

MICHIGAN STATE UNIVERSITY LIBRARIES



3 1293 00784 9197

LIBRARY
Michigan State
University

This is to certify that the

dissertation entitled

DYNAMIC PHYSICAL SYSTEM SIMULATION
AND OBJECT ORIENTED PROGRAMMING

presented by

John Douglas Reid

has been accepted towards fulfillment
of the requirements for

PhD degree in Mechanical
Engineering

R.C. Rosenberg
Joseph Whitesell
Major professor

Date 4/02/90

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.

DATE DUE DATE DUE DATE DUE		
106 79 1994 Feb 6 45	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

MSU Is An Affirmative Action/Equal Opportunity Institution

c:\circ\datedue.pm3-p.1

DYNAMIC PHYSICAL SYSTEM SIMULATION

AND

OBJECT ORIENTED PROGRAMMING

By

John Douglas Reid

A DISSERTATION

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

DOCTOR OF PHILOSOPHY

Department of Mechanical Engineering

1990

6054961

ABSTRACT

DYNAMIC PHYSICAL SYSTEM SIMULATION AND OBJECT ORIENTED PROGRAMMING

By

John Douglas Reid

Many useful software packages exist today that effectively implement physical theory in order to support dynamic physical system simulation. As simulation methodologies mature and extend, so must their software implementations. A relatively new software methodology, known as object oriented programming, is investigated to determine its impact on the future of physical system simulation. A bond-graph/block-diagram processor is developed using Smalltalk-80 to help delineate the benefits and difficulties of using the object oriented paradigm. Encapsulation, inheritance, and polymorphism are shown to provide a framework that supports extendible software on both a small and large scale that is typically lacking in conventional languages (e.g., FORTRAN and PL/I). This includes user customization of the software, as well as incorporating major enhancements to the software. Additionally, object oriented implementations are shown to follow closely the world they are modeling. This allows the software designer to concentrate on the problem being solved and not on transforming a theoretical solution of the problem into a software implementation.

To Monica...

ACKNOWLEDGMENTS

I would like to thank my major professors, Dr. Ronald C. Rosenberg and Dr. Joseph Whitesell, for their guidance and support throughout my PhD program. Also, Dr. Steven Shaw, Dr. Jon Sticklen, and Dr. Maciej Zgorzelski have each made valuable suggestions throughout the research.

Special thanks goes to GMI Engineering & Management Institute for providing me with the opportunity to pursue my educational and career interests.

Finally, for their endless love and support, I would like to thank my family, particularly my mother and father (the Professor), and especially my wife, Monica and daughter, Amanda.

TABLE OF CONTENTS

	Page
LIST OF FIGURES	vii
CHAPTER I - INTRODUCTION	1
1.1 Problem Definition and Goals	2
1.2 Benefits	6
1.3 Order of Reporting	7
CHAPTER II - BACKGROUND	8
2.1 Dynamic Physical System Simulation	8
2.1.1 A Brief History	9
2.1.1.1 Direct Programming of Equations	10
2.1.1.2 Block Diagrams	10
2.1.1.3 Bond Graphs	11
2.1.1.4 Macro Models	14
2.1.1.5 Direct Physical Description	14
2.1.2 Capabilities and Style: An Engineer's Perspective	15
2.2 Object Oriented Programming	17
2.2.1 A Brief History	18
2.2.2 Object Oriented Languages	18
2.2.3 Simulation	21
CHAPTER III - AN OBJECT ORIENTED BOND-GRAPH/BLOCK-DIAGRAM PROCESSOR	22
3.1 Model Building	26
3.1.1 Topological Diagram	27
3.1.2 Functional Specifications	32
3.1.3 Topological Macros	40
3.1.4 Enhancements: Topological and Functional	44
3.2 Model Analysis/Equation Formulation	47
CHAPTER IV - THE SMALLTALK-80 IMPLEMENTATION	52
4.1 Object Oriented Design	53
4.1.1 Responsibility-Driven Design	53
4.1.2 Incremental Development	54
4.1.3 Interactive Applications: Model-View-Controller	55
4.2 NodeGraph-80	57
4.3 The Implementation	58
4.3.1 Phase I: Primitive Bond Graph Processor	58
4.3.1.1 System Capabilities	58
4.3.1.2 Object Oriented Structure	59
4.3.2 Phase II: Additional Multiports and Functions	66
4.3.2.1 System Capabilities	66

4.3.2.2 Adding Multiports	67
4.3.2.3 Additional Function Capabilities	72
4.3.3 Phase III: Block Diagrams and View-Controller Pairs	76
4.3.3.1 System Capabilities	76
4.3.3.2 Signals	76
4.3.3.3 Blocks	78
4.3.3.4 View-Controller Pairs	81
4.3.4 Phase IV: Macros	83
CHAPTER V - CHARACTERISTICS OF OBJECT ORIENTED PROGRAMMING IN DYNAMIC PHYSICAL SYSTEM SIMULATION	87
5.1 Development	88
5.1.1 Language Constructs	89
5.1.1.1 Processing Input Requests	89
5.1.1.2 Interactive Windowing	91
5.1.1.3 Data Storage and Usage	92
5.1.1.4 Software Organization	96
5.1.2 Environment	100
5.1.3 Robustness	104
5.1.4 Time	107
5.2 Enhancements	108
5.2.1 Types	108
5.2.1.1 Functions	108
5.2.1.2 Nodes	110
5.2.1.3 Methods	115
5.2.2 Required Expertise	116
5.2.3 Controlled Access	118
5.3 Reusability	120
5.4 Portability Between Object Oriented Languages	123
5.5 Qualitative Reasons	125
CHAPTER VI - CONCLUSIONS	130
6.1 Benefits of Object Oriented Programming	130
6.2 Difficulties with Object Oriented Programming	132
6.3 Unanswered Questions and Future Research	135
APPENDIX A - INTRODUCTION TO BOND GRAPHS	138
APPENDIX B - INTRODUCTION TO OBJECT ORIENTED PROGRAMMING	146
APPENDIX C - OOBProc IMPLEMENTATION DETAILS	151
LIST OF REFERENCES	206

LIST OF FIGURES

Figure 1.1 Who Will Benefit?	6
Figure 2.1 Field Controlled DC Motor	11
Figure 2.2 Block Diagram of a Plant Process	11
Figure 2.3 Bond Graph of a Field-Controlled DC Motor	12
Figure 2.4 Radar Pedestal	13
Figure 2.5 System Graph of Radar Pedestal	13
Figure 2.6 Macro Model of the Radar Pedestal	14
Figure 2.7 Roots of Smalltalk	19
Figure 3.1 Two Wheel Drive Tractor Model: Rigid Body	22
Figure 3.2 Two Wheel Drive Tractor Model: Drive Train	23
Figure 3.3 Macro Model of Tractor	23
Figure 3.4 Details of the Right Wheel	24
Figure 3.5 The Main Menu	25
Figure 3.6 Submenu for Arcs	25
Figure 3.7 Menu for Adding Multiports	25
Figure 3.8 Node Types	26
Figure 3.9 Arc Types	27
Figure 3.10 Predefined System Equations	33
Figure 3.11 User Defined System Equations	34
Figure 3.12 List of Functions	36
Figure 3.13 Views of a Function During Specification	37
Figure 3.14 Diode Function	45

Figure 3.15	Template Function Format	46
Figure 3.16	System Equations for the Radar Pedestal	50
Figure 4.1	Model-View-Controller	56
Figure 4.2	MVC Class Hierarchy	60
Figure 4.3	Phase I Class Hierarchy	62
Figure 4.4	BGController 'addNode:' Method	65
Figure 4.5	Summary of Atomic Node Additions	67
Figure 4.6	Macro Capacitor Class Hierarchy	69
Figure 4.7	Phase II Function Class Hierarchy	73
Figure 4.8	Arc Class Hierarchy	77
Figure 4.9	Node Class Hierarchy	79
Figure 4.10	Multiple Views of a Bond Graph	82
Figure 5.1	Generic Data Defining a Node	94
Figure 5.2	Data Defining Specific Nodes	95
Figure 5.3	Complexity of User Enhancements	118
Figure 5.4	Controlled Access to Enhancement Capabilities	119
Figure 5.5	Smalltalk-80 Code for Inverting a Matrix	126
Figure 5.6	Results From Executing Matrix Inversion Code	127
Figure A.1	Variables Used For Bond Graphs	139
Figure A.2	Bond Graph Multiports	141
Figure A.3	Two Degree of Freedom Spring-Mass-Damper System	142
Figure A.4	Bond Graph of System Figure A.3	142
Figure A.5	Schematic of an Electrical Circuit	144
Figure A.6	Bond Graph Representation of the Circuit	144
Figure B.1	A Simple Class Hierarchy	149
Figure B.2	(a) Procedural vs. (b) Object Oriented Organization	150
Figure C.1	OOBProc Class Hierarchy	152

CHAPTER I

INTRODUCTION

Computer-based physical system simulation is now an integral part of the design process. The two main benefits of simulating system behavior are (1) it saves time, which in today's competitive market is vital, and (2) it saves money. Building multiple proto-type parts is expensive, while computer resources are relatively inexpensive. It also contributes to producing better designs by encouraging more "what if" exploration by design engineers.

Physical system simulation spans many disciplines. It includes mechanical systems, electrical systems, control systems, and combinations of these systems, to name a few. For example, an active suspension system for automotive vehicles includes mechanical, electrical, hydraulic and feedback control systems.

Simulating these systems is not just an application of physical theories. The methods one uses to model and analyze these systems depends on physical theory, the set of conceptual tools available, and computer implementation techniques. This research investigates how a relatively new software methodology, object oriented programming, may impact the future of physical system simulation.

1.1 Problem Definition and Goals

As physical system simulation matures, the software that implements the newer technology becomes increasingly complex. Developing simulation software has become a major challenge and expense. Cox [1] and Levy [2] discuss some of the issues associated with developing quality software, including brittleness, lack of expandability, software reusability, data type dependency, software maintenance, and the software design process as the issues pertain to procedural programming languages like FORTRAN, C, Pascal, or PL/I.

It is evident from these and other writings that research is needed into the software development process, to develop new software methodologies and design new implementation languages. Object oriented programming is one such new methodology [3], [4].

Object oriented programming is believed to be a key to great improvements in the next generation of computer software. Some of the proposed benefits of using object oriented programming are (1) software is reusable and easily identified throughout a system so that duplication of code is drastically reduced and understandability is greatly increased; (2) code can be written generically without reference to the type of data it is operating on; (3) the object oriented programming environment allows for user customizable software and for different levels of development, and (4) complex data structures, such as macro capabilities (a layering concept dealing with systems, sub-systems and components), can be modeled logically and efficiently [5], [6], [7].

The overall goal of this research was to investigate how object oriented programming could be used effectively in physical system simulation. To help determine these benefits a portion of a bond-graph/block-diagram processor was developed in the following phases.

Phase I

A limited bond graph processor that derives the equations of motion was developed. This consisted of the basic bond graph primitives Se, 1, I, R and GY elements. (An introduction to the bond graph methodology is presented in Appendix A.)

Phase II

Enhancements to this bond graph processor were investigated from two different approaches. Capabilities for including bond graph elements Sf, 0, C and TF were examined from (1) a macro perspective, and (2) an atomic perspective.

Phase III

Further enhancements were added to the bond graph processor by incorporating block diagrams into the system.

Phase IV

At this point the research could have proceeded in either of two directions: (1) provide the next step in the simulation process (i.e., numerical methods, such as integration), or (2) provide advanced model building techniques by investigating a more abstract perspective of the models (e.g., provide macro capabilities). This

research focused on the second, model-building perspective.

