



This is to certify that the

dissertation entitled

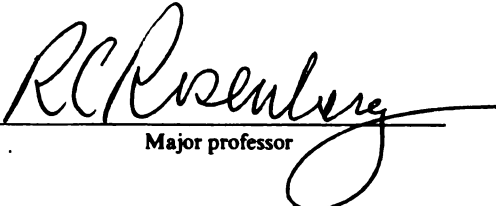
**COMPUTER AIDED ENGINEERING TOOLS FOR
STRUCTURED MODELING OF MECHATRONIC SYSTEMS**

presented by

Michael Keith Hales

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Mechanical Engineering


Major professor

Date Dec. 17, 1999

LIBRARY

Michigan State University

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.
MAY BE RECALLED with earlier due date if requested.

DATE DUE	DATE DUE	DATE DUE
MAY 28 2001		
JUN 05 2008		

**COMPUTER AIDED ENGINEERING TOOLS FOR
STRUCTURED MODELING OF MECHATRONIC SYSTEMS**

By

Michael Keith Hales

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Mechanical Engineering

1999

ABSTRACT

COMPUTER AIDED ENGINEERING TOOLS FOR STRUCTURED MODELING OF MECHATRONIC SYSTEMS

By

Michael Keith Hales

Industry is continually faced with pressures to develop improved products while decreasing design-cycle times. Complex designs of mechatronic systems, which incorporate modeling elements from multiple engineering domains and embedded control subsystems, can be particularly challenging. Computer Aided Engineering tools, such as mathematical modeling, have proven useful because they allow engineers to consider more design alternatives in shorter amounts of time. As part of the mathematical modeling effort, considerable resources can be dedicated to creating a new model for a particular purpose. Tools that help ease the burden of new model development and directly support mechatronic modeling are therefore of particular importance.

Structured, reusable mathematical models of common engineering components help to simplify the initial modeling task, to capture engineering knowledge for use in future projects, and to support group model development efforts. Many difficult issues are associated with implementing a structured modeling approach. A structured modeling framework that can accurately and simply represent systems of interest is needed. A flexible modeling environment for modifying model properties, while ensuring that models are not altered in ways that are inconsistent with the original design

intent should be developed. When many models have been defined and collected in a library, methods for efficiently locating models that are useful for a particular purpose become increasingly important. The desire to share models among various groups raises the issue of model security.

Research was conducted to investigate ways to address the above issues. As a result of this effort, a new modeling construct, the Multiport Template, was defined. The Multiport Template simplifies the creation of flexible, reusable models of mechatronic components and systems, helps ensure consistent model modification, leads to a natural, meaningful classification and ordering of models, and supports multiple library searching methods. Additional data constructs, used in conjunction with the Multiport Template, provide control of access to various model properties by different types of users.

The usefulness of the Multiport Template design was demonstrated by implementation of a particular modeling environment. Examples are presented that demonstrate how the modeling tools developed allow for completion of tasks that were not previously possible.

To my family

ACKNOWLEDGEMENTS

The completion of any significant endeavor in life, though often credited to an individual, is not the result of the efforts of one person. My completion of this advanced degree is no exception. Along the way I have been the recipient of assistance from countless individuals who have increased my understanding, directed and redirected my efforts in the right direction, corrected my errors and misconceptions, set challenging expectations for me, and given me support and understanding when I needed it the most. To all of these, I express my gratitude.

Among the many people who have contributed to my current success, I would like to call the attention to a select few. First is my academic advisor, Dr. Ron Rosenberg. I have worked with Dr. Rosenberg for six years, two years for a Master of Science degree and four years for a Ph.D. Throughout this time we have had many meetings and discussions, often trying to convince each other that our ideas were correct. In many instances they turned out to be the same idea after all, expressed from opposing perspectives. I have learned from his experience and insight and appreciate all the time and effort he has given in my behalf.

The other members of my committee, Dr. Radcliffe, Dr. Mukherjee, and Dr. Zeid, served as a valuable resource for ensuring that my work was of significant caliber. I

would like to thank them for challenging my ideas, pressing me to think about the unique contribution that I was trying to make, and for supporting me during difficult times.

I would be remiss if I did not also pause to express gratitude for the members of the family in which I grew up. Our unique blend of strengths and ambitions has been part of what has given me the drive and determination to pursue my goals. I would especially like to thank my mother for the extra support she gave at the end of this effort.

Finally, my deepest respect and gratitude goes to my wife, Julie. Without her love, sacrifice, and support, this accomplishment would have been unattainable.

TABLE OF CONTENTS

LIST OF TABLES	ix
LIST OF FIGURES	xi
CHAPTER 1 INTRODUCTION	1
1.1 Background	1
1.2 Area of Research and Scope	4
1.3 Research Issues	5
1.4 Research Objectives	9
1.5 Dissertation Organization	10
CHAPTER 2 DESIGN OF A MODELING ENVIRONMENT	12
2.1 Introduction	12
2.1.1 Design Considerations for a Modeling Environment	15
2.1.2 Previous Work	17
2.1.3 Approach	19
2.2 Definition of a General Multiport	20
2.2.1 Topological Properties	21
2.2.2 Parametric Properties	22
2.2.3 Functional Properties	23
2.2.4 Display Properties	26
2.3 Templates	27
2.3.1 Working With Instances and Templates	28
2.3.2 Working with Templates	31
2.3.3 Implications of Inheriting Constraints	32
2.3.4 Template Libraries	34
2.4 Illustrative Examples	38
2.4.1 A Template Definition	39
2.4.2 Creating and Classifying UDMT Templates	42
CHAPTER 3 IMPLEMENTATION OF A MODELING ENVIRONMENT	45
3.1 Implementation Background	45
3.1.1 Previous Work	46
3.1.2 Contents of Chapter	49
3.2 Creating and Editing Component Models	49
3.2.1 General Features	49
3.2.2 An Example	52
3.3 Creating and Editing Templates	54
3.3.1 Creating a Solenoid Template	54
3.3.2 Deriving a New Template From an Existing Template	71

3.3.3 Editing Existing Templates	74
3.4 Library Browsing Tools	76
3.4.1 Keywords Filters	76
3.4.2 Constraint Filters	77
3.4.3 Generation History Display	78
 CHAPTER 4 CONTROLLED ACCESS	80
4.1 Introduction	80
4.2 Design of a Controlled Access Environment	81
4.2.1 Access-Controllable Attributes	82
4.2.2 Users and Groups	84
4.2.3 Group Access Settings	86
4.3 An Implementation of Controlled Access	87
4.3.1 Current User And Group Members	87
4.3.2 Specifying Group Access Settings	89
4.3.3 Enforcement of Controlled Access	90
4.4 Two Illustrative Examples	92
 CHAPTER 5 CONCLUSIONS	96
5.1 Summary of Contributions	96
5.1.1 Template Design	97
5.1.2 Library Tools	98
5.1.3 Environment Implementation	99
5.1.4 Controlled Access	100
5.2 Areas for Future Research	101
5.2.1 Template Design	101
5.2.2 Library Tools	103
5.2.3 Implementation Environment	105
5.2.4 Controlled Access	105
 APPENDIX Implementation Details	108
 REFERENCES	114

LIST OF TABLES

Table 1. Port Variable Names.....	63
Table 2. A Set of Equations for a Field Controlled DC Motor.....	73
Table 3. Three Access-Controllable Objects and Their Possible Access Values.	83
Table 4. Major Files Used for Creating the MB Environment.	112

LIST OF FIGURES

Figure 1. A Scalar Gain Block.....	13
Figure 2. Three Areas Related to Modeling.....	14
Figure 3. Improperly Modified Bond Graph, 1-Port, C-Type.....	16
Figure 4. A General Multiport.....	22
Figure 5. Two Types of Multiport Components.....	26
Figure 6. Working with an Instance and Template.....	28
Figure 7. Working With Templates.....	32
Figure 8. Parent-Child Structure of Templates.....	34
Figure 9. Two Paths for Generating a Template.....	38
Figure 10. A Template of a PMDC Motor.....	40
Figure 11. Deriving New Templates.....	43
Figure 12. Model Builder Environment.....	51
Figure 13. An Instance of Bond Graph, One-Port Capacitance Component.....	52
Figure 14. Editing the Capacitance Component's Equations.....	53
Figure 15. A Solenoid Component.....	54
Figure 16. Solenoid Template: Initial Form.....	55
Figure 17. Solenoid Template: General Properties.....	57
Figure 18. Solenoid Template: Number of Signal Ports.....	58
Figure 19. Solenoid Template: Number of Power Ports, Option 1.....	59
Figure 20. Solenoid Template: Number of Power Ports, Option 2.....	60
Figure 21. Solenoid Template: Editing Power Port Properties.....	61

Figure 22. Solenoid Template: Specifying the Coil Port Properties.	62
Figure 23. Solenoid Template: Specifying the Slug Port Properties.	64
Figure 24. Inductance of Coil as a Function of Slug Position.	66
Figure 25. Solenoid Template: Default Parameters.	67
Figure 26. Solenoid Template: Defining Default Parameter.	68
Figure 27. Solenoid Template: Default Equations.	69
Figure 28. Solenoid Template: Slug Force Equation.	70
Figure 29. Solenoid Template: Summary Page.	71
Figure 30. An Instance of the Solenoid Template Icon.	71
Figure 31. A Field-Controlled, Non-Linear, DC Motor Icon.	72
Figure 32. Permanent Magnet DC Motor Icon.	74
Figure 33. Filtering Templates by Keywords.	77
Figure 34. Filtering Templates by Constraints.	78
Figure 35. Template Generation History Display.	79
Figure 36. Organization of Groups and Users.	85
Figure 37. Specifying the Current User.	87
Figure 38. Interface for Editing Group Members.	88
Figure 39. Specifying Access Settings.	90
Figure 40. Restricted Access to Equations.	91
Figure 41. A Feedback Control System.	93
Figure 42. The MB Object Diagram.	110

CHAPTER 1

INTRODUCTION

1.1 Background

Engineers have come to rely on computers to complete an increasing number of tasks. During the course of a day, it would not be unusual for an engineer to use a computer to perform a numerical analysis, communicate with a colleague through email, search the Internet for crucial engineering data, and work on a technical document. However, despite the many ways in which computers can enhance the engineering effort, it is somewhat astonishing to note that there is a tremendous gap between the way computers fundamentally operate (the binary state of a set of electrical switches) and the way humans naturally think (spatial relationships, physical images, abstract reasoning, etc.). Computers are useful because of efforts to bridge this gap. What are some of the ways in which computers are made more useful? An answer to this question can lead to directions of future efforts to further increase the usefulness of computers.

Initially, computers became more useful by teaching humans to think like computers. This result involved humans learning rather cryptic computer languages, like Assembly, to perform low-level tasks, such as "moving" and "pushing" variable values around in computer memory. This demanding approach requires large amounts of time from highly skilled individuals (computer programmers). Even simple tasks can involve complex instructions to the computer.

Another way computers can become more useful is to "teach" computers to think more like humans (Dertouzos, 1997). An early advancement in this area was the development of the FORTRAN computer language (ANSI, 1966). FORTRAN improves the way humans communicate with computers in several ways. One improvement comes from the definition of data structures that support commonly used ideas and concepts. For example, FORTRAN supports many different variables types, such as character strings, integers, and complex numbers. Another improvement is in the specification of a more "human-like", natural syntax for instructing the computer to complete common computer tasks. This feature is helpful because it reduces programming efforts and makes it easier to remember commands. Another improvement comes from an organization of data into meaningful groupings. For example, code for performing a common task can be grouped in a subroutine. This ability helps to simplify computer instructions and also makes it possible to reuse a given set of instructions in multiple contexts.

Two observations of the above discussion are important to consider. The first observation is that the existence of FORTRAN does not change the fundamental computing ability of computers. Put another way, any set of computer instructions written in FORTRAN could also be written in Assembly. If this statement is true, then what is the added value of creating improved data structures, syntax, and organization? The second observation addresses this question: the benefit is that the way computers are instructed to perform a given task is brought into closer conformity with the way humans

reason about performing that task. This benefit results in a decreased investment of time and greatly improves productivity.

The trend to simplify the way computers are instructed has continued. Object-oriented programming languages like C++ (ISO, 1998) have been developed. These languages have data structures, syntax, and an organization that more naturally reflect the ways humans reason about the world (Pressman, 1992). Computer operating systems have evolved from text based, command-line driven tools to graphical, mouse-driven interfaces. Fewer people use computer languages directly. Instead, computer programmers use the computer languages to create computer software to perform specific tasks. Tasks that are cumbersome to complete using a computer language are almost trivially performed using a computer program. For example, using FORTRAN to generate a finite element mesh for a complex, 3-dimensional object, is a formidable undertaking. However, programs like ANSYS (Swanson Analysis Systems, Inc., 1998) can be used to perform this task at the click of a button. For each of these advancements, the above observations apply, i.e., communication with computers is simplified, but fundamental computing power is not increased.

Computer software, therefore, can enhance the engineering effort by bringing the performance of the computer more in line with human thought and by reducing the computer-specific knowledge that is required to solve engineering problems. The general area in engineering that uses computer tools for this purpose is often referred to as Computer Aided Engineering (CAE). The general purpose of this dissertation is to

explore ways in which to enhance engineering efforts through improved CAE tools. The approach followed is similar to the trend previously outlined. Enhancements will be sought by searching for ways in which new data structures, problem representations, and information organization can be exploited.

1.2 Area of Research and Scope

CAE covers a broad class of tools for many different engineering applications. The research described in this document will focus on computer support for one area of increasing engineering interest, the dynamic behavior of mechatronic systems. As opposed to a static response, a dynamic response is one in which relevant system parameters significantly evolve over a time period of interest (Umez-Eronini, 1999). Whether a system is considered dynamic or not is a subjective decision and depends on engineering judgement and operating conditions (Stein and Rosenberg, 1991). For example, an automobile's rack-and-pinion steering system may be treated as a static system when evaluating its response under freeway driving conditions, but it may be considered a dynamic system when evaluating its response during a high-speed race.

Mechatronics is a relatively new term with somewhat varying definitions (Auslander, 1996; Comerford, 1994; Buur, 1992); however, common ideas have emerged. In this dissertation, a mechatronic system will be defined as one that has two general features:

- 1) it is composed of components from multiple engineering domains, such as mechanical, electrical, hydraulic, acoustic, thermal, and magnetic; and
- 2) it has integrated automatic control subsystems as an inherent part of its design.

There are many areas in which computers can support an investigation into the dynamic response of a mechatronic system. One area is system representation. Initially, a model of a system to be investigated must be constructed. The model could take one of many forms, including a set of mathematical equations, a qualitative description of how the system behaves, or a physical description of the model's properties. Another area is system transformation. For example, at some point a model must be transformed into a representation that is suitable for numerical simulation. The numerical simulation itself is another area. There are many algorithms for performing numerical simulations, and a variety of strategies for selecting a suitable algorithm.

The research described in this document deals primarily with creating and working with model representations. These issues related to these topics are addressed in more specific detail in the next section.

1.3 Research Issues

Mathematical modeling of mechatronic systems and the corresponding computer tools that support it generally have been successful in supporting mechatronic system design. "Virtual prototyping" can decrease the need for physical prototyping and thus reduce design cycle time and design development costs. However, there are constant

pressures in industry to further decrease design cycle times, reduce development costs, and consider increasingly complex designs. Also, the large resource investments, in terms of time and money, that are used to create models for a specific modeling purposes are, unfortunately, not always available to use in future efforts. These observations indicate a need for improved modeling tools that make the model building process as easy and efficient as possible and support model reuse. In general terms, an improvement in computer tools can improve productivity (Gibbs, 1997). An improved modeling environment that decreases the modeling effort benefits both industry and academia. Simplified modeling tools reduce the learning overhead and allow those using the tools to focus on more relevant modeling issues.

Another issue in mechatronic systems modeling is the inherent multidisciplinary nature of mechatronic systems. One class of modeling software is based on fixed input/output information flow, a subset of which is block diagrams. Simulink (MathWorks, 1999), SystemBuild (Integrated Systems, Inc., 1994), and Easy5 (Boeing, 1998) are typical examples. These types of software are widely used in both academia and industry. However, Otter and Cellier (1996) discuss why these modeling constructs, while quite useful for controller design, are not the best-suited tools for modeling physical systems, despite assertions to the contrary (Fritchman and Hammond, 1993; MathWorks, 1998). One reason supporting this assertion is that the fixed input/output nature of block diagrams deters model reuse, since a component model's input/output structure may depend on its use in a particular system. This weakness can be especially troubling in a large, hierarchical model. Another reason is that the representation of

component models can be confusing since interactions are limited only to signals. A third reason is that there are no general tools for working with transducer components. Cellier (1992a) indicates that a more appropriate modeling approach would include power-based interactions, enabling descriptions of physical components that are much closer to reality. The use of power-based interactions also helps a modeler to avoid some common modeling errors by enforcing conservation of energy laws.

Due to these reasons specified above, modeling data structures intended to support mechatronic systems modeling should support both data and power flows. Some examples of previously defined tools that are specifically designed for systems composed of components from multiple power domains are bond graphs (Karnopp, et. al, 1991), object diagrams (Otter, 1997), SIDOPS+ (Bruenese and Broenink, 1997), and Modelica (Mattson and Elmqvist, 1998). These tools are finding increased attention and acceptance as is evidenced by computer software that is based on these tools, such as ENPORT (Rosencode Associates, Inc., 1995), 20Sim (Controllab Product, 1998), Dymola (Dynasim, 1999), and AMESim (IMAGINE, 1996).

Another modeling issue involves the support provided for model reuse. Currently, many modeling environments support component model reuse by supplying a set of pre-defined model types, contained in a fixed library. A weakness of pre-defined model types arises when one wishes to modify certain properties of a component based on that type. The ways in which a model's properties are modified can be quite limited. Often the only properties that can be changed are the model's connectivity and parameter

values. If a pre-defined model that precisely matches the current needs cannot be found, then a new model must be generated from scratch. A good modeling environment should support increased editing capabilities of library components.

In another effort to support model reuse, a modeling environment may provide support for a specialized model definition, referred to here as User-Defined Model Types (UDMTs). This approach allows a user to define a new model type, in a way similar to a modeling environment supplying pre-defined types. In many cases, UDMTs are defined using a fixed-form, programmatically specified data structure. These types of tools require knowledge of highly-specialized data constructs and can be quite complex. Also, once a new UDMT is created, it can suffer from the same problems associated with pre-defined model components, namely that modification of model properties is limited and organization and browsing tools have limited support. On the other hand, some modeling environments allow for essentially unlimited modification of modeling components that are based on UDMTs. This feature provides increased flexibility to modify an existing model in a library to meet a new need, but it doesn't prohibit the model from being altered in ways that are inconsistent with its original design.

Engineering model libraries may come to include a large number of models, each designed for a very specific purpose. As the number of pre-defined modeling types in the library grows, searching through the library contents to find a useful model for a current modeling purpose becomes increasingly difficult. To address this issue, a design

environment should support simplified, efficient tools for organizing and browsing the contents of a component model library.

