let *V* be the set of vertices of $\mathbf{G}$ and *U* be a proper subset of *V*. If $e_{min} = (u, v)$ is an edge of lowest cost such that $u \in U$ and $v \in V - U$, then there is a minimum cost spanning tree that includes e_{min}. The proof of this property is outside the scope of this thesis but can be found in Aho *et. al.* [5]

The weight assignment is performed only one time and it has order $O(e)$ where *e* is the number of edges. The worst-case performance for the minimum-cost spanning tree algorithm is $O(n_v^{\,2})$ where n_v is the number of nodes in the graph. However, it can be made to run in $O(n_e \log_2 n_e)$ when efficient data-structures are used to represent the graph [27]. This algorithm will perform much better than that based on the selection of subgraphs of the system graph.

Once a normal tree has been selected, we can write *e* cut-set and circuit equations, and *e* terminal equations. The constraint equations together with the terminal equations constitute the set of equations for the complete solution of the mechatronic system excluding the mechanical energy domain. To completely specify the mechatronic system, these equations are combined with the dynamic equations for the 3D mechanism derived by Dynaflex. This new set of equations will be in general a set of differential algebraic equations (DAE), which constitute the set of equations for the complete system.

The MCT algorithm requires as a precondition that the graph is connected. In general the system graph $\mathbf{G}$ of a mechatronic system is non-connected. If that is the case, the problem would be to find a normal forest[1] and MCT could not be directly applied to $\mathbf{G}$. One solution approach is to apply the MCT algorithm to each component in $\mathbf{G}$ independently. This would

1. A normal forest is defined to be a set of normal trees for each connected subgraph of $\mathbf{G}$

find a normal tree for each connected component that would comprise the resulting normal forest.

5.4 Synthesis of system level dynamic equations

This section introduces an algorithm to automatically formulate the state space equations for a mechatronic system. Given a normal tree **T** of the system graph **G** derived by the algorithm presented in the previous section. The algorithm presented here is based on the branch-chord method to formulate state equations proposed by Roe [109].

Algorithm E. *(Equation synthesis)*. The algorithm takes as input a normal tree and returns the set of dynamic equations of the system. Once the normal tree is selected, the state variables of the system can be readily identified: they are the across variables corresponding to the branch *nth* order accumulator elements, and the through variables corresponding to the chord *nth* order delay elements.

E1. For all elements in **G** write the terminal equations such that they show primary variables as explicit functions of secondary variables and derivatives.

E2. Write the set of topological constraint equations for the system which are the cut-set and circuit equations derived from **G**. At this point we have $2e$ equations in $2e$ unknowns.

E3. Apply Algorithm F to reduce this number of equations to a set of *m nth* order differential equations where *m* equals the number of state variables in the system.

The reduction process is as follows:

Algorithm F. *(Reduction)*. The algorithm takes as input a set of *2e* equations and returns a minimal system of *m nth* order differential equations.

F1. Substitute the constraint equations into the terminal equations to eliminate branch through and chord across variables. This will leave the terminal equations as a function of primary variables only.

F2. Eliminate the algebraic relations from the resultant set of terminal equations. To easily identify which equations to use in the reduction process, the set of equations obtained in F1 is now partitioned into three subsets: set **A** contains all equations that include differential terms of primary variables. Set **B** contains equations which specify across and through drivers, and set **C** contains equations which can be solved algebraically for primary variables.

F3. The system of algebraic equations in set **C** is then solved for the primary variables.

F4. The solution to the system of algebraic equations together with the equations in set **B** are substituted in the set of differential equations. This will produce a set of differential equations as functions of primary variables (and their derivarites) of dynamic elements and the specified drivers.

F5. The differential equations obtained in F4 are then rearranged in canonical form to obtain the desired state space equations.

5.5 BLT form

The set of equations obtained in the previous section is causally oriented, however, no computational order of evaluation is given for the set. In this section, we explore a method to find a computational order for the set of dynamic equations, and show how to incorporate the software components in the solution of such equations.

We seek a form in which the system equations are given as a sequence of blocks of one or more equations. In this form, the state derivatives and the algebraic variables are unknowns. The equations are all first-order differential and algebraic equations. Each block can be solved as a separate problem assuming all previous blocks are solved. Therefore, what we seek is a *Block Lower Triangular* (BLT) order of the system equations [37, 38, 39, 130, 131, 138, 150]. The BLT form is a permutation of equations and unknown variables so that the *structural incidence matrix* of the system equations is triangular or block triangular. The incidence matrix is a square matrix where rows represent equations and columns represent

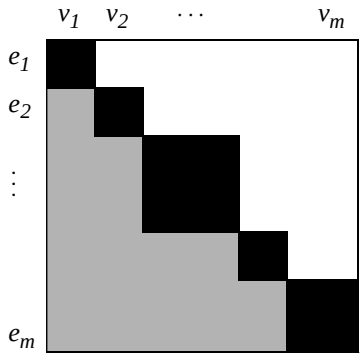


Figure 5-2. Illustration of a BLT form. Black regions indicate that the variable appears in the equation. White areas indicate that the variable does not appear in the equations while gray areas indicate that variables may or may not appear in the equation.

unknown variables. It indicates what variables appear in each equation (Figure 5-2). In this form, the state derivatives and the algebraic variables are regarded as unknowns.

The BLT form partitions the set of equations into k blocks of order n_k where n_k is the number of unknown variables to solve for in the subsystem of n_k equations. Equations and software components are sorted such that variables appearing in the equations within each block are either unknowns of the same block or variables solved from previous blocks.

To introduce software components into the incidence matrix we first define two functions associated with each software component (Figure 5-3):

$$\dot{\mathbf{x}} = f_d(t, \mathbf{x}, \mathbf{u})$$
$$\mathbf{y} = f_o(t, \mathbf{x}, \mathbf{u}) \qquad \textbf{Equation 5-19}$$

That is, a software component will have an operator that computes its outputs and one that computes its derivatives. The unknown variables of the software component are the inputs given in the input vector $\mathbf{u}$. If the output function $\mathbf{y} = f_0(t, \mathbf{x}, \mathbf{u})$ depends on the input vector $\mathbf{u}$, the software component is classified as *algebraic*; otherwise, it is classified as *non-algebraic*. This property determines when the software component should be scheduled for evaluation and will be used when we introduce the software component to the incidence matrix.

Before we generate the BLT form, the system equations are augmented with equations of the form:

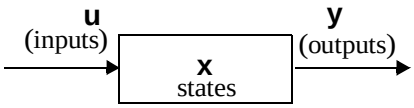


Figure 5-3. Software component.

$$y_i(t) = f_o(t, \mathbf{x}, \mathbf{u}) \qquad \textbf{Equation 5-20}$$

In addition, all occurrences of references to software components in the system equations are replaced by the y_i variables. In this way, every software component is executed at most once in the evaluation of the dynamic equations. The result of this substitution is a system of equations (5-21) where **v** represents the algebraic variables of the system.

$$\begin{aligned} \dot{x_i} &= f_{1,i}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \mathbf{v}) \\ v_i &= f_{2,i}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \mathbf{v}) \\ y_i &= f_{o,i}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{u}) \end{aligned} \qquad \textbf{Equation 5-21}$$

Where $\mathbf{u} = \left[\boldsymbol{v}\ \hat{\boldsymbol{x}}\right]$ and $\hat{\boldsymbol{x}}$ is the state vector of the dynamic equations. In this new system of equations, some of the equations are explicit assignments to state derivatives; others are assignments to algebraic variables. The third type of equations includes those introduced by the software components. For any equation of this kind, some of the unknown variables will appear as inputs to the function $f_{o,i}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{u})$. We call that subset of variables the *dependency set* for the software component.

Entries in the incidence matrix indicate whether a variable appears or not in a given equation. However, variables in the dependency set need to be treated in a special way. If the software component is algebraic, the variables in the dependency set are treated as ordinary variables, that is, the equation y_i related to the algebraic software component depends on its inputs. If on the other hand the software component is non-algebraic, the output function $f_{o,i}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{u})$ does not depend on the input variables and they are not considered in the incidence matrix. The BLT form orders the output functions of the software components only; the derivative function is evaluated at the integration stage.

Given the correct order of evaluation for the system equations that include software components, we can evaluate them numerically using the following iteration at each major and

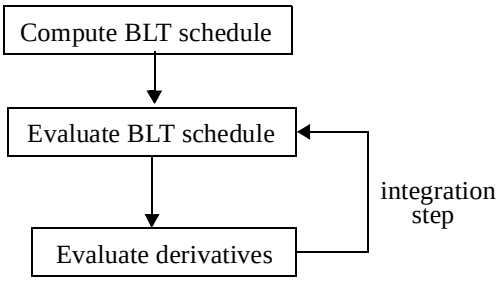


Figure 5-4. Single iteration to evaluate the system equations

minor integration step (Figure 5-4). We first evaluate the BLT that will compute the unknown derivatives and the values of unknown variables. In doing so, we evaluate the output operators of the software components. Finally, we proceed to evaluate the derivative operators of the software components. Since the variables that are computed by equations together with the output variables of the software components are known, all inputs to the derivative operators for the software components are ready and can be evaluated to complete the evaluation of the vector of derivatives. This is executed every minor/major integration step.

If the simulation kernel supports distributed processing, we may be able to evaluate the output operators of software components given in the BLT form or the derivative operators in parallel on different computers. This provides the ability to run legacy software even in different architectures, in remote computers.

5.6 Example: positioning system

To illustrate these concepts, we will use the following example: a positioning system. This system involves two energy domains: electrical and mechanical, and includes information technology components. The system graph consists of a digital controller, a DC motor, gear box, shafts, rotating inertia and angular position sensors (potentiometers) as illustrated in Figure 5-5.

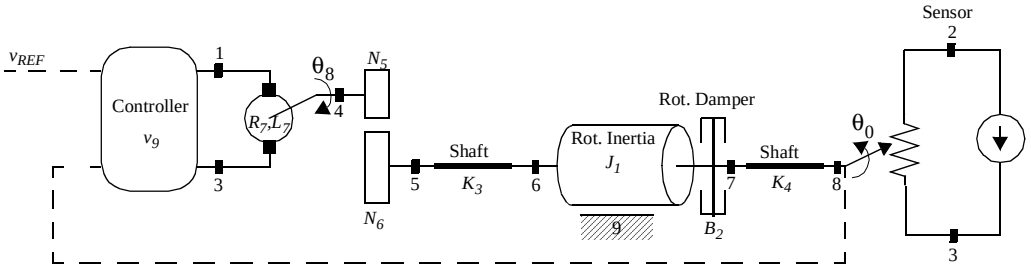


Figure 5-5. Positioning system

The terminal equations for the different components in the mechatronic system are given by:

Table 5-1. Terminal equations for the components in the positioning system

Domain	Component	Edge	Terminal equation
Electrical	Sensor	11	$v_{11}(t) = \frac{\theta_0(t) R_{11} i_{11}(t)}{\theta_m}$
	DC motor	7	$v_7(t) = k_m \dot{\theta}_8(t) + R_7 i_7(t) + L_7 \frac{d}{dt} i_7(t)$
	Current driver	10	$i_{10}(t) = I(t)$
	Variable driver	9	$v_9(t) = f_{out}(v_{11}(t), v_{REF}, t)$
Mechanical	Sensor	0	$T_0(t) = b_0 \dot{\theta}_0(t)$
	DC motor	8	$T_8(t) = b_8 \dot{\theta}_8(t) + J_8 \ddot{\theta}_8(t) - k_m i_7(t)$
	Gear train	5, 6	$\begin{bmatrix} T_5(t) \\ \theta_6(t) \end{bmatrix} = \begin{bmatrix} b_5 \frac{d}{dt} + J_5 \frac{d^2}{dt} & \frac{N_5}{N_6} \\ -\frac{N_5}{N_6} & 0 \end{bmatrix} \begin{bmatrix} \theta_5(t) \\ T_6(t) \end{bmatrix}$
	Inertia	1	$T_1(t) = J_1 \ddot{\theta}_1(t)$
	Shaft	3	$T_3(t) = K_3 \theta_3(t)$
		4	$T_4(t) = K_4 \theta_4(t)$
	Damper	2	$T_2(t) = b_2 \dot{\theta}_2(t)$

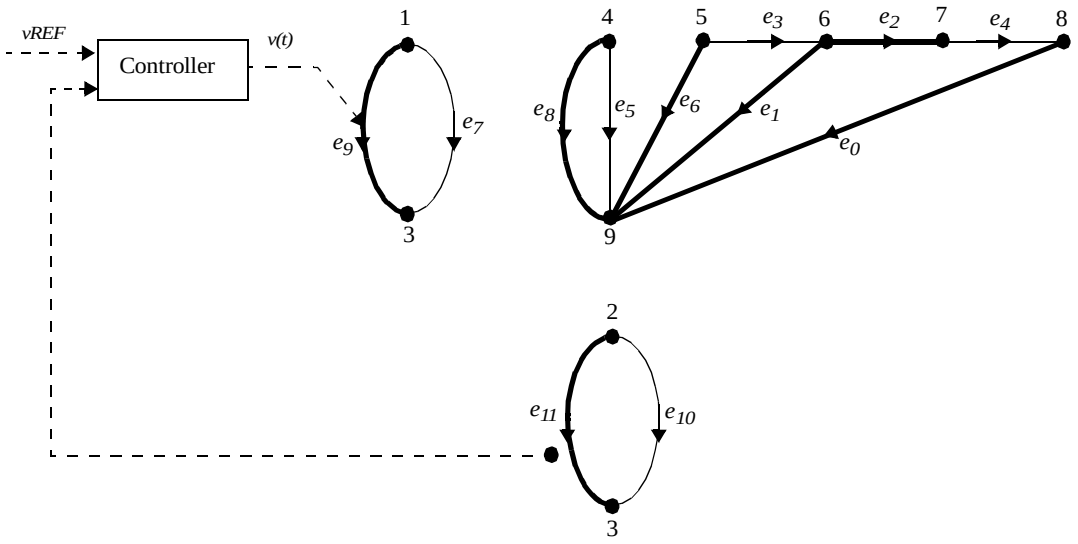


Figure 5-6. System graph for the positioning system.

The system graph for the mechatronic system is shown in Figure 5-6. Two coupling elements are used: the DC motor that couples the electrical domain to the mechanical domain, and the sensor that couples the mechanical domain back to the electrical domain. A signal-controlled voltage driver will model the controller. Following the classification rules defined before, the elements in the system graph fall in the following classes:

Table 5-2. Classification of elements in the positioning system

Edge	**Class**	**Weight**
0	first order accumulator	w_0
1	second order accumulator	w_1
2	first order accumulator	w_2
3	algebraic element	w_3
4	algebraic element	w_4
5	second order accumulator	w_5
6	generalized across driver	w_6
7	first order delay	w_7
8	second order accumulator	w_8

Table 5-2. Classification of elements in the positioning system

Edge	Class	Weight
9	signal-controlled across driver	w_9
10	through driver	w_{10}
11	algebraic element	w_{11}

where the weights w_i are selected such that

$$w_9 < w_6 < (w_8 = w_1 = w_5) < (w_0 = w_2) < (w_3 = w_4 = w_{11}) < w_7 < w_{10} \qquad \textbf{Equation 5-22}$$

Within the electrical domain, the sensor represented by element 11 is classified as an algebraic element since its two terminal variables only appear algebraically in the equation. On the other hand, element 7 is classified as a first order delay element since its through variable appears differentiated in the terminal equation. In the mechanical domain however, element 6 representing the gear train is classified as a generalized across driver. This is because the value of $\theta_6(t)$ is determined by $\theta_5(t)$. If we had neglected the dynamics of the gear train, we would have an ideal transducer of the form given by Equation 5-11 and we could have made the opposite choice. From Algorithm D, we obtain the forest containing edges $\{0, 1, 2, 6, 8, 9, 11\}$ leaving edges $\{3, 4, 5, 7, 10\}$ in the coforest as shown in Figure 5-6 in bold lines.

Having selected a normal forest, we proceed with the derivation of the topological matrices, which are given by the incidence matrix $\overline{\mathbf{A}}$, the fundamental cutset matrix $\mathbf{Q}$, and the fundamental circuit matrix $\mathbf{B}$ as follows:

$$\overline{\mathbf{A}} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & -1 & -1 & 0 & 0 \end{bmatrix} \qquad \textbf{Equation 5-23}$$

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \end{bmatrix} \qquad \textbf{Equation 5-24}$$

$$\mathbf{B} = \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad \textbf{Equation 5-25}$$

Once a suitable formulation forest has been selected, the columns of the incidence matrix for each energy domain are arranged to include the branches of the defining tree as the first $v - 1$ columns. Similarly, the columns of the circuit matrix for each energy domain are arranged to include the chords of the defining tree as the last $e - v + 1$ columns. This ordering is necessary since we know that the submatrix of the incidence matrix given by the branches of the tree is non-singular. Furthermore, we also notice that since the system graph is not connected, the incidence matrix $\overline{\mathbf{A}}$ is a direct sum. Therefore, the columns in each submatrix are arranged so that the branches come first and then the chords of each tree in the forest.

Now we proceed to write the terminal equations of the elements in the system. Since we already have a normal forest, they can be written in maximal form; i.e., a maximum number of equations with derivatives of primary variables written as explicit functions of secondary variables, as well as algebraic equations written as explicit functions of secondary variables. This selection results in the following terminal equations:

$$v_{11}(t) = \frac{\theta_0(t) R_{11} i_{11}(t)}{\theta_m} \qquad i_{10}(t) = I(t)$$

$$v_9(t) = f_{out}(v_{11}(t), v_{REF}, t) \qquad \frac{d}{dt} i_7(t) = \frac{v_7(t) - k_m \dot{\theta}_8(t) - R_7 i_7(t)}{L_7}$$

$$\dot{\theta}_0(t) = \frac{\tau_0(t)}{b_0} \qquad \tau_5(t) = b_5 \dot{\theta}_5(t) + J_5 \ddot{\theta}_5(t) + \frac{\tau_6(t) N_5}{N_6}$$

$$\ddot{\theta}_8(t) = \frac{\tau_8(t) - b_8 \dot{\theta}_8(t) + k_m i_7(t)}{J_8} \qquad \tau_3(t) = K_3 \theta_3(t)$$

$$\theta_6(t) = -\frac{N_5}{N_6} \theta_5(t) \qquad \tau_4(t) = K_4 \theta_4(t)$$

$$\ddot{\theta}_1(t) = \frac{\tau_1(t)}{J_1}$$

$$\dot{\theta}_2(t) = \frac{\tau_2(t)}{b_2}$$

Equation 5-26

The constraint equations for this system can be written from the cuset (**Q**) and circuit (**B**) matrices obtained from the system graph as follows:

$$v_7(t) = v_9(t) \qquad i_9(t) = i_7(t)$$

$$v_{10}(t) = v_{11}(t) \qquad i_{11}(t) = -i_{10}(t)$$

$$\theta_5(t) = \theta_8(t) \qquad \tau_8(t) = -\tau_5(t)$$

$$\theta_4(t) = \theta_1(t) - \theta_2(t) - \theta_0(t) \qquad \tau_6(t) = -\tau_3(t)$$

$$\theta_3(t) = \theta_6(t) - \theta_1(t) \qquad \tau_1(t) = \tau_3(t) - \tau_4(t)$$

$$\tau_2(t) = \tau_4(t)$$

$$\tau_0(t) = \tau_4(t)$$

Equation 5-27

This set of 24 equations in 24 unknowns represents a necessary and sufficient set of equations for the mechatronic system. At this point, we start the reduction process to find a set of five ODEs as indicated by the selected forest. We now proceed to substitute the constraint equations into the terminal equations for the elements to obtain the branch-chord equations

$$v_{11}(t) = -\frac{\theta_0(t) R_{11} i_{10}(t)}{\theta_m} \qquad i_{10}(t) = I(t)$$

$$v_9(t) = f_{out}(v_{11}(t), v_{REF}, t) \qquad \frac{d}{dt} i_7(t) = \frac{v_9(t) - k_m \dot{\theta}_8(t) - R_7 i_7(t)}{L_7}$$

$$\dot{\theta}_0(t) = \frac{\tau_4(t)}{b_0} \qquad \tau_5(t) = b_5 \dot{\theta}_8(t) + J_5 \ddot{\theta}_8(t) + \frac{\tau_3(t) N_5}{N_6}$$

$$\ddot{\theta}_8(t) = \frac{-\tau_5(t) - b_8 \dot{\theta}_8(t) + k_m i_7(t)}{J_8} \qquad \tau_3(t) = K_3(\theta_6(t) - \theta_1(t))$$

Equation 5-28

$$\theta_6(t) = -\frac{N_5}{N_6} \theta_8(t) \qquad \tau_4(t) = K_4(\theta_1(t) - \theta_2(t) - \theta_0(t))$$

$$\ddot{\theta}_1(t) = \frac{\tau_3(t) - \tau_4(t)}{J_1}$$

$$\dot{\theta}_2(t) = \frac{\tau_4(t)}{b_2}$$

Next, we build the system of algebraic equations (Equation 5-29) and solve it for the algebraic variables $v_{11}(t), \tau_5(t), \theta_6(t), \tau_3(t), \tau_4(t)$

$$v_{11}(t) = -\frac{\theta_0(t) R_{11} i_{10}(t)}{\theta_m} \qquad \tau_5(t) = b_5 \dot{\theta}_8(t) + J_5 \ddot{\theta}_8(t) - \frac{\tau_3(t) N_5}{N_6}$$

Equation 5-29

$$\theta_6(t) = -\frac{N_5}{N_6} \theta_8(t) \qquad \tau_3(t) = K_3(\theta_6(t) - \theta_1(t))$$

$$\tau_4(t) = K_4(\theta_1(t) - \theta_2(t) - \theta_0(t))$$

obtaining

$$v_{11}(t) = -\frac{\theta_0(t) R_{11} i_{10}(t)}{\theta_m}$$

$$\theta_6(t) = -\frac{N_5}{N_6}\theta_8(t)$$

$$\tau_3(t) = \frac{-K_3(N_5\theta_8(t) + N_6\theta_1(t))}{N_6}$$

$$\tau_4(t) = K_4(\theta_1(t) - \theta_2(t) - \theta_0(t))$$

$$\tau_5(t) = \frac{N_6^2(b_5\dot{\theta}_8(t) + J_5\ddot{\theta}_8(t)) + N_5K_3(N_5\theta_8(t) + \theta_1(t)N_6)}{N_6^2}$$

Equation 5-30

The solution to the system together with the equations defining across drivers are substituted in the differential equations to obtain the state space equations:

$$\dot{\theta}_0(t) = \frac{K_4(\theta_1(t) - \theta_2(t) - \theta_0(t))}{b_0}$$

$$\frac{d}{dt}i_7(t) = \frac{f_{out}\left(-\frac{\theta_0(t) R_{11} I(t)}{\theta_m}, v_{REF}, t\right) - k_m\dot{\theta}_8(t) - R_7 i_7(t)}{L_7}$$

$$\ddot{\theta}_8(t) = -\frac{(b_5 + b_8)\dot{\theta}_8(t)}{J_8 + J_5} - \frac{N_5K_3(N_5\theta_8(t) + \theta_1(t)N_6) - k_m i_7(t)N_6^2}{(J_8 + J_5)N_6^2}$$

$$\ddot{\theta}_1(t) = \frac{-K_3(N_5\theta_8(t) + N_6\theta_1(t))}{N_6J_1} - \frac{K_4(\theta_1(t) - \theta_2(t) - \theta_0(t))}{J_1}$$

$$\dot{\theta}_2(t) = \frac{K_4(\theta_1(t) - \theta_2(t) - \theta_0(t))}{b_2}$$

Equation 5-31

Equation 5-31 includes the output function (f_{out}) associated with the controller. In this case the output function is algebraic since it depends on the inputs to the component (i.e., the controller). To determine the computational order of the equations in Equation 5-31, first, we rewrite the system of 2nd order differential equations as a system of first order differential equations as given by Equation 5-21. With this new set of equations, the BLT form can be obtained.

5.7 Summary

The normal tree of the system graph provides the means to find a causal orientation for the system of equations. In this chapter, we showed that an efficient way of finding such a tree is by converting the problem to that of finding the minimum cost spanning tree of a weighted graph.

Additionally, when the design uses software components, we presented a method that combines the symbolic description of the dynamic equations with the software components. We also pointed out the importance of properly classifying the software components to obtain a consistent BLT form that can be used in simulation.

Chapter 6 Reconfigurable models of mechatronic systems

6.1 Introduction

A generally accepted approach of evaluating designs is by using virtual prototyping. With virtual prototyping, the designer can evaluate designs without building physical prototypes. Virtual prototyping is a useful approach to improve the design process; however, to take full advantage of the benefits of this technique, design tools must integrate analysis tools transparently to the designer.

In addition to being integrated with the design tools, analysis tools must also support the evolutionary nature of the design process. Since the design process is dynamic, the behavioral descriptions provided by these tools need to be easily adjustable to changes in the design. The reconfigurable models described in this chapter are a step toward providing such flexible behavioral descriptions.

The modeling paradigm presented in this thesis, allows the designer to model systems in a hierarchical fashion. Initially, a system can be described by high-level components and the interactions between them. The models of the high-level components can then be refined iteratively without having to modify the system-level model description. As a result, the system model can be incrementally adapted as more detailed design features become known. This is in contrast to most currently available modeling environments in which a small change in the system description may require a large change in the model structure.

6.1.1 Related work

To provide simulation support to the mechatronic design approach, we need modeling tools that capture system behavior across energy domains. In recent years, a number of modeling languages have emerged that capture mathematical models of mechatronic systems. These languages are based on object-oriented principles, and include Dymola [41], OMOLA [9], NMF [118], and—more recently—Modelica [47] and VHDL-AMS [60].

Although these modeling languages are object oriented in nature, they do not permit the model *structure* to be easily modified. Instead, only mechanisms for parameter reconfiguration are provided. Given the evolutionary nature of the design process, it would be desirable to accommodate reconfiguration of the model structure also.

In de Vries' work with polymorphic models (the MAX system), he suggests an approach to achieve structure configuration [146]. His polymorphic models are similar to our concept of reconfigurable models; however, they present the following limitations:

- An instance of a model is considered an implementation. This forces a new type to be defined for each new set of parameter values for an implementation.
- Models are represented by bond graphs, which limits their applicability to lumped parameter systems.

To overcome such limitations, we propose a system representation based on two concepts (introduced in Chapter 2): *interface* and *implementation*. In this model representation, systems are described from a systems engineering point of view where subsystems interact with their environment through energy exchange. The interface of a system describes the

interaction through a set of ports. The implementation, on the other hand, describes a system's internal behavior. The interface and implementation together define a complete model of a system. A direct consequence of this new representation is that it is possible to assign different implementations to the same system interface, thereby achieving reconfigurability of models. We call system models that are based on this modeling paradigm *reconfigurable models*.

The proposed system model has the following characteristics:

- It supports the definition of systems whose behavior is defined by both energy exchange and signal flow.
- It provides for gradual refinement of models.

6.1.2 What is a reconfigurable model?

Model reconfiguration is based on two principles: *composition* and *instantiation*. Composition is the mechanism that allows us to specify a model of a device from a collection of building blocks, while instantiation performs the realization of individual building blocks.

To illustrate the idea, consider an electric DC motor. The model of this device is commonly expressed as a set of equations:

$$\tau = ki + b\dot{\theta}$$
$$v = Ri + L\frac{di}{dt} + k\dot{\theta}$$

Equation 6-1

However, looking at the energy transformation properties of this machine [64], namely, the conversion of electrical energy into mechanical energy, a finer decomposition based on energy balance can be achieved. In this decomposition the interaction of each subcomponent is established and well defined. Figure 6-1 shows the decomposition of the electric machine into three components: *electrical system, coupling field* and *mechanical system* where:

- W_E*:* total energy supplied by the electrical source.