Smalltalk-80 [8], [9] was the object oriented language used to implement the bond graph processor. The language was chosen for the following reasons: (1) It frequently serves as the language to which other object oriented languages are compared; (2) it is a large system containing many pre-defined classes, and (3) it is a pure object oriented language. Thus old programming habits will be kept to a minimum. For example, if C++ [10] were used it might have been tempting to write much of the system using plain C. It was felt that breaking away from procedural languages entirely would provide the most insight.

It should be pointed out that the actual implementation was not the major objective. The focus was on how object oriented programming supports the transition from phase to phase.

In addition to the development of original software, the capabilities available to the users of the software are delineated. To make this clear, let us define three types of users:

Engineer

- Typical engineer with a Bachelor's degree.
- Uses physical system simulation software as one of many tools to complete the job on hand.
- Knows the basic theory of analysis but if a detailed analysis is needed would pass it off to an Analysis Engineer.
- Often needs minor enhancements to existing software.

Analyst

- Applies analytical methods on a day-to-day basis.
- Typically holds a Master's or PhD degree in engineering.
- Often wants to get more out of physical system simulation software than its original purpose.

Systems Programmer

- Responsible for administering and perhaps enhancing the software.
- Typically trained in computer science.
- Tasks include making software work on various hardware platforms, and implementing additional analysis techniques defined by an Analyst within existing software.

Currently any enhancements that are made to a complex simulation package are done by a systems programmer. This is because of the complexity of the implementation of the software.

One question to be answered is: What classes of enhancements can be made by each type of engineer if object oriented programming is used to develop the original software? By implementing the bond-graph/block-diagram processor in phases, it is hoped that this question can be answered.

1.2 Benefits

There is reason to think that object oriented programming will allow the several types of users to understand the software and its environment well enough to make reasonable modifications without becoming expert programmers. This would be a major benefit to the engineering physical system simulation community.

Additionally, both the computer science object oriented programming community and the engineering physical system simulation community will benefit from this research. This is depicted in Figure 1.1. Specific benefits for each community are then listed.

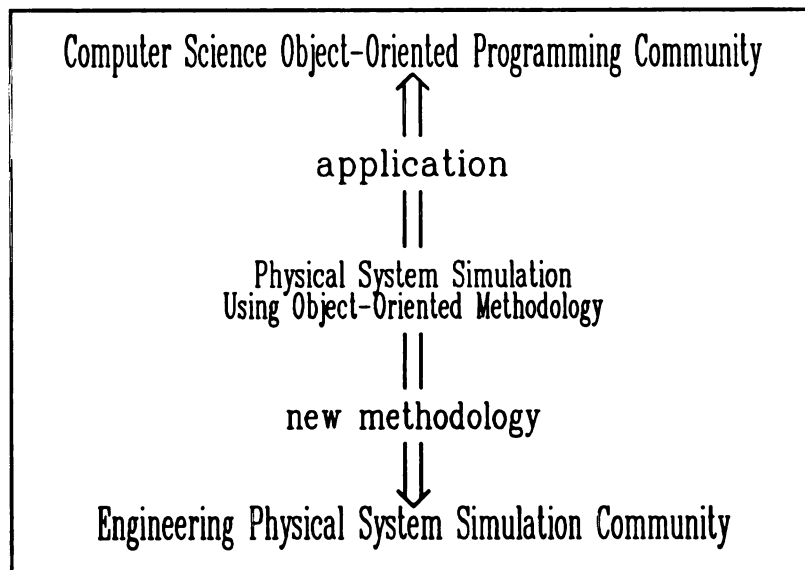


Figure 1.1 Who Will Benefit?

Benefits for the Computer Science Object Oriented Programming Community

- Serious applications are required in order to examine the theory of the object oriented methodology as it applies in practice.

- Advantages demonstrated will encourage others to use this methodology where applicable.
- Shortcomings recognized (or discovered) give researchers a direction for further work.

Benefits for the Engineering Physical System Simulation Community

- May provide a development platform that is readily extendable to include additional physical and dynamic effects.
- May provide implementations that are more readily maintained (e.g., ported, revised for hardware upgrades).
- May enable existing physical system simulation technology to run with fewer bugs (defects, errors) and faster.

In summary, if important benefits are found by applying the object oriented methodology then significant improvements in future computer software systems will be obtainable.

1.3 Order of Reporting

Chapter II gives a background for both dynamic physical system simulation and object oriented programming. Chapter III describes, from the user's point-of-view, the bond-graph/block-diagram processor (OOBProc) developed using Smalltalk-80. Chapter IV discusses the implementation technicalities of OOBProc. Chapter V details the benefits of using object oriented programming methods that were discovered during the development of OOBProc. Finally, Chapter VI lists the benefits and difficulties encountered during the research. Additionally, some unanswered questions and future research topics are given in Chapter VI.

CHAPTER II

BACKGROUND

2.1 Dynamic Physical System Simulation

The purpose of simulation is to predict the behavior, the structure or the attributes of a system [11]. Our interest is in simulation through computer software, also known as soft proto-typing. Since the advent of the computer, engineers have been developing and using soft proto-typing techniques in addition to the normally more costly and time consuming method of creating hardware proto-types.

The simulation process is complex, involving problem formulation, model building, model analysis, formulation and numerical solution, and meaningful response display [12]. Design based on simulation is an iterative process and can take anywhere from a few days to several months, depending on the complexity of the model.

The major types of simulation are discrete, continuous and combined [13]. Discrete simulation is based on instantaneous changes in the state of a system [14]. An example of discrete simulation is modeling a bank customer queuing system. Continuous simulation is based on continuous changes in the state of a system over time [15]. For example, planetary motion in our solar system would be studied with continuous simulation

methods.

Systems that are affected by both discrete and continuous changes in state are referred to as combined, or hybrid, simulations. For example, a liquid waste storage system involves continuous dynamics of the liquid in the storage tank and discrete changes due to trucks unloading of waste at different intervals.

The physical systems under consideration in this research are those that are dynamic and involve energy, as well as information, exchange between components. Examples of these types of systems include an overhead cam valve positioning mechanism as used in an automotive engine, a hydraulically controlled landing gear retraction mechanism as used in an airplane, and an electro-pneumatic transducer as used in a temperature control system. This research concentrates on bond graph and block diagram modeling techniques for performing simulations.

2.1.1 A Brief History

This sub-section describes the advances in physical system simulation technology over the last four decades. Reference [16] gives a survey of the various continuous system simulation languages developed between 1955 and 1985 in order to implement the newer technology. Of more recent history and interest to engineers are the automated modeling and analysis programs, commonly referred to as computer aided engineering (CAE) systems [17], [18].

2.1.1.1 Direct programming of equations

Original digital simulation packages were numerical integrators which emulated the analog computer style [19]. If one would describe the system equations to the computer, the software would generate volumes of (useful) data. The sophisticated packages provided graphical output for easy interpretation of the system performance. The equation derivations were left to the engineer. Of course, there were no restrictions on how the equations were obtained. For example, a set of mechanical system equations could be derived using Newton's second law, Lagrange's equations, or Hamiltonian methods. For complicated systems this could be extremely difficult. Furthermore, the system equations had to be expressed in a formal language like FORTRAN.

2.1.1.2 Block diagrams

Block diagrams are a graphical technique used to model physical systems [20], [21]. Block diagrams are used for information transfer processes, such as control. They provide a convenient and useful representation for characterizing the functional relationships among the various components of a control system. Using graphics terminals, simulation software can capture the graphical representation of the system in its block diagram form. For example, Figure 2.1 shows a block diagram for a simple field-controlled dc motor [22]. This leads to insight into the system structure that is not evident just by examining the system equations. Although the initial system equations are still needed to create the block

diagram for detailed analysis, modifications to the system are much more easily adapted into the block diagram.

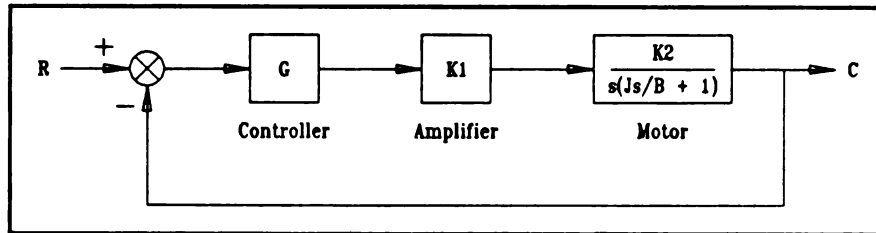


Figure 2.1 Field Controlled DC Motor

Additionally, block diagrams allow for abstractions, such as transfer functions and 'plant' models. See Figure 2.2 as an example.

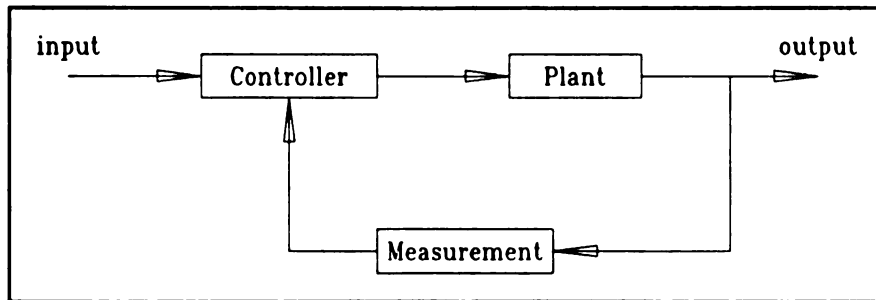


Figure 2.2 Block Diagram of a Plant Process

2.1.1.3 Bond graphs

Of more recent vintage is a modeling technique known as bond graphs [23], [24], [25]. Bond graphs are used for structured modeling of energy and power transfer processes, including mechanical, electrical, magnetic, hydraulic, and thermal processes and their combinations. Like block diagrams, bond graphs are a graphical representation form. But unlike

block diagrams, bond graphs are derived directly from the physical system model, rather than through the equations of the system. This gives the trained engineer direct insight into the physical system by examining the bond graph without interpreting any equations.

Figure 2.3 shows a bond graph representation of a field-controlled dc motor connected to a load. An introduction to bond graphs is given in Appendix A.

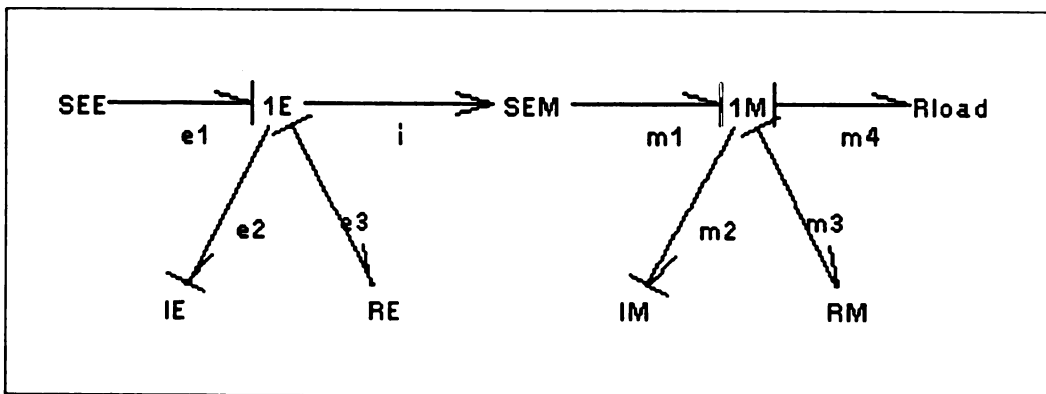


Figure 2.3 Bond Graph of a Field-Controlled DC Motor

Using a combination of bond graphs and block diagrams allows the engineer to model an extremely large set of dynamic physical systems [26]. The bond graph portion of the model captures the system dynamics, while the block diagram portion can be used to represent the control of the system.

For example, a radar pedestal used to track moving objects is shown in Figure 2.4 [27]. The system graph shown in Figure 2.5 includes bond graph elements used to model the open loop system and block diagram components used to impose feedback control. This control is used to set the angular position of the pedestal to a desired location.