A final modeling issue considered in this research involves the problems associated with sharing component models that contain data requiring restricted access. Such a situation can arise when two companies must share model information to accomplish a systems design. Some information included in the model may be proprietary to one of the companies. This situation can also arise in an academic setting when an instructor prepares a model for student investigation. Perhaps students should have the ability to use the model to determine its behavior, but not to view its properties. A simple approach to controlling the access to a model's properties is to "lock up" the entire model. This approach might be accomplished using file access control properties supplied with some operating systems. However, such an approach can lead to a design that is far from optimal, since the control of access to the model details is limited to the model as a whole. Therefore tools should be provided that allow an owner of a model to make available to a model user some of the model details, according to the user's classification. Such a feature allows for varying levels of security regarding both reading and modifying model details.

1.4 Research Objectives

The principal objective of this research is to design data structures, formulate new concepts, and organize existing information that results in significant enhancements to modeling environments for mechatronic systems. This design addresses the issues

discussed in the previous section. Specifically, the work will be divided into three areas relating to modeling of mechatronic components and systems:

- 1) creating and correctly modifying models and model types of mechatronic components and systems,
- 2) organizing and browsing the contents of a library of mechatronic model types, and
- 3) controlling the access to details of models of mechatronic components and systems.

Another objective of this research is to demonstrate the efficacy of the design through a computer implementation. The implementation shows that the issues raised in the previous section are addressed, and that the proposed design improves modeling environments for mechatronic systems.

1.5 Dissertation Organization

The remainder of this dissertation is contained in four chapters. Their contents are described briefly in this section.

Issues related to the design of modeling environment for mechatronic systems and components are covered in CHAPTER 2. A general purpose modeling structure for supporting modeling of mechatronic systems and components is defined. This data structure is called a General Multiport. Using this data structure, a novel, template-based approach for creating User-Defined Model Types is presented. It is shown how the tools

improve the way in which UDMTs are used and defined. The benefits also extend to providing enhanced methods for classifying and browsing the contents of a set of UDMTs stored in a library. A modeling environment implementation for mechatronic systems that is based on these ideas is presented in CHAPTER 3.

In CHAPTER 4 a design is presented which supports the control of access to various model properties of mechatronic components and systems. An implementation of the design is presented and its usefulness is illustrated with two examples.

A summary and conclusions are given in CHAPTER 5 along with a set of recommendations for future research in this area. The appendix gives a brief description of the implementation of the modeling environment. This document concludes with a list of references that were useful in carrying out this research.

CHAPTER 2

DESIGN OF A MODELING ENVIRONMENT

2.1 Introduction

As was discussed in CHAPTER 1, there is a need for enhanced modeling tools that decrease the burden of creating models of mechatronic systems. One strategy that has been successfully employed in the past is the use of a modeling structure that will be referred to as a *User-Defined Model Type* (UDMT). To understand what a UDMT is, it is helpful to first consider the more common modeling structure, a *Pre-Defined Model Type* (PDMT). PDMTs are basic modeling components that are specific to a modeling environment. Their fundamental definition cannot be changed; only particular attributes can be changed. As an example, consider a modeling environment that provides a pre-defined model of a common component, a scalar Gain Block. To use a Gain Block model in a system, the user creates a model *instance*, as shown in Figure 1. The only modifications that can be made to the instance are how it is connected in the system and the value of its gain parameter. This behavior, or what makes the Gain Block "act" like a Gain Block, is intrinsically defined as part of the modeling environment and cannot be changed; the Gain Block definition is fixed.

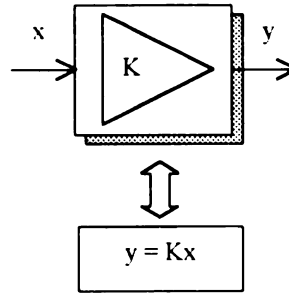


Figure 1. A Scalar Gain Block.

The use of PDMTs has been useful in the support of mechatronic systems modeling. In some environments, there is no other option for building a system model than to use the PDMTs that are supplied with the environment. In these cases, the flexibility and customizability of the environment is limited. No new modeling types can be defined without a re-definition of the environment. In response to this issue, some modeling environments provide a more general modeling structure that allows a user to not only create model instances, but to also create User-Defined Model Types.

It can be confusing to understand some of the issues regarding UDMTs. One reason for this difficulty is that when dealing with model types, a more abstract way of thinking is required. Therefore, before proceeding, it is important that the fundamental distinction between model *types* and model *instances* be clearly understood. One way to describe the fundamental difference is as follows: model instances describe the properties of physical systems and model types describe the properties of model instances. Stated another way, model instances are based on model types. A simple example will help to demonstrate this point. Figure 2 shows three different areas that may be of concern when creating a system model: (a) the physical system being studied,

(b) a system model, and (c) a set of model types. The model in (b) is a representation of the system shown in (a). It is composed of a set of component model instances. The intent of the system model is to describe relevant behavior of the physical system. There is a direct relationship between the model instances and the component or property of the physical system which they represent.

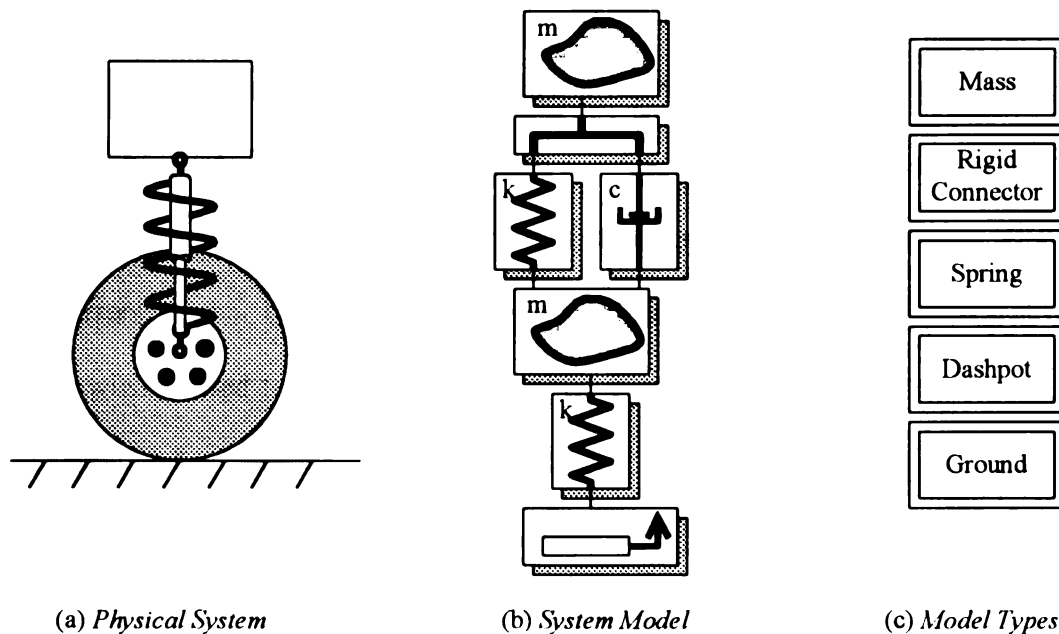


Figure 2. Three Areas Related to Modeling.

In a similar relationship, the model types in (c) are general descriptions of the component models specified in (b). The intent of a model type is to describe the behavior of its corresponding modeling instance. A single model type description is related to multiple model instances. For example, there are two spring models, but only one spring type. Each instance of the spring model maintains some data that is unique (e.g. the

stiffness value). The spring type maintains data that is common among all spring instances (e.g. the fact that there are only two connection points).

2.1.1 Design Considerations for a Modeling Environment

It is desirable for a modeling environment to support User-Defined Model Types (UDMTs) for storage in a model library. UDMTs provide a means to customize the modeling environment with the models that are most useful. Model generation time is decreased because, instead of starting from scratch each time a new system is to be modeled, an engineer can browse a library of existing UDMTs for ones that most closely meet current needs. UDMTs also serve as a repository of modeling knowledge; efforts and knowledge used to solve previous problems become resources available for solving future problems.

There are many difficult issues related to using and developing UDMTs. A good definition of a UDMT allows for sufficient detail to capture desired behavior while being flexible enough to allow for future modification, thus providing specialization for a particular purpose. The amount of flexibility in modifying properties a user should have when working with an instance of a UDMT is not universally agreed upon. Current implementations tend to fall at one of two ends of a spectrum. At one end, allowed modifications are quite limited. In many environments the only allowed modification is the setting of parameter values; meanwhile the underlying equations remain invariant. This scheme helps to ensure that a UDMT instance won't be modified in ways inconsistent with its intended use. However, it severely restricts the useful forms that it

can take. Since a model is an approximation of reality, there are generally many possible forms any given model may take. It therefore becomes necessary to store multiple, closely-related model forms of any given component model.

At the other end of the spectrum, modifications of the properties of a UDMT instance are relatively unlimited. This philosophy gives more flexibility when using a UDMT, usually at considerable effort in defining a new instance. In addition, it may mean that models can be changed in ways that are inconsistent with original intentions. Figure 3 illustrates how this philosophy can lead to an improperly modified model. A default bond graph, 1-Port C-Type is shown in part (a). Since the general form of a bond graph C-Type allows multiple equation forms, these equations should be accessible to the user. However, the lack of constraints on how these equations can be modified can lead to an improper modification, as shown in Figure 3(b) where the function relationship $e = f \cdot C$ is more appropriate for a purely energy dissipating component, like a bond graph R-Type (Karnopp, et. al., 1990).

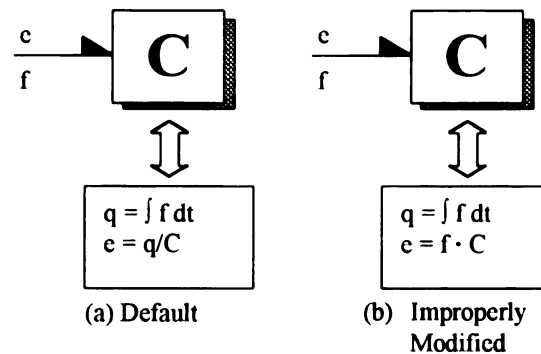


Figure 3. Improperly Modified Bond Graph, 1-Port, C-Type.

Another difficult issue in working with UDMTs arises when one wants to create a new UDMT. This task generally involves learning a specialized modeling language or set of function calls. Also, it is often necessary to start with a "blank slate" every time a new UDMT is needed, even if the new model type is only slightly different from an existing one.

A third issue involves the classification of UDMTs. When a large number of UDMTs have been defined, a clear and convenient ordering and a method for finding a model with desirable characteristics are imperative. Most commonly, model classification tools are limited to grouping model types according to model purpose or functionality. While this method can be effective, it has two weaknesses. The first weakness is that the location of a particular model type is a somewhat subjective decision. Two people may desire to place a given model in two different groups. The second weakness is that the method relies on models being assigned to "correct" groups. If a model is placed in an improper location, it could be hard to find, or worse, give an incorrect impression of the model's purpose.

2.1.2 Previous Work

The concept of using UDMTs to extend a modeling environment has been exploited previously by others, although the ideas have not been expressed in the fashion presented here. This section presents the conceptual ideas that have been explored by others in relation to this work. A more specific description of implementation details of some other groups is given in section 3.1.1.

Previous implementations of UDMTs specify a set of model properties. Instances based on a UDMT initially have the same attributes as specified by the UDMT definition. The type and manner of modifications that can be made to model instances are fixed; i.e., they are part of the UDMT definition. For example, in the Simulink modeling environment (MathWorks, 1999) model instances created from UDMT definitions only allow for modification of model parameter values. There is no way to extend the environment's ability to modify equations or to specify the way the parameters values are set. In the modeling environment 20Sim (Controllab Products, 1999) additional properties of a model instance can be modified (e.g., the model's equations), but which properties can and cannot be modified is not controlled by the creator of the UDMT, but is a function of the modeling environment.

Using a UDMT definition to prescribe the manner in which properties of a model can be modified has been suggested.. However, current thinking is limited. Thus far the only application has been to specify valid ranges of parameter and variable values, as in the SIDOPS+ modeling language definition (Bruenese, 1996). Extension to other model properties has not been previously explored.

Vries et. al. (1994, 1993) have explored methods for structuring of a set of model types. In their work they divide the properties of a UDMT into two categories, the *type specification* or *type interface* and the *implementation*. Properties that are classified in the type category are inherited when a new UDMT is derived from an existing one.

Implementation properties are not inherited. This strategy allows for an ordering of a set of UDMTs in a "class" or type structure. A similar approach is taken in the Modelica modeling language (Modelica, 1999; Mattson, et. al., 1998). However, this manner of inheriting properties is limited in that once a set of properties has been specified as part of the type interface, then all derived models have the same type interface. In Simulink there is no notion of structuring a set of models. Each UDMT is a unique entity with no relationship to any other UDMT.

2.1.3 Approach

To address the issues raised above, a new environment for describing User-Defined Model Types of mechatronic components was designed. This design improves the ways in which UDMTs are defined, created, and classified. This goal is accomplished with a new modeling structure, the Multiport Template. A Multiport Template is a combination of three items:

- 1) a modeling structure called a General Multiport,
- 2) constraints that specify the range of properties that the UDMT can have and that bound the values the properties can assume, and
- 3) default property values.

A Multiport Template (hereafter referred to as a Template) is used as the basis for creating instances of component models, prescribes the way an instance's properties can be modified, serves as a basis for deriving new Templates, and provides a mechanism for classifying Types based on functionality.

Section 2.2 describes the properties of a General Multiport. The Template design is given in Section 2.3. The usefulness and effectiveness of the design is demonstrated by two examples in Section 2.4.

2.2 Definition of a General Multiport

The definition of a fundamental modeling construct developed as part of this research is presented in this section. The data structure, called a General Multiport, establishes the range of models that can be defined by any instance of a Template. While this definition provides a foundation for building models that represent many types of dynamic systems, it is not meant to be all-inclusive. Also, many of these ideas have been studied and expressed in various formats by others. For example, power-based modeling ideas in the form of bond graphs were initially introduced by Paynter (1961) and later further explored and defined others (Breedveld, 1985; Cellier, 1991, Karnopp, et. al. 1991). Other general purpose, power-based modeling constructs have been defined that attempt to further expand the functionality of these tools (Rosenberg, et. al, 1996; Otter and Cellier, 1997; Elmqvist, et. al., 1998). Concepts related to hierarchical model representations have also been explored (Cellier, 1992b; Hales, 1995). However, the design presented here has additional features not previously considered. For example, no references have been found to indicate work done on specifying properties of a modeling structure that would ensure correct UDMT reuse and to allow for model classification schemes based on functionality.

It is convenient to divide the definition of the General Multiport into four categories. The Topological Properties are described in Section 2.2.1. Parametric properties are discussed in Section 2.2.2. Functional properties are presented in Section 2.2.3. Display properties are described in Section 2.2.4.

2.2.1 Topological Properties

The fundamental modeling entity in this system is referred to as a *Component*. Figure 4 shows a representation of a General Multiport Component, illustrating many of its properties that will be elaborated on in this and subsequent subsections. A Component can represent a physical object, like a shock absorber, or an effect, like mechanical friction. Components are classified as either *open*, meaning that they can be connected to other Components, or *closed*, meaning that they cannot be connected to other Components. A closed Component is given the special designator of *System*.

Ports are directly associated with Components. A Port indicates a site for connection between Components. Ports can be physically meaningful, like a shaft on a flywheel, or functional, like an input signal to a transfer function. Open Components have one or more Ports; closed Components have no Ports. A pair of Ports is associated by a *Connector*, thus connecting the two associated Components.

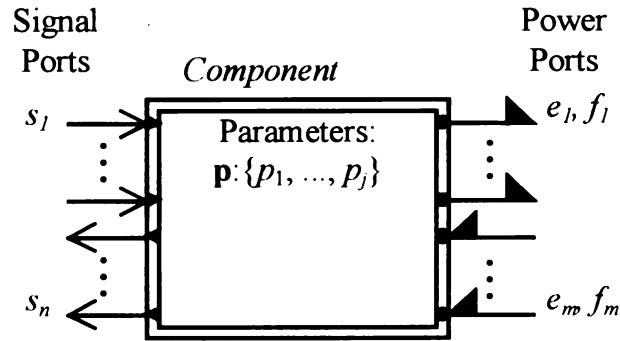


Figure 4. A General Multiport.

2.2.2 Parametric Properties

It is often useful to define a set of *Parameters* for modeling convenience. Parameters may point to physically meaningful attributes, such as material properties or geometric dimensions. It is also useful to define common symbolic constants, such as π . In this definition a Parameter's value does not change during the course of a simulation, although it may be changed between simulation runs. Although Parameters are ultimately used in Equations, that relationship is not important at this level. This distinction is meaningful because System Components, which have no Equations, may have a set of Parameters.

A Parameter can be associated with a Multiport in several different ways, depending on its intended use. A Parameter associated with a System is considered "global" and is accessible to any Component that is contained in the System. The density of hydraulic fluid is an example of a System Parameter. On occasion it is desirable to have a Parameter that is only accessible to a subset of a model's Components. In this case, a Parameter should be associated with a Subsystem Component. When a Parameter

should only be accessible to a single Component, then the Parameter should be associated with a Core Component.

2.2.3 Functional Properties

There are several aspects to the functional properties of a General Multiport. Port Variables, Internal Variables, and Equations are described in this section.

2.2.3.1 Port Variables

The Ports on a Component also serve a functional purpose. Ports have one or more *Port Variables* directly associated with them. Port Variables are dynamic; that is, they are functions of time.

Ports have one of two directions: *In* (towards the Component) or *Out* (away from the Component). The interpretation of the Port direction depends on the *Port Type*. A Port's Type classification is based on its Port Variables. A single Component can have multiple Ports of any Port Type associated with it. Currently, two Port Types have been defined, which will be discussed below.

A *Signal Port* Type has exactly one Port Variable. If the Port is directed In, then the Variable is a functional input; otherwise, it is a functional output. For example, a Block Diagram Sum Block has one or more In Ports and exactly one Out Port.

A *Power Port* Type has two Port Variables associated with it. The product of a Power Port's Variables represents a power flow. For example, a shaft Port's Variables are torque and angular velocity. The direction of a Power Port shows the positive direction of energy transfer.

One of a Power Port's Variables is used as a functional input and the other as a functional output. Which Port Variable is used as the input and which Variable is used as the output at a Power Port depends on the Port's *Causality*. Causality may be "fixed" or it may be dependent on the system assembly. For example, a model of an electric battery might have one Port with fixed Causality specifying the voltage is always a functional output. On the other hand, the Power Port of a model of an electrical resistor could have either voltage or current as a functional output, depending on how it is connected in a system model.

Additional Port Types and properties have previously been defined and studied and will not be elaborated upon here (Breunese, 1996; van Dijk, 1994). However, this design does not preclude future definition of other Port Types and properties.

For two Ports to be connected, they must at least have the same Type and complementary directions; i.e., one Port must be In and the other Out. A Port connection indicates that the corresponding Port Variables are directly coupled. The functional role of the Port Variables must also be complementary; i.e., Variables used as functional inputs on one Port must be functional outputs of the other Port.

2.2.3.2 Internal Variables

Internal Variables are Variables that are defined locally to a Component. As the name implies, the values associated with Internal Variables are not accessible to other Components, and their values can change over time. They are primarily used for convenience. For example, an intermediate computation may be assigned to an Internal Variable.

2.2.3.3 Equations

Equations are mathematical expressions consisting of Constants, Parameters, Port Variables and Internal Variables. Equations may be expressed in traditional mathematical form (e.g., algebraic and differential) or expressed as logic statements and procedures (e.g., *if-then* statements and loops).