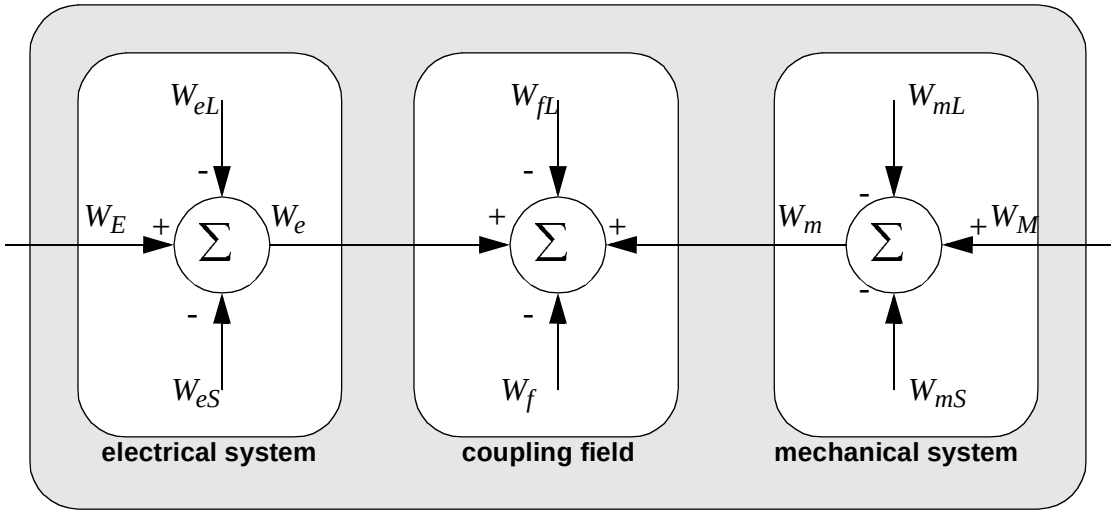


Figure 6-1. Energy-based block diagram of an electric motor.

- W_M*:* total energy supplied by the mechanical source.

Then the energy distribution law can be expressed.

$$W_E = W_e + W_{eL} + W_{eS}$$
$$W_M = W_m + W_{mL} + W_{mS}$$

Equation 6-2

where

- W_{eS}*:* energy stored in the electric or magnetic fields which are not coupled with the mechanical system.
- W_{eL}*:* heat losses associated with the electrical system. These loses occur due to the resistance of the current carrying conductors as well as the energy dissipated from these fields in the form of heat due to hysteresis, eddy currents and dielectric losses.
- W_e*:* energy transferred to the coupling field by the electrical system.
- W_{mS}*:* energy stored in the moving member and compliances of the mechanical system.
- W_{mL}*:* energy losses of the mechanical system in the form of heat.
- W_m*:* energy transferred to the coupling field.

If we define W_F as the total energy transferred to the coupling field then:

$$W_F = W_f + W_{fL}$$ **Equation 6-3**

where

- W_f: energy stored in the coupling field.
- W_{fL}: energy dissipated in form of heat due to losses within the coupling field.

The electromechanical system must obey the law of conservation of energy:

$$W_f + W_{fL} = W_e + W_m$$ **Equation 6-4**

An energy-based system decomposition for an electric motor consists of three basic building blocks, each representing one subsystem of the machine. This decomposition represents the general behavior of an electric motor where each subsystem can be described in its most general terms with no reference to the underlining equations that rule its dynamics. That is, at this level of description we are interested only in the fundamental building blocks and how they interact with each other. The specifics of each subsystem are introduced later when information about the experiment is available. For example one experiment may require the model of the machine to consider how the total magnetic flux Φ of the field coil affects the torque of the motor. This can be captured by a model of the coupling field such as:

$$\tau_e = \frac{dW_F}{d\theta}$$ **Equation 6-5**

which specifies torque as the rate of change of the energy stored in the coupling field as a function of θ, where θ is the angle between the magnetic axes [64].

Different models of the electrical subsystem can be provided as well, namely, separate excited, shunt, series, or compound. Each model describes the electrical system of the machine based on different topological connections. Similarly, the mechanical system can be described by different models including rigid bodies or non-rigid bodies, linear or non-linear contacts, etc. The basic characteristic of a reconfigurable model is that it is defined in terms of generic descriptions of submodels.

Once the model of the electric motor is defined in terms of generic subsystem components, each of the subsystems must be *instantiated*. A model is said to be an instance of a generic description if it meets the requirements that the generic description specifies for that model. From this point of view, the generic description defines a family of models that can be exchanged for one another.

Reconfigurability is achieved by the instantiation principle since every model instance that matches the requirements of its generic description is a potential candidate. This provides the ability to describe mathematical models of devices in a very structured way that can change as the requirements of the problem change. Traditional modeling allows parameter instantiation but does not provide a convenient mechanism for changing the structure of the model.

Reconfigurable models are based on the port-based modeling paradigm introduced in Chapter 3. In the next section, we will refine the port-based modeling paradigm, and in doing so, we define the basic elements that constitute a reconfigurable model.

6.2 Port-based multi-domain modeling of mechatronic systems

Recall that in a port-based multi-domain model of a mechatronic system (see Chapter 3) subsystems interact with each other through ports. Connections between ports represent the interactions between different components, which can be of two kinds: interactions that capture energy flow, for energy-based systems, or interactions that capture signal flow, for non-energy based-systems. In the first case, the connection is non-directed to reflect the non-causal nature of the energy exchange; in the second case, signal flow is represented by a directed connection.

As illustrated in Figure 6-2 the port-based model can be hierarchical (compound model) or primitive. Primitive models represent the behavior of a primitive system component in

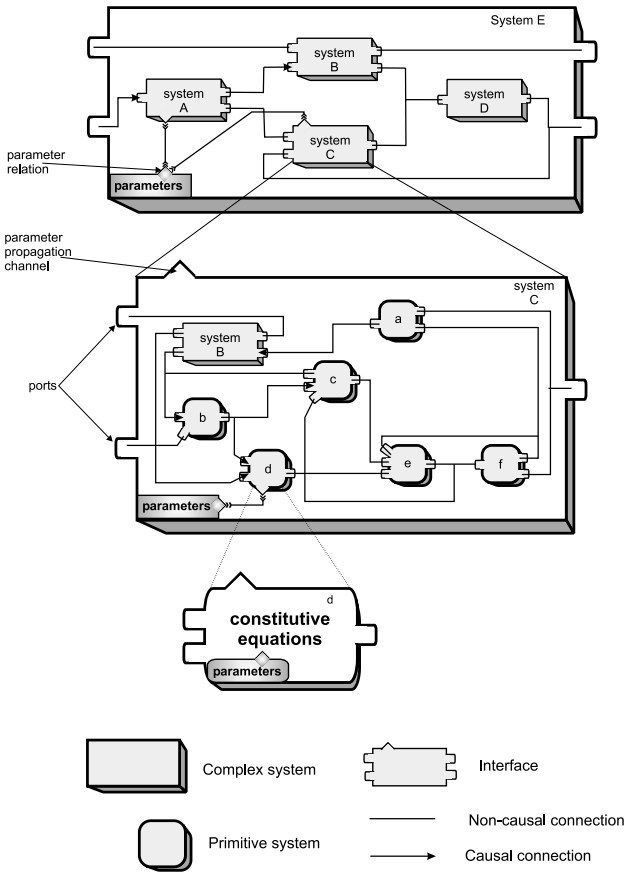


Figure 6-2. Port-based model

terms of constitutive equations. Compound models represent their behavior in terms of the structural arrangement of primitive or compound models.

6.2.1 Equation-based modeling

The behavior of a primitive system can be specified using either of two formalisms, depending on whether the system is energy-based or not. Both formalisms represent the behavior of the system with a set of *constitutive equations.* The difference between the different types of systems lies in how the constitutive equations are specified and how the system's topology is described.

In the first formalism, the behavior of non-energy-based systems is described using a procedural approach. In this case, a signal quantity represents the quantity that is available to the environment through the interface of the system. Constitutive equations of this kind are

described as assignments (causal equations) that compute the value of a signal quantity based on the inputs to the subsystem.

In the second formalism, the behavior of energy-based systems can be represented using a graph-based model [35]. Each edge in the graph has two associated quantities, called *branch quantities*: *across quantity* and *through quantity.* These quantities are analytic functions of time (i.e., they are piece wise continuous with a finite number of discontinuities). Across quantities, represent effort such as voltage, temperature, or pressure that are the result of a measurement taken across two energy ports of the system. Through quantities represent flow such as current, heat flow rate, or fluid flow rate that are the result of a measurement taken in series with the component. To illustrate this, consider an example of an electrical network. Here, the vertices of the graph represent equipotential nodes in the circuit, and edges represent branches of the circuit through which current flows. The measurement taken across two nodes defines the across branch quantity, while the measurement taken in series with the component defines the through quantity.

The constitutive equations are expressed by relating the across and through quantities of one or several branches—for example, a resistor has a single branch, and its constitutive equation (Ohm's law) relates the voltage across (the across quantity) and the current through (the through quantity) the resistor.

Constitutive equations define the behavior of a subsystem without mention of a particular causal direction. The causal direction emerges only when the equations are combined with constitutive equations of other subsystems, and the quantities that are external to the subsystem are specified. A quantity is external to the subsystem if its value is computed using a constitutive equation that is not part of the constitutive equations of the subsystem. For example, when the constitutive equation of a resistor is combined into a larger system of equations, either the voltage or the current will be defined externally; Ohm's law is causally oriented to reflect this: $v := iR$ or $i := v/R$. In the first case, the voltage v is computed from the current i, which is computed using an external equation. Similarly, in the second case, the current i is computed from the voltage v.

The equations appearing in the model can be of several kinds: *ordinary differential equations*, *algebraic equations*, or *differential-algebraic equations*. Ordinary differential equations (ODE) can represent the variation of quantities as a function of a single variable, such as time. Therefore, ODEs are used to represent lumped parameter models. Algebraic equations do not contain partial or total derivatives. When a model includes both ODE and algebraic equations, the resultant system of equations is called a differential-algebraic system of equations (DAE). In this type of system, algebraic equations represent constraints among the state variables defined by the set of ODEs.

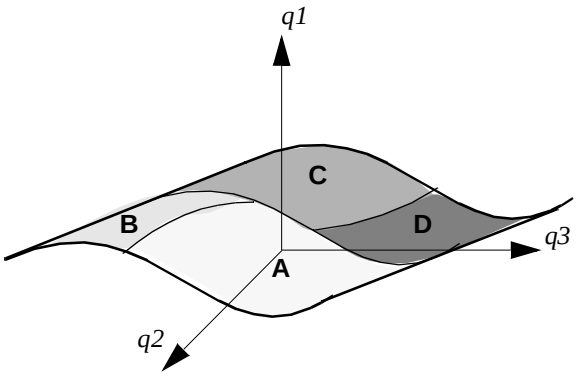
Constitutive equations of both types (i.e., non-causal and causal) may include a combination of branch quantities and signal quantities. We call these types of models *hybrid models* since they describe the interaction of energy-based systems with non-energy-based systems. An example of this type of behavioral description would be an electromechanical system controlled by a digital controller.

If the system is compound, its behavior is described according to the structural arrangement of subsystems, which in turn may be compound or primitive. This unambiguously defines the topological constraints among components. Consequently, a compound system can be reduced through a sequence of algebraic transformations into a primitive model that exhibits the same behavior.

6.2.2 Meta knowledge

Whether using a compound or primitive model to describe a system's behavior, constitutive equations do not provide sufficient information to reason about the properties of the system since they are based on implicit assumptions and approximations. In other words, the context in which a model of a system can be applied is not explicit. If such knowledge were explicit, one could not only reason about the applicability of the model to a given problem, but also decide when models having similar properties can be interchanged.

One kind of meta knowledge that we consider is the *operating region* of the model. The operation region defines the space of admissible values for the quantities of the model, which provide meaningful results. Outside this space, the model may provide erroneous




```
model process
quantities
      q1, q2, q3
equations
      if OC1(q1, q2, q3)
            RA(q1, q2, q3)=0
      elif OC2(q1, q2, q3)
            RB(q1, q2, q3)=0
      elif OC3(q1, q2, q3)
            RC(q1, q2, q3)=0
      elif OC4(q1, q2, q3)
            RD(q1, q2, q3)=0
```

Figure 6-3. Segmentation of the domain based on different operating regions

results that invalidate its application. To ensure that a model is used within its operating region, we explicitly express its bounds through the *operating conditions* of the model. Operating conditions are conditional expressions on the quantities of the model and explicitly define the sub-domain for each quantity for which the equations stated in the model are valid.

When it is necessary to define multiple operating regions, for example to capture a large portion of the system's operating region, operating conditions are disjointed. Each element of the disjunction represents a valid operating region that has an associated set of constitutive equations. In other words, operating conditions allow us to segment the domain of the quantities involved in the constitutive equations. Consequently, we can obtain a model that is applicable within a larger operation region of the system. For example, consider the operation region of the system shown in Figure 6-3. The domains of the three quantities define the solution space of the system. The operating conditions given as a function of the three quantities (represented by the functions OC_i) segment the space into four regions *(A, B, C,* and *D),* each of which has an associated set of equations (represented by the function R_j) that describe valid behavior under these conditions.

6.2.3 Current support for port-based modeling

Using object-oriented modeling principles, it is possible to describe a port-based model that is composable and hierarchical [9, 47, 60, 118]. However, in an object-oriented modeling

paradigm, there is not necessarily a clear separation between the interface of the model and the implementation of its behavior. Often both concepts are merged together into a single modeling entity. In this modeling approach, only parametric reconfiguration is allowed. However, to evaluate the design at different levels of detail, changes in structure should also be supported.

Ideally, the designer should be able to specify a system based on an understanding of its behavior and its interaction with the environment. Details of how the system achieves its behavior or about its internal structure do not become important until later, when the designer has selected a particular instance of the given system. Moreover, as the design process evolves, the designer should be able to change the structural configuration of the system. To provide a modeling paradigm that admits structural modifications as well as parametric configuration, we extend the port-based modeling paradigm to reconfigurable models in the following section.

6.2.4 Reconfigurable component models

In this section, the port-based modeling paradigm presented in the previous section is extended towards reconfigurable models. In a reconfigurable model, the interface of the model and the implementation of its behavior are considered to be two separate, but dependent, concepts. By considering these two concepts independently, it is possible to associate different implementations to a single interface, achieving a structural modification of models, and consequently, creating a *reconfigurable model* (Figure 6-4). A reconfigurable component model is a mathematical model that provides a mechanism to describe changes in structure as well as the basic parameter configuration mechanism and it is based on two principles: *composition* and *instantiation*.

The composition principle denotes the mechanism by which the behavior of the component is described in terms of interfaces of subcomponents and their interactions. Since composition unambiguously represents topological constraints among components, a composed model given by the pair $\langle \Phi | \phi_c \rangle$ can be reduced to a primitive model $\langle \Phi | \phi_p \rangle$ which exhibits

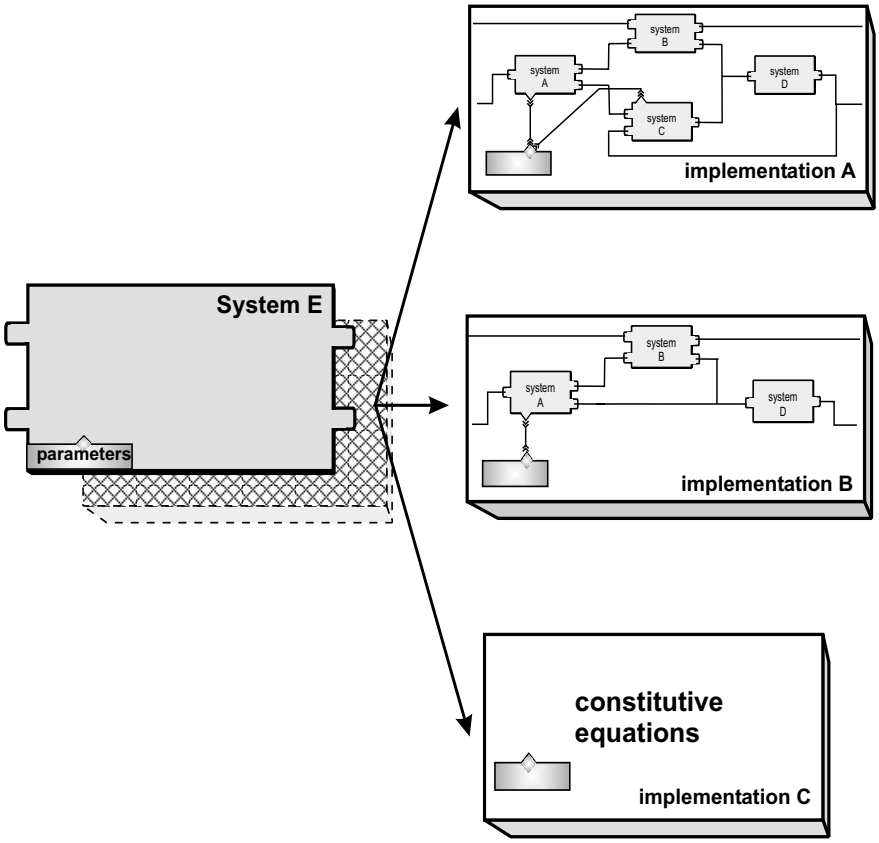


Figure 6-4. A reconfigurable system model.

equivalent behavior, where $\langle \Phi | \phi \rangle$ represents the binding of implementation ϕ to interface Φ.

As a result of the principle of instantiation, compound implementations are abstract. They are a composition of abstract interfaces, that still need to be bound to implementations to specify the behavior completely. Consequently, instantiating a compound component (i.e., a component with a compound implementation) requires the recursive instantiation of each interface in the component.

Reconfigurable models are hierarchical in nature. Based on the composition principle, we define self-contained implementations of a system in terms of the composition of sub-system interfaces; i.e., a compound implementation. However, a hierarchical system defined by reconfigurable models is not fixed. Rather, it changes as implementations are bound to the interfaces (model instantiation) that describe the compound implementation.

Specifically, binding different implementations to an interface results in a different structural arrangement and thus a different hierarchical structure.

The second principle—the principle of instantiation—describes the mechanism by which the interface of a model is bound to its implementation. An implementation that meets the requirements of an interface, can generally be bound to it. However, the semantics of the resulting model must be consistent with the context in which the model is used. For example, consider the case of a resistor and a capacitor whose interfaces both include the same set of interaction points (two electrical terminals). In such a case, an implementation that satisfies the interface for the resistor will also satisfy that of the capacitor. However, the semantics of the resulting model will differ; hence they cannot be interchanged. In summary, we will allow bindings that produce models having the same semantic meaning.

There are two kinds of implementations that can be bound to an interface and maintain a consistent semantic interpretation of the model: implementations with different representations of equivalent behavior, and implementations with different behavior. Accordingly, if τ and κ are two implementations that satisfy interface Φ, then,

$$SEMANTICS(\langle\Phi|\tau\rangle) = SEMANTICS(\langle\Phi|\kappa\rangle) \qquad \textbf{Equation 6-6}$$

The instantiation principle also provides the basis to define a family of systems. An interface that can be bound to different implementations by the instantiation principle defines a family of systems. All members of the family will show the same interaction characteristics but with different formal behavior. This method of describing membership of an element in a set is referred to as a *type* in the theory of computational objects and can be phrased as follows [1]:

> "The type of an object provides the semantic information that completely characterizes the object but not its behavior."

An interface defines a type, each member of which is a subsystem having a unique formal behavior. Based on this observation, it is possible to organize system models into a type hierarchy. This type hierarchy is derived from a type system that provides the notions of subtype and supertype [1].

Let the symbol <: represent a reflexive and transitive subtype relation between interfaces I and I', then:

Definition *Subtyping.* I' <: I if I' has the same ports and parameters as I and possibly more, and the following conditions pertain:

1. The types of the ports are subtypes of types of corresponding ports in I.
2. The types of the parameters are subtypes of the corresponding parameters in I.
3. The semantics of the parameters are equivalent to the semantics of the corresponding parameters in I.

Based on the definition of subtype (and its complement, supertype), two operations can be defined on the type hierarchy, namely, *specialization* and *generalization*. Specialization involves finding an interface for the same family of components that includes more detail, while generalization involves finding an interface (again for the same family of components) that is less specific. These operations are carried out by traversing the hierarchy in either a downward direction (specialization) or an upward direction (generalization).

Definition *Specialization*. Let *x* and *y* be two interfaces. Interface *y* is a specialization of interface *x*, denoted $y \xrightarrow{S} x$, if and only if the type of *y* is a subtype of the type of *x*, and for any system *S* that contains *x*, *y* can be substituted for *x* while maintaining the same semantics. This can be stated formally as:

$$y \xrightarrow{S} x \Leftrightarrow TypeOf(y) <: TypeOf(x) \wedge (\forall S|_x, S(x \ll y) \rightarrow SEMANTICS(S|_y) = SEMANTICS(S|_x)) \quad \textbf{Equation 6-7}$$

where the operator $S\langle x \ll y \rangle$ should be interpreted as "*x* is substituted by *y* in *S*" and $S|_x$ should be read "the system *S* evaluated with interface *x*". Similarly, it is possible to derive the property of generalization based on the supertype relation.

6.2.5 Parameter handling

In addition to the ports, it is also important to include the parameters in the interface. Model parameters describe fundamental characteristics of the system. For example, inertia and torque constants specify the invariant properties of a system that represents an electric motor; they are constant quantities that do not change value throughout the entire simulation.

In defining parameters of lower level subsystems, two kinds of parameters can be identified: *formal* and *actual* parameters. A formal parameter is defined locally in the interface of the subsystem, and it is used by a bound implementation. An actual parameter contains the value of an argument that is related to a formal parameter in a call to the model, and it is defined by the environment. The value of the argument is the value of an expression defined in terms of the formal parameters in the current scope. Parameter composition ensures that the parameters of the system are propagated to all the subsystems that were incorporated into the compound implementation.

The principles of instantiation and composition together with parameter propagation provide the infrastructure required to define reconfigurable models. In the next section, we present the structure of a reconfigurable component model. Based on this structure, later we define two important concepts that help support the design process: namely, component selection and model selection.

6.3 Component structure

In this section, we present a component structure based on an AND-OR tree that captures the complete model space for a component. In this context, the component structure is a complete behavior-based characterization of the component. It spans the space of possible system models for a given component where elements within it are instances of a reconfigurable component model; i.e., pairs of the form $\langle \Phi | \phi \rangle$ where Φ is an interface and ϕ is an implementation bound to Φ.

In an AND-OR tree representation of the modeling space, each implementation of an interface generates AND arcs. The *degree* of an AND arc is defined as the number of successor

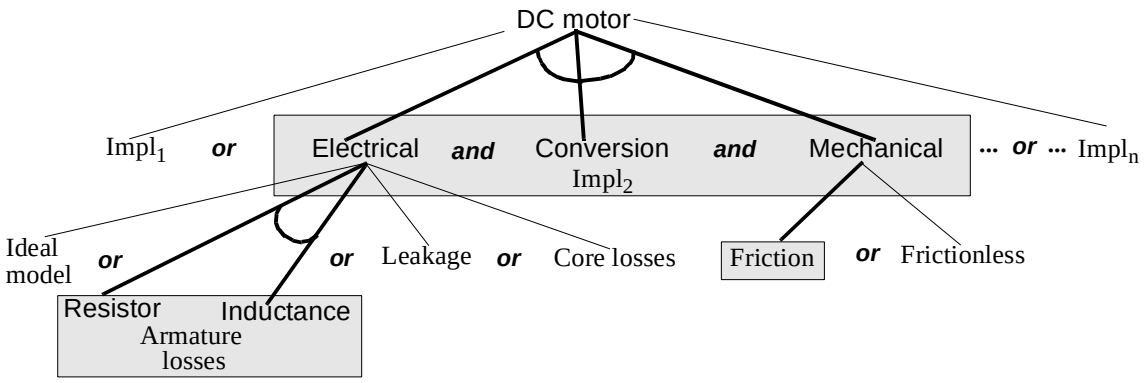


Figure 6-5. Component model structure based on an AND-OR tree

nodes the arc may point to. AND arcs of $degree > 1$ represent compound implementations and the nodes pointed to by the AND arc represent the interfaces that comprise the implementation. AND arcs of $degree = 1$ represent primitive implementations. AND arcs are indicated in Figure 6-5 with a line connecting all of the components.

In an AND-OR tree, several AND arcs may emerge from a single node, indicating alternative implementations of the interface. These are called OR arcs. OR arcs define valid changes in structure that the model may undergo, thereby populating the model space of the component.

The structure of the AND-OR tree and the principles of composition and instantiation are tightly related. The principle of composition is described by an AND arc pointing to all the constituents of the composed model. The principle of instantiation, on the other hand, is described by an OR arc since it describes alternative ways of defining the component.

For example, the AND-OR tree shown in Figure 6-5 depicts part of the structure of a permanent magnet DC motor. The top-level interface of the motor can be bound to different implementations, which indicate different alternatives to describe its behavior, i.e., OR arcs. In Figure 6-5, the AND arc of $degree = 3$ consisting of the subsystems electrical, conversion and mechanical, indicates that this particular implementation is a compound system composed of the interfaces *electrical*, *conversion* and *mechanical*. Each interface, in turn, generates an AND-OR tree that expands the possibilities in the selection of their respective implementations. For instance, the implementation “armature losses” of the

interface *electrical*, spans an AND arc of *degree* = 2 with two interfaces: *resistor* and *inductance*. Similarly, the interface *mechanical* can be described using any of the two implementations, one that considers friction and another that neglects friction.

In summary, the representation of a component model using an AND-OR tree describes different ways of modeling the component, by capturing the modeling space spanned by the reconfigurable model. Points within the space represent specific modeling instances for the component, given by the binding $\langle \Phi | \phi \rangle$ of implementation ϕ to interface Φ. The next section, describes a component model library intended to select and organize reconfigurable models based on this representation.

6.4 Model libraries

This section presents a model library of reconfigurable models with the aim of providing the designer with tools to achieve both component and model selection. By component selection, we mean that the library should supply reasonable alternatives of commercially-available components based on given requirements. For example, the library should be able to answer requests such as "Find a permanent magnet motor with a maximum torque rating of τ". Model selection means that the library will provide the user with different model alternatives or different implementations for a particular subsystem. This is important because the availability of modeling alternatives (or implementations for a given model) lets the user explore the behavior of a subsystem at different levels of detail.

Models in the library range from subsystems that are abstract to others that are concrete and completely defined. An abstract system is characterized by an interface with unknown parameters and a default implementation. A concrete system is characterized by an interface with a fixed set of parameter values and a particular implementation. However, the implementation of a concrete system does not need to be fixed. It may be the case that the designer is interested in analyzing the behavior of a concrete system under different experimental conditions, making it necessary to select a different implementation for the component.

Subsystems and components within the library are both described by an implicit AND-OR tree. Abstract subsystems are described by an AND-OR tree of level >= 1; i.e., the model space of the component. On the other hand, concrete subsystems are described by a sub-graph (i.e., an *induced* tree) of the AND-OR tree that represents particular selection of implementations for all subsystems in the component. The induced sub-graph on the component model structure (component AND-OR tree) has no OR arcs and is defined by the binding $\langle \Phi_i | \phi_k \rangle$ for $i = 1 \ldots N$, $k \in 1 \ldots n_i$ where N is the number of interfaces and n_i is the number of implementations for interface Φ_i as defined for the component. An example of an induced tree is indicated by bold arcs in Figure 6-5.