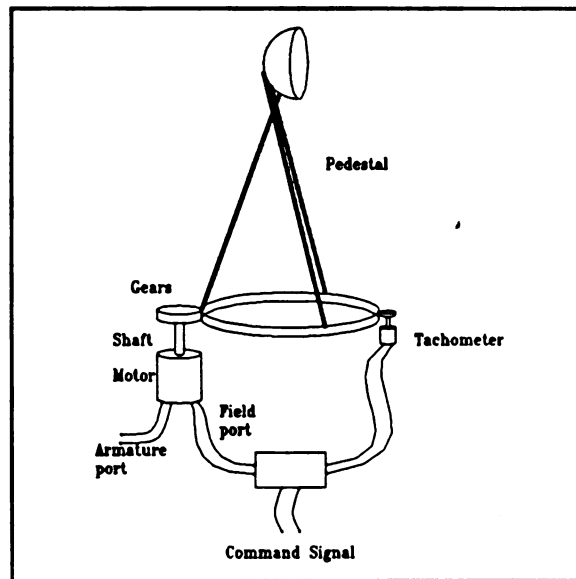


Figure 2.4 Radar Pedestal

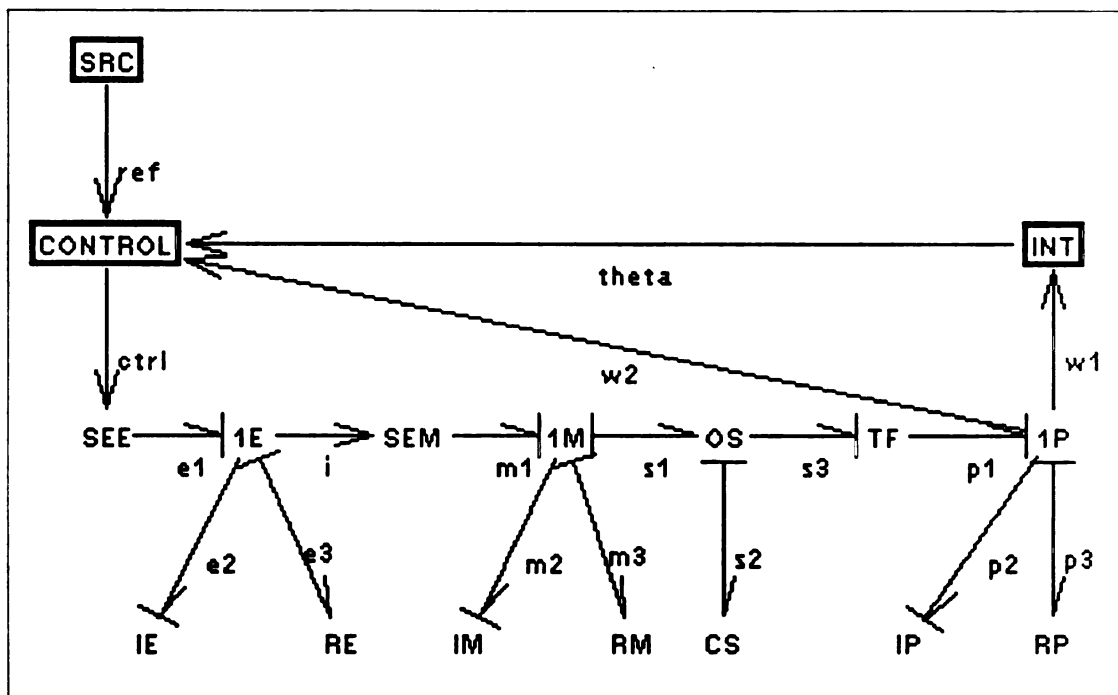


Figure 2.5 System Graph of Radar Pedestal

Once a model of the dynamic system is obtained in bond-graph/block-diagram form it can be handled by computer software, such as ENPORT, CAMP, and TUTSIM [27], [28], [29].

2.1.1.4 Macro models

Topological macros provide for storage of sub-system descriptions in order to be used in larger systems [30] . This is like a component library system. Three advantages of using macros are: (1) describing and working with large complex systems in a manageable way; (2) reusability of component models, and (3) efficiency when testing variations of the same physical component. Figure 2.6 shows a macro model for the radar pedestal shown in Figure 2.4.

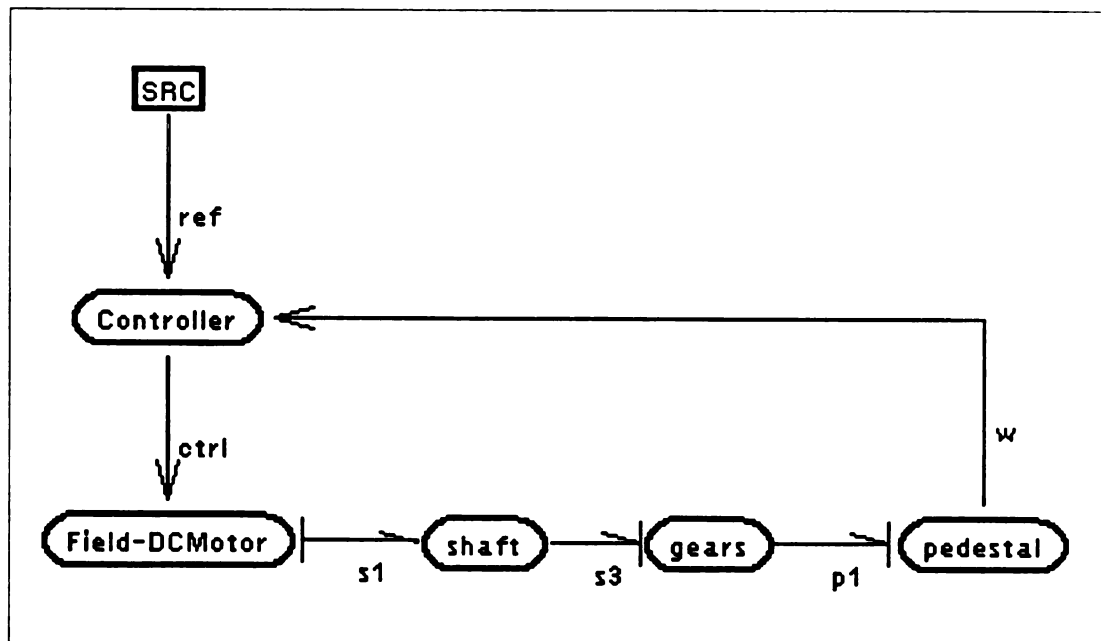


Figure 2.6 Macro Model of the Radar Pedestal

2.1.1.5 Direct physical description

Currently, a lot of attention has been directed to integrating computer aided design (CAD) systems with simulation packages. The CAD systems are

used for capturing the geometric design descriptions, generating the input to a simulation package, executing the simulation software, and then displaying the results from the simulation in a meaningful way. For example, Magic [31] is a graphical layout editor for VLSI design which interfaces with SPICE [32], a circuit simulation program. As another example, MEDS [33] is a mechanical design system that interfaces with ADAMS [34], a software package for simulating the force and motion behavior of mechanical systems.

2.1.2 Capabilities and Style: An Engineer's Perspective

When designing physical system simulation software one must consider the different groups that will be working with the software. One can identify three such groups: software developers, software maintainers, and engineers. These three groups have tasks and viewpoints that differ and are sometimes in conflict.

Software developers are responsible for creating and implementing the simulation software. Software maintainers handle system administration functions, including different hardware configuration implementations, bug fixes and minor enhancements.

The engineer's point-of-view stems from the question: What can the software do for me? This can be broken down into three main categories, those being: (1) modeling domain; (2) solution capabilities, and (3) presentations of results. The first two issues are generally of main

concern to the engineer.

The modeling domain of a software package identifies the types of systems an engineer can simulate. For example, a rigid body mechanism could be simulated using ADAMS [34] or DADS [35]. Would these packages still be appropriate for simulating flexible bodies, or for simulating an active suspension with mechanical, electrical, hydraulic and feedback control components? The engineer needs to match his/her particular applications with the appropriate simulation software.

Once the modeling domain is specified, a detailed look into the actual simulation capabilities is required. Does the software treat linear and non-linear systems? Does it cope with stiff systems? Is linearization and eigen analysis included? How well does it react to singular states or model inconsistencies? These are but a few of the questions associated with simulation capabilities.

Style also plays an important role. Style includes, for example, the user interface features and documentation support for the software and training. These issues help determine how easy a package is to learn and to use. This is extremely important for new users and also for intermittent users who will be using the software on an as needed basis. The every day users also need specialized styling features. Being very familiar with the software, they would not want to be slowed down by an extensive menu system, for example.

2.2 Object Oriented Programming

Physical system simulation software is written to imitate behavioral aspects of real physical systems. It would make sense to design the computer-based information structure to be congruent with the way we think about the physical world. For example, if you were going to assemble and operate a dynamic mechanism you would first identify its components, assemble them and then put the mechanism in motion. The initial focus would be on the components, gradually shifting to the system.

Object oriented programming is a relatively new idea in software programming that concentrates on the objects of a system rather than on the procedures that manipulate data, as conventional engineering software does (e.g., FORTRAN or PL/I) [1], [3], [8]. According to Floyd [7]:

"Our world is filled with objects, so it seems only natural to describe and solve problems in terms of objects as well. This idea is the basis for object-oriented programming."

Stefik and Bobrow [5] think object oriented programming is directly applicable to simulation:

"Objects are a uniform programming element for computing and saving state. This makes them ideal for simulation problems where it is necessary to represent collections of things that interact."

Focussing on the objects would appear to have great potential in simulating physical systems. This dissertation investigates that potential. Appendix B gives a brief overview of object oriented programming.

2.2.1 A Brief History

The origins of object oriented programming are quite diverse. It is commonly agreed among the experts that Smalltalk is considered the first language to encompass and define object oriented programming. However, the roots of Smalltalk include conventional languages (Algol and Simula), artificial intelligence (AI) languages (Lambda-Calculus, Lisp, Planner and Logo), and unique languages (Sketchpad and Flex) [36]. This is depicted in Figure 2.7

2.2.2 Object Oriented Languages

Object oriented languages are specific implementations of vendors' interpretations of what object oriented programming should be. The majority of the literature suggests that, to be a true object oriented language, the language must have the following two properties:

Encapsulation

Ability to combine data and the code that operates on that data into a single structure.

Inheritance

Ability to derive specialized structures from more general structures.

Ada [37] and Modula-2 [38] are two languages that provide encapsulation but do not provide inheritance. A few people still call these object oriented languages but the majority do not [39], [40]. A compromise is to call languages such as these object based.