Components come in two varieties, *Subsystem* and *Core*, as illustrated in Figure 5. Core Components are open, are directly associated with Equations, and are structurally irreducible. Subsystem Components are open and have a connected assembly of other Components, supporting hierarchical model descriptions. Subsystem Components have implied equations, derivable from their contained Core Components. Whether a Component is created as a Subsystem or a Core depends on modeling judgement. For example, a model of a DC motor could be represented either by an assembly of Components such as a resistor, inductor, inertia, etc., or by a stack of Equations.

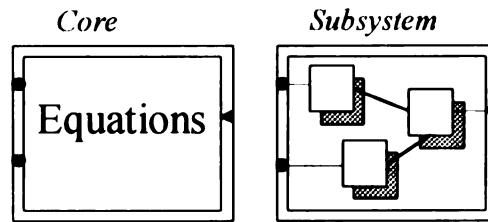


Figure 5. Two Types of Multiport Components.

An important characteristic of mathematical equations in this system is that they are in symbolic form. This feature is important for supporting reusable models (Cellier and Elmqvist, 1993). One reason this statement is true is that, using the connection scheme defined above, symbolic equations are required to accommodate Components that have system-dependent causality. That is, an equation may need to be inverted so that the functional output variable is explicitly calculated. For other connection schemes, it is possible to avoid equation reformulation, as was demonstrated by Byam (1999).

The data structure defined here makes it possible to specify an equation as having a fixed input/output form. An additional restriction on equations is that the output Port Variables must be computable for any legal causal configuration.

2.2.4 Display Properties

A Multiport's display properties include *Icon*, *Name*, and *Keywords*. These properties serve as mnemonics to quickly and conveniently locate, refer to, and classify models. It is important that the ideas implied by a model's display properties are

consistent with its actual definition. For example, a model named "Spring" with an Icon that looks like a spring should have other properties (Ports, Equations, Variables,...) that are consistently defined. Or, if the keyword "linear" is associated with a model, then the Equations should indeed be linear. Consistency between the Display and other model properties is important because the Display is often heavily relied upon when one is trying to understand or to explain the purpose of a model.

A good implementation that relies on display properties can be useful when browsing or searching the contents of a large library. For example, if models with similar purposes are grouped according to Keywords, then finding a model for a particular purpose at hand can be greatly simplified (Bruenese et. al., 1998, OLMECO, 1991). Finally, the Icon serves a practical role in a Graphical User Interface (GUI) for identifying, manipulating, and editing a model's properties.

2.3 Templates

Now that a useful modeling structure has been defined, how this information is used when working with User-Defined Model Types will be discussed. There are two general ways to work with a UDMT, depending on the task at hand: 1) creating instances and using them to build a model, and 2) defining and classifying new UDMTs. This section will discuss these two activities and describe the role of Templates.

2.3.1 Working With Instances and Templates

Figure 6 shows the relationship between an Instance based on a Template, a Template, and a User. A model is composed of an assembly of Instances. An Instance is a unique modeling object that is a realization of a configuration of some of the properties of the General Multiport. An Instance is created for a specific modeling purpose, such as for modeling a specific pump. Initially the attributes of the Instance are solely derived from the Template. User input is then used to specialize the properties of the Instance, such as specifying the pump displacement constant. Multiple Instances can be created from a single Template and each Instance maintains its own attribute data.

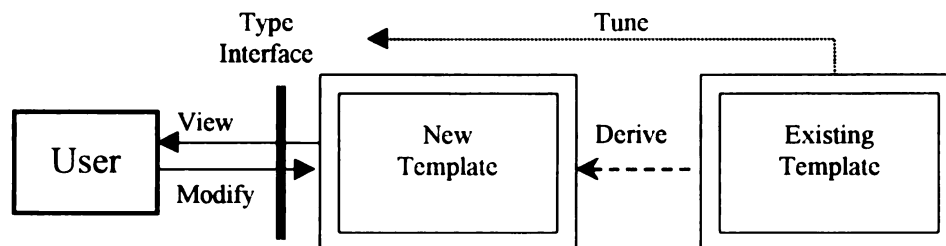


Figure 6. Working with an Instance and Template.

Interactions with an instance can be classified into two categories, viewing and modifying. It is significant that there are some properties of an Instance that can be examined by the user, but not modified. Also there are limits on how some properties can be modified. The mechanism for moderating modification of property values is the *Instance Interface*. All interactions between a User and an Instance must occur through this interface.

The key element in this arrangement is the Template. A Template is used to create Instances and specify how Instances can be modified. A Template accomplishes the goal by coordinating three sets of data.

- (1) *The General Multiport Definition.* All Templates are based on this definition, which bounds the range of models that can be defined using a Template.
- (2) *Constraints.* A Template lists a set of Constraints that are used to specify which properties of the General Multiport can be assumed by an Instance and limits the values that properties may assume. Stated another way, Constraints are used to further bound the properties that an Instance may assume. If a Template did not contain any Constraints, then it would be possible to define an Instance using any configuration defined by the General Multiport.
- (3) *Default Property Values.* A Template defines a default configuration of model properties for an Instance.

The unique feature in this design is the use of Constraints to specify a UDMT's properties. This method is different from other implementations of UDMTs, which only specify a set of properties. Constraint data is used to modify the behavior of the Interface Editor. When editing the properties of an Instance, the Editor is "tuned" according to the specified Constraints listed in the Template. In this way it becomes possible to create User-Defined Model Types that are edited correctly by all users of the model.

A brief example will help to illustrate these ideas. Consider again the 1-Port C-Component shown in Figure 3 and the three aspects of a Template discussed above that would be needed to define this Model Type.

General Multiport Definition

The constructs of the General Multiport Definition described in Section 2.2 are sufficient to support the definition of a 1-Port C-Component.

Constraints

The following statements prescribe the way instances of the C-Component can be modified:

- The C-Component must have exactly one Power Port. That is, no other Port Types are allowed, the default Port cannot be deleted, and no other Power Ports can be added.
- The state Equation (first Equation) is fixed. That is, the user cannot modify it.
- The constitutive Equation (second Equation) has the fixed form $e = \phi(q, \mathbf{p})$. That is, it can only be composed of Variables e and q and Parameters \mathbf{p} , with e as an output.

Default Property Values

The following information specifies a default instance of the C-Component.

- A single In Power Port.
- First Equation: $q = \int f \, dt$
- Second Equation: $e = q/C$.

Now consider how the Template data is used. An Instance of the model is created using the default property values of the General Multiport. When a user attempts to modify the properties of the model, the Instance Editor interprets the Constraints and enforces them. For example, the Equation editor only allows a user to modify the second Equation. In addition, the editor checks if the form of the Equation defined by the user matches the Constraint.

2.3.2 Working with Templates

Figure 7 shows the relationship between an existing Template, a new Template, and a User. A new Template is created by derivation from an existing Template. The new Template inherits the Constraints and any default properties that have fixed Constraints from the existing Template. In this way, a strict parent-child relationship is established. Any previously defined Template can be used as a parent. One special Template, the General Multiport Template, has no Constraints or default values. An instance of the General Multiport Template can be manipulated by a user to have any configuration that is supported by the General Multiport definition. The General Multiport Template is used as a "seed" for deriving a set of (child) Templates.

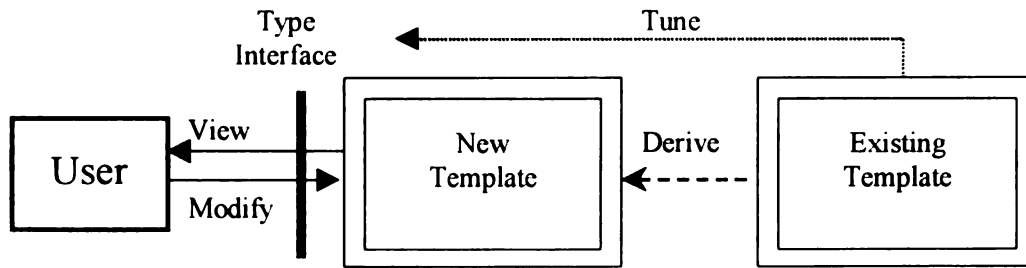


Figure 7. Working With Templates.

The input from the User in creating a new Template is limited to two types of actions. First, a User defines additional Constraints to be applied to an Existing Template. Second, default property values can be added and changed, subject to any previously defined constraints. The User works directly with the new Template through a Type Interface. This interface is tuned to enforce the rule that Constraints can only be added to an existing Template and assists the user in adding new Constraints to a new Template. This structure simplifies the process of creating a new Template, especially if a good Type Interface has been designed to present the User with the available choices of Constraints.

2.3.3 Implications of Inheriting Constraints

Recall that when a Template is created, it is derived from a parent. Derivation in this context means that the child Template inherits the parent's Constraints. During the creation process additional Constraints are specified, but Constraints specified by the parent can not be relaxed.

Further insight into the implications of this type of inheritance can be found by considering the influence it has on Template instances. The range of possible instances of a Template model is limited by its Constraints. For the General Multiport, the range of possible instances is limited only by its definition, i.e., there are no Constraints associated with the General Multiport Template. The range of possible instances of the General Multiport Template is depicted by the outer most oval in Figure 8(a).

When a new Template is derived from a parent and additional constraints are specified, as shown in Figure 8(b), then the range of possible instances decreases. More specifically, the range of possible instances is a subset of the range of possible instances of the parent. This relationship is shown in Figure 8(a). The range of instances created from Template 1 is smaller than the range of instances that can be created by the General Multiport Template. If Template 1 is in turn used as a parent for Template 2, then the range of possible instances that can be defined based on Template 2 is a subset of those that could be created using Template 1. Finally, Template 3 is also derived from the General Multiport Template, showing that multiple Templates can be derived from a single parent.

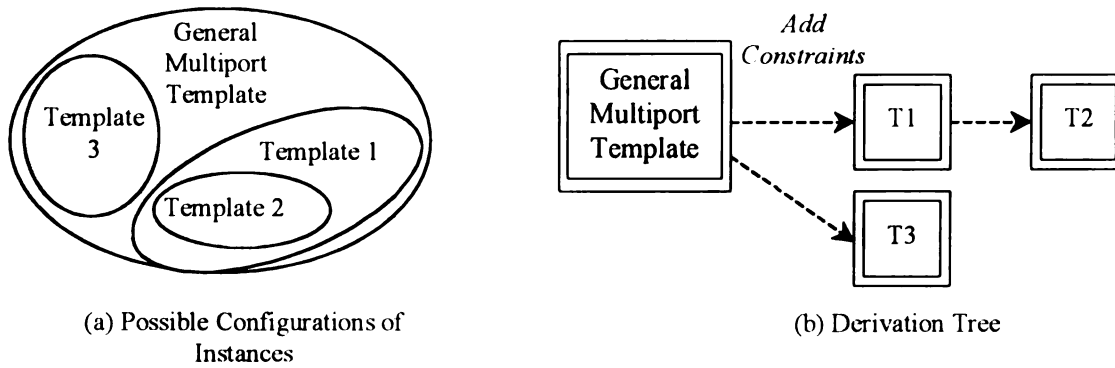


Figure 8. Parent-Child Structure of Templates.

2.3.4 Template Libraries

Many commercially available modeling environments currently provide a large number of pre-defined model types for specialized purposes. Models are often organized into groups according to similar purpose or power domain. Models are listed in the library by a name and/or an icon. Easy5 (Boeing, 1998) has a typical library architecture. Several pre-defined groups or models are defined. Each group is referred to as a library. Some of the libraries provided by Easy5 are the "Valve and Actuator Design Library", the "Electric Drive Library", and the "Aerospace Vehicle Library". When a new UDMT is created in this environment, it can be placed in any pre-existing library or in a new library specified by the user.

If a large number of UDMTs have been created and stored in a library, then it can be cumbersome to browse through a large, flat list, searching for a model type that is useful for a specific application. This observation is especially true if the set of models is

being used by someone other than the originator. Another limitation to this scheme is that a given instance of a model only resides in one library at a time. It may be difficult to locate a particular model if it cannot possibly be placed in multiple libraries, especially if the purpose and intended use of the UDMT must be deciphered from only the name and/or icon. Some of these limitation were addressed by Bruenese, et. al. (1998), who define a classification scheme that allows for models to have multiple classifications. However, another issue that is not addressed involves the fact that the classification system is based on subjective reasoning; how a model is classified by one person may not match another person's way of thinking. Also, if the number of models with a given classification becomes large, the original problem of sorting through a large, flat list of models remains.

One benefit of the Template structure is that at least three methods for organizing a list of Templates are possible: by keyword, by constraint, and by generation history. These methods help to overcome the problems listed above. A description of these methods is given below.

2.3.4.1 Organization By Keyword

Recall that part of a Template definition includes a set of keywords. By thoughtfully associating a set of keywords with the Templates in a library, a simple yet powerful mechanism can be employed for searching through a list of Templates. To begin a search, one or more keywords from a list of known keywords are specified. Next, each Template in the library is examined and its keywords are compared to the

keywords in the search list. The result of the search would be to produce a list of all the Templates that have the same keywords that are in the search list. The matching criteria can be simple, (e.g., any keyword matches any target list entry) or more complex (e.g., based on a logical construction of keywords).

This searching scheme allows a set of Templates to be classified in multiple ways, not just placed in a single group. It also means that it is possible to find a Template in a Library in multiple ways. This result is practically important, since different people organize their thinking in different ways.

2.3.4.2 Organization By Constraint

One limitation to using keywords as a searching device is that there is no control over which keywords get associated with a Template. This situation has the potential to be misleading and frustrating, with searches leading to inappropriate Templates. The idea of using keywords as a filter can then be extended to using the Constraints that are associated with the Template. For Constraints that are sufficiently general, it would be possible to search a library for Templates that have that Constraint. For example, a library could be searched for Templates that have the Constraint that no Power Ports are allowed, or for Templates that are constrained to have only Algebraic Equations.

This scheme has many of the same benefits discussed with the keyword method. An additional advantage to this idea is that the Constraints associated with a Template directly influence its functionality. This result means that it isn't possible to have a

search that produces a poorly matched Template. That is, the Templates found as a result of a Constraint search are guaranteed to exhibit the behavior specified by the Constraints. Another advantage is that classifying models based on what Constraints are applied to them is a natural way to organize one's thinking about a set of models.

One weakness of this scheme is that the list of Constraints on which to search is limited to the types of Constraints known to the system. Also, the Constraints are strictly limited to functionality; abstract classifications that are possible using Keywords are not possible. For these reasons, it is useful to support searching both by keywords and by Constraints.

2.3.4.3 Organization By Generation History

A third classification scheme takes advantage of the Parent-Child relationship between two Templates (see Figure 7). This relationship sets up a natural ordering of Templates in a tree structure instead of in a flat list. Recall that a child Template is a specialization of a parent Template. Browsing is aided by this fact. For example, if a candidate model located in a library is too general, then a child Template can be sought that specializes the behavior in an appropriate way. Conversely, if a model found in the library is too restrictive, the parent model can be considered.

An additional benefit to this classification arises from the fact that the classification structure can be patterned after the thinking of the person building it. This is possible because the classification structure is not unique. For example, consider the

task of initializing an environment with a set of electrical components that only have Power Ports. One way to go about this task is to first define a Parent Template that adds the single constraint that all the power ports must be electrical. The next step might be to derive a new Template from this Parent Template and add a single Constraint specifying that there are no Signal Ports. This path is illustrated in Figure 9(a). An equally valid option for obtaining the same results would be first to define a Template with no Signal Ports and then to use this Template to derive one that adds the Constraint that all Ports are electrical.

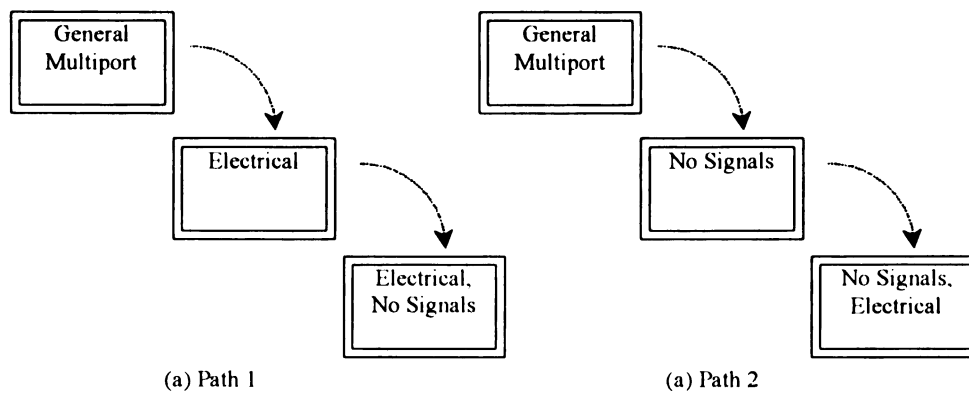


Figure 9. Two Paths for Generating a Template.

2.4 Illustrative Examples

Two examples will help to illustrate how the design ideas discussed in this chapter are used when working with Templates. The first example demonstrates how a Template is used to define and work with an instance. The second example shows how

new Templates are created based on existing Templates and the natural classification that arises.

2.4.1 A Template Definition

A representation of the information associated with a Template of a Permanent-Magnet DC Motor is shown in Figure 10. The information is divided into the four categories that were used to describe a General Multiport in Section 2.2. On the left side, default property values are given. Constraints are listed on the right hand side and correlate with the properties listed directly to the left.

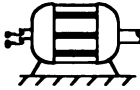
Topological		Constraints
<u>Component Type</u> Core		Fixed.
<u>Ports</u> Electrical Power In Rotational Power Out		Fixed Port Number. Fixed domain, direction. Fixed domain, direction.
Functional		Constraints
<u>Port Variables</u> Electrical: v, i Rotational: τ, ω		Prefer v as input. Prefer τ as input.
<u>Equations</u> $v = v_L + v_R + v_m$ $\tau = \tau_m - \tau_J - \tau_R$ $v_m = k_m \omega$ $\tau_m = k_m i$ $v_L = L \frac{di}{dt}$ $v_R = R_e i$ $\tau_J = J \dot{\omega}$ $\tau_R = R_m \omega$		Allow ODE and Algebraic. Fixed. Fixed. Fixed. Fixed. $v_L = \phi_5(\frac{di}{dt}, \mathbf{p})$ $v_R = \phi_6(i, \mathbf{p})$ $\tau_R = \phi_7(\dot{\omega}, \mathbf{p})$ $\tau_R = \phi_8(\omega, \mathbf{p})$
Parametric		Constraints
<u>Parameters</u> $\mathbf{p} = \{k_m, R_e, L, J, R_m\}$		$k_m \in \{k_{m1}, k_{m2}, \dots, k_{mn}\}$ $R_{e,min} \leq R_e \leq R_{e,max}$ $0 \leq L \leq L_{max}$ $J_{min} \leq J \leq J_{max}$ $0 < R_m \leq R_{m,max}$
Display		Constraints
<u>Icon</u> 		Fixed.
<u>Key Words</u> Two Port, Transducer,...		Fixed.
<u>Name</u> PMDC Motor		Fixed.

Figure 10. A Template of a PMDC Motor.

Note that the topological property of this Template that specifies the default Component Type is "Core". Recall that this attribute means that the Component is structurally irreducible and Equations are directly associated with the Component. A

Constraint on this property specifies that the "Core" designation is Fixed. This Constraint indicates that a user can not change an instance's Component Type from Core to Subsystem.