To fully define a concrete component, in addition to finding an induced sub-graph in the component structure, it is necessary to define the set of parameters that define the properties of the component.This combined process is known as *realization*. Realization of a (concrete) component is the process of finding an induced sub-graph in the component structure together with the assignment of fixed parameter values for the component. Only when parameter values are fixed in the induced sub-graph do we have a fully *realized* component.

Models within the library are organized in a hierarchy based on type. This organization is such that models that are subtypes of parent models add only new information to the interface. For instance, in Figure 6-6, the component labeled *diodeTh* (thermal diode) is a subtype of component *diode* since it adds the thermal ports and parameters needed to describe that interaction. The other ports in the model remain the same, satisfying the subtype relation stated earlier.

The library of component models defined in this section exhibit the following characteristics [18]:

- *The models in the library are reusable and reconfigurable*—the system model proposed in this chapter promotes reusability and reconfigurability of models. Moreover, since models range from abstract to concrete, they are sufficiently general to be used in different contexts.

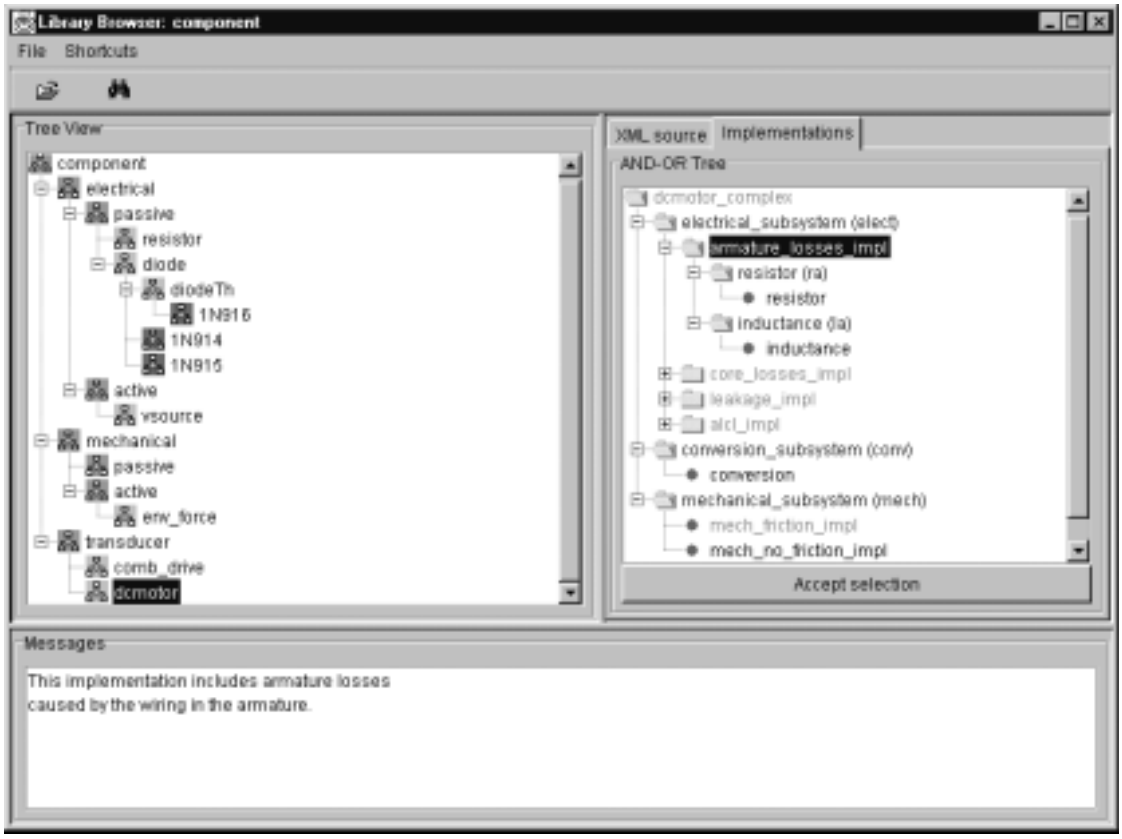


Figure 6-6. Component library browser. Green nodes represent abstract interfaces which do not have an implementation associated. Gray nodes represent generic component interfaces, that is, interfaces that have not yet been parametrized. Finally, pink nodes represent interface instantiations; i.e., fully parametrized components. These represent manufacturer components as given in a component catalog.

- *The models in the library can be shared*—system models are described using a web-savvy format (See “Component modeling markup language”) based on a language explicit enough that users others than the original author can understand it. Moreover, the designers sharing the models can use domain specific browsers dedicated to presenting the models in familiar terms.
- *The models in the library are accessible*—the browsing mechanism allows users to interactively select model instances (i.e., model selection) and to search for component alternatives based on a given set of requirements (i.e., component selection). Consequently, the availability of models in the library is brought to the attention of the designer, thereby making the information fully accessible.

The next section describes the representation language used to characterize the structure of reconfigurable models stored in the library.

6.5 Component modeling markup language

As indicated above, this section describes the component model language used to characterize reconfigurable component models. We have developed a neutral-format, model description language that captures component model structure based on the AND-OR tree representation introduced earlier. The language is in neutral format since it is based on XML (extensible markup language) [158] and can be translated to the simulation language of choice. A translator to VHDL-AMS [60], the target language used to model our simulations, has been developed. Translators to other modeling languages (e.g., modelica [47], ASCEND [101], OMOLA [9], or NMF [119]) can be written as well. In the design of this model description language, the following criteria were established:

- *Multi-energy domain*—the language should be able to capture the interactions between components in multiple energy domains.
- *Non-causal equation models*—the language should express the laws of physics without assigning causality.
- *Meta-knowledge*—the language should provide elements to represent knowledge implicit in the constitutive equations (such as assumptions and approximations). In other words, the context in which this model can be applied should be made explicit enough to provide a basis to reason intelligently about these models.

The markup language proposed in this chapter meets the above requirements and supports our concept of reconfigurable models as defined earlier.

6.5.1 Why XML?

XML was selected because it provides clear document structures and a context-free vocabulary. In addition, all XML documents share a common hierarchical structure and can be managed, read, edited, searched and presented using the same tools. We can take advantage of XML's hierarchical document structures to capture the hierarchical nature of the component models. Using XML, it was also possible to define catalogs of components and provide search mechanisms to explore the entire content of the document. In short, by defining

a markup language based on XML to represent component models the following features were attained:

- *Document sharing*—designers can use standard XML tools to view and edit models.
- *Component search*—designers can use search tools to locate component models based on desired characteristics.
- *Expressiveness*—a rich internal structure and a rich vocabulary makes model knowledge clear.
- *Reuse*—a consistent document structure makes it easier to reuse document content, extract it and apply content to different problem domains.

6.5.2 The markup language

System models are organized into a *document*, the internal structure of which captures the hierarchical structure of component models. All aspects of the structure (i.e., modeling constructs) are described with a rich vocabulary that translates into XML tags. A *document type definition* (DTD) describes the internal structure of a document and defines the symbols in the vocabulary. The use of this DTD ensures that the models will be *well formed* and *valid*. Well-formedness means that it is possible to check that the document is syntactically correct before it is processed, while validity involves checking the document structure to ensure it contains all the parts required by the DTD but no extraneous parts.

The markup language defines the two basic modeling entities, *interface* and *implementation*, as the core of the internal document structure. Symbols in the vocabulary include interface and implementation as well as symbols used to represent constitutive equations, subcomponents, interactions between components and meta-knowledge.

At the top level, a document consists of either interface and implementation declaration blocks or component declaration blocks. In the first case, the document specifies the basic component models, while in the second case, the document instantiates completely defined components by specifying the parameters and implementations of the basic component models (i.e., it defines a catalog of components). Using a single markup language, we can

describe the two kinds of system models described earlier, abstract system models and concrete system models.

In the markup language that we have defined, the interface of the system includes—along with ports and parameters—declarations that are common to all implementations of the interface, conditional statements that check the validity of parameters, and the meta-knowledge about the different implementations associated with the interface. For example, the interface declaration for the DC motor in Figure 6-5 would be as follows:

```
interface DCmotor
      parameters
            ktau: real = 1.0e-14;
            km: real = 1.0;
            Ra: real = 1.0;
            La: real = 1.0e-3;
            Jm: real = 1.0e-14;
            Bm: real = 1.0e-5;
      ports
            pos, neg: electrical;
            rotor, reference: rotational;
end DCmotor;
```

In the language, two sections describe an implementation of an interface: the *declaration section* and the *statement section*. The declaration section defines quantities or subcomponents (in the case of a compound component) that are local to the body of the implementation. The statement section defines the behavior of the component with either a set of constitutive equations or a set of connection statements.

Compound components are described with two vocabulary symbols: *component* and *connections*. Components declare the instances to be used in the model, and connections define the interactions between declared subcomponents. A component declaration is an XML sub-structure that describes the induced tree in the AND-OR tree of the component. It captures parameter propagation and the binding of interface-implementation for all subcomponents of the component.

For the DC motor illustrated in Figure 6-5, the implementation composed of the three subsystems: electrical, conversion and mechanical would be the following:

```
implementation dcmotor-cmp implements DCmotor
      declarations
            elect-subsystem elect(Ra=10,La=0.1)
                  bound-to armature-l-impl
                        resistor ra(r=Ra) bound-to resistor
                        inductance la(l=La) bound-to inductance;
            conv-subsystem conv(Km=km, Kt=ktau)
                  bound-to conv-impl;
            mech-subsystem mech(Bm=1.0e-5)
                  bound-to friction-imp;
      statements
            connections;
end dcmotor-cmp;
```

In this implementation the paths *DCmotor-Electrical-[Resistor, Inductance], DCmotor-Conversion,* and *DCmotor-Mechanical-[Friction]* shown in Figure 6-5 provide the subtree selected for the DC motor. This means that for the electrical system of the DC motor this implementation includes armature losses in the electrical subsystem. Similarly, the selected subtree indicates that the implementation considers friction in the mechanical subsystem.

Binding the electrical subsystem to the implementation *armature-l-impl* requires also binding implementations for each component declared within it. For example, in this case, implementation *armature-l-impl* is an implementation composed of three components: *ra* and *la*. These bindings are recursively specified in the declaration of the component *elect*. For example, component *ra* with interface *resistor* is bound to implementation *resistor*. Implementation *resistor* is defined by a set of constitutive equations, and it does not specify new bindings for any components; i.e., it is a primitive implementation.

Within the DC motor, subcomponent interaction is specified by means of connections in the statement section, which define the structure of the DC motor.

Listing 6-1 and Listing 6-2 show the XML representation for the DC motor AND-OR tree shown in Figure 6-5.

6.6 Summary

In this chapter, we have described a modeling paradigm based on reconfigurable component models that supports the design of mechatronic systems. In this paradigm, mathemat-

Listing 6-1. An XML representation of a DC motor model. Interface body.

```
<interface ident="DCmotor">
 <generics>
  <parameter semantics="torque_constant"
   default="1.0e-14" nature-type="real" ident="tau"/>
  <parameter semantics="motor_constant"
   default="1.0" nature-type="real" ident="km"/>
  <parameter semantics="armature_resistance"
   default="1.0" nature-type="real" ident="Ra"/>
  <parameter semantics="armature_inductance"
   default="1.0e-3" nature-type="real" ident="La"/>
  <parameter semantics="rotor_inertia"
   default="1.0e-14" nature-type="real" ident="Jm"/>
  <parameter semantics=”friction”
   default=”1.0e-5” nature-type=”real” ident=”Bm”/>
 </generics>
 <boundary>
  <terminal nature-type="electrical" name="pos"/>
  <terminal nature-type="electrical" name="neg"/>
  <terminal nature-type=”rotational” name=”rotor”/>
  <terminal nature-type=”rotational” name=”reference”/>
 </boundary>
</interface>
```

ical models consist of two elements: interface and implementation. The interface defines the mechanism by which the model interacts with its environment, while the implementation describes the behavior of the component. Model reconfiguration is achieved when the model of a component is defined by binding an implementation to an interface.

We showed that an AND-OR tree that captures all possible modeling alternatives for a particular component describes the modeling space of the component. Using this AND-OR tree representation, we showed how these reconfigurable models can be logically organized into a library of components that supports model selection and component selection.

Listing 6-2. An XML representation of a DC motor model. Implementation body.

```
<implementation compound="true"
 of-interface="DCmotor" ident="dcmotor-cmp">
 <component interface-name="elect-subsystem" name="elect">
  <parameter-binding actual-part="10" formal-part="Ra"/>
  <parameter-binding actual-part="0.1" formal-part="La"/>
  <parameter-binding actual-part="10" formal-part="km"/>
  <bound-implementation implementation-name="armature-l-impl">
   <component interface-name="resistor" name="ra">
    <bound-implementation implementation-name="resistor"/>
   </component>
   <component interface-name="inductance" name="la">
    <bound-implementation implementation-name="inductance"/>
   </component>
  </bound-implementation>
 </component>
 <component interface-name="conv-subsystem" name="conv">
  <parameter-binding actual-part="32.0e-3" formal-part="tau"/>
  <bound-implementation implementation-name="conv-impl"/>
 </component>
 <component interface-name="mech-subsystem" name="mech">
  <parameter-binding actual-part="1.0e-5" formal-part="Bm"/>
  <bound-implementation implementation-name="friction-impl"/>
 </component>
 <concurrent-statements>
  <connect terminal-B="neg" terminal-A="elect.neg"/>
  <connect terminal-B="rotor" terminal-A="mech.load"/>
  <connect terminal-B="reference" terminal-A="mech.ref"/>
  <connect terminal-B="conv.elect" terminal-A="elect.conv"/>
  <connect terminal-B="conv.mech" terminal-A="mech.conv"/>
  <connect terminal-B="pos" terminal-A="elect.pos"/>
 </concurrent-statements>
</implementation>
```

Chapter 7 Case study—Mechatronic design of a missile seeker

7.1 Introduction

In this chapter, we examine the design process of a missile seeker. The example follows the *flow of design information* model as described in [125, 136] (Figure 7-1) for three complete levels of refinement to the point where the actual mechanism of the seeker is formulated, the actuators and gears are selected, and the controllers are derived.

The flow of design information model identifies different states within a particular stage in the design process. The edges in the state diagram denote the flow of information from one state to another. Design activities transform design information and move this information from one state to another; labels attached to the edges indicate such design activities.

There exists a direct correspondence between our reconfigurable modeling paradigm and a number of states and transitions in the flow of design information model (illustrated in by

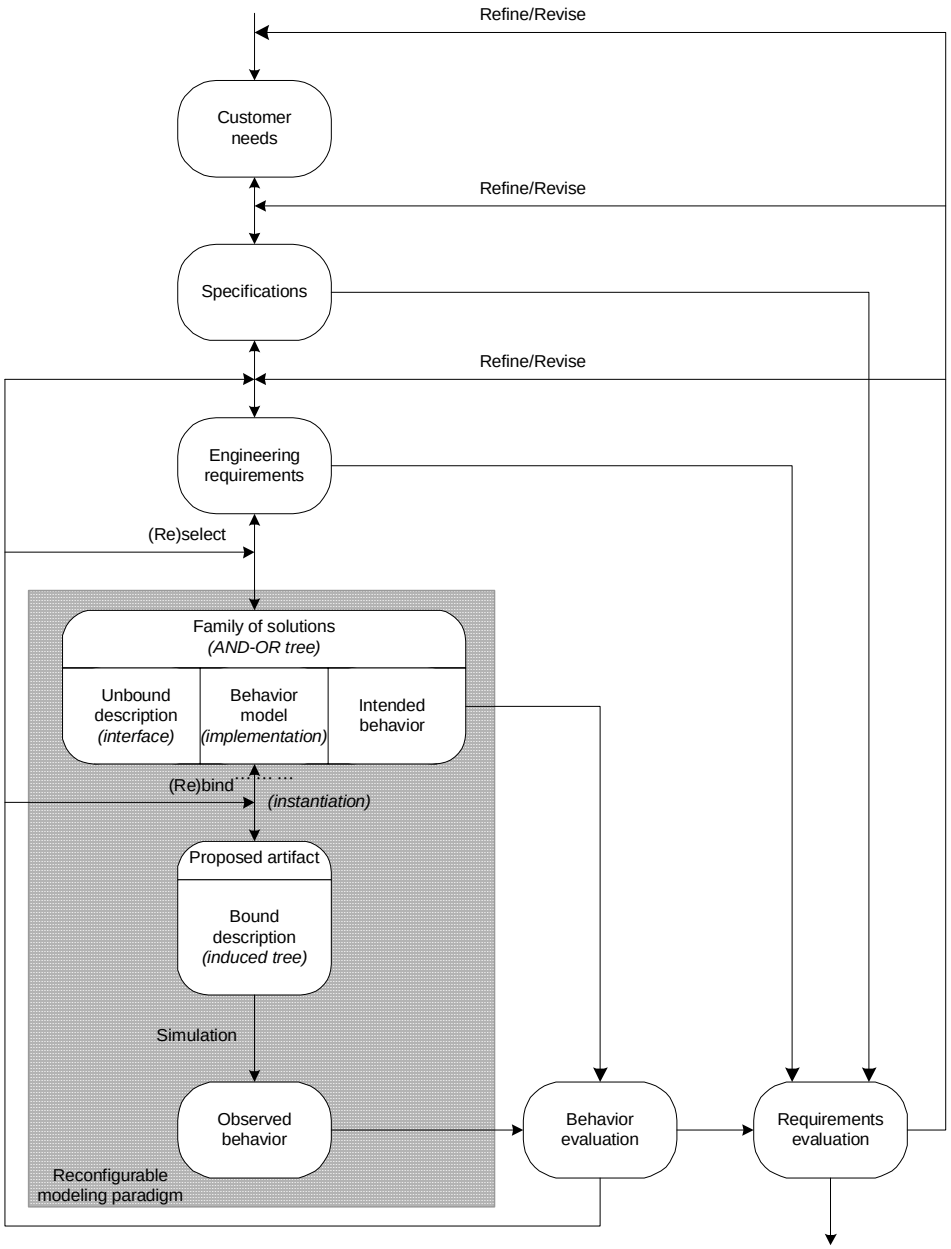


Figure 7-1. Flow of design information model. Adapted from [125, 136]

the shaded area). This correspondence is exploited in this example to allow the designer to evaluate the behavior of the proposed artifact by means of reconfigurable models.

In the flow of design information model, the *family of solutions* state represents a family of artifacts that may meet the engineering requirements. The relationship between a solution family and its member artifacts is similar to the relation between a class and its instances

in object-oriented programming [125, 136]. Similarly to a class in object-oriented programming, a solution family describes a collection of elements that share common properties. Element members of this family are equivalent to instance objects. This means that every member of the family shares the same set of attributes that describe the family, but with possibly different attribute values.

The family of solutions is equivalent to our AND-OR tree description of a component. Recall that an AND-OR tree represents the model space of a component. Elements in this model space, which share the same attributes with the rest of the members in the space, represent specific instances of a component.

Interfaces and implementations in the reconfigurable modeling paradigm have two equivalent views in the flow of design information model. Interfaces map to unbound descriptions and implementations map to behavior model. Through the principle of instantiation, we move to the proposed artifact state.

The *proposed artifact* state is reached when the designers complete the description of the artifact. The designers do so by defining parameter values in the unbound description and by binding the unbound description to a behavior model. This process corresponds to the instantiation principle for a reconfigurable model and it is represented by an induced tree in the AND-OR tree of the component.

Once the designer has selected a proposed artifact, the *observed behavior* is derived by simulating the artifact. The results of the simulation are used in the *behavior evaluation* state. In this state, designers compare the artifact's intended and observed behaviors and identify any discrepancies. Based on the nature of the discrepancies the designer may choose to instantiate a different artifact (by means of reconfigurable models) assuming that the current solution family remains promising and the proposed artifact can be improved (refined). Alternatively the designers may decide that the current solution family is not adequate and may select a different family. At this point our reconfigurable modeling paradigm can be used again to instantiate and analyze behaviors of member of the new family.

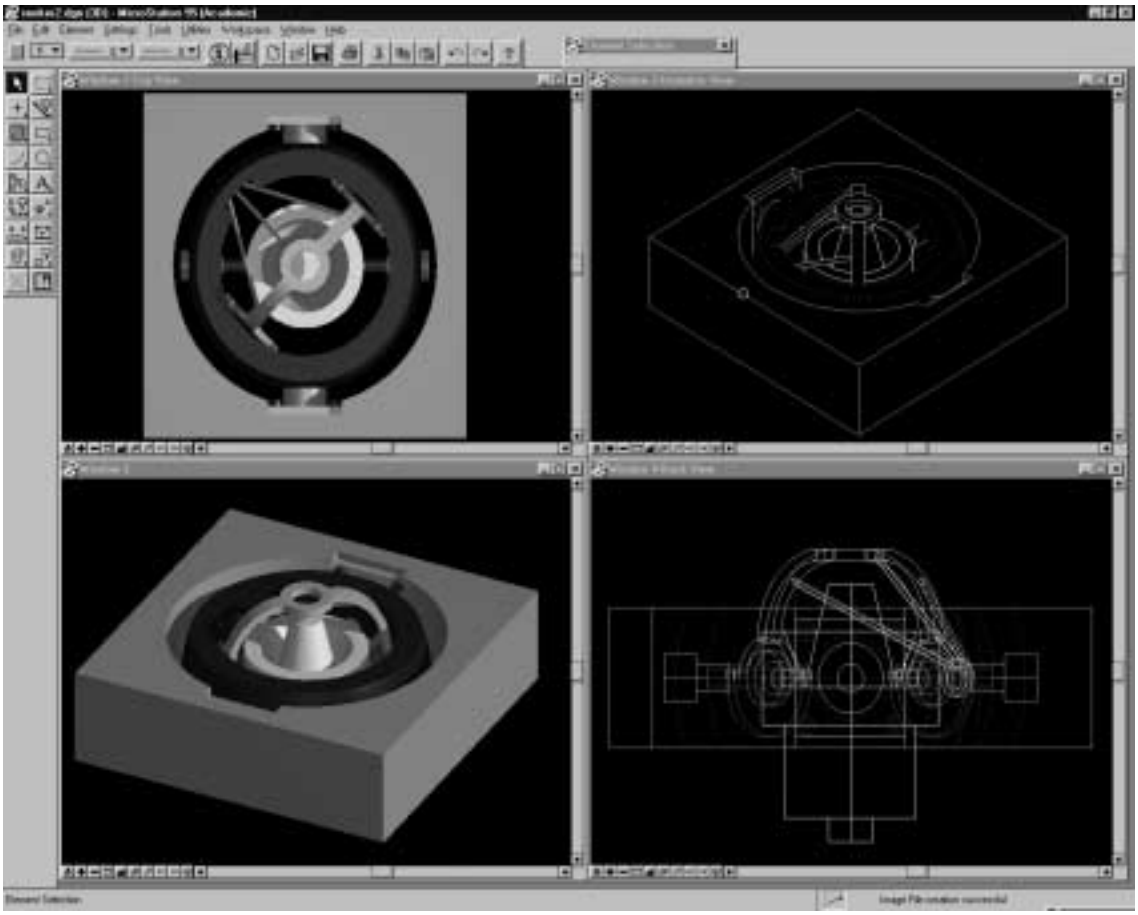


Figure 7-2. Seeker

7.2 The device

The seeker is a device of medium complexity with two rotational degrees of freedom (called pitch and yaw). The two degrees of freedom allow the device to scan a two-dimensional workspace. Figure 7-2 illustrates a complete seeker.

The design of the device must meet design specifications such as desired range of motion and desired acceleration. All this should be consider together with the physical dimensions of the device and physical properties of the selected materials. The design of the seeker also includes the selection of actuators and control systems. In this chapter, simulation will be used to verify the different alternatives available while selecting the actuators, controllers, and geometry.

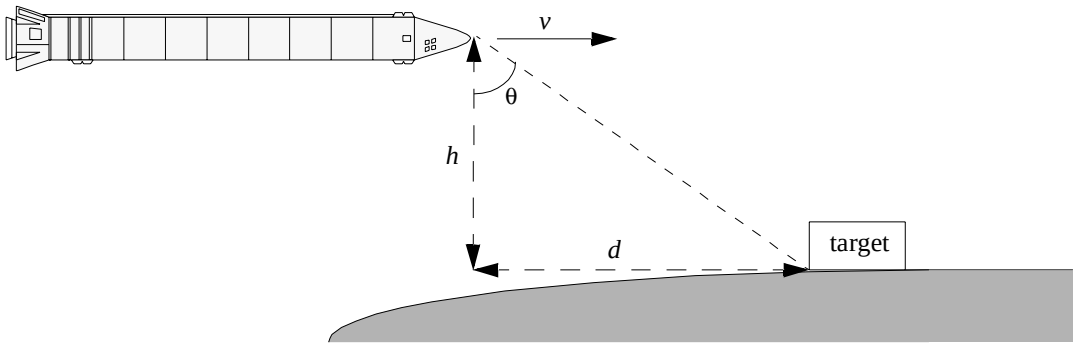


Figure 7-4. Customer needs.

The complete structure of the seeker, based on an AND-OR tree, is illustrated in Figure 7-3.

7.3 Iteration I

The design begins with the recognition of need for a mechanism that allows the scanning of a two-dimensional workspace.

7.3.1 Customer needs

The customer needs for the seeker specify the desired characteristics the intended design must achieve. With reference to Figure 7-4, these include:

1. Light weight.
2. Manageable size.
3. Scan a square two-dimensional area of 800m by 800m.
4. Minimum cruising altitude *h* is 250m.
5. The seeker is mounted on a missile traveling with a maximum cruise speed of 800Km/hr. (222.22m/sec) and it should track a stationary object on the ground.

7.3.2 Specifications

The specifications are derived from the customer needs. This results in the following specifications.