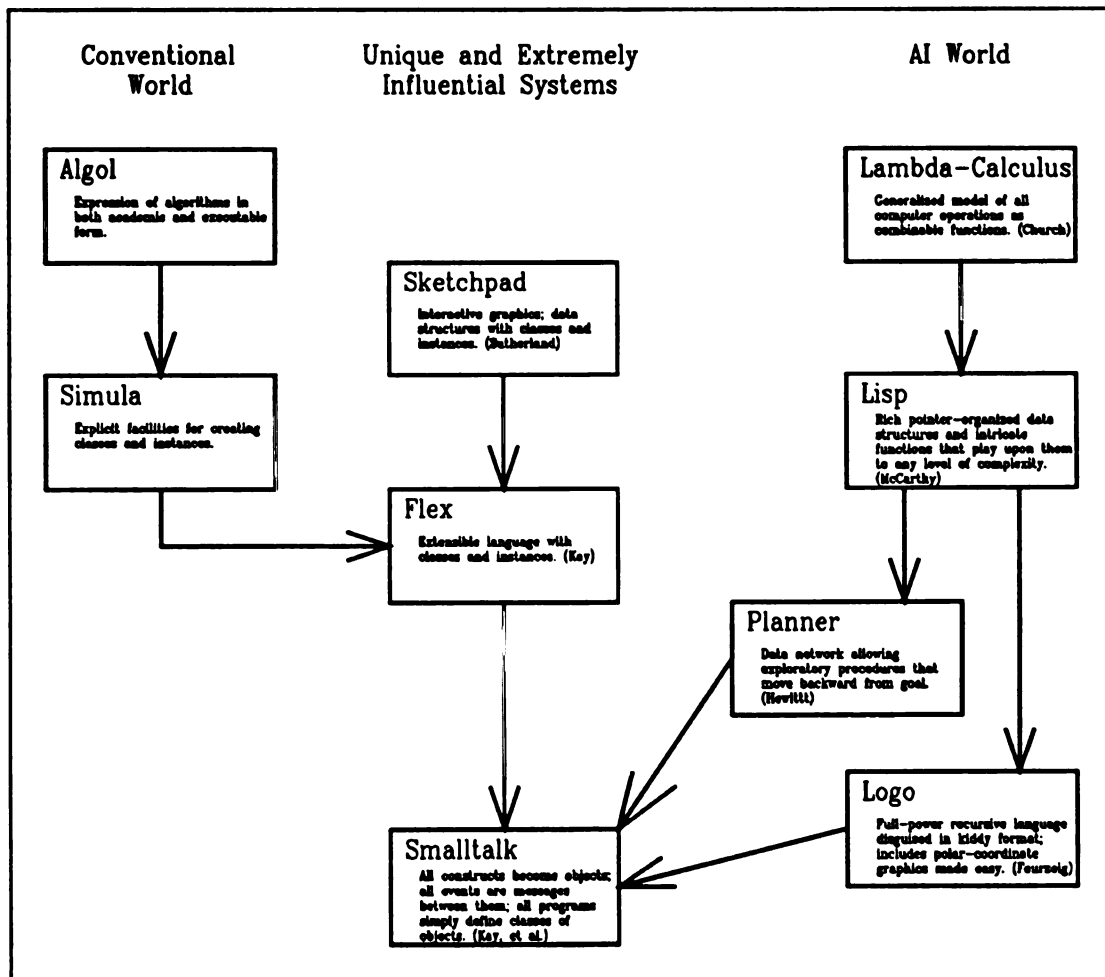


Figure 2.7 Roots of Smalltalk

A third property sometimes added to the definition of object oriented languages is:

Polymorphism

Ability to send the same message to different objects. That is, the same action can be requested from different objects without special processing.

We adopt these three properties in our approach.

There are two types of object oriented languages, pure and hybrid. There are dozens of object oriented languages available today. Rettig et al [41] discuss 17 of them.

Pure object oriented languages operate strictly within the rules of object oriented programming. This means that everything (essentially) is considered an object and that the language is built upon that concept. Smalltalk-80 [9], Trellis/Owl [42] and Actor [43] are three object oriented languages that are considered to be pure.

Hybrid languages consist of object oriented extensions to a non-object oriented language like C, Lisp or Pascal. Examples such as C++ [44] and Objective-C [45] are hybrid object oriented languages written on top of standard C. CommonLoops [46] and Flavors [47] are object oriented languages implemented using LISP. Jacky and Kalet use standard Pascal but program with the concepts of object oriented programming [48]. New releases of Pascal from both Microsoft and Borland [49] (major software corporations) include some basic object oriented programming capabilities.

2.2.3 Simulation

Object oriented simulation is relatively new but traces its roots back to Simula in the 1960's [50]. The majority of the applications to date have concentrated on discrete simulation.

General object oriented languages, like Smalltalk-80, often provide discrete simulation capabilities [51], [52]. Also, special languages have been developed for general discrete simulation applications [53], [54]. Additionally, object oriented languages have been developed for specific applications. For example: (1) Ruiz-Mier and Talavage describe a paradigm for simulating manufacturing processes [55]; (2) Larkin et al describe SERB, an object oriented language for research biologists [56], and (3) Gates et al discuss the object oriented language Ross, under development since 1982, used for military modeling and simulation [57].

Work on continuous simulation using object oriented programming is just beginning. For example, Gaush and Huntsinger [58] describe an object oriented continuous system simulation environment that integrates a few nonlinear functions that could be described by a block diagram, for instance. Sung [59] describes a kinematic solver developed in Smalltalk-80, as another example. Both works concentrate on the numerical solution aspect of simulation. The work of this dissertation concentrates on the model description and equation formulation aspects of the simulation process.

CHAPTER III

AN OBJECT ORIENTED BOND-GRAPH/BLOCK-DIAGRAM PROCESSOR

This chapter describes, from the user's point-of-view, the bond-graph/block-diagram processor developed using Smalltalk-80. For conciseness the object oriented bond-graph/block-diagram processor will be referred to simply as OOBProc.

Two examples are used through out this chapter to illustrate the features of OOBProc: a radar pedestal with positional control and a two-wheel-drive tractor. The radar pedestal was shown in Figures 2.4, 2.5 and 2.6. The two wheel drive tractor is shown in Figures 3.1 and 3.2. The details of the bond graph for the tractor were developed by Kaumbutho [60]. Figure 3.3 shows a macro model of the tractor, while Figure 3.4 shows the details that compose the right wheel node.

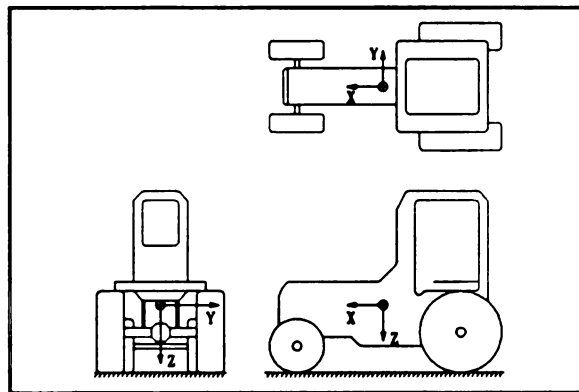


Figure 3.1 Two Wheel Drive Tractor Model: Rigid Body

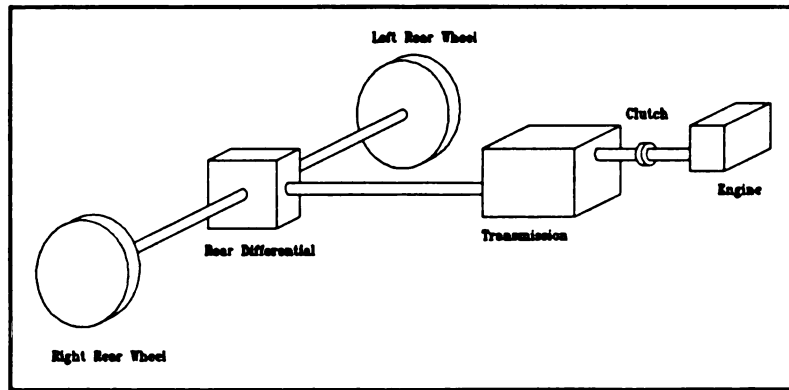


Figure 3.2 Two Wheel Drive Tractor Model: Drive Train

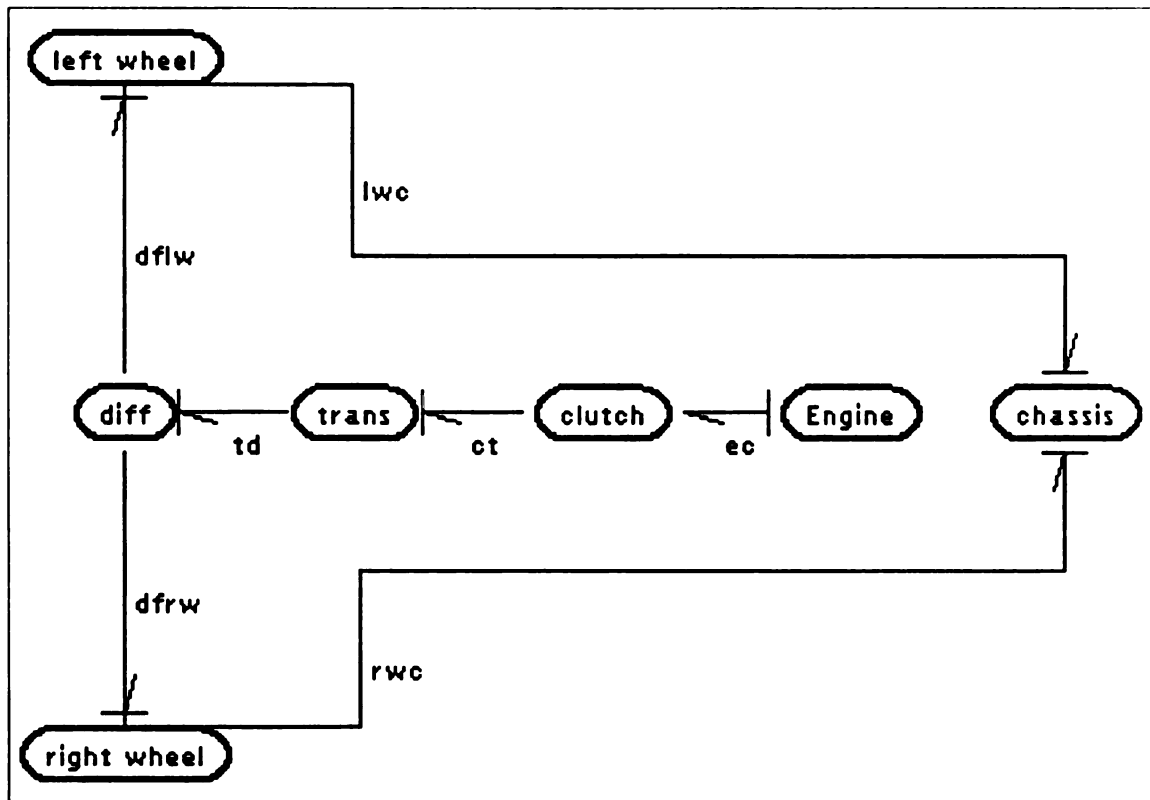


Figure 3.3 Macro Model of Tractor

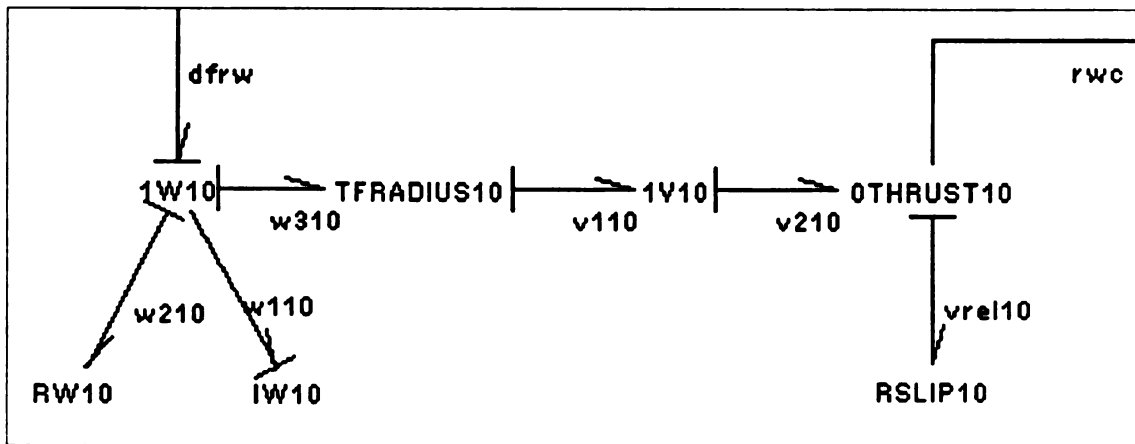


Figure 3.4 Details of the Right Wheel

With the exception of a few activities every option in OOBProc is chosen by use of a hierarchical menu; with each option there is a possibility of a sub-menu associated with it. This is recognized by options with arrows to the right.