The default data also specifies that there are initially two Ports associated with this Component: an Electrical Power In and a Rotational Power Out. The Constraint, "Fixed Port Number" indicates that the Port number can not change; i.e., no Ports can be added or deleted. The power domain and power direction of each Port are also fixed.

The default Equations listed suggest that this model of a motor nominally considers the effects of the winding resistance and inductance, back *emf*, and the inertia and friction in the motor. Although all the effects are linear by default, some non-linear equations are possible.

There are two types of Constraints on Equations shown in the figure. Some Equations are "Fixed", meaning the Equation can not be changed; others have a fixed form, meaning the equation can only have the specified variables and parameters and must have the specified variable as an output.

There are two types of Constraints on Parameters shown. The motor constant k_m has a list of allowable values. This Constraint is consistent with the fact that motors are often available with discrete values of motor constants. The other Parameters have an allowable range of values in the format shown.

The Constraints are intended to make sure that the basic model effects are captured and that the form of the equations is correct, while still providing flexibility for multiple instances. This flexibility means that a broader range of model variations is possible than if only the parameter values could be changed. For example, the mechanical resistance equation model could be changed to model a different type of friction model, such as Coulomb.

Recall that Constraints specified in the Template are enforced by the Instance interface. For example, limits on parameter values are enforced whenever a user attempts to change a parameter value. If the entered value is not within the specified range, the user is immediately notified and required to fix the error before continuing. Another way that the Instance Editor can enforce Constraints is by never presenting the user with an option to perform a task that would violate a Constraint. For example, since the number of Ports is fixed, the user is never given the option to "add" or "delete" a Port. Using the Constraint information, an instance then "behaves" according to its design.

2.4.2 Creating and Classifying UDMT Templates

When creating a new Template, the starting point is either the General Multiport Template or another existing Template. Consider the process of defining new Templates as illustrated in Figure 11.

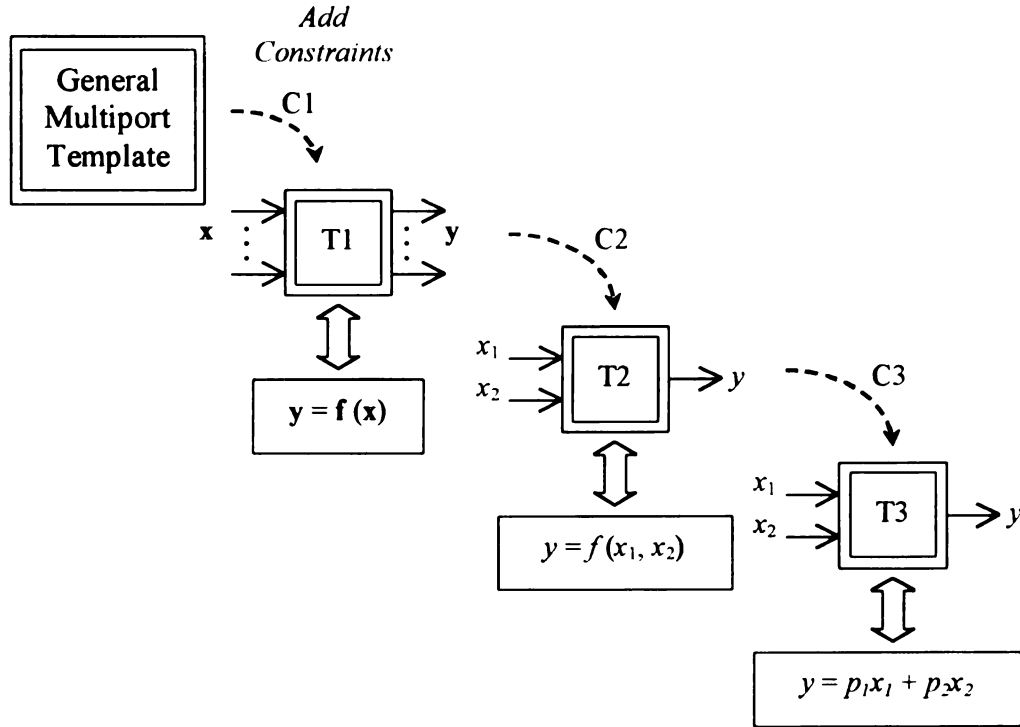


Figure 11. Deriving New Templates.

A new Type, T1, is created, based on the General Multiport Template. To accomplish this objective, Constraints C1 are added. The Constraints are as follows: no Power Ports, 1 to n In Signal Ports, 1 to m Out Signal Ports, and Algebraic Equations. These Constraints limit the possible models that may be defined using T1 as a Template, but still allow for a broad class of models.

A new Type, T2, is created, derived from the existing Type T1. The additional Constraints of exactly one Signal Out and exactly two Signal In Ports mean that models based on T2 have less flexibility than models based on T1, but they are more efficient for certain tasks.

Similarly, a new Type T3 is created based on T2. The additional Constraint in this case is that the Equations are Linear. This Constraint effectively creates a Weighted Sum Block, with the parameters p_1 and p_2 accessible for modification by the user.

Notice that Types T2 and T3 each could have been created directly from the General Multiport Template. However, this path would require more effort and input from the user than starting with an appropriate parent. It also should be emphasized that defining multiple Templates does not increase the range of models that can be defined. That range is bound by the General Multiport definition. In fact, if the only Template that was available the General Multiport Template, no generality would be lost; i.e., a user could create any model supported by the definition. However, each time an instance was created, a user would need to make a large number of modeling decisions that would eventually result in the desired model description. Instead, using models that have been appropriately constrained eases the modeling burden. Applying Constraints to a Template is therefore equivalent to a user making modeling decisions. By choosing a specific type, some modeling decisions are already being made.

The example above also demonstrates how the derivation of new Templates from existing Templates, by applying additional Constraints, leads to a natural classification of a set of Templates. Each Template is a special case or subset of its parent.

CHAPTER 3

IMPLEMENTATION OF A MODELING ENVIRONMENT

3.1 Implementation Background

In this chapter an implementation of a modeling environment for mechatronic systems is described. The environment was specifically targeted to support the General Multiport modeling construct, the use and creation of Templates, and use of the library tools described in CHAPTER 2. A brief synopsis of the goals of this design is presented here.

- The Environment will have a simple-to-use, graphically driven interface that is congruent with common software that is currently available. Although this goal is not a specific research issue, it is nevertheless a significant objective for any practical application (Mackulak, et. al., 1994).
- The Environment tools, semantics, organization, and operation will be congruent with standard modeling paradigms.
- The fundamental modeling construct will be the General Multiport.
- In this demonstration environment explicit, ordinary differential equations will be supported.
- Every modeling component defined in the system will be a User-Defined Model Type, which will be defined using Template concepts.
- The environment will use Template data to provide tools that ensure that model instances are used as intended.
- The creation of a new Template will be as simple as possible, only requiring a user to fill out a set of simple forms.

- New Templates will be derivable from existing Templates. Constraints are automatically inherited.
- A classification system is supported that supports library browsing and searching tools using keywords, constraints and generation history.

3.1.1 Previous Work

There are several existing software implementations that are used for mechatronic systems modeling that also support the definition of UDMTs. In this section, the relevant features of two modeling tools will be discussed. Simulink (MathWorks, 1999) is representative of software that is based on fixed input/out information flow. 20Sim (Controllab Products, 1999) is representative of software based on both information and power flow.

3.1.1.1 Simulink

The Simulink environment supports models with fixed input/output information flows. There is no support for power flows. UDMTs are implemented using "S-Functions". An S-Function is a text file written in the style of a program subroutine and uses one of three forms: syntax specific to Simulink, FORTRAN, or C. The name of the text file is the name of the UDMT. The syntax of an S-Function can be somewhat complex and requires knowledge of a specialized format and Simulink-defined functions. There is no automated software support for creating and editing S-Functions.

An S-Function definition provides for specification of a fixed number of Signal Ports. Each Port has a specific "width", which is the number of variables associated with it. Ports can either have a fixed or variably sized width. The actual number of variables associated with a Port with a variable width is specified when the S-Function is connected in a system.

A fixed input/output relationship among the port variables is specified in an S-Function using constructs that are common in FORTRAN and C and functions that have been defined by the Simulink environment. Once this relationship has been defined then 1) the causal form of the equation cannot be altered and 2) all instances of the S-Function must use that definition; i.e., an instance's equations cannot be modified. However, each instance can specify its own set of parameter values.

Simulink has some built in support for organizing a set of S-Functions. First, a set of "libraries" can be created. Each library can contain a set of S-Functions. A particular S-Function is found by manually searching the contents of each library. Alternatively, the location of a particular S-Function can be found using the S-Function's name and an automated search tool; i.e., if the S-Function name is known, then its model can be found.

3.1.1.2 20Sim

The software package 20Sim supports models composed of both information and power flows. All models that are defined by the system are based on the modeling

language SIDOPS+ (Bruenese and Broenink, 1997). The result is that nearly every component model in the system can be considered a UDMT. Comprehensive knowledge of SIDOPS+ is not required to create many types of component models, but at least some knowledge is required for all types of component models. The software provides some automated help in the creating of new UDMTs, mainly in the form of syntax checking.

SIDOPS+ specifications provide for either a fixed number or variable number of signal and power ports. Each port has a specific "dimension", indicating the size of a matrix that stores the port's variables.

When specifying the properties of a SIDOPS+ model, a default relationship among port variables, or a set of equations, can be supplied. 20Sim supports two features with respect to a component's equations that are of particular importance. First, a component's equations are treated *symbolically*. The result of this design is that multiple causal forms can automatically be derived. As was previously mentioned, this point is key for supporting the reuse of physical system models. Second, each instance of SIDOPS+ model maintains its own set of equations. The result of this design is that a given SIDOPS+ model can be used with greater flexibility. However, an unfortunate consequence also results. A SIDOPS+ model has limited ability to ensure that its equations are modified in a way consistent with original intentions. As an example, it is possible in the 20Sim environment to specify the equations of a bond graph, 1-port C-Component to behave as an R-Component, as discussed in Section 2.1.1.

Library support in the current version of 20Sim is limited to placing SIDOPS+ models in various directories of the operating system. However, a design for more extensive library tools has been described (Bruenese, 1997).

3.1.2 Contents of Chapter

Based on the review of currently available tools and the desired goals of this research, an implementation of an environment for modeling mechatronic systems was created. The modeling environment developed is called Model Builder (MB). General features of the MB and an example of working with a modeling component are described in Section 3.2. Using MB to define a new Template, both by starting from scratch and by using an existing Template, is illustrated in Section 3.3. Section 3.4 presents tools for browsing the contents of a library.

3.2 Creating and Editing Component Models

3.2.1 General Features

As can be seen in Figure 12, the MB environment is graphically driven and uses a common Windows interface. A list of available Templates appears on the left-hand side of the screen in the list labeled "Components". This list was specifically not labeled "Templates" to avoid forcing the advanced features of Templates onto users who simply want to use the environment for creating models based on existing Templates.

Consequently, in the rest of this chapter, "Component" is often used as a synonym for "Template". The context of the discussion will make the actual meaning clear.

The list of Components initially displays all of the Templates that are known to the system in alphabetical order. However, as is indicated in the figure, there are more sophisticated tools for browsing the list of Components. These features are described in Section 3.4.

Upon starting MB a new model is created. The top-level contents of a model are displayed in a *Model Window*. Multiple model files can be opened concurrently in the environment, with each model displayed in its own Model Window. The contents of Subsystem Components can also be displayed concurrently. To view the contents of a Subsystem Component, the right-mouse button is clicked on its icon. This action produces a "context menu" with commands that are applicable to the Subsystem Component. One of the context menu commands is "View Contents". If the Subsystem has not already been opened, then selecting this command causes a new Model Window to be created, displaying its contents. If the Subsystem has been previously opened, then selecting this command causes the Subsystem's Model Window to be brought to the top of the windows stack. There is no limit to the depth of imbedded subsystems in this environment.

In Figure 12, the top-level contents of a model called Model1 is shown on the top right-hand side. There is a single Subsystem Component in the top level of Model1,

called Subsystem_1. The contents of Subsystem_1, which is currently empty, are displayed in a Model Window on the bottom right-hand side.

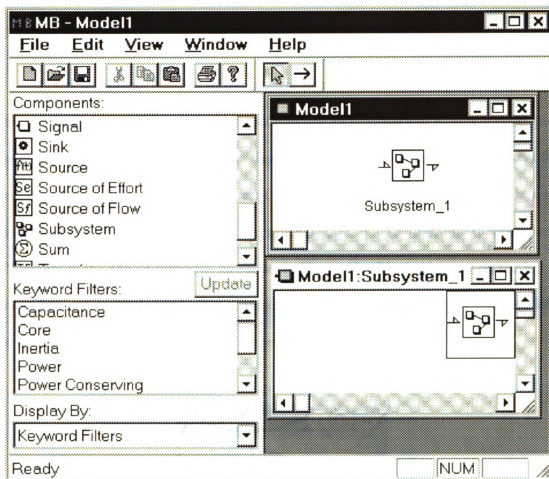


Figure 12. Model Builder Environment.

Selecting the desired Component from the Component List and dragging it into an active Model Window creates an instance of a Component. After this action an instance of that Component Template is created and its icon appears in the Model Window. The Component icon can be positioned freely in the Model Window by dragging it with the mouse. As is standard with graphically driven tools, many of the objects that are created

in a Model Window can be easily positioned by dragging, including a Component's Ports and its Label.

3.2.2 An Example

Some additional features of the general interface and the Template design can be demonstrated by considering the process of creating an instance and editing a bond graph, one-port Capacitance Component. The first step is to locate the desired Component from the Component List. Once the desired Component Template has been found, a new instance of that Component is created. At this point, an icon representing this component will appear in the Model Window, as shown in Figure 13. The half arrow on the left is a Power Port and is labeled "1". The Component label is "Capacitance_1". These labels can be edited and displayed or hidden at the user's discretion.

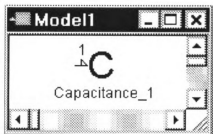


Figure 13. An Instance of Bond Graph, One-Port Capacitance Component.

The Template that defines the Capacitance Component is used to ensure correct Component "behavior". For instance, it is not possible to delete the Power Port or to add another Power Port or a Signal Port. Additional Template data restricts the way in which

the Capacitance Component's Equations are edited. Figure 14 shows the interface for editing the Capacitance Component's Equations. By default, there are two Equations. The first equation listed is a state equation. It is an inherent part of the definition of a Capacitance Component. Therefore, this equation cannot be edited or deleted. Disabling the *Add* and *Delete* buttons when the state equation is selected enforces these restrictions.

The second equation in Figure 14 is a linear constitutive law. However, since the Component is not restricted to this linear form, this equation can be edited, but not deleted. An additional constraint on editing this equation is that it must be of the form $e1 = \phi(q1)$, where $\phi(q1)$ is any function of the variable $q1$.

Equations

Definition:

q1 = Integral(f1,0)
e1 = q1/C

Add

Edit

Delete

Description:

State Equation

OK Cancel

Figure 14. Editing the Capacitance Component's Equations.

3.3 Creating and Editing Templates

In this section the tools provided for creating a new Template, for deriving a Template from an existing Template, and for editing an existing Template are discussed. The ideas are introduced by way of examples.

3.3.1 Creating a Solenoid Template

One of the design goals for this environment was to simplify the process of creating Templates. To this end an interface was designed that amounts to filling out a series of forms. Each of these forms will be reviewed for the process of creating a Template for a simple model of a solenoid, represented in Figure 15.

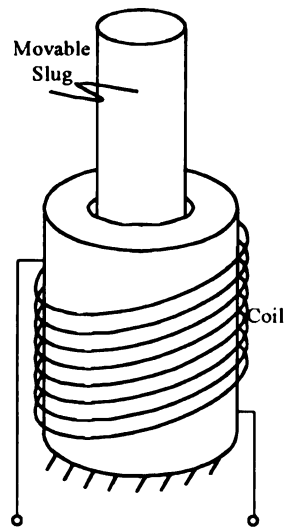
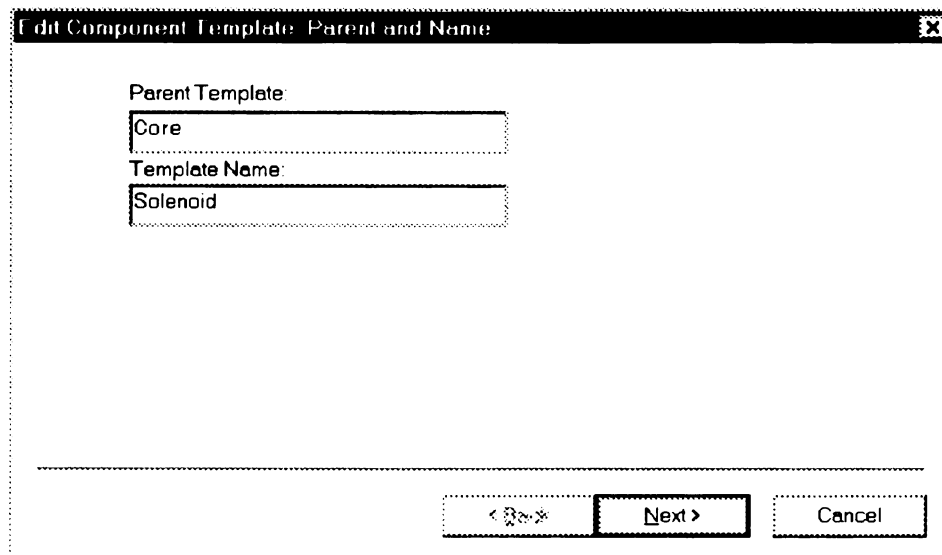


Figure 15. A Solenoid Component.

3.3.1.1 Initial Form

The first step in creating any Template is to select an existing Template from the Component Template list to act as a Parent. The constraints that are specified by the Parent are inherited. In this first example, a general Core Component is selected as the Parent. In this case, the only constraint inherited is that the new Template must also be a Core. When the parent Template has been selected, then the File menu command "Create Template..." can be selected. At this point the form shown in Figure 16 is displayed, with the Parent Template name automatically filled in. This form also provides the opportunity to give the new template a name, which must be unique when compared to the list of existing Templates. In the current case, "Solenoid" is chosen as the name for the new Template.



The image shows a dialog box titled "Edit Component Template: Parent and Name". It contains two text input fields. The first field is labeled "Parent Template:" and contains the text "Core". The second field is labeled "Template Name:" and contains the text "Solenoid". At the bottom of the dialog, there are three buttons: a button with a left arrow and a question mark, a button labeled "Next >", and a button labeled "Cancel".

Figure 16. Solenoid Template: Initial Form.

3.3.1.2 General Properties

The general properties of a Template are specified using the next form. The first piece of information is the *Component Type* that will be created. In the general case, the choices are Core or Subsystem. In the present case, when the parent Template is a Core component, there is only one option: Core.

There are two *Display Icon Types* available: Text or Bitmap. This property refers to the way in which a display icon for a Template is generated. The default option, and the simplest, is to specify a text string that will be used to create an icon. The second option is to supply a bitmap file that has the same base name as the Template that is being created. In the current case, Bitmap is chosen as the display icon type and a bitmap file called "Solenoid.BMP" must be externally generated.