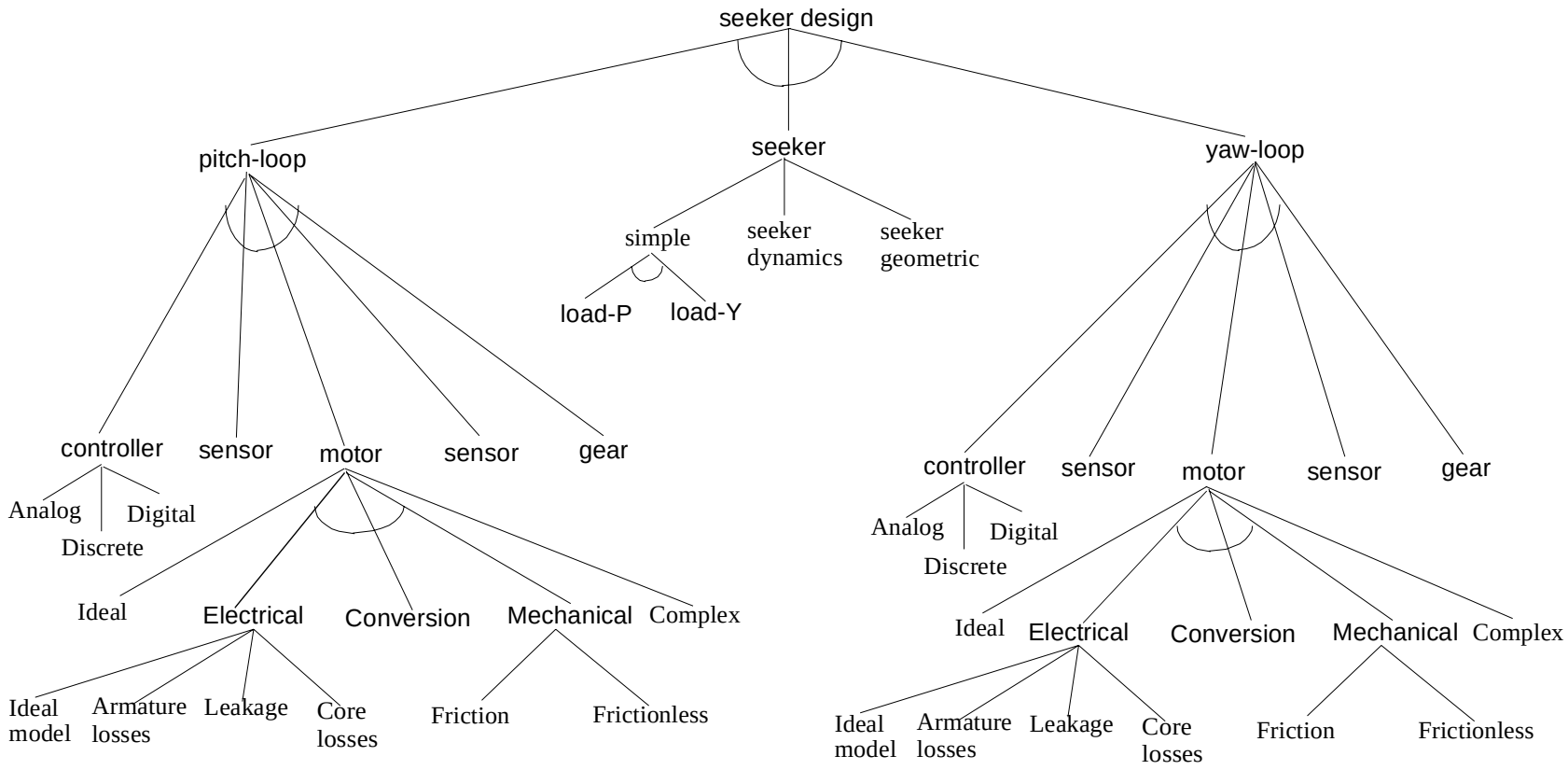


Figure 7-3. Seeker design structure based on an AND-OR tree

1. Weight limit of 800gr.
2. The size of the seeker will be determined later.
3. The seeker needs to span a 120 degrees arc (it needs to cover an area of 400m from the center of the plane). This is considering the minimum cruising altitude, which is, from the customer needs, equal to 250m. The range of motion of the device should be 1.0 rad.
4. The maximum velocity of the missile (800 Km/hr.) is translated into a required angular velocity (for a single degree of freedom) as follows: let $\theta = \operatorname{atan}\left(\frac{d}{h}\right)$ where d is the distance of the seeker to the target and h is the missile cruising altitude. Taking the derivatives with respect to time we find the angular velocity:

$$\frac{d\theta}{dt} = \frac{d}{dt}\operatorname{atan}\left(\frac{d}{h}\right) = \frac{v}{h\left(1+\frac{d^2}{h^2}\right)} \qquad \textbf{Equation 7-1}$$

which has a maximum value, when $d = 0$, of 0.888 rad/sec. To compensate for sudden changes in direction, we will require that the seeker should be able to track a trajectory with a frequency of 2Hz ($T = 0.5$ sec.). Under this requirements, we define a desired trajectory $q_d(t)$ with components:

$$\theta_{pd} = g_1 \sin\left(\frac{2\pi t}{T}\right)$$
$$\theta_{yd} = g_2 \cos\left(\frac{2\pi t}{T}\right) \qquad \textbf{Equation 7-2}$$

with period $T = 0.5\,\text{sec}$ and amplitudes $g_i = 1\,\text{rad}$.

The desired trajectory renders a desired velocity and desired acceleration of

$$\omega_p = 4\pi\cos(4\pi t) \quad \alpha_p = -16\pi^2\sin(4\pi t)$$
$$\omega_y = -4\pi\sin(4\pi t) \quad \alpha_y = -16\pi^2\cos(4\pi t) \qquad \textbf{Equation 7-3}$$

from which the maximum desired angular velocity is $\omega_{max} \approx 12.5\frac{\text{rad}}{\text{sec}}$ and the maximum angular acceleration is $\alpha_{max} \approx 160\frac{\text{rad}}{\text{sec}^2}$ for both degrees of freedom.

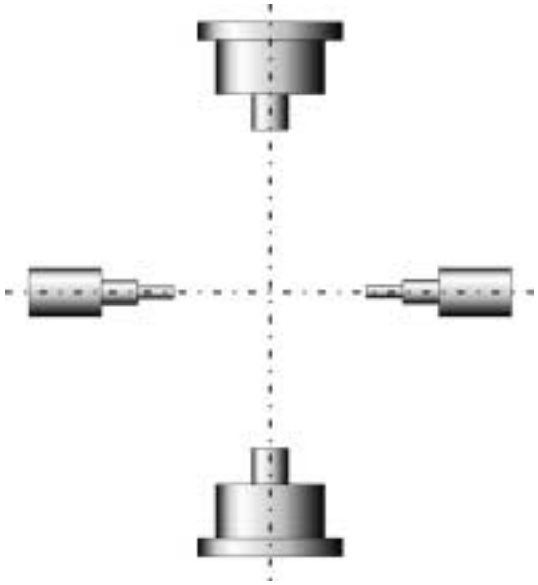


Figure 7-5. Initial kinematic model.

7.3.3 Engineering requirements

The engineering requirements formalize the specifications to a structure that facilitates its realization [125, 136]. At this stage we specify the form requirements of the missile seeker as follows:

1. Perpendicular axes for the two degrees of freedom.(Figure 7-5).
2. Symmetry.
3. Size restriction.
    a. Length: 30 cm
    b. Width: 10cm
    c. Height 30cm.

The design will consist of a gimbal ring, which will provide the yaw motion, and a component that contains the optics of the seeker, which will provide the pitch motion. The gimbal ring will support the optics housing and the two will have perpendicular axes with respect to each other. To comply with the size restriction, the gimbal ring will have an initial diameter of 20cm and a width of 4.5cm. The optics housing will have a diameter of 8cm and

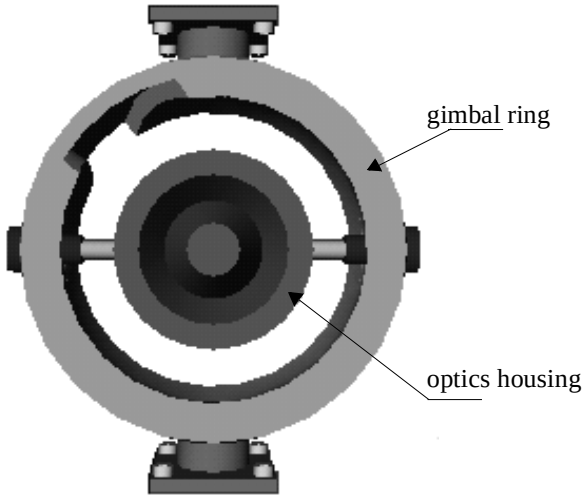


Figure 7-6. Geometric model of the seeker design.

width of 5cm. The combined actuation of the pitch and yaw degrees of freedom would provide the required workspace coverage. The desired kinematic design is illustrated in Figure 7-6.

7.3.4 Family of solutions

After establishing the engineering requirements we must explore possible solutions. In this case we turn to the description of the seeker based on the AND-OR tree (Figure 7-3). At this stage we define the high-level components that comprise the device; these represent broad concepts that suggest the structure of the device and the interaction between subcomponents (Figure 7-7).

The component graph defines the device as a collection of high-level concepts and their interactions. In this graph, we can identify two subsystems: a positioning system for each degree of freedom controlling a rotating mass (Figure 7-8). From the engineering requirements and the specifications, it is estimated that the components composing the yaw motion will be of 600gr (gimbal ring and optics housing), and the component comprising the pitch motion will be of 300gr (optics housing). This would provide a (estimated) rotational inertia of $J = 0.002$ Kg-m^2 for the yaw mechanism and of $J = 0.001$ Kg-m^2 for the pitch mechanism.

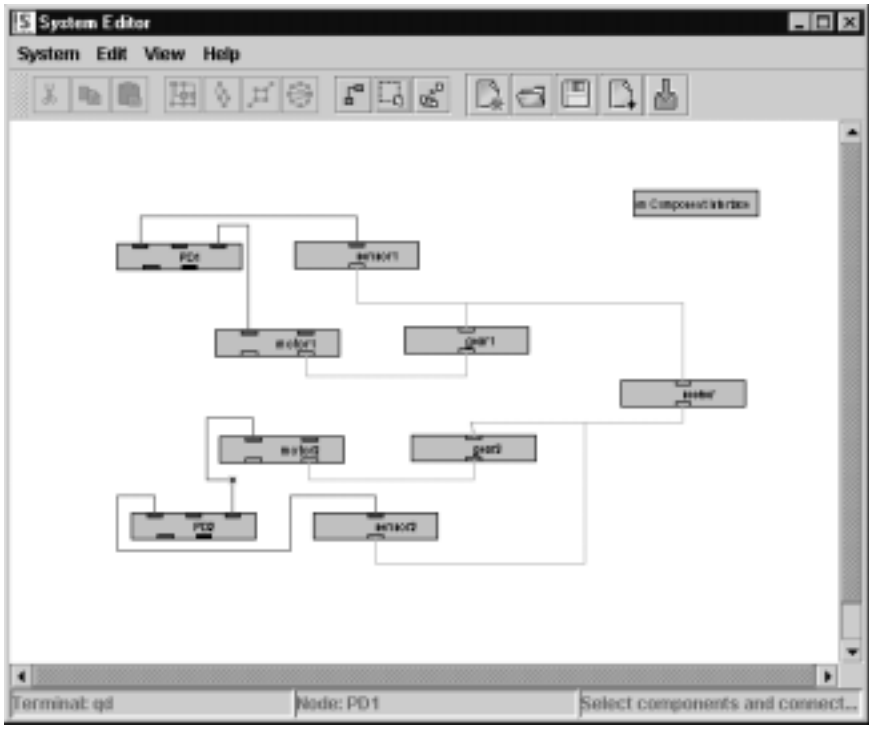


Figure 7-7. Conceptual design of the missile seeker.

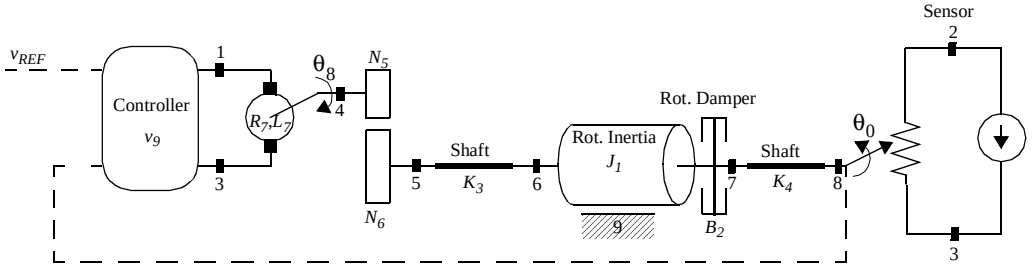


Figure 7-8. Positioning system

The load data is used to determine the amount of torque the motors need to provide. In this case, we will use a motor and a gear box to provide the desired torques. We have selected the MicroMo series GNM 26A and GNM 31 with nominal input voltage of 24 volts, speeds up to 4000 rpm, and torque up to 130×10^{-3} N-m and 240×10^{-3} N-m [85], for the pitch and yaw degrees of freedom respectively. In addition, a gearbox with ratio 30:1 would achieve the required speed and the required torque.

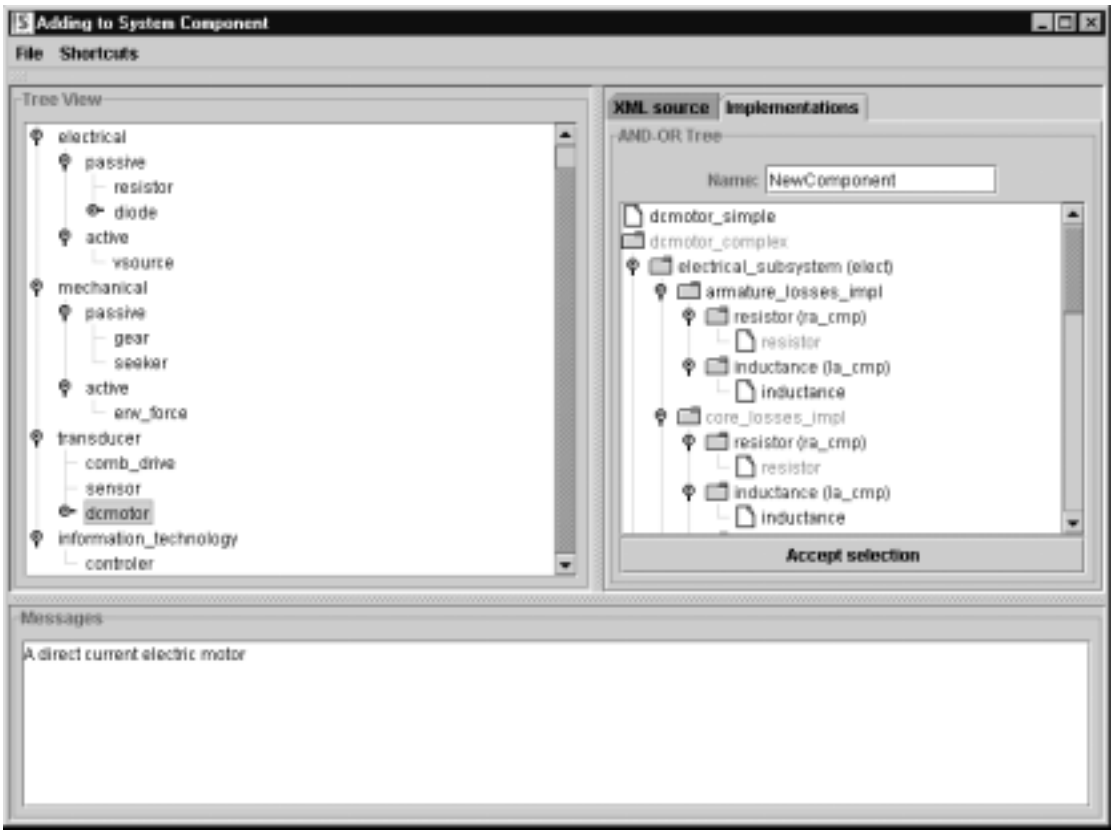


Figure 7-9. Model selection tool

7.3.5 Behavior evaluation

The behavior evaluation involves comparing the simulated behavior with the intended behavior. To perform the analysis, we select the behavior for each component in our component graph using the model selection tool (Figure 7-9) and the AND-OR tree description of the design shown in Figure 7-3.

The goal of this analysis is to verify that the motors can provide the required torque without going into saturation. A simple model of the motors will be used for this purpose. We will use a model for the selected motors with no friction and with no armature losses. The controller will be a simple analog proportional controller. In this case, the bindings of implementations to interfaces are the following: <*motor, ideal*>, <*controller, analog*> and <*seeker, simple*>. Interfaces *sensor* and *gear* provide one implementation (the default implementation) and thus no explicit binding is indicated.

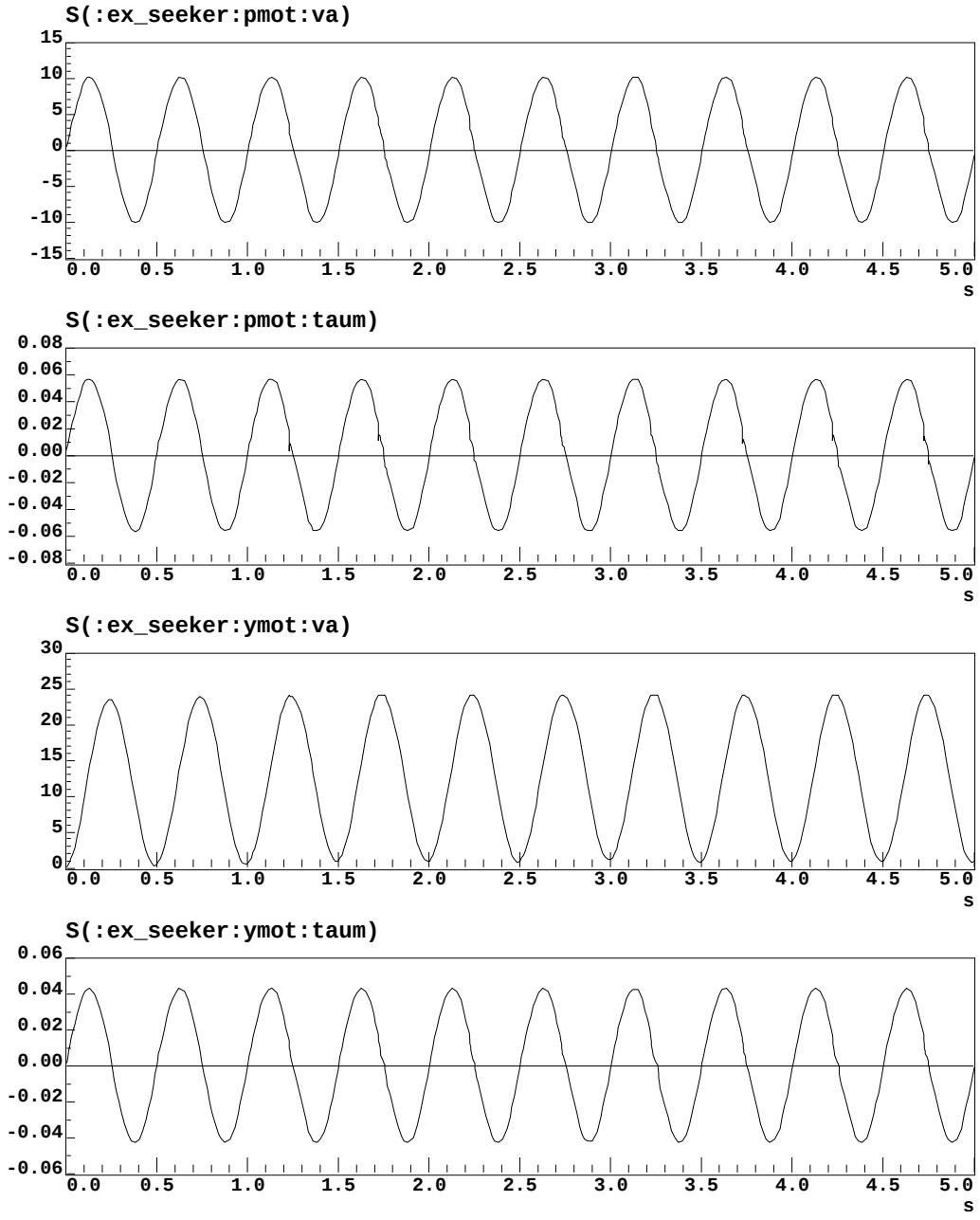


Figure 7-10. Iteration I: Input voltages and generated torques of the selected motors using the estimated loads.

As illustrated in Figure 7-10, the input voltage to the motors does not exceed the nominal supply voltage of each motor and the torque generated by each motor is within bounds. We can conclude that the motors are appropriate to drive the loads. The next step is to focus on the design of the actual geometry of the device.

7.4 Iteration II

In the second iteration we refine the model of the seeker. A refinement for the seeker model involves to define the geometry and materials.

7.4.1 Engineering requirements

The dimensions of the seeker are now established and fixed according to the customer needs (Figure 7-11). It is assumed that the geometry is synthesized using a design advisor such as the one presented in [127]. Next, we choose the type of material we will use to manufacture the seeker. We have selected an ABS polymer for the gimbal ring with density of $\rho = 1.2\frac{g}{cc}$, while the ABS polymer for the optics housing has density of $\rho = 1.07\frac{g}{cc}$ [79]. Using this information and the volume computed (we use the geometric kernel to extract that information), we find the mass of the gimbal to be 0.35 Kg, and the mass of the camera housing to be 0.33 Kg.

7.4.2 Observed behavior

Querying the geometric kernel we find that the gimbal provides a moment of inertia about the *z* axis (yaw angle) of 0.0018 Kg-m^2 while the optics housing provides a moment of inertia about the *y* axis (pitch angle) of 0.0012 Kg-m^2. Both values of moment of inertia are close to the estimated values in the conceptual design. This suggest that the motors selected in the previous section would be able to drive the load without going into saturation. To corroborate this, we derive the dynamic model of the seeker [33] and perform a new analysis on the refined system.

In this analysis we will also refine the models of the motors. The refined model of the motor includes the armature inductance as well as friction in the mechanical component. The refined model of the seeker includes the dynamic model as it was derived from the geometric model. As in the previous step, the new models are selected from the browser tool and the new binding for the models of the motors is <*motor, complex*>, and the new binding for the model of the seeker will be <*seeker, dynamics*>. The new implementations are bound to the interfaces given in the conceptual graph without having to change the concep-

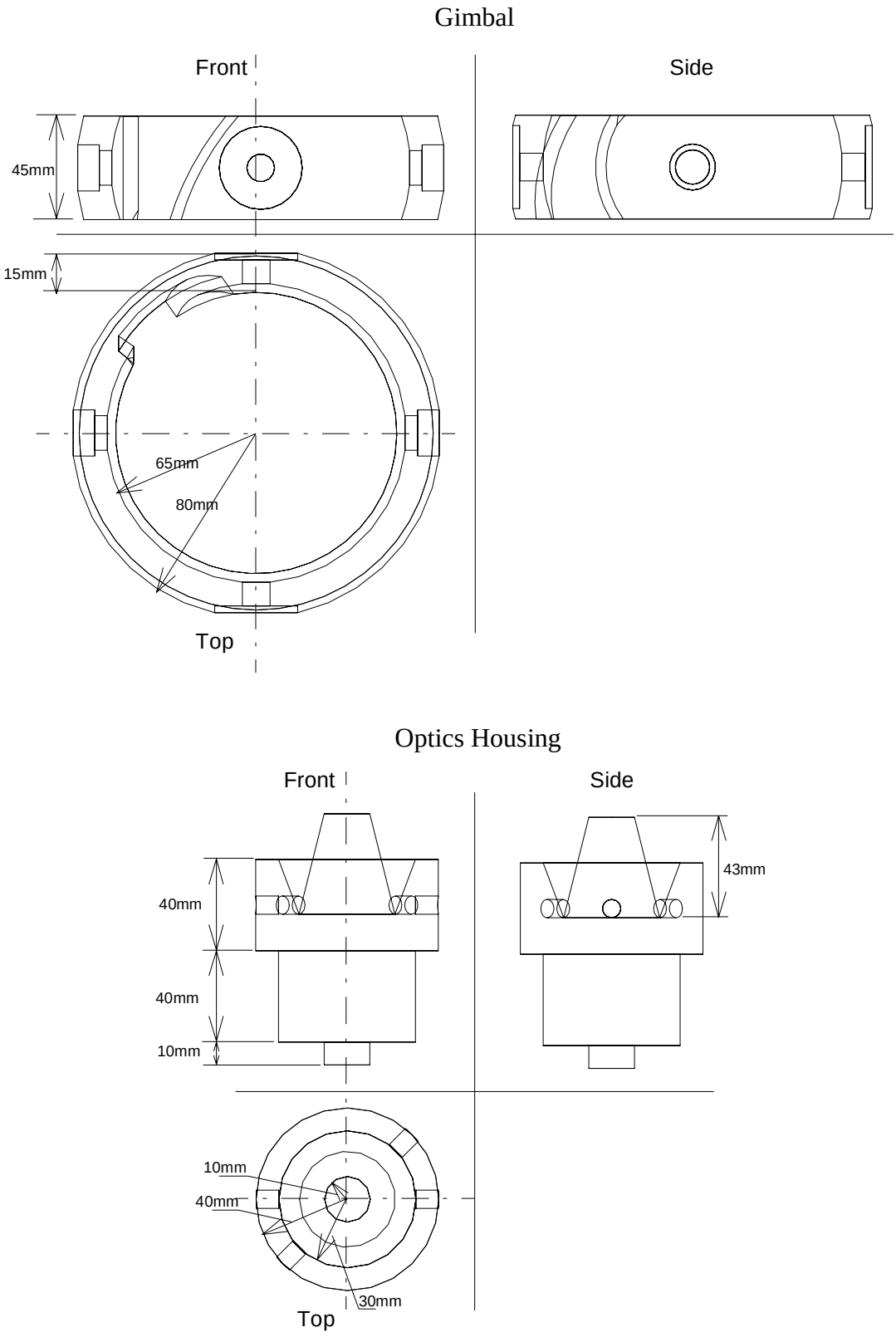


Figure 7-11. Seeker dimensions.

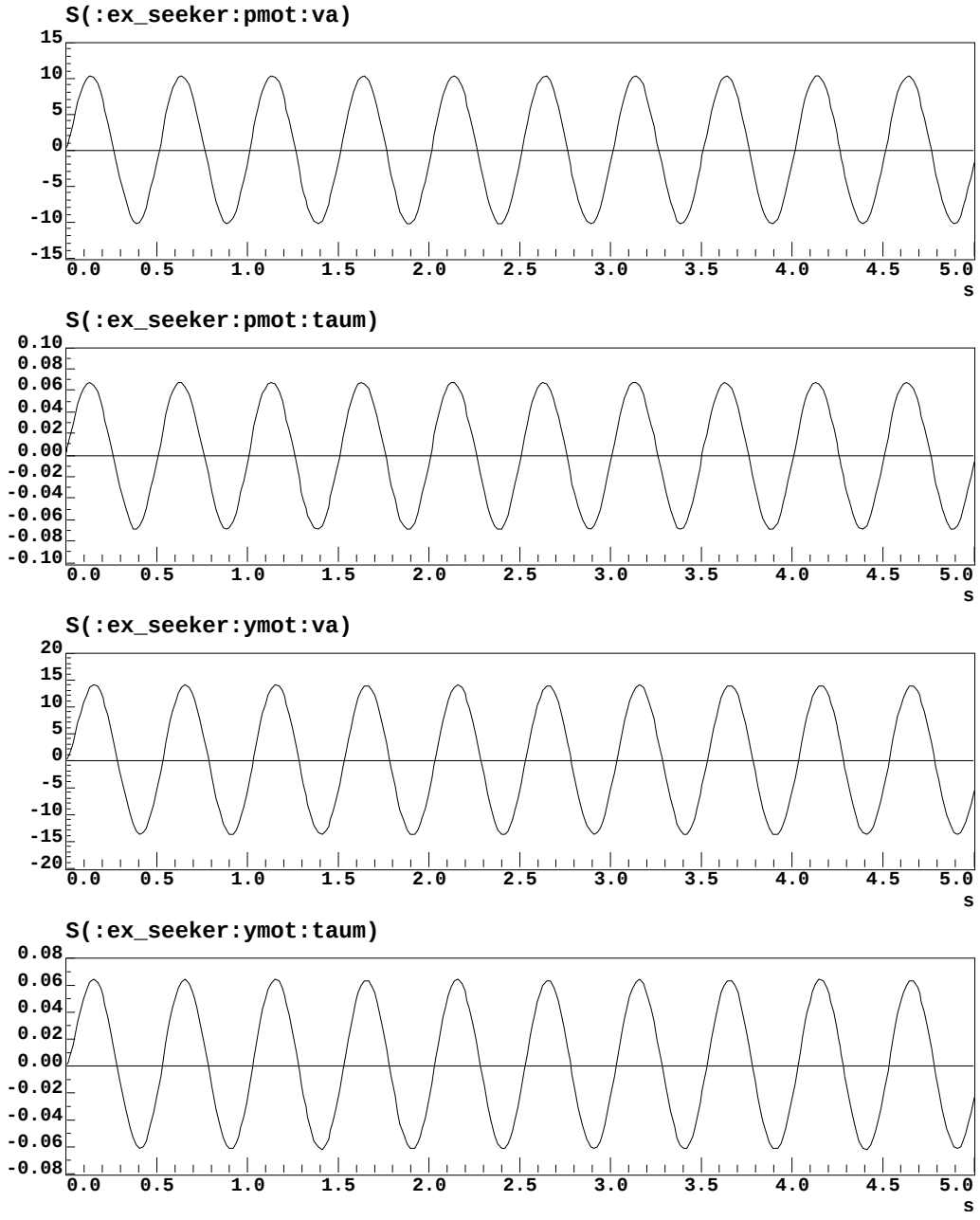


Figure 7-12. Iteration II: Input voltages and generated torques for the selected motors using refined models for the motors and the seeker.

tual description of the design. The result of the simulation at this stage is illustrated in Figure 7-12.