The main menu of OOBProc is shown in Figure 3.5. To bring up a sub-menu all one needs to do is to drag the mouse to the higher option desired and the sub-menu is brought up automatically. See Figures 3.6 and 3.7 for example. A specific request is submitted to the bond graph processor when the user finally selects a menu option that has no sub-menu associated with it.

Throughout this chapter picking an option from the menu refers to working through the menu hierarchy until that option is selected.

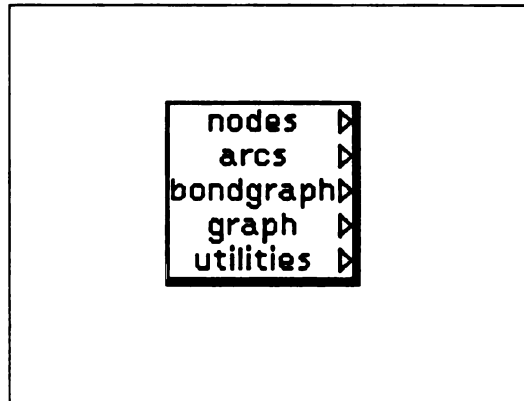


Figure 3.5 The Main Menu

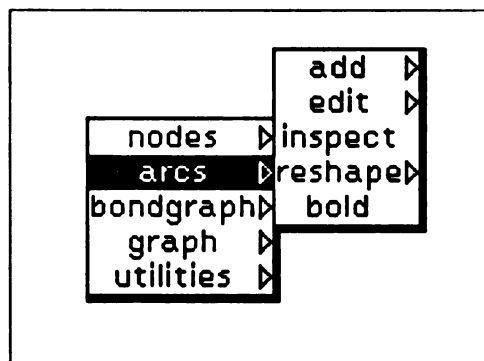


Figure 3.6 Submenu for Arcs

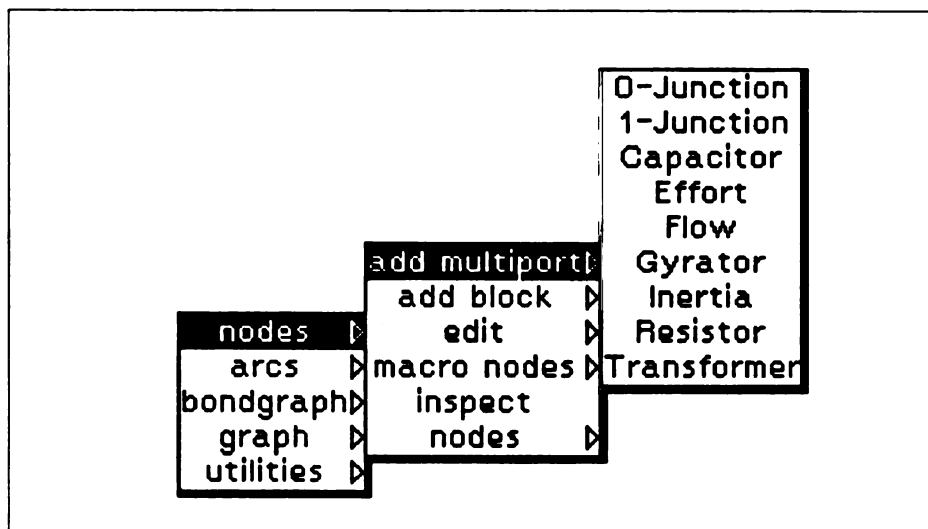


Figure 3.7 Menu for Adding Multiports

3.1 Model Building

The first stage of the simulation process is to describe the physical model to the software application. For a bond graph this means describing the topology and the functional specifications of the nodes.

A graph consists of nodes and arcs. Bond graph nodes are referred to as multiports while their arcs are called bonds. Block diagram nodes are called blocks and their arcs are referred to as signals. Figures 3.8 and 3.9 list the specific types of nodes and arcs available in OOBProc respectively. Methods to add, modify, manipulate and remove these components are needed for proper model building.

Multiports	0-Junction 1-Junction Capacitor Effort Flow Gyrator Inertia Resistor Transformer
Blocks	Distributor Gain Integrator Source Sink Weighted Summer

Figure 3.8 Node Types

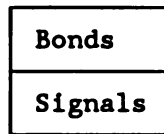


Figure 3.9 Arc Types

Associated with each node are functional specifications. These specifications are mathematical relationships that depend upon the topology. Methods that define the functions and keep them consistent with the topology are required. Function definition is part of the model building process.

Advanced model building capabilities have been included in the bond graph processor. These include: (1) topological macros, which are nodes composed of multiple nodes and arcs; (2) user addition of new function types, and (3) addition of new multiport and block types not supplied in the original application.

This section describes in detail the model building features of OOBProc. This work concentrates on building models using graphical tools rather than through keyboard entry.

3.1.1 Topological Diagram

The topological diagram is a description of the connection network describing the system. This describes what components exist, how they are connected and how they interact with each other. Describing the topology

of the bond graph to OOBProc consists of adding, modifying, manipulating and removing components or groups of components.

ADDING COMPONENTS

Each node and arc in the system is restricted to have a unique label (i.e., name) in order to communicate efficiently between the user and the software. This is checked at creation time for nodes and arcs.

An icon is the term used for the graphical representation of anything to be displayed on the screen. The icon of a multiport node is just a graphical representation of its label, while that of a block is its label with a box around it.

An arc requires a `from_node` and a `to_node`. Bonds are restricted to connect exactly two multiports. Signals can start from a block, a 1-Junction or a 0-Junction; they must end in a block or any multiport but a 1 or 0- Junction. Additionally, arcs have internal points associated with them to allow for polyline or spline display, rather than just straight lines from beginning node to ending node. This and other graphical features are discussed later in this section.

MODIFYING COMPONENTS

At some point in the simulation, functional specifications are made for each node. Modifying the topology will affect these specifications if they have been set previous to the modifications. These implications will

be discussed in section 3.1.2. Here topological modifications are discussed as if functional specifications have not been set. However, the methods for modifying components do not change even after functions have been assigned. The following topological modifications are permitted:

1. The label of any node or arc can be changed as long as the new label is unique.
2. The type of a node may be changed. However, a multiport can only be changed into another multiport, and a block can only be changed into another block. For example, a resistor can be changed into a capacitor but not into an integrator.
3. The direction of a bond can be reversed. A signal was not made reversible because the input/output characteristics of blocks do not lend themselves to switching signal directions effectively.
4. The `from_node` and/or `to_node` of any arc can be changed with the same restrictions as for new bonds and signals as mentioned previously.

MANIPULATING COMPONENTS

There are multiple types of graphical manipulation provided, including the following:

1. Any node can be dragged around on the display window simply by picking the node and moving the mouse. This occurs dynamically and all connecting arcs are continuously re-displayed as the node moves.
2. The entire graph can be dragged by picking a minimum distance away from any node and moving the mouse.
3. The display of an arc can be made into any one of the following: (1) straight; (2) polyline; (3) rectilinear polyline, or (4) spline. Polylines and splines can be edited dynamically by dragging their defining points. The arcs themselves can be displayed as bold (i.e., thicker lines than the default display).
4. The graph can be scaled to fit into a defined rectangle or into the current displayed view.
5. The graph can be zoomed in and out either to look at details more closely (zoom in) or to look at the overall graph better (zoom out).

Another type of manipulation is to work on multiple nodes, and the internal arcs defined by these nodes, at one time. This is referred to as a group.

A group can be moved around graphically just as if it was a single node. A group can also be removed from the graph. This is done one node at a time (see removing components below).

A buffer is supplied to allow for what is known as copy, cut and paste for a group. Copy makes a copy of the group and stores it in the buffer. Cut behaves as copy but additionally removes the group from the graph. Paste makes a copy of whatever is in the buffer and adds it to the graph.

Pasting the buffer to the graph requires that unique labels are adhered to. This may require re-labeling of the nodes and arcs. This can be done either manually or automatically.

Pasting also requires a spot in the graph to put the new components. Four options are provided for this:

- a. insert as is
- b. insert scaled to rectangle
- c. paste as is
- d. paste scaled to rectangle

Insert spreads apart current nodes to allow for the insertion. Paste puts the new components directly on top of whatever happens to be in the spot the user chooses as the reference point for pasting. Scaled to rectangle for both options allow for scaling of the new components to a rectangle before adding them to the graph. After the buffer is copied into the graph, that group of components is made dynamically movable for precise

placement.

REMOVING COMPONENTS

Finally, if a node is deleted, then all of its attached arcs are removed also. This is because an arc without both ends defined is meaningless for a bond graph. For protection, confirmation is required before the deletion of a node takes place.

3.1.2 Functional Specifications

As mentioned previously, associated with each node are mathematical equations. The collection of the equations from all of the nodes in the system graph is referred to as the system equations. The topology defines some of the system equations (see Figure 3.10), while other equations are user defined (see Figure 3.11). This section discusses specifying the user defined functions.

The functions listed for the multiports in Figure 3.11 are for the simple cases when no block diagrams are used. When a signal is directed towards a valid multiport, that multiport is referred to as modulated. For example, the voltage input source SEE in the radar pedestal is a modulated source based on the signal labelled 'ctrl'. The functions associated with modulated multiports then include the input signals as part of the functional definition input variables.

<u>Multiports</u>	
0-Junction	Common Efforts Sum of Flows = 0
1-Junction	Common Flows Sum of Efforts = 0
Capacitor	Displacement = integral(Flow) or Flow = derivative(Displacement)
Gyrator	Effort1 = modulus1 * Flow2 and Effort2 = modulus1 * Flow1 or Flow1 = modulus2 * Effort2 and Flow2 = modulus2 * Effort1 The modulus is user defined.
Inertia	Momentum = integral(Effort) or Effort = derivative(Momentum)
Transformer	Flow1 = modulus1 * Flow2 and Effort2 = modulus1 * Effort1 or Flow2 = modulus2 * Flow1 and Effort1 = modulus2 * Effort2 The modulus is user defined.
<u>Blocks</u>	
Distributor	output1= input . . outputn= input
Gain	output = constant * input
Integrator	output = integral(input)
Source	output = function(time)
Weighted Summer	output = constant1 * input1 + constant2 * input2 . . + constantn * inputn

Figure 3.10 Predefined System Equations

Multiports	
Capacitor	Effort = function(Displacement) or Displacement = function(Effort)
Effort	Effort = function(valid system variables)
Flow	Flow = function(valid system variables)
Gyrator	modulus = function(valid system variables)
Inertia	Momentum = function(Velocity) or Velocity = function(Momentum)
Resistor	Effort = function(Flow) or Flow = function(Effort)
Transformer	modulus = function(valid system variables)
Blocks	The user can override any predefined function type of a block, if desired, with the exception of the Distributor and Sink block types. However, in doing so, the consistency between the topology and the functional specifications is lost.

Figure 3.11 User Defined System Equations

A function consists of a type, output variables, input variables and parameters. The topology of the system graph is used, by default, to determine the outputs and inputs for each node. The user can override the input variables for a particular node but not the output variables.

In general, one would probably not want to override input variables because this makes the equation structure inconsistent with the topology, resulting in a system that would be hard to interpret. However, special cases do exist where this overriding convenience is warranted. Thus, this feature is provided in the bond graph processor. It would be possible to provide a switching mechanism that could be turned on to force the functional variables to be consistent with the topology, if desired.

Valid system variables consist of the effort, flow, momentum and displacement on all bonds, the values of all signals, and time. System variables are determined by the labels of the arcs. Signal variables are just the signal labels, while bond variables are prefaced by characters to distinguish between effort (E.label), flow (F.label), momentum (P.label) and displacement (Q.label) variables.