The next general Template property is the *Default Label*. Each Component that exists in a Model Window must have a unique label. The reason for this restriction is that the label is used to generate unique System variable names. For Signal Ports, which have one variable associated with them, System Variables names are automatically generated by concatenating the Port name with the Component name. For Power Ports, which have two variables associated with them, System Variables are automatically generated by concatenating the name of the each Variable with the Port name and the Component name. This naming strategy makes it possible for two instances of a component to have the same port name set.

Every Component instance that is created will automatically have the default label specified in the Template appended by the instance number. For example, if the default label is "Solenoid" then first Solenoid created in a Model Window will be called "Solenoid1" and the second will be called "Solenoid2" and so forth. After an instance is created, its label can be changed by the user, subjected to the constraint that the label is unique among all other Components in the same Model Window.

Finally, a set of *Keywords* can be associated with the Template. There are no restrictions on what can be specified as a keyword, but each one should be thoughtfully chosen. As will be demonstrated, the keywords associated with a Template can be used to help search through a set of Templates. The completed form for the general properties of the Solenoid Template is shown in Figure 17.

Figure 17 is a screenshot of a software dialog box titled "Edit Component Template: Specify General Properties". The dialog box contains several input fields and a list box. On the left side, there is a "Component Type" dropdown menu with "Core" selected, a "Display Icon Type" dropdown menu with "Bitmap" selected, and a "Bitmap File" text box containing "Solenoid BMP". On the right side, there is a "Default Label" text box containing "Solenoid" and a "Keywords" list box containing "Core", "Power Conserving", and "Transducer". At the bottom right of the dialog box, there are three buttons: "< Back", "Next >", and "Cancel".

Figure 17. Solenoid Template: General Properties.

3.3.1.3 Number of Ports

A form for specifying the number of In Ports and the number of Out Ports that can be associated with a Component must be completed for each Port Type. Currently, two types of Ports have been defined, Signal and Power. For this Solenoid Template, it is specified that there should be no Signal Ports of either direction. Specifying that the number of In Ports and Out Ports as fixed at 0 completes this objective. Figure 18 shows the completed form for the number of Signal Ports on the Solenoid.

Number of Signal Ports

In

☒ Fixed ☐ Variable

Number: 0

Out

☒ Fixed ☐ Variable

Number: 0

Total

☒ Fixed ☐ Variable

< Back Next > Cancel

Figure 18. Solenoid Template: Number of Signal Ports.

The next decision to be made is the number of Power Ports to be associated with the Solenoid. In this case two Power Ports are needed, one to represent the flow of electrical power to and from the solenoids coils, and another to represent the flow mechanical power to and from the solenoid's slug.

A design question arises in deciding on the direction of the Ports. One possibility is that the direction of the power flow is unimportant and can be freely specified for any instance of the Template. In this case, the form is completed as shown in Figure 19. Here, the total number of Ports is fixed, but the In Port and Out Port numbers are variable from 0-2 and the default numbers are 1. This combination of constraints will ensure that the total number of Ports is always two (the sum of the default number of In and Out Ports), but the directions of either port can be set to be In or Out.

Number of Power Ports			
In			
<input type="radio"/> Fixed	Min:	Max:	Default:
<input checked="" type="radio"/> Variable	0	2	1
Out			
<input type="radio"/> Fixed	Min:	Max:	Default:
<input checked="" type="radio"/> Variable	0	2	1
Total			
<input checked="" type="radio"/> Fixed			
<input type="radio"/> Variable			

< Back Next > Cancel

Figure 19. Solenoid Template: Number of Power Ports, Option 1.

Another possibility is to specify that the In Port and Out Port numbers are fixed at 1. The completed form to accomplish this objective is shown in Figure 20. In this case, it is assumed that the solenoid is used by applying electrical power to generate mechanical power, although power can still flow in either direction. Therefore, the alternative to have a fixed number of In and Out Power Ports is selected.

Number of Power Ports

In

☒ Fixed ☐ Variable

Number: 1

Out

☒ Fixed ☐ Variable

Number: 1

Total

☒ Fixed ☐ Variable

< Back Next > Cancel

Figure 20. Solenoid Template: Number of Power Ports, Option 2.

3.3.1.4 Default Port Properties

After the number of Ports has been determined, additional Port Properties can be specified. The next form, shown in Figure 21, is used for this purpose.

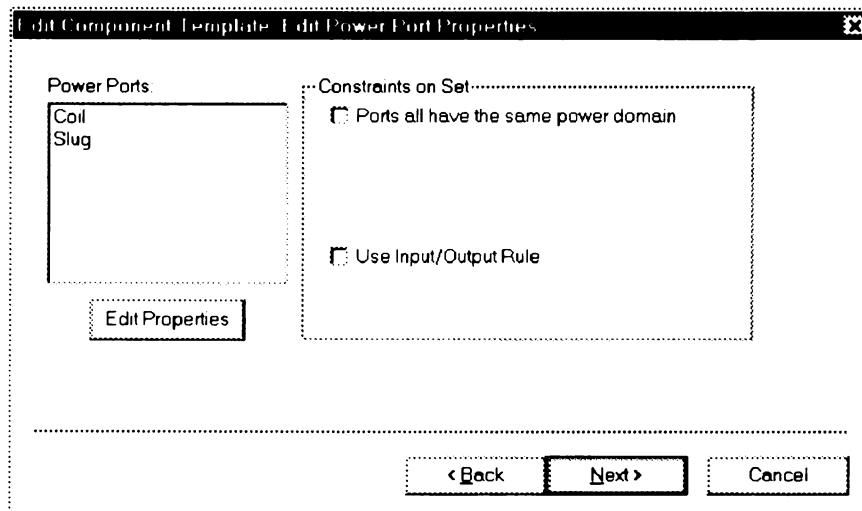


Figure 21. Solenoid Template: Editing Power Port Properties.

There are two Power Port properties that may need to be applied to the all the Ports as a set. First, all the Power Ports may be required to have the same power domain. Such is the case when defining a bond graph 0- or 1- junction. However, this is clearly not the case in the current situation. Second, the variables that are used as input and outputs may depend on a rule such as on a bond graph 0- junction where only one "effort" variable is to be used as in input, but it doesn't matter which port specifies the input effort variable. The solenoid does not require such a rule.

This form also lists default names for each of the default Ports that were specified on the previous form. The properties of each Port listed are specified by selecting the Port of interest and pressing the "Edit Properties" button. In the current case, the In Port represents the electrical coils and the Out Port represents the slug.

Figure 22 shows the form for editing the properties of the Port representing the electric coil. The Port *Label* is used in generating unique variable names. Therefore, each Port must have a unique Label. A useful Label in the current case is "Coil".

Default Port Properties	
Label: Coil	Location (Degrees): -90
Direction: In	Input/Output Constraint <input checked="" type="checkbox"/> Fixed Preferred: Effort In/Flow Out
Power Domain: <input checked="" type="checkbox"/> Fixed Electrical	State Variable: <input checked="" type="checkbox"/> Fixed p (momentum)
<div>OK</div> <div>Cancel</div>	

Figure 22. Solenoid Template: Specifying the Coil Port Properties.

The initial *Location* of the Port is specified here. The value given represents the degrees from a horizontal line extending from the center of the Component to the right. Angles are measured using counter clock-wise as positive. To be congruent with the schematic shown in Figure 15, the Port will be initially placed at -90 degrees. The *Direction* of the Port is displayed for reference purposes, but it cannot be changed here.

The *Power Domain* of the Port is specified on this form. The system understands five different power domains: General, Mechanical Translation, Mechanical Rotation, Electrical, and Hydraulic. The power domain value for a port is used for two purposes. First, the power domain determines the names of the Variables associated with the Port.

The Variable names and descriptions for each of the Power Domains are listed in Table 1.

Second, the system ensures that only Ports with the same power domain are connected.

Table 1. Port Variable Names.

Power Domain	Variables	
	Names	Descriptions
General	e	Effort
	f	Flow
	q	Displacement
	p	Momentum
Mechanical Translation	F	Force
	V	Velocity
	d	Distance
	pt	Momentum
Mechanical Rotation	T	Torque
	w	Angular Velocity
	Theta	Angular Displacement
	h	Angular Momentum
Electrical	v	Voltage
	i	Current
	qe	Charge
	Lambda	Flux Linkage
Hydraulic	P	Pressure
	Q	Volumetric Flow Rate
	Vh	Volume
	ph	Pressure Momentum

In the current case, the power domain for the coils is chosen to be "Electrical". Additionally, every instance of this Port should always be defined as an electrical Port. Selecting the "Fixed" check box that is next to the Power Domain options specifies this constraint.

As has just been mentioned, Power Ports have two variables associated with them: one functional input and one functional output. Which Port Variable is an input

and which Port Variable is an output is influenced by the *Input/ Output Constraint*. For the case of the coil, the effort variable, or voltage, is specified as preferred as an input. This constraint has to do with the preferred form of the equations that will be specified shortly. As a device which stores electrical power, a *State Variable* is associated with this port, which represents the flux linkage of the coils. A corresponding form for the slug Port is shown in Figure 23.

Default Port Properties [X]	
Label: Slug	Location (Degrees): 90
Direction: Out	Input/Output Constraint: <input checked="" type="checkbox"/> Fixed Preferred: Effort Out/Flow
Power Domain: <input checked="" type="checkbox"/> Fixed Mechanical Translation	State Variable: <input checked="" type="checkbox"/> Fixed q (displacement)
OK	Cancel

Figure 23. Solenoid Template: Specifying the Slug Port Properties.

3.3.1.5 Equations

The next set of forms deals with the functional aspects of the solenoid. In this example, a set of parameters and a set of equations are defined. Some planning is needed to set up a meaningful set of parameters and equations that will be useful and flexible for a user working with an instance of the Solenoid Template. The goals in defining the functional properties of the solenoid are to

- 1) provide a default set of parameters and equations that will describe the solenoid's behavior in a meaningful way;
- 2) Allow for the user to modify the default equations; and
- 3) Assist the user when editing the equations in ways that minimize errors.

Recall that the inductance value of the coil varies as the slug moves through it. This effect is due to the differences in the permeability of air and the slug material. A general form of the relationship between the coil's inductance vs. the position of the slug is shown in Figure 24. When x is zero, the slug is centered in the coil. As the slug moves out of the coil in either direction, the inductance decreases until it reaches its minimum value. This relationship can be represented by the parameterized equation given below.

$$L(x) = L_{\min} + \left(\frac{(L_o - L_{\min})}{1 + \left(\frac{x}{c}\right)^2} \right) \quad [1]$$

In Equation 1 L_o is the maximum inductance value of the coils, L_{\min} is the minimum inductance value of the coils, and c is a general measure of how quickly the inductance changes from L_o to L_{\min} . In practical terms, the parameters L_o , L_{\min} , and c can be chosen so that shape of Equation 1 matches the desired characteristic.

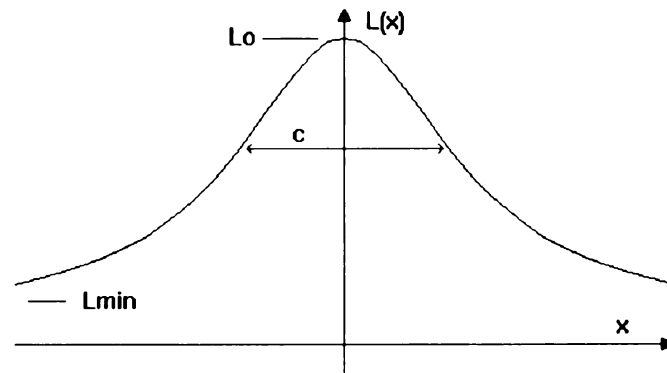


Figure 24. Inductance of Coil as a Function of Slug Position.

Two equations can be used to describe the behavior of the solenoid at its ports. The general form of these equations is given below.

$$i = i(\lambda, x) \quad [2]$$

$$F = F(\lambda, x) \quad [3]$$

For a linear inductance model Equation 2 becomes

$$i = \lambda / L(x) \quad [4]$$

For a more complete discussion of the behavior of a solenoid device, see Karnopp, et. al. (1991), pp. 284-289.

Using the information above, the Parameter form for the solenoid Template can be completed as shown in Figure 25. On the right the constraints that can be applied to the set of parameters is shown. If the "Fixed" option was checked, then the parameter equations would all be considered constants, i.e., the user could not edit them. This is not

true in the current case. On the left the parameters used in Equation 1, with default values, are specified.

Figure 25. Solenoid Template: Default Parameters.

Each parameter has a set of properties. The form for the parameter L_0 is shown in Figure 26. The parameter is defined and given a default value in the top line of the form. A Parameter can be specified using a constant value, like shown in the figure, or as an equation made up of constants and previously defined parameters. A list of the parameters that have been previously defined is listed here for quick reference. Note, however, that Parameters cannot make use of the time variable and must be explicitly computable; i.e., algebraic loops among Parameters is not allowed.

The Parameter Properties form allows for a definition to be associated with the Parameter that will be used to assist the user in specifying its value. Two types of

constraints can be applied to an individual parameter. If the parameter is to be treated as a constant, then it is constrained as "fixed". In the current case, there is a constraint that the parameter must always be defined within a specified range. Similar forms can be completed for the other two parameters.

Define Parameter Properties

Default Parameter:
Lo = 0.004

Description:
Inductance value when the slug is in completely in the coil.

☐ Fixed ☒ Value Limits

Min: 0.002 Max: 0.01

Available Variables:
c
Lmin

OK Cancel

Figure 26. Solenoid Template: Defining Default Parameter.

After the Parameter properties have been specified, a set of Equations can be defined. Figure 27 shows the form with the default equations specified for the solenoid Template. A list of constraints that may be associated with the Equations as a set is listed on the right-hand side of the screen. The meaning of these constraints is described below:

Fixed - None of the default equations can be edited by the user.

Algebraic - The equations can only be defined using algebraic operators. Specifically, the integral and derivative operators cannot be used.

No Use of Time Variable - None of the equations can make explicit use of the time variable.

Edit Component Template: Define Equations

Default Equations

```
lambdaCoil = Integral(vCoil,0)
dSlug = Integral(vSlug,0)
L = Lmin + (Lo-Lmin)/(1+(x/C)^2)
dL_dSlug = -2*(dSlug/c)*(Lo-Lmin)/(1+(dSlug/c)^2)^-2
iCoil = lambdaCoil/L
FSlug = -lambdaCoil^2*dL_dSlug/(2*L^2)
```

Constraints on Set

☐ Fixed

☐ Linear

☐ No Use of Time Variable

Add Edit Delete

< Back Next > Cancel

Figure 27. Solenoid Template: Default Equations.

On the left-hand side of the form are listed the default Equations that have been defined as part of the Solenoid Template. The first two Equations are automatically defined and cannot be edited or deleted at this point. This result is a consequence of choosing the coil and slug Ports with state variables as discussed in Section 3.3.1.4. The other four Equations are defined and modified using the "Add" and "Edit" buttons.

Each Equation has a set of properties. Figure 28 shows the properties for the default Equation that defines the Force at the slug Port. As was previously discussed, this

equation has a fixed form. Selecting two Constraints specifies this behavior. First, the "Edit Right-Hand Side Only" constraint is chosen. This constraint indicates that the FSlug variable must always appear on the left-hand side of the equation. The second constraint restricts this equation to only be composed of the variables listed. Similar forms can be completed for each of the other Equation shown in Figure 27.

The screenshot shows a dialog box titled "Define Equation Properties". It contains the following fields and options:

- Default Equation:** A text box containing the equation $F_{\text{Slug}} = -\lambda_{\text{Coil}}^2 dL_{\text{dSlug}} / (2 \cdot L^2)$.
- Description:** A text box containing "Default Force equation".
- Fixed:** An unchecked checkbox.
- Edit Right-Hand Side Only:** A checked checkbox.
- Restrict to Linear:** An unchecked checkbox.
- Restrict to Algebraic:** An unchecked checkbox.
- No Explicit Use of Time Variable:** An unchecked checkbox.
- Restrict Set of Available Variables:** A checked checkbox next to a list box containing the following variables: dL_{dSlug} , dF_{slug} , F_{Slug} , i_{Coil} , L , λ_{Coil} , and λ_{min} .
- Buttons:** "OK" and "Cancel" buttons at the bottom.

Figure 28. Solenoid Template: Slug Force Equation.

3.3.1.6 Properties Summary

After completing the above forms, the solenoid Template definition is complete. At this point a summary of the properties that were specified in the previous forms is given, as shown in Figure 29. After pressing the *Finish* button, the name of the new

Template will appear on the list of Components in the main MB window and be available for use. An instance of the Solenoid will appear as shown in Figure 30.

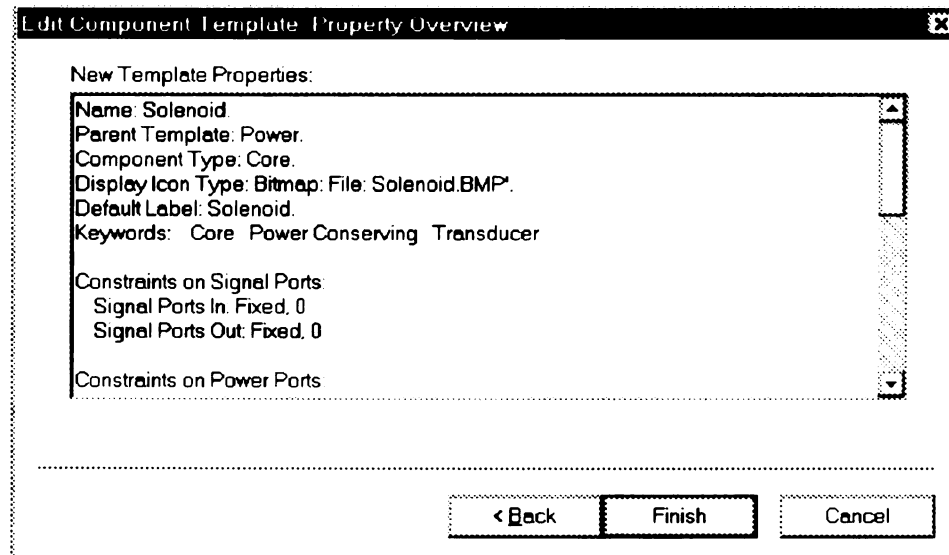


Figure 29. Solenoid Template: Summary Page.

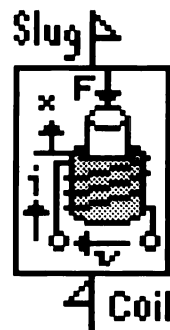


Figure 30. An Instance of the Solenoid Template Icon.

3.3.2 Deriving a New Template From an Existing Template

As can be noted by working through the previous example, there are many questions that need to be addressed when creating a new Template. Although using a set

of forms simplifies the process, an additional simplification can be achieved by exploiting the fact that the constraints specified in a Template's parent are inherited.

In recognizing this fact, it is useful to "seed" a Template library with a set of Template definitions that have selected properties constrained. For example, it can be useful to define a general "Signal" template that only adds the constraint that there should be no Power Ports. Once this is done, then the Signal Template can be used as a parent when generating a set of block diagram Templates. Then, when creating a new Template based on the Signal Template, the decision on how many Power Ports the new Template should have is already made.