7.4.3 Behavior evaluation

From the simulation results shown in Figure 7-12 we verify that the selected motors can drive the loads. The pitch motor provides a maximum torque of 0.07 N-m, which results in a torque slack of 0.06 N-m. The yaw motor, on the other hand, provides a maximum torque of 0.06 N-m, which results in a torque slack of 0.18 N-m. We now proceed to complete the design of the controllers.

7.5 Iteration III

In this iteration we concentrate on the design of the controllers that must be used to optimize the behavior of the seeker. So far we have used an analog proportional controller. However, in a practical implementation, we must use a digital controller that can be implemented in a micro-controller. In this last iteration, we refine the model of the controller first by providing a refinement from the analog domain to the discrete domain. The last refinement takes the controller from the discrete domain to the discrete domain using a PWM amplifier at the output.

For this last iteration, the user requirements, specifications and engineering requirements remain the same.

7.5.1 Family of solutions

The family of solutions for this stage of the design is the subtree of the AND-OR tree rooted at the node *controller*. This family includes an analog controller, a discrete controller and a digital controller. In this case, we are interested in the bindings <*controller, discrete*> and <*controller, discrete, SM*>, which, through our reconfigurable modeling paradigm, can be selected to study different system performance.

In the principle of PWM, a dc power supply is rapidly switched at a fixed frequency *f* between “ON” and “OFF”. This frequency is often in excess of 1KHz. The high value is held during a variable pulse width *t* during the fixed period *T*. The resulting asymmetric waveform has a ***duty cycle***, defined as the ratio between the ON time and the period of the waveform, usually specified as a percentage:

$$\text{duty cycle} = \frac{t}{T}100\% \qquad \textbf{Equation 7-4}$$

As the duty cycle is changed (by the controller), the average voltage of the motor will change, causing changes in speed and torque at the output. It is primarily the change in the duty cycle and not the value of the power supply that determines the output characteristics of the motor [59].

There are two alternatives provided in the family of solutions for each controller based on the information encoded in the PWM signal. These are <*controller, discrete, SLA*>, <*controller, discrete, SM*>, <*controller, digital, SLA*>, and <*controller, digital, SM*> (see Figure 7-3). The behavior *SLA* (i.e., simple, locked anti-phase PWM) consists of a single, variable duty-cycle signal in which is encoded both direction and amplitude information. A 50% duty-cycle PWM signal represents zero drive, since the average voltage delivered to the motor is zero. On the other hand, *SM* (Sign/magnitude PWM) consists of separate direction (sign) and amplitude (magnitude) signals. The (absolute) magnitude signal is duty-cycle modulated, and the absence of a pulse signal (a continuous logic low level) represents zero drive. Voltage delivered to the motor is proportional to pulse width.

7.5.2 Observed behavior

The observed behavior is obtained for the discrete instance of the *controller* component (Figure 7-13). For this design, we have selected the sign/magnitude PWM behavior. The observed behavior of the discrete controller is shown in Figure 7-14.

7.5.3 Behavior evaluation

The error achieved by the yaw controller is 0.009 rad while the error for the pitch controller is 0.008 rad. At this last stage of the design of the seeker, we consider the error values attained by the discrete controllers to be acceptable.

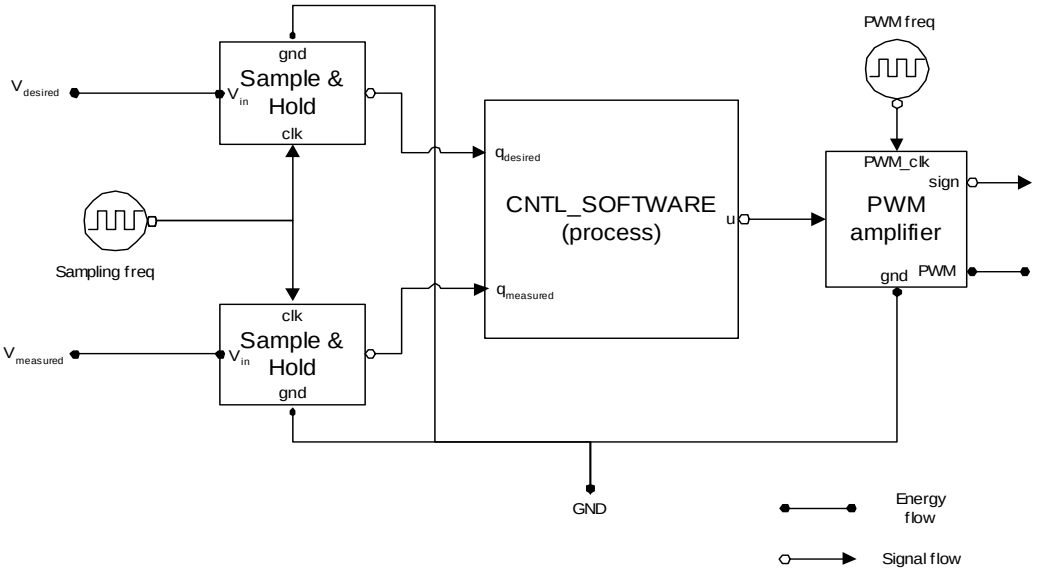


Figure 7-13. Discrete controller with PWM amplifier.

7.6 Lessons learned

Throughout the example we observed that our simulation-based design environment can help in the creation of different analysis settings, which can help in the evaluation of the design without too much effort. By being able to analyze the behavior of the design early in the design process, the designer is able to make more educated estimates that will reduce design conflicts in later stages.

The quality of the design improves because more design alternatives can be explored through the composition and instantiation of components. This is true since the use of reconfigurable models allows the designer to test different alternative components and to analyze the behavior of the system with these new components.

The port-based modeling paradigm permits reusable hierarchical models to be composed. This is also an advantage because by having a minimal set of building blocks the designer can compose a large number of designs that can be used later in new design problems.

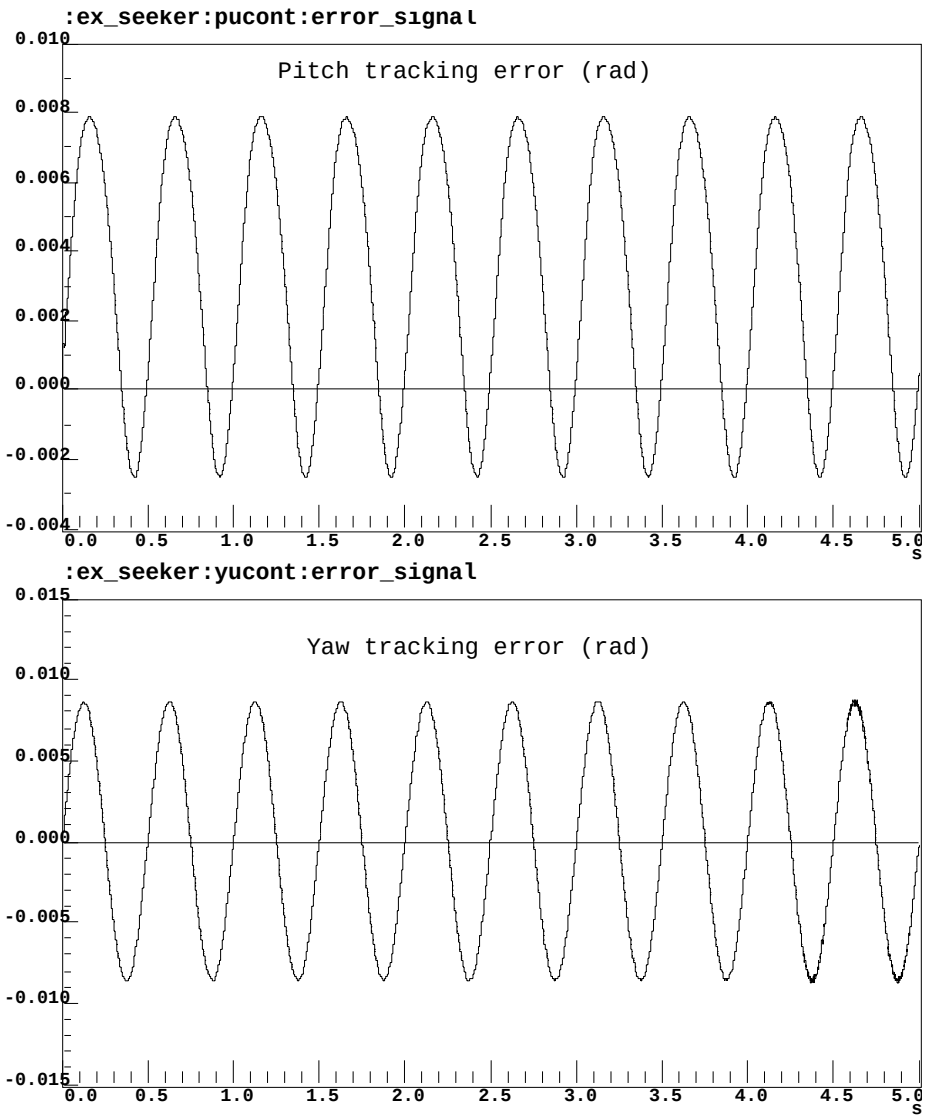


Figure 7-14. Iteration III: Tracking errors for yaw and pitch using a discrete controller.

7.7 Summary

In this chapter, we walked through a number of iterations to design a missile seeker. We started off by defining the customer needs and the requirements based on these needs. Based on the flow of design information model we identified several areas that map directly to our reconfigurable modeling paradigm. As a result, a seamless integration between the design process and evolutionary simulation (based on our reconfigurable models) was presented. The complete simulation-based design framework is shown in Figure 7-15.

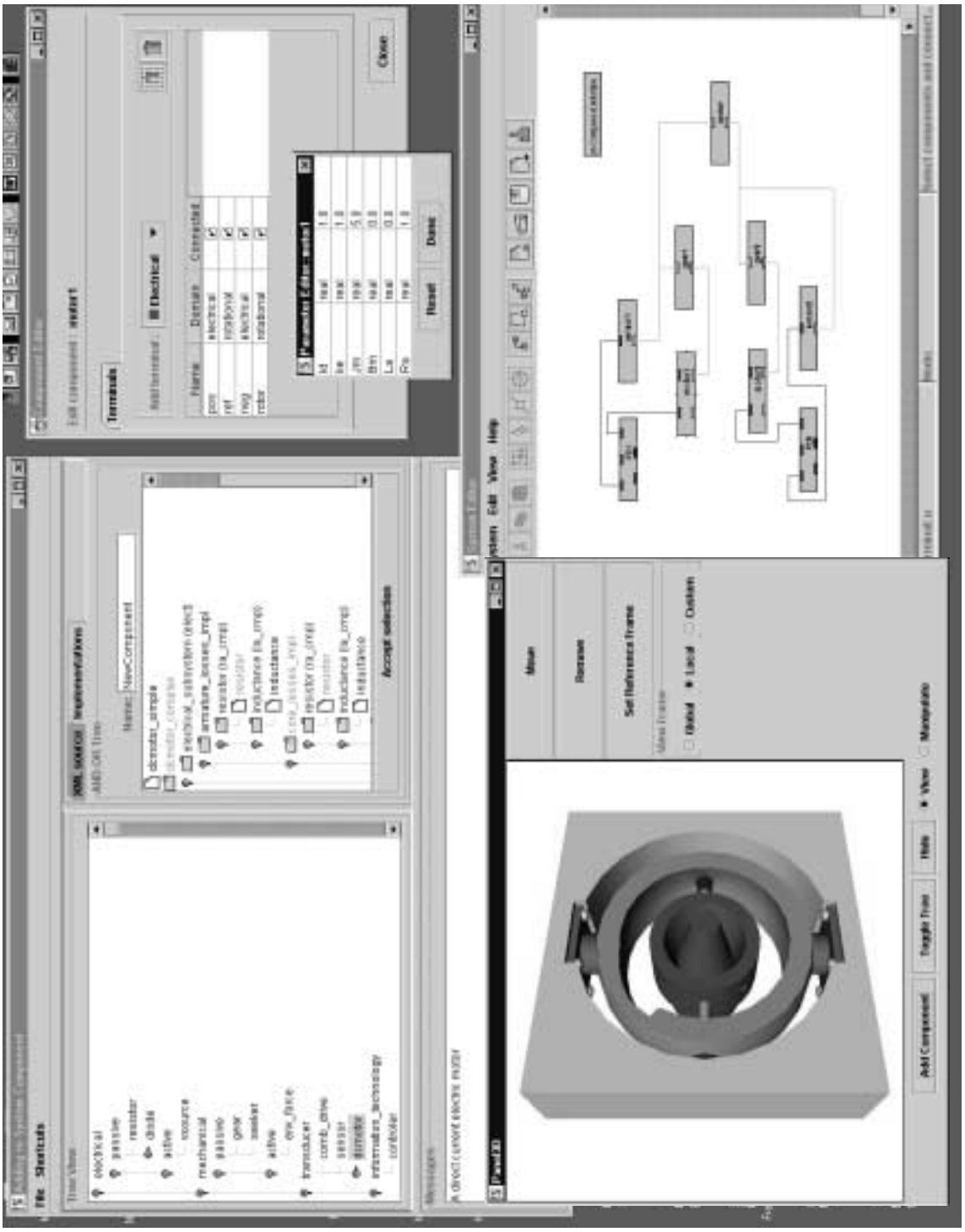


Figure 7-15. The composable modeling and simulation environment.

Chapter 8 Conclusions

8.1 Contributions

While developing a new modeling paradigm for mechatronic systems that provides modeling and simulation support to the design process, significant contributions in the area of modeling multi-domain systems were achieved. We can classify these as follows:

- Composable simulation.
- Port-based multi-domain modeling of mechatronic systems.
- Reconfigurable models.
- Structural knowledge representation.
- Multidisciplinary modeling and simulation representation.

We have grouped these contributions into two larger groups: intellectual and implementation contributions. Intellectual contributions include new ideas, and new algorithms, while

implementation contributions include new framework and new representational structures. Within the intellectual contributions of this work we can include composable simulation, port-based multi-domain modeling of mechatronic systems, and reconfigurable models. The implementation contributions of this work include the structural knowledge representation and multidisciplinary modeling and simulation environment.

8.1.1 Intellectual contributions

8.1.1.1 Composable simulation

In this thesis, we developed the idea of *composable simulation*. By composable simulation we mean the ability to generate system-level simulations automatically by simply organizing the system components.

Composition is the basis for assembling simulation models of multi-energy domain physical components. It is through composition that our port-based models are assembled into a complete model. When these models are combined into a complete system, our framework automatically combines them into a system-level simulation.

Raising the level of user interaction to composition of system components rather than composition of simulation models will result in a significant reduction of effort in creating and modifying system-level simulations and will reduce the simulation and modeling expertise required of the user.

Our framework for composable simulation will therefore enable designers to verify their physical designs with much less effort and time than is required in current simulation environments.

8.1.1.2 Port-based multi-domain modeling of mechatronic systems

Multi-domain modeling of physical systems requires means to capture the interaction between components within a single energy domain and across energy domains. To this end, we developed a novel modeling paradigm based on port-based objects [133]. The port-based object approach allows us to model system components by describing their behavior and their interaction with the environment. Interaction paths capture energy flow (for

energy-based systems) or signal flow (for non-energy based systems). In this way, we can describe a system as a graph where nodes represent high-level system components and edges represent their interactions.

Port-based objects can be compound or primitive. Compound port-based objects define the behavior of a system as a structural arrangement of subsystems (also modeled as port-based objects), while primitive port-based objects are defined by the constitutive equations describing the behavior of the object.

The port-based modeling paradigm is the basis for our multidisciplinary modeling and simulation environment, as well as for our concept of reconfigurable models. A port-based object is transformed into a hybrid mathematical representation based on linear graphs and block diagrams.

8.1.1.3 Reconfigurable models

To support the evolutionary nature of design, we extended the port-based modeling paradigm to support reconfigurability of system models. Reconfigurable models are a powerful abstraction that allows the designer to change the simulation models on the fly. The modeling paradigm of reconfigurable models is based on the separation of the boundary of the component (i.e., collection of ports) from the description of its behavior (which can be given by equations—for primitive components—or by a composition of subcomponents—for compound components). We called the boundary of the component its *interface*, and we called its behavior the *implementation*.

Using the concept of subtyping, we organize the component interfaces into a semantic network. An important virtue of this network is that by traversing it (upward or downward) we define two operations: *refinement* and *generalization*. Reconfigurability is achieved when an implementation is bound to an interface. Therefore, this network completely defines the basic operations that are required to support reconfigurable models, namely, specialization, generalization, and reconfiguration.

8.1.2 Implementation contributions

8.1.2.1 Structural knowledge representation

During design, it is necessary to have access to a set of simulation models for a given component. This set of models can be used to perform simulations at different levels of detail and at different stages of the design process. We call this group of models the model space of the component. We developed a representation to describe the model space of a reconfigurable component. The representation is based on an AND-OR tree [105]. The AND-OR tree representation systematically organizes a family of possible structures of a system, hence describing the model space of this system. Using the properties of the AND-OR tree, OR arcs denote modeling alternatives, while AND arcs denote the elements comprising an individual modeling alternative.

The structure of the AND-OR tree and the principles of composition and instantiation defined in this work are tightly related. The principle of composition is described by an AND arc pointing to all the constituents of the composed model. The principle of instantiation, on the other hand, is captured by an OR arc since it describes alternative ways of defining the component.

Based on this structure, we developed models of concrete components. These models are characterized by an induced tree of the AND-OR tree. The collection of reconfigurable models represented by this component structure are stored in a library of components.

To describe this model structure, we developed a neutral markup specification language based on XML. The purpose of this language is to facilitate sharing of reconfigurable models among the members of a team of designers.

8.1.2.2 Multidisciplinary modeling and simulation representation

We developed a novel multidisciplinary modeling paradigm that combines energy-based and non-energy based systems into a single modeling representation. The formalism used to represent a multidomain system is based on linear graphs [143]. We extended this formalism and created a hybrid representation for mechatronic systems. In this representation, energy-based systems are modeled using the linear graph formalism, and non-energy-based

systems are modeled using block diagrams. We have combined the two formalisms into a hybrid representation that allows the description of both types of systems. New elements—variable elements—seamlessly interface the two formalisms.

We developed algorithms to automatically synthesize the linear graphs for all energy domains involved, including signal, electrical, and mechanical domains. The algorithms that synthesize the linear graph for the mechanical energy domain take care to simplifying the graph. This simplification is done in order to minimize the possibility of obtaining both high-index algebraic differential equations and fully constrained mechanisms. The simplification algorithm works by identifying and removing redundant kinematic joints (i.e., joints that have coincident joint axes).

We formalized the causality problem as that of finding a minimum cost spanning tree on the linear graph. This provided a convenient way for finding causal directions for all the equations in the system. To incorporate the equations derived from the non-conservative system, we defined an extension of the classic *Block Lower Triangular* algorithm to find a feasible order of evaluation of the DAEs.

8.2 Future directions

This work is a foundation for a different approach to modeling. Our paradigm has raised several issues that need to be addressed to develop its full potential. These issues provide the basis for future research in the areas of reconfigurable models and improved support of the design activities. Some of these issues include the following:

1. *Automated model selection*—A solution to the problem of automated model selection is required to have intelligent simulation-based design advisors. A simulation-based design advisor is a design tool that can suggest appropriate simulation models based on information about the kind of analysis that is to be performed. From this information, the system should explore the model space for the component (the component AND-OR tree) and find a subset of implementations that have to be bound to the interfaces

used in the design. To accomplish this goal, we consider that the following questions should be resolved, however, these questions present only a starting point from which research in this area can leverage:

1.1. What semantic description (i.e., language representation) should we use to describe an experiment?

1.2. What is the information embedded in a class of experiments that is relevant to select a subset of models? This area would be useful to reduce the search space when working with a set of experiments.

1.3. What is the semantic content of a model?

1.4. What is the mapping of the semantic content of an experiment to model semantics?

1.5. How would the computational requirements affect the model selection?

2. *Model aggregation*—Composable simulation and port-based modeling allows us to create models of physical systems by putting together simpler models. An important problem is that of aggregation of these models. This is important because we want to be able to abstract all details of the model being created such that the abstraction can be used in larger models. The issues in this area include:

   2.1. How does the context in which the aggregate is to be used influence the aggregation boundaries? It may be possible to have many different views of a composition depending on the context in which the aggregate will be used. The simplest case would be to place an envelope around all components.

   2.2. How can the ports and terminals of the elements of the aggregate be combined into a meta-port (which may include ports and terminals)? If the composition can have different views, it must have different ways of interacting with the environment. Ports and terminals that are visible in one view may not be in another.

3. *Expanding the expressiveness of the modeling paradigm*—As we pointed out in Chapter 3, the port-based modeling paradigm can describe component interactions in any energy domain as long as the interaction is not distributed but lumped. As a result, only lumped parameter models can be described with this modeling paradigm. An

important set of physical models, however, require the use of *distributed parameter models*. Therefore, to expand the modeling capabilities of our approach, extensions to distributed parameter modeling are required. The issues involved in this area include:

3.1. What representational structures should we use to describe a distributed parameter model?

3.2. Can the existing representations be used to extend our framework?

3.3. How can we relate the distributed parameter model to the lumped parameter model?

4. *Use of modeling in collaborative design*—Design of mechatronic systems is a process that is characterized by the involvement of a number of experts in different areas, for example mechanical, electrical and software engineering. Therefore, it is necessary to provide simulation support in a collaborative design environment. In this area, teams of experts working on different portions of the design should be able to share the model of the design such that others have an updated view of the problem. The issues are related to software systems architectures, since the objective is to implement a distributed repository of models that is accessible from anywhere in the organization. These include, version control, model sharing and locking, and consistency maintenance.

5. *Product structure*—The product structure represents the physical organization of components. It provides information not available in a simulation model of the device. This information may influence the selection of the kind of models, and how these models interact. An example of such case is the physical proximity or actual mechanical contact between two components. If the experiment we are performing considers the physical proximity of components, for example to capture the effects of heat radiation of one component onto its neighboring components, the models that are selected should account for those interactions. Considering the topology of the component in a simulation run, would improve the fidelity of the simulation and give the designer a better understanding of the interactions between components that may be otherwise excluded. The issues to be addressed in this area include:

5.1. What is the taxonomy of interactions that can be derived from the product structure? Physical proximity and physical contact are two examples, but are there more?

5.2. What semantic description should these interactions have to be able to perform inferences on the type of models required?

5.3. What semantic content is required to describe the models?

6. *Intelligent behavioral search of models*—Typically, in design, one starts from functional requirements, which after selection of the appropriate physical processes, are transformed into form [96]. Once the form is synthesized, mathematical modeling can provide a description of the behavior of the form. It would be desirable to provide tools that can search for a component that meets given functional requirements. An example could be to search for an electric motor that meets some torque or angular speed requirements. This translates into a search problem on the space of available components. The result of the query should bring a subset of components that best match the requirements. Behavioral search is different from model selection. We can think of the problem of formulating a model for a physical device as being divided into two steps. One is to find the device that matches the functional requirements (behavioral search) and once the device is given, we need to select a model for that device (model selection) that matches the requirements of some experiment. The issues involved in the area of behavioral search include:

   6.1. What is the semantic description of the query (i.e., a query language)?

   6.2. What is the semantic content of a model?. This semantic content may be different from the semantic content in 1.3 above since it should reflect the properties of a model related to the design requirements.

   6.3. What is the best internal organization of models to make the query efficient? This question is related to the design of the component repository. Should we use an object-oriented data-base?, relational database?, distributed?

7. *Functional models*—The function of a device is its intended behavior. The functional specification describes the device's goals. Functions are achieved through form. The form of a device is where the physical processes associated with the device take place. If we associate behaviors with form and combine them into a single component model, we obtain component models containing a description of their form and one or more behavioral descriptions. Using functional models in a simulation-based design environment requires that we address the following issues:

    7.1. Mapping from function to form. If we associate the behavioral description of a device to form, it may be possible to synthesize form from function. The combined use of behavior and form could be used to test, through simulation, that the behavior of a device matches the given functional requirements; this problem is related to the issue of intelligent behavioral search of models (above). However, once a set of (behavior of) devices has been identified to match the functional requirements, further analysis is required to find the appropriate physical realization (form) of the device; i.e., to verify that the physical characteristics of the selected device match the physical requirements (dimensions, materials, etc.)

    7.2. Mapping from form to behavior. In this context, the problem is to find a kinematic model that describes the form. This kinematic model is used in conjunction with the behavioral description of the system to provide a complete model of the system. To this end, work is in progress in our center. This work deals with the automatic synthesis of behavioral models that describe interaction between geometric components [126]. Given the components for these interactions, the dynamic model of the form can be synthesized and parametrized by obtaining lumped parameters from the CAD model.

8. *Integration with commercial CAD programs*—We envision that the composition of simulation models will consider the geometry of the physical device. In the current implementation, the geometry is taken into account using a CAD package developed in our group. If the modeling paradigm and software environment presented in this dissertation are to be used in an engineering setting, the framework must be integrated to commercial CAD packages like I-DEAS or ProEngineer.

8.3 Conclusions

In this dissertation, we presented a composable simulation framework to support the design of mechatronic systems. Our framework allows simulations to be assembled from high-level component descriptions, which results in a significant reduction in time, and hence cost, of developing new simulations. Designers can take full advantage of these features to test new designs or to test changes to existing designs. This could translate in a larger design space being considered which may result in better designed products.

The concept of reconfigurable models provides three basic operations: specialization, generalization, and reconfiguration. This allows the designer to create simulations at different levels of detail by simply changing the implementations associated with the interfaces in the design.

Reconfigurable models incorporate parameterization, typing, and port-based interfaces, which allow building models as networks of encapsulated, reusable subsystems that are explicitly classified. The classification of these models is given in the type hierarchy that we used to organized the models in the library. It is explicit because we can determine the properties of a model from the type hierarchy through inheritance. The type hierarchy provides the information needed about a particular model at any time.

When we consider a component such that it is divided into two parts, an interface that defines essential properties and an implementation that defines non-essential properties, we achieve modularization of the component. Furthermore, when combining subtyping with modularization, it becomes possible to define instantiations of an abstract concept (interface) with an implementation. We can also define specializations or generalizations of an abstract concept with the subtype relation.

Our framework allows the designer to consider concurrently the integrated system at different levels of abstraction, ranging from low-resolution models to high-resolution models. Additionally, it gives the designer an unequaled flexibility to manipulate the model of the design.

Appendix A Linear Graph Theory

A.1 Introduction

Linear graph theory, a branch of combinatorial mathematics, has proved to be a useful tool for the study of large and complex systems. Leonhard Euler wrote one of the first papers on graph theory and laid the foundation for the theory when he published the solution to the Königsberg bridge problem in 1736. However, its first application to an engineering problem did not arise until 1847, when Gustav Kirchhoff applied it to the study of electrical networks.