Briefly, for block diagrams, the signal specifies the direction of its variable. For bond graphs, a technique known as causality determines the direction of a bond's effort and flow variable. These variable directions on the arcs uniquely determine the inputs and outputs for all nodes.

Before assigning functions to the nodes, the system graph must:

a. be complete

Have a valid number of bonds and/or signals attached to each node.

b. have assigned causality

In order to determine the input/output variables for multiports.

If these constraints are not satisfied then the user is notified of the problem and is not allowed to assign functions until the problems are fixed.

The list of available functions in OOBProc is shown in Figure 3.12. The current list of functions is just a representative choice of functions a more complete and practical system would have.

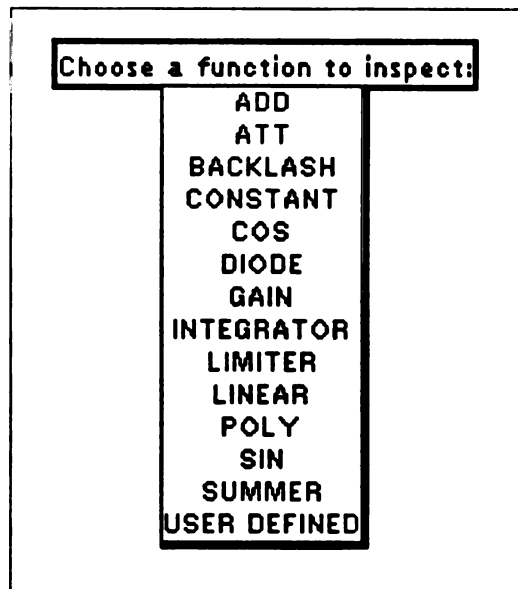


Figure 3.12 List of Functions

To aid the user in specifying the input variables and parameters of the function, two views of the function are brought up on the screen. For example, Figure 3.13 shows the shaft compliance in the radar pedestal being changed. The view on the left is the generic template used to specify functions in general and the view on the right is that function as it is currently defined for the node. In this case, the left view shows the generic GAIN function definition and the right view shows the current definition of the shaft effort equation.

Enter new system variable for input 1	
Q.s2 _a	
Generic function definition: GAIN # Inputs = 1 # Outputs = 1 # Parameters = 1 output 1 = parm 1 * input 1	Current function definition: $E.s2 = 5000 * Q.s2$ <div style="background-color: #cccccc; height: 50px; width: 100%;"></div>

Figure 3.13 Views of a Function During Specification

When modifications are made to the system topology (section 3.1.1) it is required to investigate the functional specifications in order to see if they are still valid. The following cases must be handled:

a. Re-labeling a node.

In general, node labels are not used for functional

specifications, thus no special handling is required. However, the gyrator and transformer moduli functions do require the node name for unique identification. Thus, if the label of a gyrator or a transformer is changed then their function is updated appropriately.

b. Re-labeling an arc.

Arc labels are used as the system variables. If an arc changes its label then all functions using that label as a variable must either be (1) modified to use the new label or (2) reset to the default function because the old label would no longer exist, thus invalidating the old function. The first case is desired and occurs if the user follows the topology for determining system variables allowable for each node. Case (2) occurs if a different variable (which is modified) is used to override the default variables for a node. This method is used because it was determined that a global system search of all functional definitions was not reasonable just to check if the user has used the special case of overriding system variables.

c. Removing an arc (or changing an arcs nodes).

Removing an arc may cause the functions of the nodes that use that arc as a system variable to be invalid. These nodes will be reset to their default functions. If an arcs' to and from nodes are changed the nodes affected in the change must also be reset.

d. Removing a node.

The arcs attached to that node are also removed. This requires handling as described in c. above.

e. Adding nodes.

Adding nodes do not directly effect the existing system except that this is usually followed by adding new arcs, which does affect the system equations.

f. Adding arcs.

Adding arcs affects the topological system variables associated with the nodes that the new arcs are attached to. By default, it is assumed the user will want to change those node functions and they are thus reset to default conditions.

Before proceeding to functional specifications after completing topological modifications the checks for completeness and valid causality are made. At this time the functional changes described in b.(2), c, d, e and f are actually made. This is because the assignment of causality checks to make sure that the existing functions are still valid for the modified system. If not, then those functions are reset to default functions. Most of the cases described above are discovered at this time.

Another possible change in functions is due to a possibly different causality assignment on the system graph than the previously assigned causality. Recall that causality determines the input and output variables for a node. If causality changes then so do the input and

output variables for the nodes attached to the changed causality bonds. These nodes must be reset to their default functions.

3.1.3 Topological Macros

Topological macros are needed for: (1) describing and working with large complex systems in a manageable way; (2) reusability of component models, and (3) efficiency when testing variations of the same physical component.

Macros are incorporated in OOBProc by use of what is referred to as a macro node. The reason for this will be explained in section 4.3.4. A macro node is a node in the system graph that is composed of other nodes (either multiport, block or other macro nodes) and arcs. A macro node can not contain itself.

For example, the tractor model shown in Figure 3.3 is shown as composed entirely of macro nodes. The details of each node are hidden from the observer. The expanded tractor model contains over eighty multiport nodes and seventy bonds. Without macros to organize the tractor model the editing and understandability of the tractor would be extremely complex. The macro nodes provide for managing large and complex systems efficiently.

A macro node behaves as any other node in a several ways. For example, the graphical capabilities described in section 3.1.1 apply to macro nodes in the same manner as they did for multiport and block nodes. This point is

very useful for developing the software in an object oriented language, which will be discussed in the next chapter.

A macro node is also different from a block or multiport in the following ways:

- a. Different methods for creation and modifications are required.
- b. Viewing the internals of a macro is required.
- c. Expanding a macro to un-create it but leave its composition in the system is required.
- d. Saving and restoring macro nodes by themselves are required.

Creating a macro node requires the user to define a rectangle that encloses the nodes that are to make up the macro. The arcs that are defined within these nodes become the internal arcs of the macro node. The arcs that are only defined by one end point among these nodes become the external arcs of the macro node. These external arcs are the connections between the macro node and the rest of the system.

Making modifications to a macro node include:

1. Changing its internal structure.

Modifications to the internal topological structure of a macro

are not allowed directly. If the user wants to change the internals he/she must un-create (expand) the macro into its component parts and then modify those components. This restriction is used to help keep consistent macro part libraries. If the user could modify directly the internals of a macro it would lead to non-standardized parts fairly quickly.

2. Modifying its functional specifications.

Changing the functional specifications of multiport and block nodes is required even when those nodes are part of a macro node. For this reason, if the user selects a macro node to modify its functional specification (see section 3.1.2), the macro temporarily expands itself in a graphical view in order for the user to select the desired node to change. Since modifying a functional specification is only valid for a multiport or block, this expansion process continues in recursion until one of these types of nodes is selected.

3. Changing its external arcs.

The arcs that connect a macro node to the rest of the system have to be user definable and modifiable. For this reason, when the user adds an arc to the system and selects either its to_node or from_node and if one of these happen to be a macro node, the macro node will expand itself in order for the user to select one of its internal nodes. This is required because an arc must attach itself directly to a multiport or block for

meaning (see section 3.1.1). When selecting the `to_node` and `from_node` of an arc, the user is prohibited from selecting these nodes from inside the same macro node. Otherwise item 1. above would be violated.

Viewing a macro means to expand a macro in a graphical view in order to display its composition. This is required for both 2. and 3. above. This capability is also provided separately in order to inspect the composition of a system as demonstrated for the tractor in Figure 3.4.

The two wheel drive tractor requires two rear wheels that are the same. Making a macro out of a generic wheel allowed for reusability of the wheel macro. Inside the tractor model are four tires that are patterned after the same tire model. It is quickly determined that large macro libraries would be very beneficial for reusability of common models. The difference between generic macro models and specific ones would be in their functional specifications. For example, there are different size tires between the front and rear on the tractor. Providing functional modifications as described in 2. above makes the system very useful in this regard.

Finally, the radar pedestal shown in Figure 2.6 contains a field controlled DC motor. If one wants to change the field motor to a shunt motor in the bond graph processor all that needs to be done is:

- a. Read in the shunt motor macro from file. (Assuming this has already been created before hand.) For the user this

is handled like the paste options discussed in 3.1.1.

- b. Change the connecting arcs (labelled 'ctrl' and 'cl') from the field motor to the shunt motor.
- c. Remove the field motor from the graph.
- d. Modify the functional specifications of the default shunt motor to the specific one desired.

This capability demonstrates the ease in which testing various types of the same component (in this case motors) can be done when macros are available.

3.1.4 Enhancements: Topological and Functional

Providing a variety of node types and a large choice of functions to choose from is needed in a useful bond graph simulator. However, there will always be users who have special applications that require nodes and functions other than the ones provided. This section describes the process used to make these types of enhancements.

The functions assigned to a node are implemented in a way so that users can add, modify or remove functions from the system. This is done through a concept referred to as template functions.

The function definitions are stored in template form in text files that are read into the bond graph processor as needed. For example, the generic GAIN function template was shown in Figure 3.13. The DIODE function is shown in Figure 3.14 as another example.

```
DIODE
# Inputs = 1
# Outputs = 1
# Parameters = 2
if ( input1 >= 0.0 )
then output1 = parm1 * input1
else output1 = parm2 * input1
endif
```

[click here to continue](#)

Figure 3.14 Diode Function

Template functions are written by editing a text file formatted as shown in Figure 3.15, containing the following information:

- a. Function name
- b. Number of input variables. These variables are referred to as input1, input2, and so on.
- c. Number of output variables. These variables are referred to as output1, output2, and so on.

- d. Number of parameters. These parameters are referred to as parm1, parm2, and so on.
- e. The functional definition based on the input variables, output variables and parameters. This is currently written in a FORTRAN style of definition.

```

<name>
# Inputs - <n in>
# Outputs - <n out>
# Parameters - <n parms>
Text to define function.

```

Figure 3.15 Template Function Format

A text editor option is provided in the bond graph processor in order to ease the definition of the functions. However, these functions can be edited by any editor outside of the program and will be automatically incorporated into the system if they are put in the file directory in which OOBProc looks for them.

When used, the template files are read in from memory and the generic inputs, outputs and parameters are replaced by the specific ones associated with the node that uses the template.

The bond graph processor has no means for checking to see if the functions are 'correct' since complete definition of correctness would not come into focus until the system equations are actually solved, which is not part of this system. The user, however, can check to see if the functions are as

desired by assigning the functions to desired nodes and inspecting the resulting system equations. (This feature is discussed in section 3.2).

Although the user probably would not need to create new node types as often as new functions, the capability to add them efficiently and relatively easily is important. With object oriented programming this capability can be realized. Here we briefly discuss what is required for a new node. The implementation and associated details can be found in the next chapter and in the Appendix.

Adding a new node type involves three main acts: (1) Creating the node class; (2) defining its methods, and (3) incorporating the node into OOBProc.

For example, creating the source of flow multiport required: (1) Creating the class named Flow; (2) writing three methods (the rest of the Flow's behavior is inherited), and (3) modifying the method that defines the menu for OOBProc to include the class Flow.

3.2 Model Analysis/Equation Formulation

Once a description of the model exists in OOBProc model processing can be performed. Many procedures deal with converting the topological model into a mathematical model useful for numerical solution. However, some analysis can be done on the model from its topological description. Four important issues are:

1. Model Analysis
2. Equation Structure (sorting)
3. Coupled Algebraic Equations
4. Explicit/Implicit State Equations

Model analysis refers to performing operations on the model without necessarily generating any equations. For rigid body mechanisms the choice of coordinates to be used is very important and needs to be done in a meaningful way [61]. The analogy in bond graphs is the assignment of causality. This can be done with the standard sequential causality assignment procedure (SCAP) [24] or by other methods, such as Lagrangian bond graphs [62].