The feature of inheriting constraints from a parent Template can also be useful when two similar templates are to be specified. As an example, consider a Template that defines a field-controlled DC Motor that allows for non-linear mechanical resistance. Such a Template might be defined with three Ports: an In Electrical Power Port, for the motor armature, an Out Rotational Power Port, for the motor shaft, and a Signal Port, for the field current. This information is represented in Figure 31. The default equations that might be associated with this Component Template, along with the constraints for editing them, are listed in Table 1.

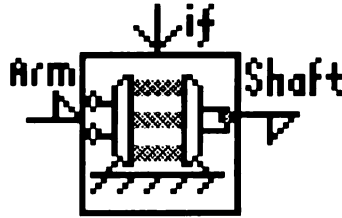


Figure 31. A Field-Controlled, Non-Linear, DC Motor Icon.

Table 2. A Set of Equations for a Field Controlled DC Motor.

<u>Equations</u>		<u>Constraints</u>
v_{Arm}	$= v_L + v_R + v_M$	Fixed
T_{Shaft}	$= T_M - T_J - T_R$	Fixed
\dot{i}_{Arm}	$= \text{der}(i_{Arm})$	Fixed
\dot{w}_{Shaft}	$= \text{der}(w_{Shaft})$	Fixed
K_m	$= k_m \cdot i_f$	Restricted Vars: K_m, k_m, i_f Edit RHS
v_L	$= L \cdot \dot{i}_{Arm}$	Fixed
v_R	$= R_e \cdot i_{Arm}$	Fixed
v_M	$= K_m \cdot w_{Shaft}$	Fixed
T_M	$= K_m \cdot i_{Arm}$	Fixed
T_J	$= J \cdot \dot{w}_{Shaft}$	Fixed
T_R	$= R_m \cdot w_{Shaft}$	Restricted Vars: T_R, R_m, w_{Shaft} Edit RHS

Defining such a Template takes care, planning, and time. The result is a new model type that can be instantiated with relatively little effort. The Template is also flexible in that it not only allows for specification of some model parameters, but it also allows for modification of the mechanical resistance equation while ensuring that this function has the correct form.

The initial Template design effort can be further exploited if a model of a Permanent-Magnet DC Motor with all linear equations is to be defined, as shown in Figure 32. In this case, the previous Template definition can be used as a Parent Template. This decision would greatly reduce the effort required to define a new Template. Only a few additional constraints need to be added, namely that the Signal Port is not needed, the K_m value is a constant, and the resistance equation is Fixed to a linear form.

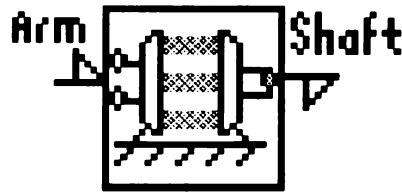


Figure 32. Permanent Magnet DC Motor Icon.

3.3.3 Editing Existing Templates

New model types based on a Template definition are intended to be a robust tool for assisting those who later create instances of the Template for a particular modeling purpose. The task of creating a Template that is precisely defined and has the desired characteristics can be an iterative process. A typical process would be to specify the properties of a new Template, create an instance of the Template, "test" if the behavior is as expected, and, where needed, modify the properties of the Template. This process may be completed many times until the desired Template design is achieved.

It is obvious that the modeling environment should provide tools for editing the properties of an existing Template. In MB, this task is easily accomplished by first selecting the Template that is to be modified. Then, selecting the *Edit* menu command "Edit Template" allows the user to review the set of forms that were filled out when the Template was initially completed, and to make changes where desired.

Although the benefits of editing an existing Template are obvious, there are some dangers in using this feature that could potentially cause problems and confusion. One issue involves the concept of inheriting constraints from a parent Template. If a Template that is being edited specifies a different Parent than originally chosen, then some of the current properties of the Template could possibly violate constraints set by the new Parent. For example, consider a Template that specifies one Signal In Port. Upon editing this Template, suppose a new parent was selected that specified zero Signal Ports. At this point, there would be a conflict between the current Template definition and its parent Template's constraints. A similar situation could arise if the Template being edited has been used as a Parent itself. Changes in a parent Template could create conflicts with existing children Templates.

To address the issue of the potential for conflict between the parent and Templates, the environment has been given two restrictions. First, a Template's parent can not be changed during the editing process. Second, a Template that has been used as a parent Template cannot be edited at all.

Another similar issue involves the relationship between a Template and any instances that have been defined. Adding a constraint to an existing Template when editing it could create a situation where a pre-existing instance has properties that violate constraints specified by the Template. At this point in time, a strategy for automatically dealing with this potential conflict has not been defined and warrants additional effort. In the mean time, knowledge of this issue can serve as a guide for the editing of Templates.

3.4 Library Browsing Tools

To address the problems of searching through a large list of Templates in a library three different organization schemes were developed: by Keyword, by Constraint, and by Generation History. These ideas were discussed previously in Section 2.3.4. In the next three sections the implementation of these ideas in the MB environment is discussed.

3.4.1 Keywords Filters

Associated with each Template defined in the MB environment is a list of keywords. In the main interface of the MB environment (see Figure 12) the set of all Components known to the system is displayed in a scrollable list, alphabetically ordered. To facilitate searching through this list, the MB environment provides a feature for "filtering" the list of Components that are displayed by a set of keywords. Beneath the list of Components is a selectable list of all the keywords that have been associated with all of the Templates defined in the system. Initially, none of the keywords are selected, which indicates that the entire list should be displayed. A set of one or more keywords

can be selected in this list to become the current "Filter". When an "Update" button is pressed the system examines the keywords associated with each Template and displays only those that match the set of keywords in the filter. An example of this feature is shown in Figure 33. The keyword filter is "Signal". All of the Templates that have this keyword associated with it are displayed in the list above.

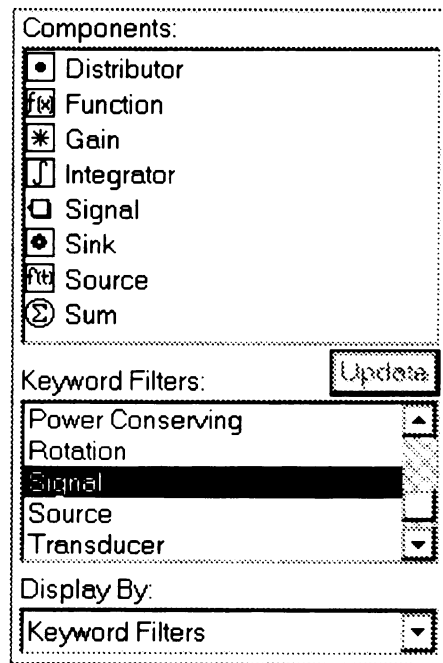


Figure 33. Filtering Templates by Keywords.

3.4.2 Constraint Filters

As was pointed out in the previous chapter, one limitation to using keywords as a searching device is that there is no control over what keyword gets associated with a Template. An additional search method proposed was to use Constraints instead of Keywords as the searching tool. This option was implemented in the MB environment in the following way. For each Constraint that is associated with a Template, a "Constraint

String" is generated. The Constraint String is a brief text description of the Constraint. This way, Constraint Strings can be used as a search tool in a fashion that is directly analogous to the way that Keywords are used.

One advantage to this idea is that the strings that represent the Constraints associated with a Template are not arbitrarily chosen; they are automatically assigned. This way there is always a one-to-one correlation between a Constraint String and the actual Constraint associated with it. Also, the Constraint String is always guaranteed to represent an actual characteristic of the Component.

Using Constraint Strings as a filter is illustrated in Figure 34. The Constraint string used was "No Power Ports". The result of this filtering produces the same list of Components was found with the previous Keyword filtering search. However, this time it is certain that every Component displayed can never have a Power Port. The same statement cannot be made in the previous case.

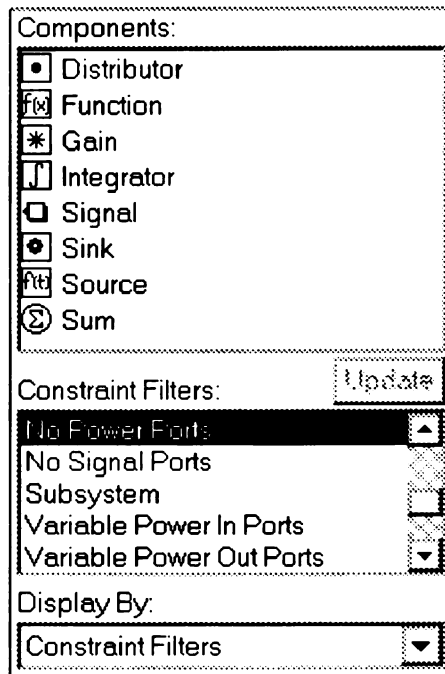


Figure 34. Filtering Templates by Constraints.

3.4.3 Generation History Display

During the Template creation process, a Parent Template must always be chosen. Although it is possible to derive each Template in the system directly from the General Multiport Template, as was previously discussed, it can be helpful to use a thoughtfully chosen Parent. This process naturally leads to a tree-based ordering of templates. This feature is exploited in the MB environment by allowing for the Templates to be displayed based on their parent-child relationships.

Figure 35 shows a particular parent-child ordering of a set of Templates. Templates that have been used as parent Templates but do not currently have their children displayed on the screen have a box with a plus sign next to them. Double

clicking on the plus sign causes the list of the Template's children to be displayed and for the plus sign to be changed to a minus sign.

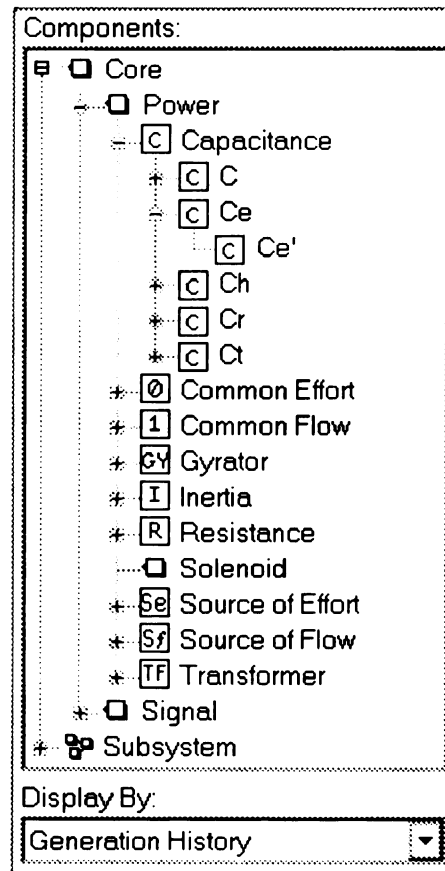


Figure 35. Template Generation History Display.

CHAPTER 4

CONTROLLED ACCESS

4.1 Introduction

In the previous two chapters, issues related to design of a modeling environment that supports mechatronic systems modeling and methods for organizing and searching the contents of a model library were addressed. This chapter discusses the third major topic addressed by this research, the issue of controlling the access to internal model details. Such considerations arise when two companies must share model information to accomplish a system design. Some model information may be proprietary to one of the companies. This situation is becoming more common as larger corporations outsource component designs. This situation may also be of concern in an educational setting where a course instructor, a teaching assistant, and students all use the information in a model file. However, persons in each of these groups should not necessarily have the same ability to view and modify the contents of the model.

A simple solution commonly used to protect information is to prevent access to the entire contents of a component model. In this case, use of such a model is greatly restricted, with access often limited to specifying parameter values and port connections. Such an approach, while protecting the interests of the model provider, can be frustrating to practicing engineers who use the model. Models may be difficult to use correctly due to a lack of understanding of model details that are inaccessible. If some changes become necessary due to design changes, the model provider must be employed to change the model.

In this chapter a concept, design, and implementation for improving the way models are shared is presented. The goal is to provide a facility that permits the owner of a model to control access to various model details, according to the user classification. Such a feature allows for varying levels of security regarding both reading model details and modifying them. The ideas presented here combine the General Multiport modeling data structure presented in Section 2.2, the need for sharing information among different groups, and the desire to control access to information. While these concepts have been exploited in different venues, they have not been combined for effective use in a mechatronic systems modeling environment.

This chapter is organized as follows. Section 4.2 presents design properties of a modeling environment that supports the control of access to various model properties. An implementation based on this design is discussed in Section 4.3. The usefulness of the design is illustrated by the presentation of two examples in Section 4.4.

4.2 Design of a Controlled Access Environment

The MB modeling environment that was presented in the previous chapter serves as a base implementation environment for tools to support the control of access to properties of a model. In addition to the data structures already introduced, one additional data structure must also be defined: a *Model File*. When a model is built, a user defines the properties of a set of General Multiports and connects their Ports. The set of connected Multiports is stored in a Model File. A Model File is generally stored on

a permanent medium, such as a hard drive or disk. In the current discussion on sharing of model libraries, the Model File is considered the fundamental shared object. Sharing is achieved by copying a Model File using standard operating system facilities.

4.2.1 Access-Controllable Attributes

As was previously observed, it can become useful to prevent access to the details of a model for various reasons. A common method for accomplishing this objective is to control the access to a model by restricting access to a Model File. Tools provided with some operating systems, such as Windows NT or Unix, can be employed for this purpose. Under these circumstances there are three general possibilities for controlling access. In the first case, there are no restrictions for opening and modifying the contents of a model file. In the second case, the contents can be examined or "Read", but modifications are not allowed. In the third case, the contents can not be accessed at all.

While this feature can be useful in controlling how and when a model is read and modified, the application of access settings are limited to a Model File as a whole. A more sophisticated approach would allow for individually controlling the access to various aspects of a model. Instead of simply "locking up" an entire model, only critical parts of the model are protected. More flexibility is provided and greater utility can be extracted from a model.

Once the concept of controlling the access to individual model attributes is accepted, it then becomes necessary to identify various model attributes for which it may

be beneficial to establish access control. Such model attributes are referred to as *access controllable*. In this design three access-controllable objects were identified in order to demonstrate the usefulness of the ideas. Table 3 lists these objects and their possible access values. For each access-controllable object, exactly one of the three access setting values is applicable at any given time.

Table 3. Three Access-Controllable Objects and Their Possible Access Values.

Object	Access Values		
Core Equations	None	Read	Read/Modify
Subsystem Contents	None	Open	Open/Modify
Model File	None	Load	Load/Save

For Core Equations, an access value of "None" indicates that a Core Component's Equations can neither be read nor modified. "Read" indicates that equations can be studied, but not modified. "Read/Modify" indicates that there are no access restrictions on how equations can be modified.

For Subsystem Contents, an access value of "None" indicates that the Contents of a Subsystem Component cannot be viewed or modified. "Open" indicates that a Subsystem can be opened and its contents examined, but modifications, such as adding and deleting Components are not allowed. "Open/Modify" indicates that there are no restrictions on examining or modifying the contents of a Subsystem.

For a Model File, an access value of "None" indicates that the contents of a Model File will not be loaded when the file is opened. "Load" indicates that the file contents can be loaded, but changes cannot be saved. "Load/Save" indicates that there are no restrictions on loading or saving the contents of a Model File.

4.2.2 Users and Groups

To give further flexibility to controlling access to the properties of a model, additional data structures are used which are patterned after the UNIX file access control (Stallings, 1998). First, a *Current User* is defined. The Current User is a system parameter used to identify the person currently working in the modeling environment. The value of the Current User Variable can be changed at any time, which is equivalent to "logging off" and "logging on".

Next, as shown in Figure 36, each Model File contains a set of *Groups*. In this initial design four groups are defined: Owner, Group 1, Group 2, and World. Although the number of groups is currently fixed, in principle the number of groups is variable. Associated with each of the first three groups is a unique list of *Users*; i.e., each User is a member of exactly one group.

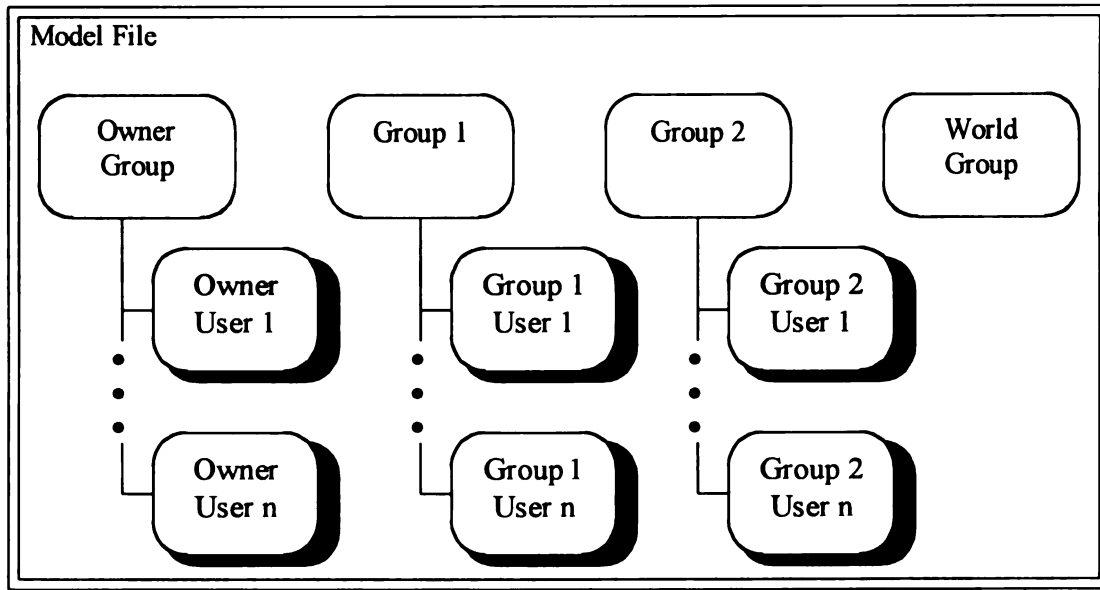


Figure 36. Organization of Groups and Users.

Using this classification scheme, the Current User is always identified as belonging to exactly one Group. If the Current User is among the Users on the Owner list, then the Current User is considered an Owner. If the Current User is among the Users on the Group 1 list, then the Current User is consider a Group 1 User. If the Current User is not contained on any list, then the Current User is considered a World User.

The functional purpose of the data structure above is to classify the Current User as a member of one of the Groups. When a new Model File is created, the Current User is automatically added to the list of Users of the Owner Group. The lists associated with the other Groups are initially empty. When an existing Model File is opened, then the Current User is identified as an Owner, Group 1, Group 2, or World User.

4.2.3 Group Access Settings

Each access-controllable object maintains an independent set of access values for the Group 1, Group 2, and World Groups. For example, a Core Component can specify that Group 1 Users can read and modify its Equations, that Group 2 Users can only read its equations, and that Word Users can neither read nor modify its equations.

Users who are classified as an Owner always have complete access to all data in a model. In addition, Owners can add and delete Users from any of the Groups. The sole exception is that the Current User cannot be removed from the Owner List. (If this option were available, then it would be possible for a Model to be permanently "locked up".)

Part of the design philosophy recognizes that a majority of models built in this modeling environment will not need to define any type of access control. Therefore any implementation should incorporate the following principles:

- Setting Controlled Access values is optional.
- Controlled Access features are exposed only if it is desired to set some control values, or if control values have been previously set and are being enforced.
- The default Controlled Access Values are such that there are no restrictions on any user.