This appendix presents a review of the basic concepts in linear graph theory, namely, the topological and algebraic properties. The material is based on the seminal work by Branin [17] on network analogies for physical systems, and the work by Seshu and Reed [121] on electrical networks.

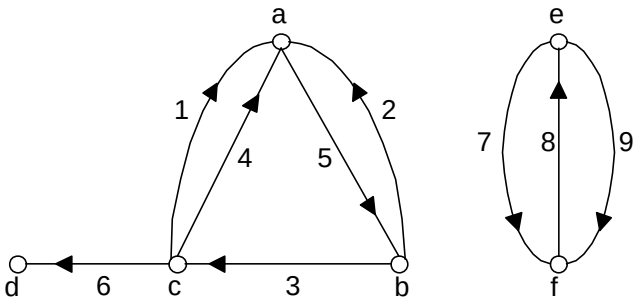


Figure A-1. A directed linear graph

A.2 Basic definitions of linear graphs

The basic elements comprising a linear graph are line segments (called edges) each having two end points (called vertices). If a direction is specified on each edge, they become oriented edges and the graph is referred to as directed graph. Schematically, we can indicate the edge orientation two ways, namely, by attaching a + and a – to the two ends of the edge or by attaching an arrowhead directed from the + end to the – end of the edge.

Definition *Linear graph*. A *linear graph* $\mathbf{G}$ is a collection of edges, no two of which have a point in common that is not a vertex.

Definition *Connected graph*. A graph $\mathbf{G}$ is connected if there exists a path between any two vertices of the graph. If the graph is not connected, it contains *p* connected components.

Figure A-1 shows an example of an oriented linear graph. This graph is a non-connected graph having two connected components. The arrowheads on each edge indicate the directionality of the edges.

The following definitions set the basic terminology for linear graph theory.

Definition *Subgraph*. A *subgraph* $\mathbf{G}_s$ is a subset of edges of the graph $\mathbf{G}$. Subgraph $\mathbf{G}_s$ is a proper subgraph if it does not contain all edges of $\mathbf{G}$.

Definition *Incidence*. Edge *k* is incident to a vertex *p* if *p* is an endpoint of *k*.

Definition *Degree of vertex*. The degree of a vertex is the number of edges incident to that vertex.

Definition *Edge sequence*. An *edge sequence* is any subset $\mathbf{E}_s$ of the edges of $\mathbf{G}$ where $\mathbf{E}_s$ can be ordered such that an edge k in $\mathbf{E}_s$ has a vertex in common with the preceding edge $k-1$, and the other vertex in common with the succeeding edge $k+1$.

Definition *Path*. A *path* is an edge sequence where each internal vertex has a degree of exactly two and each terminal vertex is of degree 1.

Definition *Circuit* or *loop*. A *circuit* or *loop* is a closed edge sequence where all vertices are of degree 2.

A tree $\mathbf{T}$ is a connected acyclic subgraph of a connected graph $\mathbf{G}$ that contains all the vertices of $\mathbf{G}$ but contains no loops. The edges that are not part of the tree form a subgraph $\bar{\mathbf{T}}$ called *cotree*. Edges in the tree are referred to as *branches,* while edges in the cotree are referred to as *links* or *chords*. For a graph with e edges and v vertices, there are exactly $v-1$ branches. Consequently, the number of chords in the cotree equals $e-v+1$. If the graph $\mathbf{G}$ is not connected, a tree does not exists because by definition, a tree is a connected subgraph of a connected graph. However, a tree can be found for each of the connected components of $\mathbf{G}$. The collection of such trees is called a *forest*. Similarly, each component of $\mathbf{G}$ defines a cotree and the collection of all cotrees is called a *coforest.* Therefore, in a graph with e edges, v vertices and p connected components there will be $v-p$ branches in the p trees (forest) and $e-v+p$ chords in the p cotrees (coforest).

Let $\mathbf{G}$ be a connected graph with v vertices and e edges (i.e., $p = 1$) and $\mathbf{T}$ be a tree of $\mathbf{G}$. If we add any chord between any two vertices in the tree, we establish a circuit. Since in a connected graph there are $e-v+1$ chords for a given tree $\mathbf{T}$, there are as many unique circuits defined by the chords of $\mathbf{T}$. The fundamental circuits (*f-circuits)* of a connected graph $\mathbf{G}$ for a tree $\mathbf{T}$ are the $e-v+1$ circuits formed by each chord of the given tree $\mathbf{T}$.

As an example, consider the graph $\mathbf{G}$ shown in Figure A-2. The tree $\mathbf{T}$=(a, b, g, d) is indicated by bold edges in the figure. The number of edges and vertices is nine and five, respec-

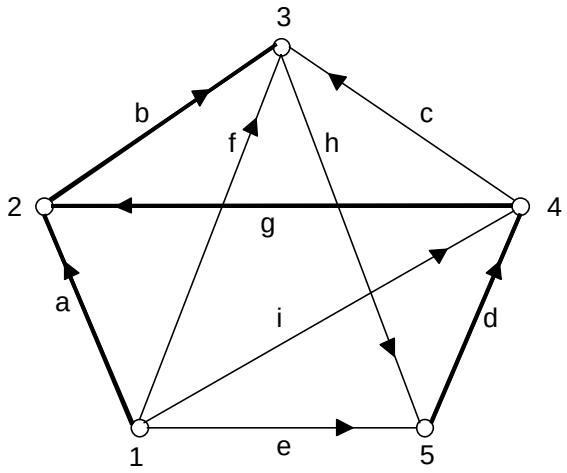


Figure A-2. A connected graph with a tree **T** indicated by bold edges

tively; therefore, the number of chords in the graph is five. The five fundamental circuits defined by these chords and their tree paths are listed in Table A-1.

Table A-1. Fundamental circuits for the tree of figure Figure A-2

Chord	**Tree path**	**f-circuit**
(f)	(a, b)	(a, b, f)
(h)	(b, g, d)	(b, g, d, h)
(c)	(b, g)	(b, g, c)
(i)	(a, g)	(a, g, i)
(e)	(a, g, d)	(a, g, d, e)

The table includes the f-circuits defined with respect to the tree **T** = (a, b, g, d). It has been proved that *any circuit in a graph can be made an f-circuit with respect to some tree* [121]. This property is of utmost importance when we analyze the algebraic structure of the graph since it allows us to select the causality of the equations of the physical system being modeled.

Since the graph is directed, it is appropriate to consider the f-circuits oriented. As stated above, a chord uniquely defines an f-circuit; therefore, it is natural to assign the orientation of the f-circuit to be consistent with the direction of the defining chord.

A circuit in a linear graph has a dual called cut-set. A cut-set of a connected graph $\mathbf{G}$, is defined as a set C of edges of $\mathbf{G}$ such that the removal of these edges from $\mathbf{G}$ leaves $\mathbf{G}$ partitioned in exactly two connected components.

A *fundamental system of cut-sets* (f-cutsets) with respect to a tree $\mathbf{T}$ of a connected graph $\mathbf{G}$ is the set of $v - 1$ cut-sets in which each cut-set includes a branch of $\mathbf{T}$. The fundamental cut-set orientation is to agree with the orientation of the defining branch.

We end this section by presenting an interesting property of a fundamental cut-set: if e_1 represents a branch of a tree $\mathbf{T}$ in a connected graph $\mathbf{G}$, and $(e_1, e_2, e_3, \ldots, e_n)$ represents the cut-set defined by e_1, then each of the f-circuits defined by the chords $e_2, e_3, \ldots, e_n$ includes e_1. To illustrate this property, let e_1 be edge a in the graph of Figure A-2. The f-cutset defined by this branch is (a, f, i, e) since the removal of this set leaves the graph in two components one of which is the isolated vertex 1. From Table A-1, we can see that the f-circuits defined by the chords (f, i, e) all include edge a.

A.3 Matrix representations of linear graphs

The connectivity relations of any oriented linear graph can be completely specified by means of the augmented incidence matrix, denoted $\bar{\mathbf{A}}$.The incidence matrix contains information both about the orientation of edges in the graph and how they are joined to form nodes. For a directed graph is $\mathbf{G}$ with v vertices and e edges the incidence matrix is a $v \times e$ matrix with entries $\bar{a}_{ij}$ defined by:

$$\bar{a}_{ij} = \begin{cases} 1 \text{ if edge } j \text{ is incident at vertex } i \text{ and is oriented away from vertex } i \\ -1 \text{ if edge } j \text{ is incident at vertex } i \text{ and is oriented toward vertex } i \\ 0 \text{ if edge } j \text{ is not incident at vertex } i \end{cases}$$

In general, for a graph with p connected components, the incidence matrix is a *direct sum*. A matrix $\mathbf{M}$ is said to be a direct sum of $\mathbf{M}_1, \mathbf{M}_2, \ldots \mathbf{M}_p$ if for any $\mathbf{M}_k$ in $\mathbf{M}$ no nonzero element lies in a row or column of $\mathbf{M}$ associated with any of the other submatrices [143]. The existence of a direct sum in a matrix always indicates the existence of subgraphs; therefore,

the $\mathbf{M}_k$ matrices can be regarded as the incidence matrices of each of the p connected components.

The incidence matrix for the graph shown in Figure A-1, is given by Equation A-1. As expected, this matrix is a direct sum; there are two connected components in the graph, which show up in the incidence matrix.

$$\bar{\mathbf{A}} = \left[\begin{array}{cccccc|ccc} -1 & -1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 \end{array}\right] \qquad \textbf{Equation A-1}$$

Consider the incidence matrix $\bar{\mathbf{A}}$ of a connected graph ($p = 1$) $\mathbf{G}$. Since the sum of all rows of $\bar{\mathbf{A}}$ equals zero, its rows are linearly dependent. Removing any row from $\bar{\mathbf{A}}$ will leave $v-1$ linearly independent rows. We call this new matrix the reduced incidence matrix, denoted $\mathbf{A}$. From graph theory [121], we know that, if $\mathbf{T}$ is a tree of a connected graph $\mathbf{G}$, the $v-1$ columns of $\mathbf{A}$ that correspond to the branches of the tree $\mathbf{T}$ constitute a nonsingular matrix. Thus if a tree is chosen and the columns of $\mathbf{A}$ are properly arranged, the matrix $\mathbf{A}$ can be partitioned into the $(v-1)\times(v-1)$ submatrix $\mathbf{A}_T$, referring to the branches of the tree only, and the $(v-1)\times(e-v+1)$ submatrix $\mathbf{A}_C$, referring to the chords or to the cotree.

$$\mathbf{A} = \left[\mathbf{A}_T\ \mathbf{A}_C\right] \qquad \textbf{Equation A-2}$$

Two new matrices can be defined to describe the topology of the graph. The *fundamental circuit matrix* (designated $\mathbf{B}$) captures the connectivity relations between circuits and edges, and the *fundamental cut-set matrix* (henceforth referred to as the cut-set matrix) designated $\mathbf{Q}$. Matrix $\mathbf{Q}$ captures the connectivity between cut-sets and edges.

The *fundamental circuit matrix* $\mathbf{B}$ of a directed graph $\mathbf{G}$ with respect to a tree $\mathbf{T}$ is defined by the $e-v+1$ circuits formed by each chord as follows:

$$b_{ij} = \begin{cases} 1 \text{ if edge } j \text{ is in circuit } i \text{ and the orientation of the circuit and the edge coincide} \\ -1 \text{ if edge } j \text{ is in circuit } i \text{ and the orientation of the circuit and the edge do not coincide} \\ 0 \text{ if edge } j \text{ is not in circuit } i \end{cases}$$

If the columns of matrix **B** are properly arranged matrix **B** can be partitioned into the $(e - v + 1) \times (v - 1)$ submatrix $\mathbf{B}_T$ referring to the branches of the tree and the $(e - v + 1) \times (e - v + 1)$ submatrix $\mathbf{B}_C$ referring to the chords of the cotree. However, since each chord appears exactly once in any given f-circuit in the positive sense, the matrix $\mathbf{B}_C = \mathbf{U}_C$; i.e., a unit matrix. Then we can write

$$\mathbf{B} = \left[\mathbf{B}_T \; \mathbf{U}_C\right] \qquad \textbf{Equation A-3}$$

A closer look to the matrices **A** and **B** will reveal a very interesting and fundamental property of linear graphs. We can define two linear vector spaces associated with the graph **G**, namely, the vector space V_Q spanned by the rows of matrix **A**, and the vector space V_B spanned by the rows of matrix **B**. These two vector spaces are subspaces of the linear vector space V_G of dimension *e*. It can be shown [121] that the matrices **A** and **B** satisfy the following relation:

$$\mathbf{A}\mathbf{B}^{\mathsf{T}} = \mathbf{0} \text{ and } \mathbf{B}\mathbf{A}^{\mathsf{T}} = \mathbf{0} \qquad \textbf{Equation A-4}$$

This implies that the two vector subspaces V_Q and V_B are orthogonal complements of the *e*-dimensional linear vector space V_G. This fact is known as the *orthogonality principle*.

Using this fact, we can derive an equation to find matrix $\mathbf{B}_T$:

$$\left[\mathbf{A}_T \; \mathbf{A}_C\right] \begin{bmatrix} \mathbf{B}_T^{\mathsf{T}} \\ \mathbf{U}_C \end{bmatrix} = \mathbf{A}_T\mathbf{B}_T^{\mathsf{T}} + \mathbf{A}_C\mathbf{U}_C = \mathbf{0} \qquad \textbf{Equation A-5}$$

It follows that

$$\mathbf{B}_T = -\mathbf{A}_C^{\mathsf{T}}(\mathbf{A}_T^{-1})^{\mathsf{T}} \qquad \textbf{Equation A-6}$$

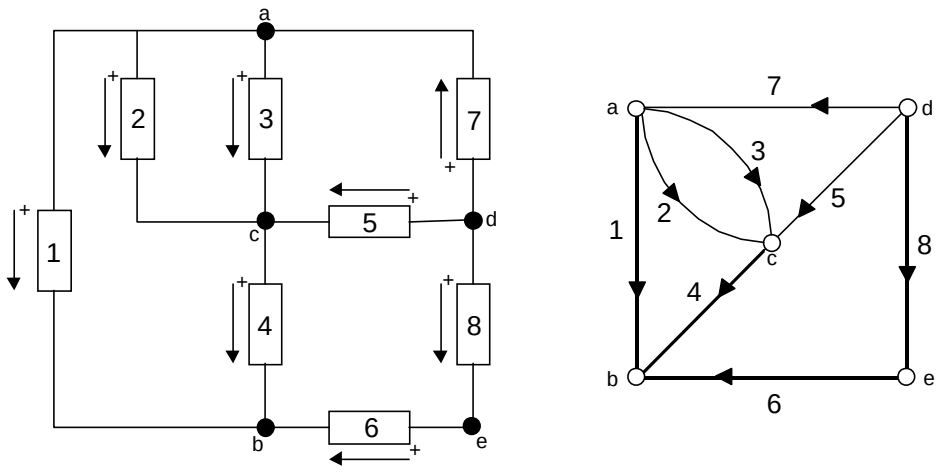


Figure A-3. An electrical network and its associated linear graph.

When the columns of matrix **A** are properly arranged such that the first $v - 1$ columns of **A** are in direct correspondence with the branches of some tree **T** of a graph **G**, an equivalent matrix—two matrices **A** and **Q** are equivalent when their rows span the same vector space—**Q** can be derived. This matrix is derived from matrix **A** by applying row operations to **A**. Matrix **Q** represents the *fundamental system of cut-sets* with respect to the tree **T**. It includes the $v - 1$ cut-sets of **G** in which each cut-set includes only one branch of **T**. Then

$$\mathbf{Q} = \left[\mathbf{U}_T \; \mathbf{Q}_C\right] \qquad \textbf{Equation A-7}$$

Since matrices **A** and **Q** are equivalent, the following relation holds

$$\mathbf{B}\mathbf{Q}^{\mathsf{T}} = \mathbf{0} \qquad \textbf{Equation A-8}$$

It follows from Equation A-5 that

$$\mathbf{B}_T = -\mathbf{Q}_C^{\mathsf{T}} \qquad \textbf{Equation A-9}$$

A.4 The algebraic structure associated with a linear graph

We will illustrate the algebraic structure associated with a linear graph, with the analysis of a simple electrical network. Figure A-3 shows an electrical network and a linear graph topologically equivalent to the network. The numbering of the elements in the network and

the edges in the linear graph makes it easy to see the correspondence between them. In the figure, all current directions and voltage references have been indicated.

It is a trivial exercise to derive the Kirchhoff's current and voltage equations for this network and they are given as follows:

$$\begin{aligned} i_1(t) + i_2(t) + i_3(t) - i_7(t) &= 0 \\ -i_1(t) - i_4(t) - i_6(t) &= 0 \\ -i_2(t) - i_3(t) + i_4(t) - i_5(t) &= 0 \\ i_5(t) + i_7(t) + i_8(t) &= 0 \\ i_6(t) - i_8(t) &= 0 \end{aligned}$$

Equation A-10

and

$$\begin{aligned} -v_1(t) + v_2(t) + v_4(t) &= 0 \\ -v_1(t) + v_3(t) + v_4(t) &= 0 \\ v_4(t) + v_5(t) - v_6(t) - v_8(t) &= 0 \\ v_1(t) - v_6(t) + v_7(t) - v_8(t) &= 0 \end{aligned}$$

Equation A-11

Rewriting the two systems of equations (A-10) and (A-11) in matrix form we obtain

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & -1 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} i_1(t) \\ i_2(t) \\ i_3(t) \\ i_4(t) \\ i_5(t) \\ i_6(t) \\ i_7(t) \\ i_8(t) \end{bmatrix} = \mathbf{0}$$

Equation A-12

and

$$\begin{bmatrix} -1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & -1 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} v_1(t) \\ v_2(t) \\ v_3(t) \\ v_4(t) \\ v_5(t) \\ v_6(t) \\ v_7(t) \\ v_8(t) \end{bmatrix} = \mathbf{0} \qquad \textbf{Equation A-13}$$

The coefficient matrices in the previous equations can be recognized as the augmented incidence matrix and the fundamental circuit matrix respectively of the directed graph shown in Figure A-3. This observation is general and it is applicable to any system for which a directed linear graph can be obtained (i.e., the network problem [17]). This fact leads to the definition of the two Kirchhoff theorems that state that the sum of currents leaving a node equals zero and the sum of voltages around a loop equals zero. This can be written as:

$$\bar{\mathbf{A}}\mathbf{i}(t) = \mathbf{0} \qquad \textbf{Equation A-14}$$

Where $\bar{\mathbf{A}}$ is the incidence matrix of the directed graph and $\mathbf{i}(t) = \begin{bmatrix} i_1(t) & i_2(t) & \dots & i_e(t) \end{bmatrix}$ where $i_j(t)$ is associated with edge *j*. Similarly,

$$\mathbf{B}\mathbf{v} = \mathbf{0} \qquad \textbf{Equation A-15}$$

Where $\mathbf{B}$ is the fundamental circuit matrix of the directed graph with respect to some tree $\mathbf{T}$, and $\mathbf{v}(t) = \begin{bmatrix} v_1(t) & v_2(t) & \dots & v_e(t) \end{bmatrix}$ where $v_j(t)$ is associated with edge *j*. For the example above, the tree $\mathbf{T}$ = (1, 4, 6, 8).

We know that the incidence matrix is a singular matrix for which we can remove a row to obtain the reduced incidence matrix $\mathbf{A}$, which is full rank. Thus, in a connected graph with *v* vertices, there are exactly $v - 1$ linearly independent Kirchhoff's current equations. In general, if the graph contains *p* connected components, there are $v - p$ linearly independent Kirchhoff's current equations. Similarly, there are $e - v + p$ linearly independent Kirchhoff's voltage equations for a network of *p* connected components.

Theorem A-I. If $\mathbf{T}$ is any tree of a connected graph, the voltage functions of the chords of

$\mathbf{T}$ can be expressed as linear combinations of the voltage functions of the branches of $\mathbf{T}$, and the current functions of the branches of $\mathbf{T}$ can be expressed as linear combinations of the current functions of the chords of $\mathbf{T}$ [121].

Proof. To prove the first part of the theorem, let us assume that the columns of $\mathbf{B}$ are properly arranged such that they include the chords of the defining cotree as the last $e - v + 1$ columns, and the vector $\mathbf{v}$ is arranged accordingly:

$$\begin{bmatrix}\mathbf{B}_f & \mathbf{U}\end{bmatrix}\begin{bmatrix}\mathbf{v}_b(t)\\ \mathbf{v}_c(t)\end{bmatrix} = \mathbf{0} \qquad \textbf{Equation A-16}$$

Expanding Equation A-16 we obtain $\mathbf{B}_f\mathbf{v}_b(t) + \mathbf{v}_c(t) = \mathbf{0}$, which can be solved for $\mathbf{v}_c(t)$:

$$\mathbf{v}_c(t) = -\mathbf{B}_f\mathbf{v}_b(t) \qquad \textbf{Equation A-17}$$

This shows that the chord voltages can be expressed as linear combinations of the branch voltages.

For the second part of the proof, we recognize that the cut-set matrix $\mathbf{Q}$ of $v - 1$ cut-sets and rank $v - 1$ defines a set of equations $\mathbf{Q}\mathbf{i}(t) = \mathbf{0}$, which are equivalent[1] to the Kirchhoff's current equations $\mathbf{A}\mathbf{i}(t) = \mathbf{0}$. If the columns of $\mathbf{Q}$ are properly arranged to include the branches of the defining tree as the first $v - 1$ columns, and the vector $\mathbf{i}$ is arranged accordingly we have

$$\begin{bmatrix}\mathbf{U} & \mathbf{Q}_f\end{bmatrix}\begin{bmatrix}\mathbf{i}_b(t)\\ \mathbf{i}_c(t)\end{bmatrix} = \mathbf{0} \qquad \textbf{Equation A-18}$$

Expanding Equation A-18 we obtain $\mathbf{i}_b(t) + \mathbf{Q}_f\mathbf{i}_c(t) = \mathbf{0}$, which can be solved for $\mathbf{i}_b(t)$:

$$\mathbf{i}_b(t) = -\mathbf{Q}_f\mathbf{i}_c(t) \qquad \textbf{Equation A-19}$$

This equation defines the branch currents as linear combinations of the chord currents. •

1. Two systems of linear equations are equivalent if they have the same solution.

Equations (A-17) are referred to as *fundamental circuit equations* and equations (A-19) are referred to as *fundamental cut-set equations.*

Appendix B MDL Grammar

In this appendix, we present the grammatical rules that define the high-level language used to describe component models and configurations.

```
module_def     ::=  module identifier module_qualifier
                    module_body endmodule ;

module_qualifier ::=is | isa configuration with

module_body    ::=  interface_def |
                    body_def |
                    interface_def body_def

interface_def ::=  interface interface_constituent ;

interface_constituent ::=
                    sizes_decl |
                    interface_constituent ; sizes_decl

body_def       ::=  submodules body_decl ;
                    connections body_connections ;
                    initialization body_initializations ;
```

```
body_decl     ::=  body_item_decl |
                   body_decl ; body_item_decl

body_connections ::=
                   body_connection_decl |
                   body_connections ; body_connection_decl

body_initializations ::=
                   body_init_decl |
                   body_initializations ; body_init_decl

body_item_decl::=  name_list isa module_decl

module_decl   ::=  module identifier

body_connection_decl::=
              in [ number ] @ identifier . in [ number ] |
              out [ number ] @ identifier . out [ number ] |
              identifier . in [ number ] @ in [ number ] |
              identifier . out [ number ] @ out [ number ] |
              identifier . in [ number ] @
                   identifier . out [ number ] |
              identifier . out [ number ] @
                   identifier . in [ number ]

body_init_decl::=  setstate ( init_statement_args ) |
                   setparam ( init_statement_args )

init_statement_args ::=identifier , array_def

sizes_decl    ::=  declare ( field_decl , bool_or_dim )

field_decl    ::=  inputs | outputs |
                   states | dft

bool_or_dim   ::=  true | false | num

array_def     ::=  [ expr_list ] |
                   { array_element_def }

array_element_def ::=
                   array_element |
                   array_element_def , array_element

array_element ::=  [ expr_list ]

expr_list     ::=  expr |
                   expr_list , expr
```

```
expr          ::=  num |
                   expr + expr |
                   expr - expr |
                   expr * expr |
                   expr / expr |
                   - expr |
                   expr ^ expr |
                   ( expr )

name_list     ::=  identifier |
                   name_list , identifier
```

Appendix C C-language interface component specification

C.1 C-language interface specification for software components

The C-language interface specification for software components provides an application programming interface (API) to the kernel model. The API provides a collection of standard methods, which are implemented as C functions. The methods in this API are classified in two mayor groups: methods called by the simulation kernel in response to the state of the module, and methods called by the implementation of the design entity to access its internal data structures maintained by the kernel. Methods called by the kernel include: `sbsDerivatives`, `sbsOutputs`, `sbsInitializeSizes` and `sbsTerminate` (Listing C-2). Two additional methods are shown Listing C-2, namely, `DECLARE_CLASS` and `ssGetFcnParams`. These methods are used to register a new kernel module, and to define aliases to the module's parameters. Methods called by the implementation of the

Listing C-2. Template for implementing a design entity

```
#include “sbs.h”

DECLARE_CLASS(class_name)

#define parameter_1ssGetFcnParam(S,0)[0]

static void
sbsDerivatives(double t, const double* x, const double* u,
                double* dx, sbsTask* S) {
}

static void
sbsOutputs(double t, const double* x, const double* u,
                double* y, sbsTask* S) {
}

static void sbsInitializeSizes(sbsTask* S) {
}

static void sbsTerminate(sbsTask* S) {
}
```

design entity include methods to declare and query the sizes and values of its input, output and states vectors, and access operations for work areas.

`DECLARE_CLASS(module)`—This macro is used to register `module` the new class in the kernel and prepare all internal data structures to handle the new class. It automatically forward declares all of the class functions (e.g., `sbsOutputs`) and generates the necessary code to initialize the internal virtual function table used by the kernel to fetch the appropriate class method.

`ssGetFcnParams(sbsTask*, int)`—This macro is called to access the parameters of a module that has been previously defined in a configuration. The first argument is a pointer to an `sbsTask` structure that represents the module class and the second parameter is an index in the parameter array defined in the MDL specification. The macro returns a list of array elements where the 0-th element is the first array in the list; the first element is the second array etc.

`sbsDerivatives()`—Once the task is spawned this function is called by the kernel to compute the derivatives of the design entity. The function is called every major and

minor integration step.

`sbsOutputs()`—Similarly to the sbsDerivatives function, this function is called by the kernel to compute the outputs of the module implementing the design entity. The function is called every major and minor integration step.

`sbsInitializeSizes()`—This function is called once at the moment when the task is spawned in the kernel to set up all the sizes of all internal data structures used to keep the state of the instance of the class. These include the number of continuous and discrete states, the number of inputs and outputs, and the sizes of the work areas.

`sbsTerminate()`—This function is called once at the end of the integration. It may be used to save information that is stored in the internal buffers associated with the class instance onto secondary storage. It may also be used to free space allocated to local work vectors assigned to the class instance as a request in the `sbsInitializeSizes` function.

The implementation code of a design entity is reentrant; therefore, these methods *must not* define any local storage. If they do, then all instances will use the same storage causing data corruption and unpredictable results. For this reason, each instance of a class has independent storage work areas that are initialized and accessed through a set of macros. There are three work areas that can be used for this purpose: real work vector, integer work vector, and pointer work vector. The macros used in the initialization of the length of these work areas have the form `ssSetNum?Work()` where '`?`' can take any of the following values: `R`, `I`, or `P` for real, integer or pointer type vectors.

Two operations are defined on these work areas, namely, read and write. These operations are implemented by a set of macros, which have the general forms `ssGet?Work` and `ssSet?Work.` The former is used to read values from the work area, and the later is used to write values to the work area. Similarly to the initialization macros, the '`?`' can take values R, I, or P.