In electrical networks a proper choosing of a tree structure can lead to insights not apparent from the initial model. Similarly, assigning causality to a bond graph determines state variables and identifies dependent energy storage components. If a system has dependent energy storage components it is referred to as having differential causality. This means that the energy variables used to describe the system are not dynamically independent from each other. This information may suggest to an engineer where trouble spots might arise (or already exist).

Causality is displayed on the system graph as shown in the radar pedestal and tractor models. When macro models are used one can not tell directly if there exists differential causality. For this reason, OOBProc recognizes and notifies the user of cases where differential causality is assigned.

Additionally, macro model causality gives the engineer insight into how the different components interact with one another in a system context. For example, the engine in the tractor model (Figure 3.3) supplies an angular velocity to the clutch. In return, the engine 'feels' the torque due to the clutch.

Related to each node in the system graph are a set of equations. The equation set of all of the node equations in a bond-graph/block-diagram system is called the system equations. It is desirable to collect these equations such that input and output variables are identified (i.e., such that the equations are written as output variables are functions of input variables). In OOBProc this is handled by the use of causality during the functional specification stage (see section 3.1.2). Figure 3.16 shows a portion of the system equations as developed in OOBProc for the radar pedestal of Figure 2.5.

Two special cases that make the equation structure complex are systems that result in implicit equations and coupled algebraic equations. Implicit equations result from dependent energy storage elements (i.e., differential causality). Coupled algebraic equations result from coupled dissipation elements.

```

P.p2 = int(E.p2)
F.p2 = 0.003125 * P.p2

ctrl = -1 * w2
      + -1 * theta
      + 1.0 * ref

E.p3 = 10.67 * F.p3

E.e1 = 1.0 * ctrl

P.m2 = int(E.m2)
F.m2 = 4 * P.m2

E.m3 = 0.33 * F.m3

E.s1 = E.s2
E.s3 = E.s2
F.s2 = + F.s1 - F.s3

E.e3 = 5 * F.e3

theta = int( w1 )

P.e2 = int(E.e2)
F.e2 = 10 * P.e2

F.s3 = MOD.TF * F.p1
E.p1 = MOD.TF * E.s3
MOD.TF = 30

F.p3 = F.p2
F.p1 = F.p2
w2 = F.p2
w1 = F.p2

```

Figure 3.16 System Equations for the Radar Pedestal

OOBProc allows for both algebraic coupling and implicit equations in the stack of system equations. Actually solving these equations numerically is a difficult procedure. What can be supplied from the model is a set of checks to identify such equations. This is helpful because special solution techniques could then be used for processing them.

Once the system equations are analyzed and sorted they are ready for the next stage in the simulation process, the equation analysis and numerical/symbolic solution phase. This portion of the simulation process is left for future research as it relates to object oriented programming techniques.

In summary, this chapter has presented a detailed description of OOBProc developed in Smalltalk-80. The description has been made from a user's point-of-view. The features include a graphical environment, views to aid in functional specifications, topological macros, and user enhancements for new template functions and new simple node types. To date, all of these features have not been available in a single bond graph simulator system. However, some of these features have been implemented on a limited basis in existing systems.

CHAPTER IV

THE SMALLTALK-80 IMPLEMENTATION

This chapter discusses the object oriented implementation of the bond-graph/block-diagram processor (OOBProc) described in chapter III. The implementation has been broken down into three main areas, those being (1) object oriented design; (2) NodeGraph-80, and (3) the actual implementation.

Object oriented design is a methodology used to design a software system. Because system requirements change over time it is important that the design be flexible enough to support these changes. This is part of what is referred to as incremental development. The model-view-controller is Smalltalk-80's concept for organizing/designing an interactive user environment.

NodeGraph-80 is a software product written in Smalltalk-80 that supports generic graph modeling. This product was used as an initial base for OOBProc.

An overview of the design approach for the four phase implementation is described. Some detail is required here in order to set the stage for describing the benefits and problems associated with object oriented programming that will be discussed in later chapters. The details of the

object oriented implementation are given in Appendix C.

4.1 Object Oriented Design

4.1.1 Responsibility-Driven Design

Object oriented design is the term used for the design phases of a software system. According to Jacobson [63], using object oriented design techniques promotes a seamless transition from systems analysis all the way to the actual code. However, according to Wirfs-Brock [64], in order to achieve maximum benefits one must use a responsibility-driven approach to the design rather than a data-driven approach.

Data-driven design focuses on the data in the system and what algorithms are to be applied to that data. What this amounts to is the adaptation of abstract data type design to object oriented programming techniques. This type of design is a natural transition for procedural language programmers. The problem with this is, according to Wirfs-Brock:

"Even though the goal of data-driven design is to encapsulate data and algorithms, it inherently violates that encapsulation by making the structure of an object part of the definition of the object. This in turn leads to the definition of operations that reflect that structure (because they were designed with the structure in mind). Attempts to change the structure of an object transparently are destined to fail because other classes rely on that structure. This is the antithesis of encapsulation."

Responsibility-driven design concentrates on the actions and the information sharing requirements of the objects in a system (i.e., the

protocols of the objects). At this stage, no attempt is made to define the data structure of the objects. This means that the data of an object will be better encapsulated since the object is designed without knowledge of its data structure.

During the development of OOBProc the design was initially done by the data-driven method. As progress was made the designing switched to the responsibility-driven method. As a greater understanding of object oriented techniques was obtained the responsibility-driven method was found to work better.

4.1.2 Incremental Development

Initial design of a system is important but the software lifecycle requires that the resulting software be flexible with respect to changes [65]. This includes:

- a. Support for system requirement changes.
- b. Capabilities for software restructuring due to a greater understanding of the actual system as you continue to work with it.
- c. Implementing portions of the system during the design process but with enough flexibility that you would not become tied down to the proto-typed software

implementation.

The term incremental development is used to represent the above ideas.

The bond graph processor was implemented in four phases specifically to test object oriented programming's incremental development capabilities.

4.1.3 Interactive Applications: Model-View-Controller

Designing an interactive system in Smalltalk-80 requires the use of the Model-View-Controller (MVC) metaphor. Figure 4.1 shows a schematic of the MVC. The model is responsible for the application domain state and its behavior. The controller handles the user interaction. The controller receives requests from various input devices and processes them, generally sending specific application behavior messages to the model or viewing messages to its associated view. The view is responsible for displaying the model.

Associated with each view is a controller, each having exactly one model (usually the same one). A model may have multiple view/controller pairs.

Basically the view is designed around how you want to see and interact with the model. For example see the radar pedestal system graph in Figure 2.5. The view handles the actual displaying of the bond graph on the screen.

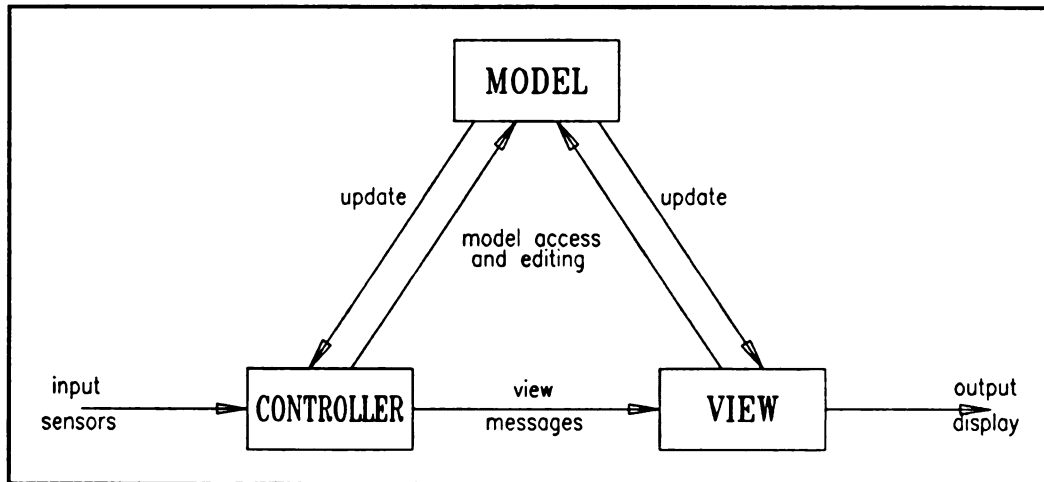


Figure 4.1 Model-View-Controller

The controller is designed around what you want to do to the model and to the display of the model. In a limited sense, the controller doesn't actually do the work, it processes the requests from the user.

The model concentrates on the fundamental state and behavior of an application, leaving the user interaction and display to the controller and view. The behavior of the model (or any object) is determined by what you want that model to do, or how that model behaves in real life.

The model-view-controller are three separate objects each with their own responsibilities, working together in an application. This concept follows the responsibility-driven method described in section 4.1.1 above.

4.2 NodeGraph-80

NodeGraph-80 is a user interface tool kit developed in Smalltalk-80 to provide a structural and user interface foundation for Smalltalk-80 applications which make use of directed node graph representations [66]. This tool kit can either be used as an application itself for the generation of various types of graphs, or as abstract classes to be subclassed and specialized in some way.

OOBProc was developed using NodeGraph-80 as a starting point. In order to cleanly separate new code from existing code, all NodeGraph-80 classes were subclassed to handle modifications and additions to the original system. All of NodeGraph-80's classes begin with the characters NG80. This helps in reading the class hierarchies presented when describing OOBProc (see Figure 4.2, for example). For the most part, the discussions that follow treat NodeGraph-80 as if it were part of Smalltalk-80 itself. Section 5.3 discusses the usefulness of NodeGraph-80 in the development of OOBProc.

4.3 The Implementation

4.3.1 Phase I: Primitive Bond Graph Processor

4.3.1.1 System Capabilities

Phase I consisted of developing a bond graph processor with the following capabilities/limits.

1. Handle bond graphs consisting of the basic primitive multiport nodes Se, 1, I, R and GY, and bonds.
2. Graphical user interface to input/modify the graph description.
3. Modify equation definitions for the Se, I, R and GY nodes.
Function types available: Constant, gain, linear, sin and backlash.
4. Assign causality to the bond graph. Retaining, if possible, node equation definitions when the bond graph is modified and causality is re-assigned.
5. Derive system equations based on causality.

6. Display capabilities

- Graphical representation of the bond graph, with and without causality.
- Dynamic placement and movement of graph components.
- System equation list.

4.3.1.2 Object Oriented Structure

Describing an object oriented system consists of describing the objects in the system and how these objects interact with each other. According to Goldberg [8] this is best done in Smalltalk-80 by browsing and running the actual software. Since this is not possible in a written document, the next best tool to use is the class hierarchy.

Figure 4.2 shows the relevant portion of the class hierarchy for the Model-View-Controller (MVC) of phase I. BGGraph, BGView and BGController are the classes created for the model, view and controller of phase I, respectively. Inheritance is read from top to bottom following the lines. For example, Object is the top level class. All other classes are subclasses of Object and inherit all of the properties (data structure and behavior) of Object. As another example, class BGGraph inherits from Object, NG80Object, BGObject and NG80Graph.

A running bond graph processor results when an instance of the MVC is created. The instances of the appropriate classes are called objects.