4.3 An Implementation of Controlled Access

4.3.1 Current User And Group Members

The access control tools described in Section 4.2 are dependent on the value of the current user. This value is a system parameter that can be set at any time. Initially, the current user is automatically defined as "Anonymous". If controlling the access to model properties is not an issue, then this value need never be changed. However, if the access control tools are desired, the first step should be to specify a new Current User. This task is accomplished by selecting the *Specify Current User* command from the Edit Menu. Selecting this command produces the dialog shown in Figure 37.

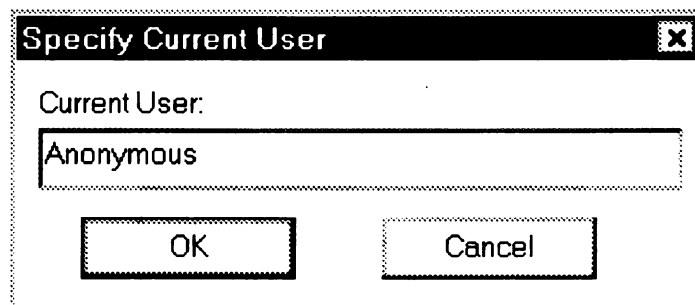


Figure 37. Specifying the Current User.

Recall that each Model File maintains its own lists of known users and classifies them in one of the four Groups: Owner, Group 1, Group 2, and World. The Current User is always classified as belonging to exactly one of these groups. If the Current User is a member of the Owner Group, then that user has the ability to add and remove Users from the first three groups. A list of Users is not maintained for the World Group. That group

is used as a default classification for a User who is not a member of any of the other three groups.

Figure 38 shows the dialog used for specifying the members of the Owner, Group 1, and Group 2 Groups. This dialog can only be accessed if the Current User is a member of the Owner Group. The Group for which the Users are to be specified is selected from the list on the top of the dialog. Figure 38 indicates that the list of Users for Group 1 is currently being specified. The list of Users of the currently selected group appears in the bottom half of the display. Figure 38 also indicates that three Users currently belong to Group 1. An Owner can freely edit the list of Users of any Group, with one exception: the Current User cannot be removed from the Owner list.

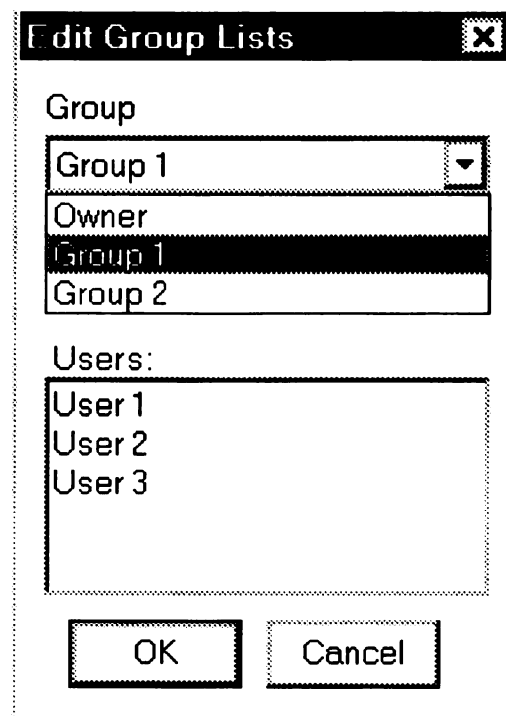


Figure 38. Interface for Editing Group Members.

4.3.2 Specifying Group Access Settings

At any time during the process of building a model, an Owner User can set the access values for any existing access-controllable object. Figure 39 shows the dialog for defining access settings for a Core Component's Equations. Right-mouse clicking on an existing Core Component accesses this dialog. Right clicking on an object in the MB environment causes a list of menu options that are specific to that object to appear in a "context menu". If the Current User is an Owner, then the context menu command "*Access Settings...*" can be selected.

On the left-hand side of the dialog the Group for which Access Values are to be set is selected. On the right-hand side of the dialog the access values for the currently selected Group are specified. The current options shown in Figure 39 indicate that any user belonging to Group 1 only has the ability to view the Equations associated with the currently selected Core Component. Similar dialogs are available for specifying the access settings for a Subsystem Component and a Model File.

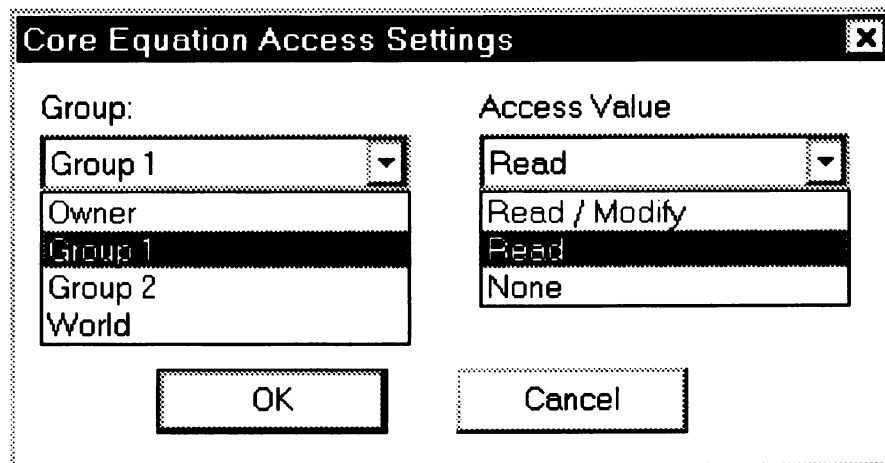


Figure 39. Specifying Access Settings.

4.3.3 Enforcement of Controlled Access

When a model has been created with various access-control settings, enforcement is then left to the MB environment. Enforcement is accomplished by determining the access value for the Current User to each of the Access-Controllable objects, as described below.

Core Component's Equations

- **None** - Upon attempting to access the equations, a Dialog similar to the one shown in Figure 40 appears. This dialog is the interface used for editing Equations. However, notice that instead of displaying the list of Equations associated with the Component, a set of asterisks appears. Also, the buttons on the right-hand side of the screen which normally support the adding, editing, and deleting of equations are disabled.

- **Read** - The actual Equations are displayed in the dialog of Figure 40. The Add, Edit, and Delete buttons are disabled.
- **Read/Modify** - The actual Equations are displayed in the dialog of Figure 40 and the Add, Edit, and Delete buttons are Enabled.

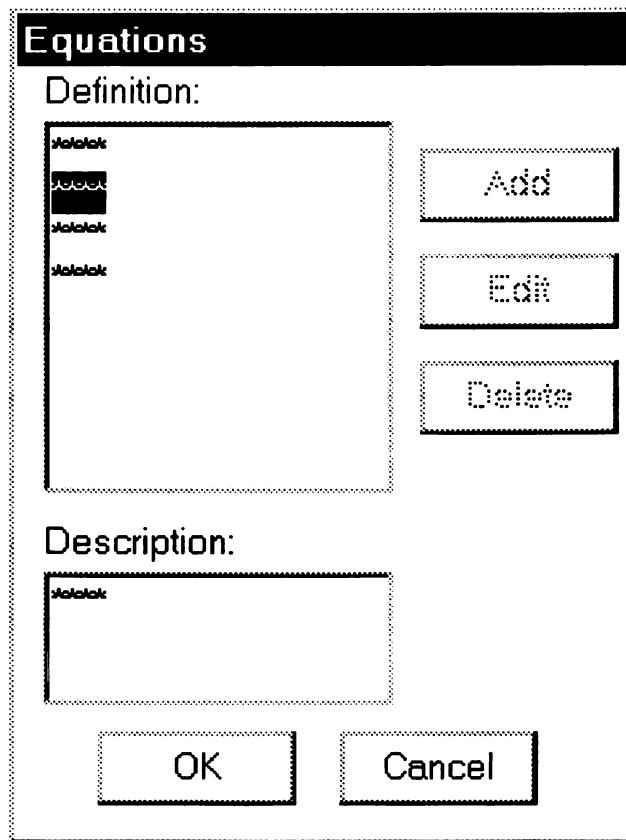


Figure 40. Restricted Access to Equations.

Subsystem Component Contents

- **None** - When an attempt is made to open a Subsystem, a message is displayed indicating that the Current user does not have permission to open the subsystem.

- **Open** - The contents of the Subsystem can be examined, but substantive modifications are not allowed. Some types of restricted modifications are editing Equations and Adding and Deleting Components, Ports or Connectors.
- **Open/Modify** - The system behaves as normal, with no restrictions on modifying the Subsystem's contents.

Model Files

- **None** - When a Model File is opened and it is discovered that the Current User doesn't have the proper access permissions, then a message is displayed indicating this fact. The contents of the file are not loaded.
- **Load** - There are no restrictions on modifying the contents of the Model file, but it is not possible to save any changes that were made. Upon attempting to save a Model File, a message is displayed indicating this fact.
- **Load/Save** - The system behaves as normal, with no restrictions on loading or saving Model Files.

4.4 Two Illustrative Examples

The utility of the features described in the previous section can be illustrated by considering two scenarios where the use of the tools would be of benefit. First, consider a company that has been commissioned to design a model of a mechatronic system. The model is to be placed in a feedback control system. The commissioning company desires to use the model and test various control strategies using various inputs. Under these

conditions, the commissioned company might develop the model shown in Figure 41. In this figure, the Duty_Cycle and Sum Junction are Core Components and the Control, Plant and Measurement are Subsystem Components.

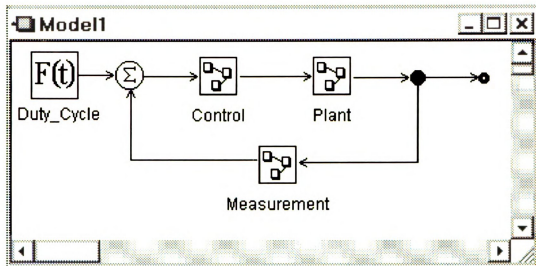


Figure 41. A Feedback Control System

Consider four groups of individuals that may have access to this model file and the way they would use the modeling environment. First is the company that is commissioned to create the model. When the person who initiates the project creates a new Model file, that person's user name is automatically added to the Owner list. Then, if the model development was a team project, that person would add the user names of the other people on the team to the Owner group. After a working model was completed, it would be the responsibility of one of the Owners to decide which parts of the model should be accessible members of Group 1, Group 2, and World.

Suppose that the development team decides that an in-house team should be formed to validate the model before sending it off to the customer. They further decide that they don't want any changes to be made to the plant model by the development team, but want them to be able to examine it. In this case, the user names of the members of the validation team would be placed on the Group 1 list and the access value for the plant Subsystem set to "Open". When the validation team uses the model, they find that they can look at the contents of the plant model, but are not allowed to modify any significant properties.

The development team then decides to specify one user name for their customer and puts that name on the Group 2 list and sets access to the plant Subsystem to "None", allowing the customer the ability to use the plant model, but not to have access to its details. Finally, the development team decides that anyone else who might gain access to the model should have no access to the file at all. The World group access value is therefore set to "none".

As a second scenario, consider an educational environment where the modeling environment is to be exploited as a "virtual laboratory", in a way similar to the scenario described by Rosenberg (1991). In this situation students are given a description of a mechatronic system and asked to create a model of the system. To make this learning experience as valuable to the students as possible, it would be particularly useful if the students were asked to "validate" the models they generate by comparing their numerical

results with experimental results. Unfortunately, as is often the case, the lack of resources often make this goal unattainable.

However, using the tools of controlled access, a professor, classified as the model owner, creates a "true" model of the system that captures the response of the system with the desired level of fidelity. The instructor adds a User Name for the teaching assistant to the course to the Group 1 list. The access values for Group 1 are set such that the TA has complete access to view the model contents, but no access to change model properties. The list of students in the class is then added to the list of member of Group 2. The access permission settings for Group 2 would then be specified to give the students appropriate access to the model details, as determined by the instructor. Students would then be able to construct their models, design "experiments" on the instructor's model, and "validate" their results by comparing them to response of the instructor's model. Finally, the instructor decides to post the model file on the course web page for easy student access. Not wanting to discourage curiosity by others not participating in the class, but not wanting to give free access to the material meant for students, the instructor sets the same access values for World as for Group 2. In addition, the Model File access value for the World members is set to "Load" so that member of the World group cannot run experiments based on different model settings.

CHAPTER 5 CONCLUSIONS

5.1 Summary of Contributions

The completed research described in this document represents a unique and significant contribution to the area of mechatronic systems modeling. The principal objective of designing data structures, formulating new concepts and organizing existing information for the use in mechatronic systems modeling has been accomplished. Effective use of the ideas formulated in this document help to simplify the modeling process, decrease the effort required in generating new mechatronic modeling components and systems, and make it possible to control access to a finer level of model details. These benefits support both industrial needs, where design cycle times are a critical factor, and academic needs, where simplified designs allow students to focus on relevant modeling issues, not on implementation details.

A specific discussion of the major contributions of this work is presented in the next four sections. Section 5.1.1 addresses contributions that were made in relation to Template design. Library tools that were designed and implemented are listed in Section 5.1.2. An implementation environment that was created as part of this work is covered in Section 5.1.3. Section 5.1.4 presents the concepts and organization of ideas concerning controlled access to a mechatronic system's properties.

5.1.1 Template Design

In CHAPTER 2 the importance of a relatively new modeling paradigm, the specification of User-Defined Model Types (UDMTs), was discussed. Currently there are several modeling environments that support this feature. As part of this work a new data structure, called a Template, was defined. A Template describes the properties of a UDMT. The Template design presented here represents a new way of organizing information about mechatronic system models. The organization provides a different perspective for thinking about the way in which UDMT are defined and their role in creating models of mechatronic systems. The Template definition includes a modeling structure, called the General Multiport, a set of constraints that prescribe desired behavior for instances of UDMTs, and a set of default properties. The explicit use of constraints to prescribe the way in which instances of UDMTs can be modified and classified is a new and unique.

A definition of the General Multiport modeling structure was also part of this work. The General Multiport is specifically designed to support models of mechatronic systems and components, hierarchical design, and model reuse. The properties of a General Multiport describe the range of properties of possible models that can be built. The fundamental definition presented is an extension of previous work. The definition is sufficiently general to allow for definition of a large class of practical engineering models.

A Template defines a unique way for organizing the properties of a UDMT and provides ways to specify functionality in a modeling environment that was not previously possible. Specifically, the Template definition makes it possible to

- (1) bound the allowable configuration of a model's properties,
- (2) derive new Templates based on existing Templates, and
- (3) structure and classify a set of Templates in useful and meaningful ways.

The usefulness of these features was demonstrated by considering possible ways that they could be exploited in a modeling environment for mechatronic systems. The first feature makes it possible to ensure that the way model instances are used is consistent with the Template definition. The second feature provides a means to simplify greatly the process of defining a new Template and gives an improved method for providing a natural structure to a set of Templates in a library. The third feature aids in the process of browsing the contents of a model library with a large number of model types. This benefit is discussed in the next section.

5.1.2 Library Tools

Tools for browsing the contents of a library of model types were designed and demonstrated in this work. The Template data structure made this design possible. The way in which these tools are designed and operate has not been previously exploited. Three methods were presented for searching the contents of a library of modeling templates.

- *By Keyword* - One of the attributes of a General Multiport definition is a set of keywords. MB provides a mechanism for searching through a list of Templates and finding ones with matching keywords.
- *By Constraint* - The constraints that are specified in a Template help to define the way in which a model instance behaves. MB provides tools for searching through a list of Templates according to constraints. This tool can be particularly useful since, unlike keywords, the constraint description is guaranteed to relate directly to the actual Template definition.
- *By Generation History* - A new Template is created by using an existing Template as a parent. The new Template inherits the constraints specified by the parent. This parent-child relationship presents a natural a tree-structured ordering of a set of Templates, and it can be useful when searching the contents of a Template Library.

5.1.3 Environment Implementation

The efficacy of the ideas expressed regarding Template design and Library Tools were demonstrated by implementing a mechatronic system modeling environment, called the Model Builder (MB), that is based on Template concepts. In this environment the only pre-defined modeling type is the General Multiport, from which a set of User-Defined Modeling Types can be generated. This feature makes it possible for the environment to be flexible and customizable, suiting the specific needs of the user. It also provides the opportunity to exploit the environment as a teaching tool. Requiring students to thoughtfully create a new model type for a specific design would expose them to a broader range of modeling experiences.

Another benefit of this implementation is that it serves as a framework for future research efforts. The investment that is required to create a user-friendly, graphically driven, general-purpose environment for mechatronic systems modeling is significant. Much of this work does not involve research-specific issues and can be time consuming and tedious. Fortunately, now that a fundamental environment exists, it can be exploited and extended in new areas related to mechatronic systems modeling.

5.1.4 Controlled Access

A need for more sophisticated tools for limiting the ways in which models of mechatronic systems are viewed and modified was identified. In response to this need the concept of controlling the access to various model properties was proposed. This design represents a unique confluence of three areas.

- *Data Structures of Mechatronic System Models* - The attributes of the General Multiport were used as a basis for specifying the various access properties.
- *Information Sharing* - The need for sharing files between individuals jointly working on a modeling effort and the trend to make models available in a public setting motivated the concept of information sharing.
- *Control of Access to Data* - Well-understood concepts associated with protecting data files stored on an operating system were used as the basis for controlling access to data.

The ideas presented make it possible for a person who accesses a model to be classified into one of several pre-defined groups. According to the specification of the model owner, each group can have a unique combination of access permissions to various aspects of a model. Permissions range from unfettered access, to read-only access, to no access.

The concept of controlled access was demonstrated by implementing a design of these features in the MB modeling environment. Controlled access was implemented after MB was completed according to the goals of Template design. The implementation effort of controlled access was relatively simple. This result supports the assertion that the MB environment can serve as a framework for future research.

5.2 Areas for Future Research

This section lists several areas that could be further investigated. The suggestions are divided into four groups. Section 5.2.1 presents ideas for tools related to the design of a Template. Library related suggestions are listed in 5.2.2. Issues relevant to the implementation environment are covered in Section 5.2.3. Topics regarding controlled access are given in 5.2.4.

5.2.1 Template Design

- 1) Enhanced General Multiport definition - The design of the General Multiport presented here is useful for modeling a large class of physical systems. However,

there are other modeling structures that have been explored by others that should be incorporated in this design. Some of these areas are discussed below.

- a) Discrete variables - Many mechatronic systems designs are *hybrid* in nature, in the sense that they combine continuous and discrete variables (Elmqvist, et. al, 1993). A system that uses a digital controller is a typical example. The General Multiport definition should support hybrid systems.
 - b) Discrete events - A related topic to discrete variables is discrete events, such as the opening or closing of an electrical switch. Methods for handling discrete events have been proposed, (Karnopp, 1988; Lorenz, 1993), but further research in this general area is needed. However, ways to incorporate discrete events in the General Multiport definition should be explored.
 - c) Vector Ports - It can be helpful to represent a group of Ports as a data single structure. Many issues regarding this concept have been previously explored (Breedveld, 1985). This feature can greatly simplify both the way a model is displayed and the way in which the model is conceptualized.
- 2) Improved parent-child relationships for Templates - In the current implementation environment, when a child Template is derived from a parent, the inherited properties are passed on by essentially making a copy of the inherited attributes and passing them on to the child. This scheme leads to a duplication of data. A more sophisticated implementation would exploit the relationship information as a mechanism of inheritance; e.g., a child Template could refer to its parent to determine some of its properties.