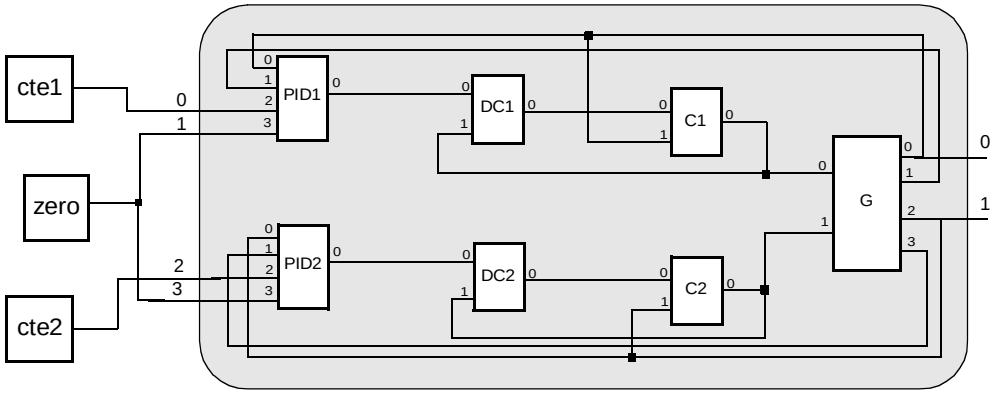


Figure C-1. Missile seeker system

An example of the use of API and the MDL language is presented next. With reference to Figure C-1 Listing C-3 shows the MDL specification for the missile seeker, while Listing C-4 shows the implementation of software component DC using the API defined in this appendix.

Listing C-3. System-level definition of the missile seeker.

```
module system isa configuration with
  submodules
    seeker_m isa module seeker;
    cte1, cte2, zero isa module constant;

  connections
    cte1.out[0] @ seeker_m.in[0];
    cte2.out[0] @ seeker_m.in[2];
    zero.out[0] @ seeker_m.in[1];
    zero.out[0] @ seeker_m.in[3];

  initialization
    setParam(cte1, [0.14]);
    setParam(cte2, [-0.14]);
    setParam(zero, [0]);
endmodule;
```

Listing C-4. C--language implementation of software component G.

```
/* Module: dcmotor.c
     Comments: Implementation of the dcmotor device interface*/

#include <stdio.h>
#include "sbs.h"

DECLARE_CLASS(dcmotor)

#define Jm  ssGetFcnParam(S,0)[0]
#define bm  ssGetFcnParam(S,0)[1]
#define Km  ssGetFcnParam(S,0)[2]
#define K1  ssGetFcnParam(S,0)[3]
#define n   ssGetFcnParam(S,0)[4]

static void sbsDerivatives(
     double t,        /* current simulation time       */
     const double* x, /* the states vector             */
     const double* u, /* the input vector              */
     double* dx,      /* the derivatives vector        */
     sbsTask* S       /* the sbs struct for this block */
) {
     double tau_m = Km * K1 * u[0];
     dx[0] = x[1];
     dx[1] = (tau_m - bm * x[1] - (u[1] / n)) / Jm;
}

static void sbsOutputs(
     double t,            /* current simulation time       */
     const double* x,     /* the states vector             */
     const double* u,     /* the input vector              */
     double* y,           /* the output vector             */
     sbsTask* S           /* the sbs struct for this block */
{
     *y = x[0];
}

static void sbsInitializeSizes(sbsTask* S) {
     ssSetNumContStates(S, 2); ssSetNumDiscStates(S, 0);
     ssSetNumInputs(S, 2); ssSetNumOutputs(S, 1);
     ssSetNumRWork(S, 0); ssSetNumIWork(S, 0);
     ssSetNumPWork(S, 0);
}

static void sbsTerminate(sbsTask* S) { }
```

C.5 C API implementation

This section presents the C application programming interface provided to develop simulation software modules.

```
/* File: sbs.h
 * Comments:
 *  Data structures and access methods for sbs-tasks
 *  Any model is an sbs-task. The sbsTask contains all entry points
 *  to the sbs-task (e.g., sbsOutputs) as well as any data
 *  associated with the sbs-task.
 */

#ifndef _SBS_H
#define _SBS_H

#define _QUOTE1(name) #name
#define _QUOTE(name) _QUOTE1(name)

#define DECLARE_CLASS(classname) sbsTask _sbsClass_##classname; \
 static sbsTask* _S = &_sbsClass_##classname; \
 void sbsDerivatives(),sbsOutputs(); \
 void sbsInitializeSizes(), sbsTerminate(); \
 static char _sbsfcnName[] = "SIMKIT sbs-task \"" _QUOTE(classname)
"\""; \
 void _sbsInitializeClassFcnPointers_##classname() \
 { \
      ssSetModelName(_S, _sbsfcnName); \
      ssSetsbsInitializeSizes(_S,sbsInitializeSizes); \
  ssSetsbsOutputs(_S,sbsOutputs); \
  ssSetsbsDerivatives(_S,sbsDerivatives); \
  ssSetsbsTerminate(_S,sbsTerminate); \
 }

typedef struct sbsTask_tag sbsTask;

/*
 * _sbsFcnModelMethods:
 *   sbsInitializeModel       -  Initialize sbsTask sizes array
 *   sbsOutputs               -  Fill output vector
 *   sbsDerivates             -  Compute the derivatives
 *   sbsTerminate             -  End of model housekeeping
 */

struct _sbsFcnModelMethods {
  void (*sbsInitializeSizes)(sbsTask* S);
  void (*sbsTerminate)(sbsTask* S);
  void (*sbsOutputs)(double t, const double* x, const double* u,
                     double* y, sbsTask* S);
```

```
  void (*sbsDerivatives)(double t, const double* x, const double* u,
                         double* dx, sbsTask* S);
};

struct _sbsSizes {
  int numContStates; /* number of continuous states */
  int numDiscStates; /* number of discrete states */
  int numOutputs;    /* number of outputs */
  int numInputs;     /* number of inputs */
                     /* ------ Work vectors --------*/
  int numIWork;      /* size of integer work vector */
  int numRWork;      /* size of double work vector  */
  int numPWork;      /* size of pointer work vector */
};

struct _sbsStates {
  double* U;   /* input vector */
  double* Y;   /* output vector */
  double* X;   /* State vector */
  double* dX;  /* derivative vector */
};

struct _sbsFcnParams {
  int count;           /* number of function parameters passed in */
  const double** params;      /* the function parameters */
};

struct _sbsWork {
  int* iWork;                 /* integer work vector */
  double* rWork;              /* real work vector */
  void** pWork;               /* pointer work vector */
  int* mapVector;           /* pointer to the global array that maps
                               inputs to outputs */
};

struct sbsTask_tag {
  const char* modelName;              /* Name of the model */
  struct _sbsSizes sizes;             /* sizes */
  struct _sbsFcnParams fcnParams; /*function parameters passed in*/
  struct _sbsStates states;      /* state and derivative vectors */
  struct _sbsWork work;           /* various work areas */
  struct {
      struct _sbsFcnModelMethods sbsFcn;  /* model methods */
  } modelMethods;
};


/*====================================*
 * sbsTask Get and Set Access methods *
 *====================================*/
```

```
/*----------------------- S->modelName ------------------------*/
#define ssGetModelName(S) \
      (S)->modelName                          /* (const char*) */
#define ssSetModelName(S, name) \
        (S)->modelName = (name)

/*-------------------------------- S->sizes ----------------------
-----------*/
#define ssGetNumContStates(S) \
      (S)->sizes.numContStates                  /*  (int)     */
#define ssSetNumContStates(S,nContStates) \
        (S)->sizes.numContStates = (nContStates)

#define ssGetNumDiscStates(S) \
      (S)->sizes.numDiscStates                  /*  (int)     */
#define ssSetNumDiscStates(S,nDiscStates) \
        (S)->sizes.numDiscStates = (nDiscStates)

#define ssGetNumTotalStates(S) \
      (ssGetNumContStates(S) + ssGetNumDiscStates(S)) /*  (int) */

#define ssGetNumOutputs(S) \
      (S)->sizes.numOutputs                     /*  (int)     */
#define ssSetNumOutputs(S,nOutputs) \
        (S)->sizes.numOutputs = (nOutputs)

#define ssGetNumInputs(S) \
      (S)->sizes.numInputs                      /*  (int)     */
#define ssSetNumInputs(S,nInputs) \
        (S)->sizes.numInputs = (nInputs)

#define ssGetNumRWork(S) \
      (S)->sizes.numRWork                       /*  (int)     */
#define ssSetNumRWork(S,nRWork) \
        (S)->sizes.numRWork = (nRWork)

#define ssGetNumIWork(S) \
      (S)->sizes.numIWork                       /*  (int)     */
#define ssSetNumIWork(S,nIWork) \
        (S)->sizes.numIWork = (nIWork)

#define ssGetNumPWork(S) \
      (S)->sizes.numPWork                       /*  (int)     */
#define ssSetNumPWork(S,nPWork) \
        (S)->sizes.numPWork = (nPWork)

/*---------------- S->fcnParams --------------------*/

#define ssGetFcnParamsCount(S) \
```

```
       (S)->fcnParams.count                        /*   (int)      */
#define ssSetFcnParamsCount(S,n) \
        (S)->fcnParams.count = (n)

#define ssGetFcnParamsPtr(S) \
      (S)->fcnParams.params                       /*   (double **)  */
#define ssSetFcnParamsPtr(S,ptr) \
        (S)->fcnParams.params = (ptr)

#define ssGetFcnParam(S,index) \
      (S)->fcnParams.params[index]                /*   (double*)   */
#define ssSetFcnParam(S,index,mat) \
        (S)->fcnParams.params[index] = (mat)

/*---------------------- S->states --------------------------*/
#define ssGetU(S) \
      (S)->states.U                               /*   (double *)    */
#define ssSetU(S,u) \
        (S)->states.U = (u)

#define ssGetY(S) \
      (S)->states.Y                               /*   (double *)    */
#define ssSetY(S,y) \
        (S)->states.Y = (y)

#define ssGetX(S) \
      (S)->states.X                               /*   (double *)    */
#define ssSetX(S,x) \
        (S)->states.X = (x)

#define ssGetdX(S) \
      (S)->states.dX                              /*   (double *)    */
#define ssSetdX(S,dx) \
        (S)->states.dX = (dx)

/*----------------------- S->work --------------------------*/
#define ssGetIWork(S) \
       (S)->work.iWork                             /*   (int *)     */
#define ssSetIWork(S,iwork) \
        (S)->work.iWork = (iwork)

#define ssGetIWorkValue(S,iworkIdx) \
      (S)->work.iWork[iworkIdx]                    /*   (int)      */
#define ssSetIWorkValue(S,iworkIdx,iworkValue) \
        (S)->work.iWork[iworkIdx] = (iworkValue)

#define ssGetRWork(S) \
      (S)->work.rWork                             /*   (double *)    */
#define ssSetRWork(S,rwork) \
        (S)->work.rWork = (rwork)
```

```
#define ssGetRWorkValue(S,rworkIdx) \
      (S)->work.rWork[rworkIdx]                 /*  (double)   */
#define ssSetRWorkValue(S,rworkIdx,rworkValue) \
        (S)->work.rWork[rworkIdx] = (rworkValue)

#define ssGetPWork(S) \
      (S)->work.pWork                          /*  (void **)   */
#define ssSetPWork(S,pwork) \
        (S)->work.pWork = (pwork)

#define ssGetPWorkValue(S,pworkIdx) \
      (S)->work.pWork[pworkIdx]                 /*  (void*)    */
#define ssSetPWorkValue(S,pworkIdx,pworkValue) \
        (S)->work.pWork[pworkIdx] = (pworkValue)

#define ssGetmapV(S) \
      (S)->work.mapVector                       /*  (int *)    */
#define ssSetmapV(S,mVec) \
        (S)->work.mapVector = (mVec)

/*------------------ S->modelMethods.sbsFcn --------------------*/
#define ssSetsbsInitializeSizes(S,initSizes) \
       (S)->modelMethods.sbsFcn.sbsInitializeSizes = \
          (void (*)(sbsTask*)) (initSizes)
#define fcnInitializeSizes(S) \
       (*(S)->modelMethods.sbsFcn.sbsInitializeSizes)(S)

#define ssSetsbsOutputs(S,outputs) \
       (S)->modelMethods.sbsFcn.sbsOutputs = \
          (void (*)(double, const double*, \
                    const double*, double*, sbsTask*)) (outputs)
#define fcnOutputs(t, x, u, y, S) \
        (*(S)->modelMethods.sbsFcn.sbsOutputs)(t,x,u,y,S)

#define ssSetsbsDerivatives(S,derivs) \
       (S)->modelMethods.sbsFcn.sbsDerivatives = \
          (void (*)(double, const double*, \
                    const double*, double*, sbsTask*)) (derivs)
#define fcnDerivatives(t, x, u, dx, S) \
        (*(S)->modelMethods.sbsFcn.sbsDerivatives)(t,x,u,dx,S)

#define ssSetsbsTerminate(S,housekeeping) \
        (S)->modelMethods.sbsFcn.sbsTerminate = \
           (void (*)(sbsTask*)) (housekeeping)
#define fcnTerminate(S) \
        (*(S)->modelMethods.sbsFcn.sbsTerminate)(S)

#define ssCopyModelMethods(T,S) \
        (T)->modelMethods.sbsFcn.sbsInitializeSizes = \
```

```
          (S)->modelMethods.sbsFcn.sbsInitializeSizes; \
        (T)->modelMethods.sbsFcn.sbsTerminate = \
          (S)->modelMethods.sbsFcn.sbsTerminate; \
        (T)->modelMethods.sbsFcn.sbsOutputs = \
          (S)->modelMethods.sbsFcn.sbsOutputs; \
        (T)->modelMethods.sbsFcn.sbsDerivatives = \
          (S)->modelMethods.sbsFcn.sbsDerivatives

#endif

/* Eof: sbs.h */
```

Appendix D Composable Simulation Markup Language

```
<?xml encoding="US-ASCII"?>

<!--*************************************************************
  **                 Carnegie Mellon University                  **
  **         Institute for Complex Engineered Systems            **
  **                                                             **
  **          Antonio Diaz-Calderon & Chris Paredis              **
  **         Composable Simulation Markup Landuage DTD           **
  **                         May, 2000                           **
  **                                                             **
  *************************************************************
-->

<!ENTITY % vhdl-ams-mode "INCLUDE">
<!ENTITY % non-vhdl-ams-mode "IGNORE">

<!-- Names definitions -->
<!ENTITY % IDENT "NMTOKEN"> <!-- an identifier -->
<!ENTITY % URL "CDATA"> <!-- an URL -->
<!ENTITY % CONNECTOR "CDATA"> <!-- a connector -->
<!ENTITY % EXPRESSION "CDATA"> <!-- an expression -->
```

```
<!ENTITY % definition "(interface | implementation | record)">
<!ENTITY % _common_declarations "signal | constant | type |
                                 sub-type | variable | package">

<!ENTITY % _entity_header "generics?, boundary?">
<!ENTITY % _entity_declarative_item "%_common_declarations;">
<!ENTITY % _entity_statements "assertions">
<!ENTITY % _entity_meta_knowledge "semantics?, implementations?">
<!ENTITY % _ebody "(%_entity_header;,
                    (%_entity_declarative_item;)*,
                     %_entity_statements;?,
                     %_entity_meta_knowledge;)">

<!ENTITY % _architecture_declarative_part
     "branch-quantity | free-quantity |
      spectral-quantity | noise-quantity |
      terminal | component | function | %_common_declarations;">
<!ENTITY % _abody "((%_architecture_declarative_part;)*,
                   concurrent-statements, description?)">

<!ENTITY % _prim_statements "process | break | equation | assert">
<!ENTITY % _comp_statements "connect">

<![%non-vhdl-ams-mode;[
<!ENTITY % _concurrent_statements "((%_prim_statements;)+ |
                                    %_comp_statements;+)">
]]>

<![%vhdl-ams-mode;[
<!-- <!ENTITY % _concurrent_statements "(%_prim_statements; |
                                    %_comp_statements;)+"> -->
<!ENTITY % _concurrent_statements "(#PCDATA | %_prim_statements; |
                                    %_comp_statements;)*">
]]>

<!-- Valid natures provided by the modeling environment -->
<!ENTITY % NATURES "(electrical | magnetic | fluidic |
  thermal | mechanical | translational | rotational |
  electrical_vector)">


<!ENTITY % mach-limit "big | small | bigint | -big | -small | -
bigint">


<!--********************* D O C U M E N T **********************-->
<!ELEMENT document (require*, (((library | package)*,
                   (%definition;)*) | component*))>
<!ATTLIST document
  version CDATA #REQUIRED
```

```
  is_library (true | false) "false">

<!ELEMENT require EMPTY>
<!ATTLIST require
  url CDATA #REQUIRED>

<!--********************** L I B R A R Y **********************-->
<!ELEMENT library EMPTY>
<!ATTLIST library
  name NMTOKEN #REQUIRED>

<!--********************** P A C K A G E **********************-->
<!ELEMENT package EMPTY>
<!ATTLIST package
  name NMTOKEN #REQUIRED>

<!--********************** I N T E R F A C E **********************-->
<!ELEMENT interface %_ebody;>
<!ATTLIST interface
  ident %IDENT; #REQUIRED
  abstract (true | false) "false"
  super-type (NMTOKEN | null) "null"
  sub-types (NMTOKENS | null) "null">

<![IGNORE[
<!--************* C O M P O N E N T - I N S T A N C E *************-->
<!ELEMENT component-instance
     (parameter-binding*, bound-implementation)>
<!ATTLIST component-instance
  ident %IDENT; #REQUIRED
  instance-of %IDENT; #REQUIRED>
]]>

<!--********************** G E N E R I C S **********************-->
<!ELEMENT generics (parameter)+>

<!--********************** B O U N D A R Y **********************-->
<!ELEMENT boundary (terminal | quantity | interface-signal)+>

<!--********************** T E R M I N A L **********************-->
<!ELEMENT terminal EMPTY>
<!ATTLIST terminal
  name %IDENT; #REQUIRED
  nature-type %NATURES; "electrical">

<!--********************** Q U A N T I T Y **********************-->
<!ELEMENT quantity EMPTY>
<!ATTLIST quantity
  name %IDENT; #REQUIRED
  nature-type (real | NMTOKEN) "real"
```

```
  range (true | false) "false"
  direction (in | out) #REQUIRED
  default %EXPRESSION; #IMPLIED
  row-l CDATA #IMPLIED
  row-h CDATA #IMPLIED>

<!--****************** I N T E R F A C E  S I G N A L ***********-->
<!ELEMENT interface-signal EMPTY>
<!ATTLIST interface-signal
  name %IDENT; #REQUIRED
  nature-type CDATA #REQUIRED
  range (true | false) "false"
  signal-kind (in | out | inout | buffer) #IMPLIED
  guarded (true | false) "false"
  default %EXPRESSION; #IMPLIED
  row-l CDATA #IMPLIED
  row-h CDATA #IMPLIED>

<!--****************** A S S E R T I O N S *********************-->
<!ELEMENT assertions (assert)+>

<!ELEMENT assert EMPTY>
<!ATTLIST assert
  label NMTOKEN #IMPLIED
  postponed (true | false) "false"
  condition CDATA #REQUIRED
  report CDATA #IMPLIED
  severity CDATA #IMPLIED>

<!--******************** S E M A N T I C S *********************-->
<!ELEMENT semantics (qprop | assumption | pprop)+>

<!--********************** Q P R O P *********************-->
<!ELEMENT qprop EMPTY>
<!ATTLIST qprop
  sign (pos | neg | none) "pos"
  Q1 CDATA #REQUIRED
  Q2 CDATA #REQUIRED>

<!--******************** A S S U M P T I O N ******************-->
<!ELEMENT assumption EMPTY>
<!ATTLIST assumption
  def CDATA #REQUIRED>

<!--********************** P P R O P *********************-->
<!ELEMENT pprop (#PCDATA)>

<!--***************** I M P L E M E N T A T I O N S *************-->
<!ELEMENT implementations (specification)+>
<!ELEMENT specification EMPTY>
```

```
<!ATTLIST specification
  name NMTOKEN #REQUIRED
  url %URL; #REQUIRED
  default (true | false) "false">

<!--********************** R E C O R D **********************-->
<!ELEMENT record (parameter)+>
<!ATTLIST record
  ident %IDENT; #REQUIRED>

<!--******************* D E S C R I P T I O N *******************-->
<!ELEMENT description (#PCDATA)>

<!--****************** I M P L E M E N T A T I O N ***************-->
<!ELEMENT implementation %_abody;>
<!ATTLIST implementation
  ident %IDENT; #REQUIRED
  of-interface %IDENT; #REQUIRED
  compound (true | false) "false"
  configure (true | false) "false"
  is-default (true | false) "false"
  vrml CDATA #IMPLIED>

<!ELEMENT concurrent-statements %_concurrent_statements;>

<!--******************* Q U A N T I T I E S ********************-->
<!ELEMENT free-quantity EMPTY>
<!ATTLIST free-quantity
  nature-type NMTOKEN #REQUIRED
  ident %IDENT; #REQUIRED
  range (true | false) "false"
  default %EXPRESSION; #IMPLIED
  min (NMTOKEN | %mach-limit;) "-big"
  max (NMTOKEN | %mach-limit;) "big"
  row-l NMTOKEN #IMPLIED
  row-h NMTOKEN #IMPLIED
  col-l NMTOKEN #IMPLIED
  col-h NMTOKEN #IMPLIED
  semantics NMTOKEN #IMPLIED>

<!ELEMENT spectral-quantity EMPTY>
<!ATTLIST spectral-quantity
  ident %IDENT; #REQUIRED
  nature-type NMTOKEN #REQUIRED
  magnitude NMTOKEN #REQUIRED
  phase NMTOKEN #REQUIRED>

<!ELEMENT noise-quantity EMPTY>
<!ATTLIST noise-quantity
  ident %IDENT; #REQUIRED
```

```
  nature-type NMTOKEN #REQUIRED
  definition CDATA #REQUIRED>

<!ELEMENT branch-quantity EMPTY>
<!ATTLIST branch-quantity
  across-vars NMTOKENS #IMPLIED
  through-vars NMTOKENS #IMPLIED
  plus-terminal CDATA #REQUIRED
  minus-terminal CDATA #IMPLIED>

<!--********************* C O N S T A N T **********************-->
<!ELEMENT constant EMPTY>
<!ATTLIST constant
  ident %IDENT; #REQUIRED
  nature-type NMTOKEN #REQUIRED
  range (true | false) "false"
  default %EXPRESSION; #IMPLIED
  min (NMTOKEN | %mach-limit;) "-big"
  max (NMTOKEN | %mach-limit;) "big"
  row-l CDATA #IMPLIED
  row-h CDATA #IMPLIED
  col-l CDATA #IMPLIED
  col-h CDATA #IMPLIED
  semantics NMTOKEN #REQUIRED>

<!--********************* V A R I A B L E **********************-->
<!ELEMENT variable EMPTY>
<!ATTLIST variable
  ident %IDENT; #REQUIRED
  nature-type NMTOKEN #REQUIRED
  range (true | false) "false"
  default %EXPRESSION; #IMPLIED
  shared (true | false) "false"
  min (NMTOKEN | %mach-limit;) "-big"
  max (NMTOKEN | %mach-limit;) "big"
  row-l CDATA #IMPLIED
  row-h CDATA #IMPLIED
  col-l CDATA #IMPLIED
  col-h CDATA #IMPLIED>

<!--******************* S I G N A L ****************************-->
<!ELEMENT signal EMPTY>
<!ATTLIST signal
  name %IDENT; #REQUIRED
  nature-type CDATA #REQUIRED
  range (true | false) "false"
  signal-kind (register | bus) #IMPLIED
  default %EXPRESSION; #IMPLIED
  row-l CDATA #IMPLIED
  row-h CDATA #IMPLIED>
```

```
<!--********************** T Y P E **********************-->
<!ELEMENT type EMPTY>
<!ATTLIST type
  name %IDENT; #REQUIRED
  type-definition CDATA #REQUIRED>

<!--********************** S U B - T Y P E **********************-->
<!ELEMENT sub-type EMPTY>
<!ATTLIST sub-type
  name %IDENT; #REQUIRED
  subtype-indication CDATA #REQUIRED>

<!--********************** C O M P O N E N T **********************-->
<!ELEMENT component (parameter-binding*,
                     bound-implementation?,
                     candidate-implementation*,
                     position?)>
<!ATTLIST component
  final (true | false) "false"
  name %IDENT; #REQUIRED
  interface-name %IDENT; #REQUIRED
  vrml CDATA #IMPLIED
  url CDATA #REQUIRED>

<!ELEMENT parameter-binding EMPTY>
<!ATTLIST parameter-binding
  formal-part %IDENT; #REQUIRED
  actual-part CDATA #REQUIRED>

<!ELEMENT bound-implementation (component*)>
<!ATTLIST bound-implementation
  implementation-name %IDENT; #REQUIRED>

<!ELEMENT candidate-implementation (component*)>
<!ATTLIST candidate-implementation
  is-default (true | false) "false"
  implementation-name %IDENT; #REQUIRED>

<!--********************** F U N C T I O N **********************-->
<!ELEMENT function (function-args, function-body)>
<!ATTLIST function
  name %IDENT; #REQUIRED
  return-type NMTOKEN #REQUIRED>

<!ELEMENT function-args (formal-arg)+>

<!ELEMENT formal-arg EMPTY>
<!ATTLIST formal-arg
  name %IDENT; #REQUIRED
```

```
  nature-type NMTOKEN #REQUIRED
  range (true | false) "false"
  default %EXPRESSION; #IMPLIED
  row-l CDATA #IMPLIED
  row-h CDATA #IMPLIED
  col-l CDATA #IMPLIED
  col-h CDATA #IMPLIED>

<!ELEMENT function-body (#PCDATA)>

<!--********************** P R O C E S S **********************-->
<!ELEMENT process (#PCDATA)>
<!ATTLIST process
  label NMTOKEN #IMPLIED>

<!--********************** B R E A K **********************-->
<!ELEMENT break (#PCDATA)>
<!ATTLIST break
  label NMTOKEN #IMPLIED>

<!--********************** E Q U A T I O N **********************-->
<!ELEMENT equation (#PCDATA)>
<!ATTLIST equation
  label NMTOKEN #IMPLIED>

<!--********************** C O N N E C T **********************-->
<!ELEMENT connect EMPTY>
<!ATTLIST connect
  terminal-A %CONNECTOR; #REQUIRED
  terminal-B %CONNECTOR; #REQUIRED>

<!--********************** T R A N S F O R M **********************-->
<!ELEMENT position EMPTY>
<!ATTLIST position
  x NMTOKEN "0"
  y NMTOKEN "0"
  z NMTOKEN "0"
  roll NMTOKEN "0"
  pitch NMTOKEN "0"
  yaw NMTOKEN "0">

<!--********************** P A R A M E T E R **********************-->
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter
  nature-type CDATA #REQUIRED
  ident  %IDENT; #REQUIRED
  range (true | false) "false"
  default %EXPRESSION; #IMPLIED
  min (NMTOKEN | %mach-limit;) "-big"
  max (NMTOKEN | %mach-limit;) "big"
```

```
  row-l CDATA #IMPLIED
  row-h CDATA #IMPLIED
  col-l CDATA #IMPLIED
  col-h CDATA #IMPLIED
  semantics NMTOKEN #IMPLIED>

<!-- EOF csml.dtd -->
```

Bibliography

[1] M. Abadi and L. Cardelli, *A theory of objects*. Monographs in Computer Science, Springer-Verlag, New York, 1996.