- 3) **Additional constraint definitions** - In the current implementation, a set of constraints was defined that prescribe properties such as the number of Ports, the direction of the Ports, the form of the equations as a set, and the form of individual equations. This set of constraints can be used to define a set of model types. However, it would be beneficial to investigate additional way in which to constrain the properties of a General Multiport, giving rise to more specialized behavior.
- 4) **Additional types of Templates** - The Template ideas expressed in this dissertation were only discussed in relation to a General Multiport. However, the notion of a Template description for any modeling object is equally applicable. For example, it is possible to define Port Templates and Equation Templates. In fact, this idea was investigated and used in the MB modeling environment to some extent; a partial definition of a Port Template was defined and exploited. However, a more thorough investigation is required.
- 5) **Educational benefits** - The concept of using constraints to prescribe a desired behavior of a very general modeling construct is unique. However, it is a useful way to organize one's thinking about modeling in general. The potential benefits of presenting Template ideas in a learning environment should be more carefully explored.

5.2.2 Library Tools

- 1) **Enhanced logical combination of filters** - In the current implementation, a set of filters is chosen. For a Component in the library to match the filter, it must have all of the keywords listed in the filter. This result can be thought of the logical "AND"

combination of the items on the filter. Allowing for different logical combinations of the filter items would make the search tools more flexible. For example, it would be useful to be able to specify a search for any Component that matches the keyword "Two Ports" OR "Transducer".

- 2) Keyword management - Currently, when a new template is created, a set of keywords is associated with it. The keywords are used later as a searching tool to assist a user in finding a useful model. However, the searching benefit can be degraded if the keywords used to describe a given model are not carefully chosen. For example, suppose two modeling components that were defined were energy conserving. If one model was given the keyword "Conservative" and the other was given the keyword "Energy Conserving", searching efforts could be hampered.

One way to deal with this issue is to have a database of keywords. When a user must decide upon the set of keywords to associate with a model, the keywords must be chosen from the database. If this scheme is used, then there must also be tools for managing the contents of the keyword database. These ideas require additional thinking and effort and would be of great value in a modeling environment.

- 3) Automated tools for sharing templates - In the current implementation of the MB environment, automated modeling tools have been defined for creating and editing Templates. However, after a Template is created in one environment, there are no automated tools for exporting the template for use in another system. This area should be investigated to support this facility.

- 4) Browsing model instances - The library browsing tools discussed in this document apply to a library of Templates. A useful feature would extend this ability to browse the contents of model instances.

5.2.3 Implementation Environment

- 1) Improved visualization tools - Additional insight into a system model can be gained by using various visualization techniques, such as those discussed by Ermer (1994). Some possible extensions in the MB environment include 3-dimensional model representations and power flow animations of simulation results.
- 2) Interface with other environments - The MB modeling environment provides tools for a subset of Computer Aided Engineering. It would be useful to support dynamic interfacing with other major types of CAE tools, such as finite element modeling tools. In this way the strengths of both environments can be exploited.
- 3) Compiled simulation code - In the current implementation, the equations of the system are ported to MATLAB as an "M-File" (Mathworks,1999). While this approach is functional for smaller problems, it can increase simulation times greatly when models get larger.

5.2.4 Controlled Access

- 1) Provide for a finer level of access control - In this dissertation, the principle of controlled access that has been previously applied to files as a whole was extended to various features of a multiport model. This extension gives a greater flexibility in

specifying how an instance of model should accessed. Applying access control concepts to even finer levels of detail of a model can extend this basic concept even further. For example, instead of controlling access to an entire set of equations associated with a Component, it might be useful to restrict access to a subset.

- 2) Controlled Access to Templates - The ideas of controlled access were applied instances of models. The benefits of controlled access could also be exploited by controlling the access to Templates.

APPENDIX

APPENDIX

IMPLEMENTATION DETAILS

The Model Builder Environment that was developed as part of this research effort consists of over 55,000 lines of code, expressed in over 100 files and nearly 95 classes. Due to the scope of this project, it was not deemed useful to present a verbatim listing of the code text. Instead, this appendix presents an overview of the implementation and directions for understanding the code structure. The information presented here, intended as a guide to assist future development, is given in two major categories. First, the organization of the major C++ classes that define the MB environment are presented. The class diagrams follow the Object Modeling Technique (OMT) described by Rumbaugh et. al. (1991). Second, the name and general purpose of the major text files that are used to create the MB environment are given.

Class Structure

The Object Modeling Technique was developed in an attempt to create a standard for communicating an abstract description of code structure based on object-oriented programming paradigms. These ideas are not related to any particular object-oriented programming language. However, since MB was developed in C++, the OMT notation will be discussed in terms specific to this language.

Using the OMT, squares represent a C++ class. Lines connecting two classes indicate a relationship between the classes. The precise nature of each relationship

depends on the classes involved in the relationship. An Annotation at one end of a relationship line indicates the role that the class plays in the relationship. A circle at one end of the relationship indicates that multiple class instances may participate in the relationship. The relationships are generally implemented using object pointers. A triangle indicates a parent-child relationship exists between two classes. For a more complete description of the notation used, consult Rumbaugh, et. al. (1991).

An Object Diagram, representing relationships between the major classes in the Model Builder environment, is shown in Figure 42¹. A general class for describing objects that are stored in a model or a *graph* is the CGraphObject class. This class is an abstract class that defines data common to objects that might be contained in a model. There are no direct instantiations of this class. Four classes are derived from the CGraphObject class, CComponent, CPort, CConnector, and CLabel. The CComponent class is further derived into the CCompMacro class, for Macro Components, and the CCompAtom class for Atom Components. The CCompMacro class stores a list of pointers to other CGraphObjects. Each CCompMacro may display its objects in a window created from the CMBView class. The CComponent class is associated with a CTmpClass, which store Template information for a Component. A parallel structure exists for Ports, although the Port Template ideas are not fully exploited at this point.

¹ The names of two model properties changed from the time the code was originally generated to the time this document was prepared. In the code Subgraph Components are called *Macro* Components, and Core Components are called *Atom* Components. The original names are used in this section.

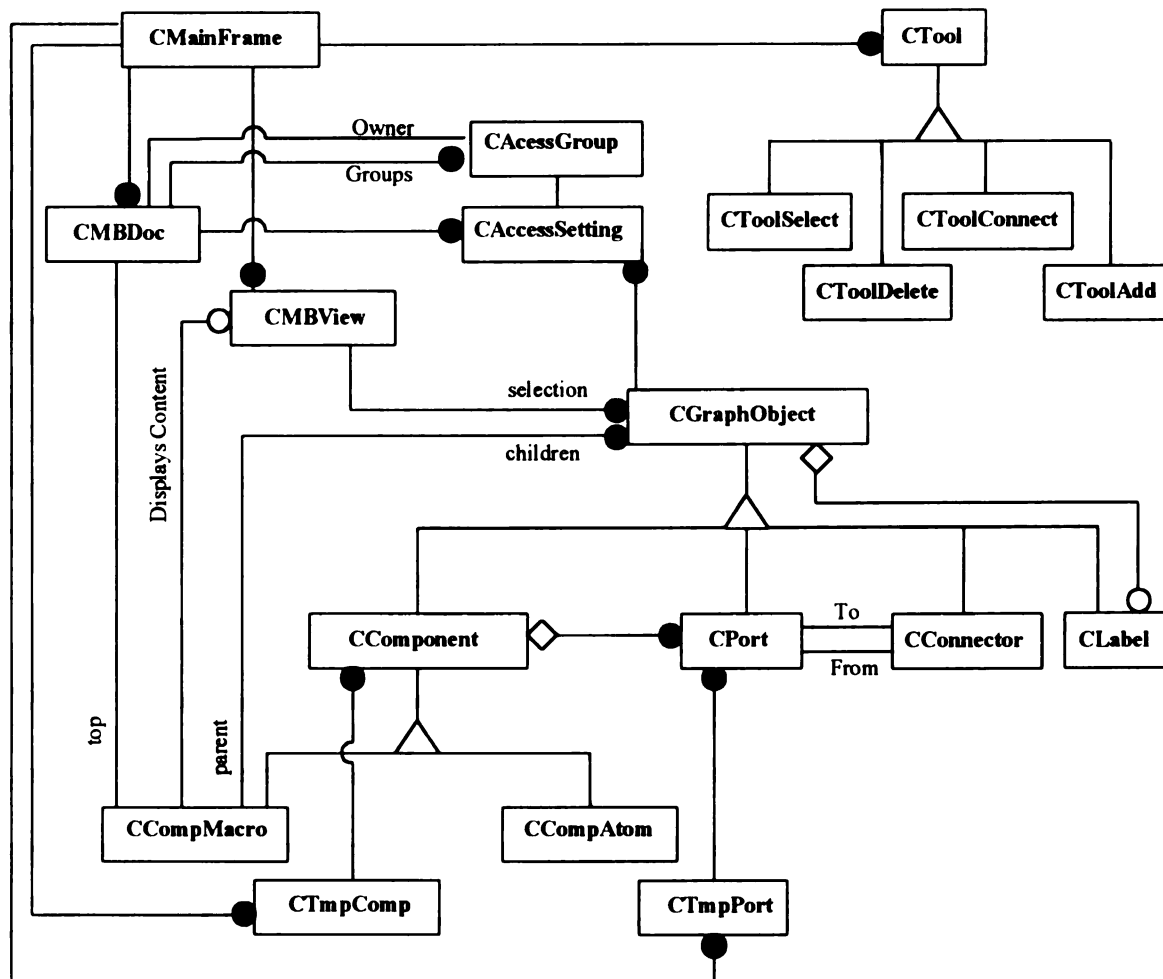


Figure 42. The MB Object Diagram.

The CMainFrame class is used to store application specific data, i.e., data that does not depend on the current model. MB has a Multiple Document Interface (MDI), meaning that more than one model file can be opened at one time. This structure is supported by having multiple CMBDoc classes related to the CMainFrame class. Each CMBDoc stores and manages data specific to a model file. The most important information stored in the CMBDoc class is a pointer to the "top" CCompMacro. Each

CMBDoc class stores exactly one CCompMacro. The CMainFrame class also stores a list of CTmpComp and CTmpPort classes.

A set of tools for operating on the contents of a model are associated with the CMainFrame class: the abstract class CTool and its derived classes, CToolSelect, CToolDelete, CToolConnect, and CToolAdd. These classes group tasks for carrying out actions to manipulate the contents of a model or graph.

The tools to support Controlled Access are contained in the two classes CAccessGroup and CAccessSetting. Each instance of the CMBDoc class stores a set of pointers to CAccessGroup classes. The CAccessGroup class stores a list of strings that represent the users that belong to it. The CAccessSetting stores an access value and a pointer to a CAccessGroup. This information is used to define restrictions to accessing the attributes of various other classes. CAccessSetting classes are associated with the CMBDoc class, to control access to model files, and to the CGraphObject class. Classes derived from CGraphObject must implement their own functions to support Controlled Access. The CCompMacro class implements Controlled Access for its contents and the CCompAtom class implements Controlled Access for its Equations.

File Structure

The above class structure was realized in a set of C++ class definitions, contained in text files. The main files used in the MB environment and a brief description of their contents is given in this section.

Table 4. Major Files Used for Creating the MB Environment.

File	Contents
MBClasses.h	Headers for most of the classes used in MB.
Components.cpp	CGraphObject, CComponent, CCompAtom, and CCompMacro class definitions.
GraphObjects.cpp	CPort, CConnector, and CLabel class definitions.
MainFrame.cpp	CMainFrame class definition.
MBDoc.cpp	CMBDoc class definition.
MBView.cpp	CMBView class definition.
Templates.cpp	CTmpComp and CTmpPort class definitions.
EditTools.cpp	CTool, CToolSelect, CTool Connect, and CToolAdd class definitions.
Equation.cpp	Class definitions for defining and manipulating equations.
Constraint.cpp	Class definitions used to specify and evaluate constraints.
Access.cpp	CAccessGroup and CAccessSetting class definitions.
PSTmpComp.h	Headers for the classes used in the Template creating/editing wizard.
Common.cpp	A set of general purpose, global functions.

REFERENCES

REFERENCES

- ANSI, 1966, *American National Standard Programming Language FORTRAN*, ANSI X3.9-1966.
- Auslander, D.M., 1996, "What is Mechatronics?", *IEEE Transactions on Mechatronics*, Vol.1, No.1, pp.1-5.
- Boeing, 1998, *Easy5 User Guide*, Boeing Computer Services, Seattle, WA.
- Breedveld, P.C., 1985, "Multibond graph elements in physical systems theory", *Journal of the Franklin Institute*, 319(1/2), pp.1-36.
- Bruenese, A.P.J., J.L. Top, J.F. Broenink, and J.M. Akkermans, 1998, "Libraries of Reusable Models: Theory and Application", *Simulation*, Vol.71, No.7, pp.7-22.
- Bruenese, A.P.J. and Broenink, J.F., 1997, "Modeling Mechatronic Systems Using the SIDOPS+ Language", *International Conference on Bond Graph Modeling and Simulation*, Vol.29, No.1, pp. 310-306.
- Bruenese, A.P.J., 1996, *Automated Support in Mechatronic Systems Modeling*, Ph.D. Thesis, University of Twente.
- Buur, J., 1992, "Does Mechatronics need a special design attitude?", *Mechatronic Systems Engineering*, Vol.1, No.4, pp.293-300.
- Byam, Brooks, "Modular Modeling of Engineering Systems Using Fixed Input-Output Structure", Ph.D. Dissertation, Department of Mechanical Engineering, Michigan State University.
- Cellier, F.E. and Elmqvist, H., 1993, "Automated Formula Manipulation Supports Object-Oriented Continuous System Modeling", *IEEE Control Systems*, Vol.13, No.2, pp.28-38.

- Cellier, F.E., 1992a, "Bond Graphs: The Right Choice for Educating Students in Modeling Continuous-Time Physical Systems", *Simulation*, Vol.64, No.3, pp.154-159.
- Cellier, F.E., 1992b, "Hierarchical Non-Linear Bond Graphs: A Unified Methodology for Modeling Complex, Physical Systems", *Simulation*, Vol.58, No.4, pp.230-248.
- Cellier, F.E., 1991, *Continuous System Modeling*, Springer-Verlag.
- Comeford, R., August 1994, "Mecha...what?", *IEEE Spectrum*, pp.46-49.
- Controllab Products, 1999, *20-Sim Reference Manual*, Version 3.0, Enschede, The Netherlands.
- Dertouzos, M.L., 1997, "Creating the People's Computer", *MIT's Technology Review*, April, pp.20-28.
- Dynasim AB, 1999, Dymola - Dynamic Modeling Laboratory, [Online] Available <http://www.dynasim.com/>, November 16, 1999.
- Elmqvist, H. Cellier, F.E, and Otter, M., 1993), "Object-Oriented Modeling of Hybrid Systems", *Proceeding European Simulation Symposium*, Delft, Netherlands, pp.31-41.
- Ermer, G., 1994, "Improving Engineering System Design Through Scientific Visualization Methods", Ph.D. Thesis, Department of Mechanical Engineering, Michigan State University.
- Fritchman, B. M. and Hammond, R. A., 1992, "A New Method for Modeling Large Flexible Structures", *Simulation* Vol.61, No.1, pp.53-59.
- Gibbs, W. July 1997, "Taking Computers to Task", *Scientific American*, pp. 82-89.
- Hales, M. 1995, "A Design Environment for the Graphical Representation of Hierarchical Engineering Models", Master of Science Thesis, Department of Mechanical Engineering, Michigan State University.
- IMAGINE, 1996, *AMESim and AmeSet Manual*, Version 1.5, Roanne France.

- Integrated Systems, Inc. 1994, *SystemBuild User's Guide Version 4.0*, Santa Clara, CA.
- ISO, 1998, Information Technology - Programming Languages - C++, ISO/IEC 14882-1998.
- Karnopp, D.C., Margolis, D.L., and Rosenberg, R.C., 1990, *System Dynamics: A Unified Approach, 2nd ed.*, Wiley Interscience, New York.
- Karnopp, D.C., 1988, "General Method for Including Rapidly Switched Devices in Dynamic System Simulation Models", *Transactions of the Society for Computer Simulation*, Vol.2, No.1, pp.155-168.
- Lorenz, F., 1993, "Discontinuities In Bond Graphs: What is Required?", *Proceedings 1993 International Conference on Bond Graph Modeling*, San Diego.
- Mackulak, G.T., Cochran, J.K., and Savory, P.A., 1994, "Ascertaining Important Features for Industrial Simulation Environments", *Simulation* Vol.63, No.4, pp.211-221.
- MathWorks, 1999, MATLAB 5.3 / Simulink 3.0, Natick, Massachusetts.
- MathWorks, 1998, "Using Simulink and Stateflow in Automotive Applications", *Simulink-Stateflow Technical Examples*, 9521v00 2/98.
- Mattson, S.E., H. Elmqvist, and M. Otter, 1998, "Physical system modeling with Modelica," *Control Engineering Practice*, Vol.6, pp.501-510.
- Modelica, 1999, Modelica: Language Design for Multi-Domain Modeling [Online] Available <http://www.modelica.org/>, November 16, 1999.
- Otter, M. and Elmqvist, H., 1997, "Energy Flow Modeling of Mechatronic Systems Via Object Diagrams", *IMACS Symposium on Mathematical Modeling*, pp.705-710.
- Otter, M. and Cellier, F., 1996, "Software for Modeling and Simulating Control Systems", *The Control Handbook*, CRC Press, Boca Raton, FL, pp. 415-428.
- OLMECO, 1991, *OLMECO: Open Library for Models of mEchatronic COmponents*, Part I-III, ESPRIT proposal EC 6521, Brussels, Belgium.

- Pressman, R. S., 1992, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, Inc.
- Rosenberg, R.C., 1991, "Modeling for the Masses: Simulation as a Soft Lab", *Proceedings ASME Dynamic Systems and Control Division DSC-Vol.36*, pp.53-60.
- Rosenberg, R., Hales, M., and Minor, M., 1996, "Engineering Icons for Multidisciplinary Systems", *Proceedings ASME Dynamic Systems and Control Division DSC-Vol.58*, pp.665-672.
- Rosencode Associates, Inc., 1995, *The ENPORT User's Manual Version 5.4*, Okemos, MI.
- Stallings, W., 1998, *Operating Systems: Internals and Design Principles*, Prentice Hall, Upper Saddle River, N.J.
- Stein, J. and Rosenberg, R., 1991, "The Art of Physical System Modeling: Can It Be Taught? A Symposium Retrospective", *Proceedings ASME Dynamic Systems and Control Division DSC-Vol.36*, pp.1-3.
- Swanson Analysis Systems, Inc., 1998, *ANSYS Engineering Analysis System - User's Manual*, Houston, PA.
- Umez-Eronini, Eronini, 1999, *System Dynamics & Control*, Brooks/Cole Publishing Company, Pacific Grove, CA.
- van Dijk, J., 1994, *On the role of bond graph causality in modeling mechatronic systems*, Ph.D. thesis, University of Twente, Netherlands.
- Vries, T.J.A. de, 1994, *Conceptual Design of Controlled Electro-Mechanical Systems*, Ph.D. thesis, University of Twente, Netherlands.
- Vries, T.J.A. de, Breedveld, P.C., and Meindertsma, 1993, "Polymorphic Modeling of Engineering Systems", *Proceedings 1993 International Conference on Bond Graph Modeling*, San Diego, pp.17-22.

MICHIGAN STATE UNIV. LIBRARIES



31293020586503