[2] G. Abowd, R. J. Allen, and D. Garlan, "Formalizing style to understand descriptions of software architecture," Pittsburgh, PA: School of Computer Science, Carnegie Mellon University.

[3] S. Addanki, R. Cremonini, and S. J. Penberthy, "Reasoning about assumptions in graphs of models," presented at IJCAI-89 1989.

[4] S. Addanki, R. Cremonini, and S. J. Penberthy, "Graphs of models," *Artificial Intelligence*, vol. 51, pp. 145-177, 1991.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data structures and algorithms.* Addison-Wesley, Reading, Massachusetts, 1987.

[6] R. J. Allen, "The Wright architectural description language," unpublished.

[7] R. J. Allen and D. Garlan, “Formalizing architectural connection,” presented at 16th International Conference on Software Engineering, Sorrento, Italy, May 1994.

[8] R. J. Allen and D. Garlan, “Formal connectors,” School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Tech. Report CMU-CS-94-115, March 1994.

[9] M. Anderson. “Object-oriented modeling and simulation of hybrid systems,” Ph. D. Thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1994.

[10] G. C. Andrews, M. J. Richard, and R. J. Anderson, “A general vector-network formulation for dynamic systems with kinematic constraints,” *Mechanisms and Machine Theory*, vol. 23, no. 3, pp. 243-256, 1988.

[11] D. Baraff, “Analytical methods for dynamic simulation of non-penetrating rigid bodies,” *SIGGRAPH: Compputer Graphics*, vol. 23, no. 3, pp. 223-232, 1989.

[12] D. Baraff, “Interactive simulation of solid rigid bodies,” *IEEE Computer Graphics and Applications*, pp. 63-75, 1995.

[13] P. I. Barton and C. C. Pantelides, “Modeling of combined discrete/continuous processes,” *AIChE Journal*, vol. 40, no. 6, pp. 966-979, 1994.

[14] R. Bhaskar and A. Nigam, “Qualitative physics using dimensional analysis,” *Artificial Intelligence*, vol. 45, pp. 73-111, 1990.

[15] D. Bobrow, B. Falkenhainer, A. Farquhar, R. Fikes, K. Forbus, T. Gruber, Y. Iwasaki, and B. Kuipers, “A compositional modeling language,” unpublished.

[16] A. M. Bos and M. J. L. Tiernego, “Formula manipulation in the bond graph modeling and simulation of large mechanical systems,” *Journal of the Franklin Institute*, vol. 319, no. 1/2, pp. 51-65, 1985.

[17] F. H. Branin, “The algebraic-topological basis for network analogies and the vector calculus,” presented at Symposium on Generalized Networks, Polytechnic Institute of Brooklin, April 12-14 1966.

[18] A. P. J. Breunese, J. L. Top, J. F. Broenink, and J. M. Akkermans, “Libraries of reusable models: Theory and application,” *Simulation*, 1996.

[19] A. P. J. Breunese. “Automated support in mechatronich systems modeling,” Ph. D. Thesis, Department of Electrical Engineering, University of Twente, Enschede, The Netherlands, 1996.

[20] Cadsim Engineering, “CAMP-G,” 2000.

[21] F. E. Cellier, *Continuous system modeling.* Springer-Verlag, 1991.

[22] F. E. Cellier and H. Elmqvist, “Automated formula manipulation supports object-oriented continuous system modeling,” *IEEE Control Systems*, vol. 13, no. 2, pp. 28-38, 1993.

[23] F. E. Cellier, “Object-oriented modeling: means for dealing with system complexity,” presented at 15th Benelux Meeting on Systems and Control, Mierlo, The Netherlands 1996.

[24] F. E. Cellier, H. Elmqvist, and M. Otter, “Modeling from physical principles,” www.ece.arizona.edu/~cellier, 1996.

[25] E. Christen, K. Bakalar, A. M. Dewey, and E. Moser, “Analog and mixed-signal modeling using the VHDL-AMS language,” 36th Design and Automation Conference, New Orleans, 1999.

[26] Controllab Products B. V., “20-sim,” 1999.

[27] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*. MIT Press, Cambridge, 1990.

[28] J. de Kleer, “Problem solving with the ATMS,” *Artificial Intelligence*, vol. 28, pp. 197-224, 1986.

[29] J. de Kleer, “An assumption-based TMS,” *Artificial Intelligence*, vol. 28, pp. 127-162, 1986.

[30] J. de Kleer, “Extending the ATMS,” *Artificial Intelligence*, vol. 28, pp. 163-196, 1986.

[31] A. Diaz-Calderon, C. J. J. Paredis, and P. K. Khosla, “A modular composable software architecture for the simulation of mechatronic systems,” presented at ASME Design Engineering Technical Conference, 18th Computers in Engineering Conference, Atlanta, GA, September 1998.

[32] A. Diaz-Calderon, C. J. J. Paredis, and P. K. Khosla, “Automatic generation of system-level dynamic equations for mechatronic systems,” *Computer-Aided Design*, vol. 32, pp. 339-354, 2000.

[33] A. Diaz-Calderon, C. J. J. Paredis, and P. K. Khosla, “On the synthesis of the system graph for 3D mechanics,” presented at American Control Conference, San Diego, CA, June 2-4 1999.

[34] A. Diaz-Calderon, C. J. J. Paredis, and P. K. Khosla, “Combining information technology components and symbolic equation manipulation in modeling and simulation of mechatronic systems,” presented at IEEE International Symposium on Computer Aided Control Sysyem Design, Island of Hawaii, HI, August 1999.

[35] A. Diaz-Calderon, C. J. J. Paredis, and P. K. Khosla, “A composable simulation environment for mechatronic systems,” presented at SCS 1999 European Simulation Simposium, Erlangen, Germany, October 1999.

[36] v. J. J. Dixhoorn, “Bond graphs and the challenge of a unified modeling theory of physical systems,” in *Progress in Modeling and Simulation*, F. E. Cellier, Ed. London: Academic Press, 1980, pp. 207-245.

[37] I. S. Duff and J. K. Reid, “An implementation of Tarjan's algorithm for the block triangularization of a matrix,” *ACM Transaction on Mathematical Software*, vol. 4, no. 2, pp. 137-147, 1978.

[38] I. S. Duff, “On algorithms for obtaining a maximum traversal,” *ACM Transactions on Mathematical Software*, vol. 7, no. 3, pp. 315-330, 1981.

[39] I. S. Duff, M. A, Erisman, and J. K. Reid, *Direct methods for sparse matrices*. Monographs on numerical analysis, Oxford Science Publications, Oxford University Press, Oxford, 1989.

[40] W. K. Durfee, M. B. Wall, D. Rowell, and F. K. Abbott, “Interactive software for dynamic system modeling using linear graphs,” *IEEE Control Systems*, vol. 11, no. 4, pp. 60-66, 1991.

[41] Dynasim AB, “Dymola,” 1999.

[42] H. Elmqvist, M. Otter, and F. E. Cellier, “Inline integration: A new mixed symbolic/numeric approach for solving differential-algebraic qeuation systems,” : Dynasim AB, Research Park Ideon, S-223 70 Lund, Sweden.

[43] H. Elmqvist, F. E. Cellier, and M. Otter, “Object-oriented modeling of hybrid systems,” presented at European Simulation Symposium, Delft, The Netherlands, October 1993.

[44] H. Elmqvist and M. Otter, “Methods for tearing systems of equations in object-oriented modeling,” presented at ESM 94 European simulation multiconference, Barcelona, Spain, 1994.

[45] H. Elmqvist and D. Brück, “Constructs for object-oriented modeling,” presented at Eurosim Simulation Congress, Vienna, Austria, September 1995.

[46] H. Elmqvist and S. E. Mattsson, “An introduction to the physical modeling language Modelica,” presented at European Simulation symposium, Passau, Germany, October 19-22 1997.

[47] H. Elmqvist, S. E. Mattsson, and M. Otter, “Modelica: The new object-oriented modeling language,” presented at The 12th European Simulation Multiconference, Manchester, UK, June 16-19 1998.

[48] B. Falkenhainer and K. D. Forbus, “Setting up large-scale qualitative models,” in *Readings in qualitative reasoning about physical systems, The Morgan Kaufmann Series in Representation and Reasoning*, D. S. Weld and J. de Kleer, Eds. San Mateo, CA: Morgan Kaufmann Publisher, Inc., 1990, pp. 553-558.

[49] B. Falkenhainer and K. D. Forbus, “Compositional modeling: Finding the right model for the job,” *Artificial Intelligence*, vol. 51, pp. 95-143, 1991.

[50] B. Falkenhainer, A. Farquhar, D. Bobrow, R. Fikes, K. Forbus, T. Gruber, Y. Iwasaki, and B. Kuipers, “CML: A compositional modeling language,” Knowledge Sysyems Laboratory, Stanford University, Palo Alto, CA, Tech. Report KSL-94-16, January 1994.

[51] J. B. Ferris and J. L. Stein, “Development of proper models of hybrid systems: A bond graph formulation,” presented at International Conference on Bond Graph Modeling and Simulation, Las Vegas, NV 1995.

[52] S. Finger and J. R. Rinderle, “A transformational approach to mechanical design using a bond graph grammar,” presented at The 1989 ASME Design Technical Conferences, 1st International Conference on Design Theory and Methodology, Montreal, Quebec, Canada, September 1989.

[53] S. Finger and J. R. Rinderle, “Transforming behavioral and physical representations of mechanical designs,” presented at First International Workshop in Formal Methods in Design, Manufacturing and Assembly, Colorado Springs, January 1990.

[54] K. D. Forbus, “Qualitative process theory,” *Artificial Intelligence*, vol. 24, pp. 85-168, 1984.

[55] K. D. Forbus, “The qualitative process engine,” in *Readings in qualitative reasoning about physical systems*, *The Morgan Kaufmann Series in Representation and Reasoning*, D. S. Weld and J. de Kleer, Eds. San Mateo, CA: Morgan Kaufmann Publisher, Inc., 1990, pp. 220-235.

[56] K. D. Forbus, “The QPE user's manual,” The Institute for the Learning Sciences, Northwestern University January 1992.

[57] P. Fritzson and V. Engelson, “Modelica: A unified object-oriented language for system modeling and simulation,” presented at The 12th European Conference on Object-Oriented Programming, Brussels, Belgium, July 20-24 1998.

[58] M. R. Genesereth and R. E. Fikes, “Knowledge interchange format version 3.0 reference manual,” Stanford University, Palo Alto, CA, Tech. Report KSL-92-86, June 1992.

[59] M. B. Histand and D. G. Alciatore, *Introduction to mechatronics and measurement systems*. Boston: Mc Graw-Hill, 1998.

[60] IEEE 1076.1 Working Group, “Analog and mixed-signal extensions to VHDL,” vhdl.org/vi/analog, 1999.

[61] D. C. Karnopp, D. L. Margolis, and R. C. Rosenberg, *System dynamics: A unified approach*, 2nd ed. John Wiley & Sons, Inc., New York, 1990.

[62] D. E. Knuth, *The art of computer programming*, vol. 1. Addison-Wesley, Reading, Massachusetts, 1973.

[63] H. E. Koenig, Y. Tokad, H. K. Kesavan, and H. G. Hedges, *Analysis of discrete physical systems.* MacGraw-Hill, New York, 1967.

[64] P. C. Krause, *Analysis of electric machinery.* McGraw-Hill series in electrical engineering, McGraw-Hill, New York, 1986.

[65] B. Kuipers, "Qualitative simulation," *Artificial Intelligence*, vol. 29, pp. 289-338, 1986.

[66] B. J. Kuipers, C. Chiu, D. T. Dalle Molle, and D. R. Throop, "Higher-order derivative constraints in qualitative simulation," *Artificial Intelligence*, vol. 51, pp. 343-379, 1991.

[67] F. S. Lee, T. J. Moon, and G. Y. Masada, "Modeling of distributed electromechanical systems using extended bond graphs," *Journal of the Franklin Institute*, vol. 331B, no. 1, pp. 43-60, 1994.

[68] E. A. Lee and others, "Ptolemy II," Department of Electrical and Computer Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 2000.

[69] T. W. Li and G. C. Andrews, "Application of the vector-network method to constrained mechanical systems," *Mechanisms, Transmissions, and Automation in Design*, vol. 108, pp. 471-480, 1986.

[70] LMS CADSI, "DADS," 1999.

[71] L. S. Louca, J. L. Stein, G. M. Hulbert, and J. Sprague, "Proper model generation: An energy-based methodology," presented at International Conference on Bond Graph Modeling 1997.

[72] D. C. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz, "Partial ordering of event sets and their application to prototyping concurrent timed systems," Computer Systems Laboratory, Stanford University, Stanford, CA CSL-TR-92-515, April 1992.

[73] D. C. Luckham and J. Vera, "An event-based architecture definition language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717-734, 1995.

[74] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, “Specitication and analysis of system architecture using Rapide,” *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336-355, 1995.

[75] D. C. Luckham, J. Vera, and S. Medal, “Three concepts of system architecture,” Computer Systems Laboratory, Stanford University, Stanford, CA, Tech. Report CSL-TR-95-674, July 1995.

[76] D. L. Margolis, “A survey of bond graph modeling for interacting lumped and distributed systems,” *Journal of the Franklin Institute*, vol. 319, no. 1/2, pp. 125-135, 1985.

[77] T. A. Mashburn and D. C. Anderson, “Automatically deriving behavior constraints for performance variables in mechanical design,” *Research in Engineering Design*, vol. 6, pp. 85-102, 1994.

[78] S. E. Mattsson and H. Elmqvist, “An overview of the modeling language Modelica,” presented at Eurosim '98 Simulation congress, Helsinki, Finland, April 14-15 1998.

[79] MatWeb, “The online materials information resource,” www.matls.com, 2000.

[80] J. J. McPhee, M. G. Ishac, and G. C. Andrews, “Wittenburg's formulation of multibody dynamics equations from a graph-theoretic perspective,” *Mechanism and Machine Theory*, vol. 31, no. 2, pp. 202-213, 1996.

[81] J. J. McPhee, “On the use of linear graph theory in multibody system dynamics,” *Nonlinear Dynamics*, vol. 9, pp. 73-90, 1996.

[82] J. J. McPhee, “A unified graph-theoretic approach to formulating multibody dynamics equations in absolute or joint coordinates,” *Journal of the Franklin Institute*, vol. 334B, no. 3, pp. 431-445, 1997.

[83] J. J. McPhee, “Automatic generation of motion equations for planar mechanical systems using the new set of "branch coordinates",” *Mechanism and Machine Theory*, vol. 33, no. 6, pp. 805-823, 1998.

[84] Mechanical Dynamics Inc., “ADAMS,” 1999.

[85] MicroMo Electronics, www.micromo.com, 2000.

[86] B. J. Muegge. “Graph-theoretic modeling and simulation of planar mechatronic systems,” MASc. Thesis, Systems Design Engineering, University of Waterloo, Waterloo, 1996.

[87] J. M. Nataf, “A direct translator from neutral model format to the SPARK simulation environment,” *Enery and Buildings*, vol. 23, no. 2, pp. 131-139, 1995.

[88] P. P. Nayak, L. Joskowicz, and S. Addanki, “Automated model selection using context-dependent behaviors,” presented at Fifth International Workshop on Qualitative Reasoning about Physical Systems 1991.

[89] P. P. Nayak and L. Joskowicz, “Efficient compositional modeling for generating causal explanations,” *Artificial Intelligence*, vol. 83, no. 2, pp. 193-227, 1996.

[90] P. E. Nikravesh and I. S. Chung, “Application of euler parameters to the dynamic analysis of three-dimensional constrained mechanical systems,” *Journal of Mechanical Design*, vol. 104, pp. 785-791, 1982.

[91] N. Orlandea. “Node-analogous, sparsity-oriented methods for simulation of mechanical dynamic systems,” Ph.D. Thesis, Mechanical Engineering, University of Michigan, 1973.

[92] N. Orlandea, M. A. Chace, and D. A. Calahan, “A sparsity-oriented approach to the dynamic analysis and design of mechanical systems - Part 1,” *Journal of Engineering for Industry*, August, 1977.

[93] N. Orlandea, M. A. Chace, and D. A. Calahan, “A sparsity-oriented approach to the dynamic analysis and design of mechanical systems - Part 2,” *Journal of Engineering for Industry*, August, 1977.

[94] O. M. Oshinowo and J. J. McPhee, “Object-oriented implementation of a graph-theoretic formulation for planar multibody dynamics,” *Numerical Methods in Engineering*, vol. 40, no. 4097, 1997.

[95] M. Otter, H. Elmqvist, and F. E. Cellier, “Modeling of multibody systems with the object-oriented modeling language Dymola,” presented at NATO/ASI Computer-Aided Analysis of Rigid and Flexible Mechanical Systems, Troia, Portugal, June27-July 9 1993.

[96] G. Pahl and W. Beitz, *Engineering design: A systematic approach*, 2nd Edition, Springer-Verlag, London, U.K., 1996.

[97] H. M. Painter, *Analysis and design of engineering systems*. MIT Press, Cambridge, MA, 1961.

[98] C. C. Pantelides, "The consistent initialization of differential-algebraic systems," *SIAM J. Sci. Stat. Comput.*, vol. 9, 1988.

[99] C. C. Pantelides and P. I. Barton, "Equation-oriented dynamic simulation: Current status and future perspectives," *Computers in Chemical Engineering*, vol. 17, S263 1993.

[100] P. Piela. "ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis," Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, 1989.

[101] P. C. Piela, T. G. Epperly, K. M. Westerberg, and A. W. Westerberg, "ASCEND: An object oriented computer environment for modeling and analysis. 1 - The modeling language," *Comput. Chem Engng*, vol. 15, no. 1, pp. 53-72, 1991.

[102] R. Rajagopalan, "Qualitative modeling and simulation: A survey," in *AI Applied to Simulation*, vol. 18, *Simulation Series*, E. J. H. Kerckhoffs, G. C. Vansteenkiste, and B. P. Zeigler, Eds. The Society for Computer Simulation.

[103] S. Y. Reddy and K. W. Fertig, "Design Sheet: A system for exploring design space," *Artificial Intelligence in Design*, 1996.

[104] R. C. Redfield, "Bond graphs as a tool in mechanical system conceptual design," presented at ASME, Automated Modeling 1992.

[105] E. Rich and K. Knight, *Artificial Intelligence*, 2nd edMcGraw-Hill, New York, 1991.

[106] M. J. Richard and R. J. Anderson, "Dynamic simulation of three-dimensional rigid bodies using vector-network techniques," *Mathematics and Computers in Simulation*, vol. 26, pp. 289-296, 1984.

[107] M. J. Richard, R. J. Anderson, and G. C. Andrews, "The vector-network method for the modeling of mechanical systems," *Mathematics and Computers in Simulation*, vol. 31, pp. 565-581, 1990.

[108] M. J. Richard, I. Bindzi, and C. M. Gosselin, “A topological approach to the dynamic simulation of articulated machinery,” *Journal of Mechanical Design*, vol. 117, pp. 199-202, 1995.

[109] P. H. Roe, *Networks and systems.* Electrical Engineering, Addison-Wesley, Reading, Massachusetts, 1966.

[110] R. C. Rosenberg, “State-space formulation for bond graph models of multiport systems,” *Journal of Dynamic Systems, Measurement, and Control*, pp. 35-40, 1971.

[111] R. C. Rosenberg and T. Zhou, “Power-based model insight,” presented at Automated Modeling for Design, The Winter Annual Meeting of the ASME, Chicago, Ill, November 1988.

[112] R. C. Rosenberg and D. C. Karnopp, *Introduction to physical system dynamics*. Series in Mechanical Engineering, McGraw-Hill, New York, 1983.

[113] R. C. Rosenberg and Y. Y. Wang, “Some structuring issues in modeling,” presented at ASME, Automated Modeling 1992.

[114] RosenCode Associates Inc., *The ENPORT reference manual*, Lansing, Michigan, 1989.

[115] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-oriented modeling and design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[116] J. Rumbaugh, I. Jacobson, and G. Booch, *The unified modeling language reference manual*. Reading, Mass. Addison-Wesley, 1999.

[117] P. Sahlin and E. F. Sowell, “A neutral model format for building simulation models,” presented at IBPSA Building Simulation conference, Vancouver, Canada, June 1989.

[118] P. Sahlin. “Modeling and simulation methods for modular continuous systems in buildings,” Ph. D. Thesis, Department of Building Sciences, Division of Building Services, Royal Institute of Technology, Stockholm, Sweden, 1996.

[119] P. Sahlin, “NMF handbook,” Department of Building Sciences, Division of Building Services, Royal Institute of Technology, Stockholm, Sweeden, Tech. Report ASHRAE RP-839, June 1996.

[120] W. Schienhlen, *Multibody systems handbook*. Springer-Verlag, Berlin, 1990.

[121] S. Seshu and M. B. Reed, *Linear graphs and electrical networks.* Addison-Wesley, Reading, Massachusetts, 1961.

[122] M. Shaw and D. Garlan, “Characteristics of higher-level languages for software architectures,” School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Tech. Report CMU-CS-94-210, December 1994.

[123] D. Shetty and R. A. Kolk, *Mechatronics system design*. PWS Publishing Company, Boston, 1997.

[124] P. Shi. “Flexible multibody dynamics: A new approach using virtual work and graph theory,” Ph. D. Thesis, Systems Design Engineering, University of Waterloo, Waterloo, Canada, 1998.

[125] S. B. Shooter, W. Keirouz, S. Szykman, and S. J. Fenves, “A model for the flow of design information,” presented at 2000 ASME Design Engineering Technical Conferences (12th International Conference on Design Theory and Methodology), paper No. DETC2000/DTM-14550, Baltimore, MD, September 2000.

[126] R. Sinha, C. J. J. Paredis, S. K. Gupta, and P. K. Khosla, “Capturing articulation in assemblies from component geometry,” presented at ASME Design Engineering Technical Conference, Atlanta, GA, September 1998.

[127] R. Sinha, C. J. J. Paredis, and P. K. Khosla, “Kinematics support for design and simulation of mechatronic systems,” presented at 4th IFIP Workshop on Knowledge Intensive CAD, Parma, Italy, May, 2000.

[128] S. S. Skiena, *The Algorithm Design Manual*. Springer-Verlag, New York, 1997.

[129] J. L. Stein and L. S. Louca, “A component-based modeling approach for systems design: Theory and implementation,” presented at International Conference on Bond Graph Modeling and Simulation, Las Vegas, NV, January 1995.

[130] D. V. Steward, “On an approach to techniques for the analysis of the structure of large systems of equations,” *SIAM Review*, vol. 4, no. 4, pp. 321-342, 1962.

[131] D. V. Steward, “Partitioning and tearing systems of equations,” *SIAM J. Numer. Anal.*, vol. 2, no. 2, pp. 345-365, 1965.

[132] D. B. Stewart and P. K. Khosla, “Rapid development of robotic applications using component-based real-time software,” presented at IEEE/RSJ International Conference on Intelligent Robots and Systems, August 1995.

[133] D. B. Stewart. “Real-time software design and analysis of reconfigurable multi-sensor based systems,” Ph. D. Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1996.

[134] D. B. Stewart and P. K. Khosla, “The Chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 2, pp. 249-277, 1996.

[135] J. C. Strauss and others, “The SCI continuous system simulation language (CSSL),” *Simulation*, vol. 9, no. 6, pp. 281-303, 1967.

[136] S. Szykman, S. J. Fenves, S. B. Shooter, and W. Keirouz, “A foundation for interoperability in next-generation product development systems,” presented at 2000 ASME Design Engineering Technical Conferences (20th Computers and Information in Engineering Conference), paper No. DETC2000/CIE-14622, Baltimore, MD, September 2000.

[137] M. Takác, “Fixed point classification method for qualitative simulation,” in *Progress in Artificial Intelligence: 8th Portuguese Conference on Artificial Intelligence, EPIA '97*, vol. 1323, *Lecture Notes in Artificial Intelligence*, E. C. A. Cardoso, Ed. Coimbra, Portugal: Springer, 1997, pp. 255-266.

[138] R. Tarjan, “Depht-first search and linear graphs algorithms,” *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146-160, 1972.

[139] The Boeing Company, “Easy5 Engineering analysis system,” 1999.

[140] The Mathworks Inc., “Matlab/Simulink,” 1999.

[141] J. L. Top. “Conceptual modeling of physical systems,” Ph. D. Thesis, University of Twente, Enschede, The Netherlands, 1993.

[142] J. Top and H. Akkermans, “Tasks and ontologies in engineering modeling,” *Human-Computer Studies*, vol. 41, pp. 585-617, 1994.

[143] H. M. Trent, “Isomorphisms between oriented linear graphs and lumped physical systems,” *The Journal of the Acoustical Society of America*, vol. 27, no. 3, pp. 500-527, 1955.

[144] M. J. L. Tiernego and A. M. Bos, “Modeling the dynamics and kinematics of mechanical systems with multibond graphs,” *Journal of the Franklin Institute*, vol. 319, no. 1/2, pp. 37-50, 1985.

[145] T. J. A. de Vries, A. P. J. Breunese, and P. C. Breedveld, “MAX: A mechatronic model building environment,” presented at Computer Aided Conceptual Design, Lancaster 1994.

[146] T. J. A. de Vries. “Conceptual design of controlled electro-mechanical systems: A modeling perspective,” Ph.D. Thesis, University of Twente, 1994.

[147] T. J. A. de Vries and A. P. J. Breunese, “Structuring product models to facilitate design manipulations,” presented at International Conference on Engineering Design, Praha, August 22-24 1995.

[148] D. S. Weld, “Automated model switching: Discrepancy driven selection of approximation reformulation,” Department of Computer Sicence and Engineering, University of Washington, Seattle, WA, Tech. Report 89-08-01, October 1989.

[149] D. S. Weld, “Reasoning about model accuracy,” Department of Computer Science and Engineering, University of Washington, Seattle, WA, Tech. Report 91-05-02, June 1991.

[150] A. W. Westerberg, H. P. Hutchison, R. L. Motard, and P. Winter, *Process flowsheeting*. Cambridge University Press, Cambridge, 1979.

[151] P. M. Will, “Simulation and modeling in early concept design: An industrial perspective,” *Research in Engineering Design*, vol. 3, no. 1, pp. 1-13, 1991.

[152] B. C. Williams and J. de Kleer, “Qualitative reasoning about physical systems: A return to roots,” *Artificial Intelligence*, vol. 51, pp. 1-9, 1991.

[153] B. H. Wilson and J. L. Stein, “An algorithm for obtaining minimum-order models of distributed and discrete systems,” *Journal of Dynamic Systems, Measurement and Control. Transactions of the ASME*, vol. 117, no. 4, 1992.

[154] B. H. Wilson, J. H. Taylor, and B. Erylmaz, “A frequancy domain model-order-deduction algorithm for linear systems,” presented at ASME Dynamic Systems and Control Division 1995.

[155] J. Wittenburg, *Dynamics of systems of rigid bodies.* B. G. Teubner, Stuttgart, 1977.

[156] J. Wittenburg, “Analytical methods in mechanical system dynamics,” in *Computer Aided Analysis and Optimization of MEchanical System Dynamics*, vol. F9, *NATO ASI Series*, E. J. Haug, Ed. Berlin: Springer-Verlag, 1984, pp. 89-127.

[157] J. Wittenburg and U. Wolz, “MESA VERDE: A symbolic program for nonlinear articulated-rigid-body dynamics,” presented at ASME Design Engineering Conference, Cincinnati, OH, September 1985.

[158] World Wide Web Consortium, “Extensible Markup Language (XML),” www.w3.org/XML, 1999.

[159] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems*, 2nd ed. San Diego: Academic Press, 2000.