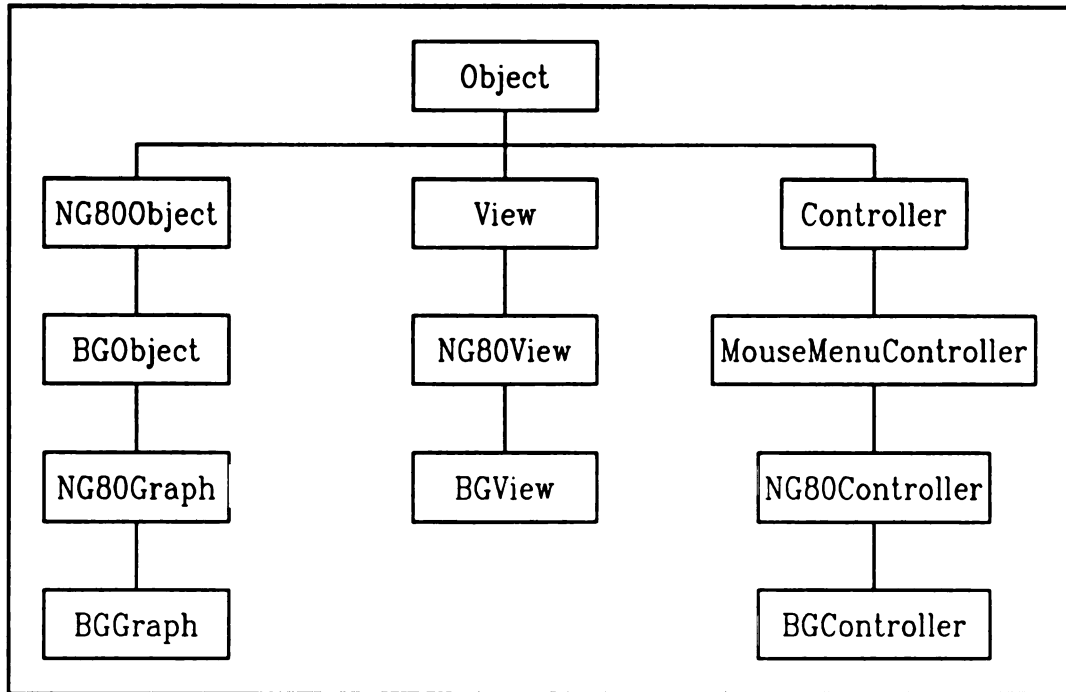


Figure 4.2 MVC Class Hierarchy

The controller object, an instance of `BGController`, handles all user interactions. Associated with this object are the menus presented to the user for running the system. When a menu option is chosen this object receives the message associated with that menu option. The controller then processes the request. This is done by the methods of `BGController`. An example of this will be given shortly.

The view object, an instance of `BGView`, is responsible for a graphical display of its associated models bond graph description. Figure 3.3 shows an example of a `BGView` object for the tractor model. The view must keep track of things like its size, its location on the terminal screen, where different objects are to be displayed on it, and how to display objects.

The model object, an instance of BGGraph, handles the behavior and storage of the actual bond graph. This is designed by first asking the question of "What do you want the graph to be able to do?". This corresponds to responsibility-driven design discussed in a previous section. That is, first determine the behavior and then design its implementation.

The behavior of the model includes:

1. Adding, removing, and modifying nodes and arcs.
2. Assigning and clearing causality.
3. Accumulating system equations.

This helps determine the protocol of the BGGraph class. The implementation then requires data structuring to further refine objects that compose a bond graph. Figure 4.3 shows the main class hierarchy created during phase I for this purpose. Many classes supplied with the original system were also used extensively by these new classes by means of instance variables and temporary variables. In particular, collection sub-classes (Dictionary, TextCollection, OrderedCollection and String) were used for storing relationships between objects and for attributes of objects.

An abbreviated version of how this was implemented in OOBProc is as follows. Bond graphs consist of nodes and arcs. Thus, there are two collection variables associated with a BGGraph object, one for nodes and one for arcs. These variables are called instance variables.

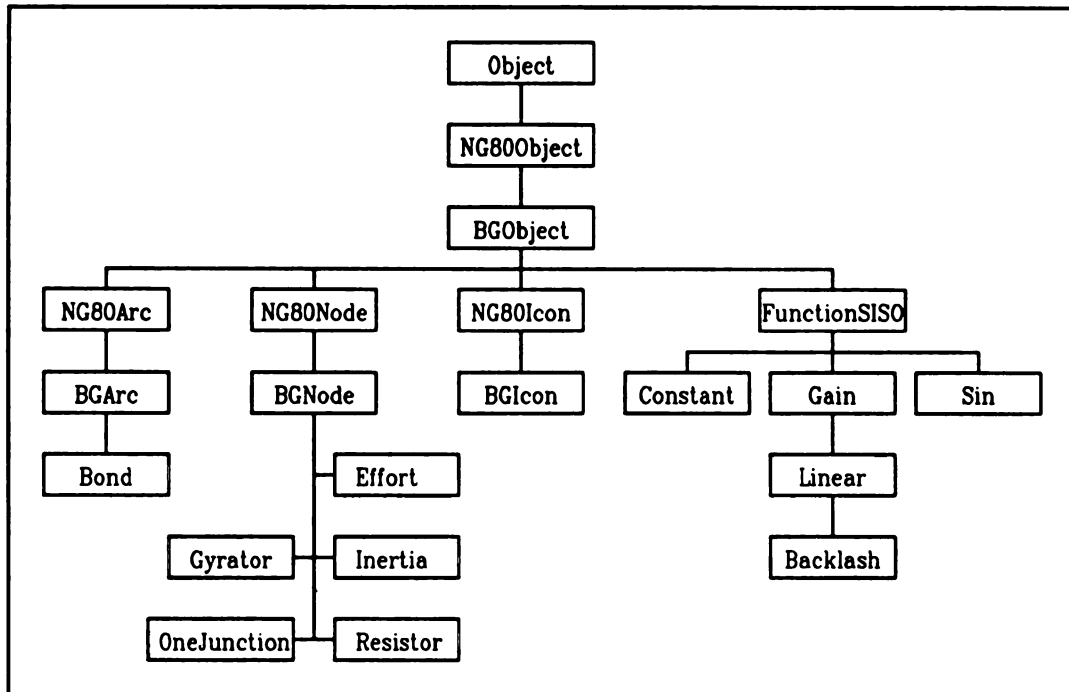


Figure 4.3 Phase I Class Hierarchy

In phase I there was only one type of arc, namely a bond. This was implemented by the class **Bond**. **Bond** inherits general arc behavior from class **BGArc**. For example, arcs have two ends attached to nodes, referred to as the `from_node` and `to_node`. (For our purposes we are considering only directed arcs.) Thus **BGArc** is responsible for storing and accessing these nodes. The **Bond** class implements specific behavior related to a bond, such as causality.

There were five different nodes in phase I (classes **Effort**, **Gyration**, **Inertia**, **OneJunction** and **Resistor**). These share common attributes that are captured by the class **BGNode**. One of these attributes is an associated graphical picture of the node. This graphic is handled by the class **BGIcon**. Another common attribute is an associated function (or functions)

with each node. The icon and function of a node are stored as instance variables for that node.

Specific behavior of these nodes not shared by each other are, for example, the handling of different causality assignments and different input/output variables of the node. This behavior is implemented by the individual node types.

Abstract class FunctionSISO handles common properties of single-input/single-output functions. Specific functions that inherit from FunctionSISO are handled by the classes Constant, Sin, Gain, Linear and Backlash. These objects can be used by nodes or by any other object that requires a function definition. This is an example of encapsulation and reusability as defined in object oriented programming.

The class hierarchy is very useful but it does not indicate how the various objects behave and interact with each other in an actual running environment. The Smalltalk-80 environment provides many tools for tracking object behavior. Currently, there is no clean and precise way to describe this interaction similar to the class hierarchies structure outside of the software environment itself.

The following is a condensed version of object interactions occurring during the addition of an inertia to the bond graph.

1. User selects Inertia from the add multiport menu.
2. The BGController will receive the message 'addNode: Inertia'. The method with the same name will control the addition of the inertia. For reference, this method is shown in Figure 4.4.
3. The message 'getUniqueLabel:' is sent in order to retrieve a unique name for the new node. The model (i.e., the BGGraph object) will be consulted in this process in order to check for uniqueness.
4. Next, the new inertia will be created and initialized (an instance of class Inertia). The inertia will set its label and create its graphical icon.
5. The model is then sent the message 'addNode: inertiaObject'. The model will add the node to its node collection, clear causality and broadcast to its views that it has been changed. The BGView object will add the inertias icon to the view at a default location.
6. Finally, the BGController will send the message 'placeNode: inertiaObject' to itself in order to let the user dynamically place the inertia at the desired location. 'placeNode:' will interact with the BGView for this.


```

addNode: aType
    "Add a node of type aType to my model."

    | label node |

    "1. Get a unique label for the new node."
    label <- self getUniqueLabel:
        ('Enter the label for the new ' + (aType asString)).

    label isNil
        ifTrue:
            [self model messageHandling: 401
                from: ' the new ' + (aType asString).
                view flash.
                ^nil].

    "2. Create the new node and initialize it."
    node <- (Smalltalk at: aType) new.
    node initialize.
    node changeLabel: label withCRs.

    "3. Add the node to my model."
    self model addNode: node.

    "4. Dynamically place the node on my view."
    self placeNode: node.

    ^node

```

Figure 4.4 BGController 'addNode:' Method

4.3.2 Phase II: Additional Multiports and Functions

The objectives of Phase II were to enhance the capabilities of the bond graph processor developed in phase I by adding additional bond graph nodes and functional capabilities.

4.3.2.1 System Capabilities

The bond graph nodes added in phase II were the source of flow, 0-Junction, capacitor and transformer. This was done from two perspectives:

- a. Macro - A capacitor can be composed by combining a gyrator and an inertia.
- b. Atomic - A capacitor can also be defined independently.

Also, additional node functions were investigated from a user perspective. A common request by many dynamic system analysts is the ability to define specialized functions for standard components. For example, a linear spring function is given by $F = kx$. A researcher might desire to test a spring with a function given by $F = kx^3$. If that function is not supplied by the original software the user will want to add it to the system.

4.3.2.2 Adding Multiports

The addition of the new multiport elements as individual classes can be summarized in Figure 4.5.

Type	Patterned after	New methods	Modifications required	Time
Capacitor	Inertia	5	1 method	1 hr
Flow	Effort	3	1 method	30 min
ZeroJunction	OneJunction	5	1 method	1 hr
Transformer	Gyrator	7	1 method	2.5 hr

Figure 4.5 Summary of Atomic Node Additions

The meaning of the table titles are as follows.

Type

This is the name used for the new class which matches its multiport type.

Patterned After

Each new class had very similar behavior to an existing class. Methods were copied from the patterned after class listed and modified to handle the new class (thus making new methods). Although this sounds like subclassing, it actually isn't for this application. For example, the inertia element desires causality towards it and works with flow and momentum variables, while the capacitor element desires causality away from it and works with effort and displacement variables.

Their methods look similar in handling behavior, but one could not inherit from the other.

New Methods

This is the number of methods written for the new class.

Modification Required

The existing system had to be modified minimally to add the new classes. For each new class, only one method had to be changed. This was the initialize method in the BGController class. This method contains the list of allowable node types from which the menu is created for the user to pick from. One line is added to this method for each new node type added to the system.

Time

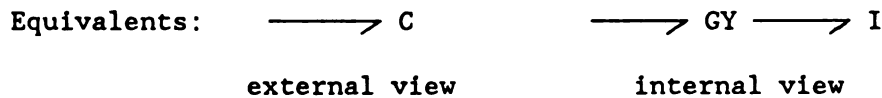
This was the time it took to program and test (on a small scale) the new node.

The "patterned after" design concept was devised due to familiarity with the bond graph methodology.

The next step is to determine what it would take to add these new node types from a macro perspective. The results from this portion determined that a macro implementation of these basic node types is not the best approach in an object oriented environment. This will be shown by looking at the capacitor element.

This is not to imply that macros are not needed. In fact, they are a very useful tool for dealing with large systems composed of many components, each with its own complex definitions. This viewpoint of macros is deferred until phase IV of the project.

As a macro node the capacitor would behave as a standard bond graph element to the outside world. But internally it would be composed of different atoms, specifically, a gyrator connected by a bond to an inertia. This is depicted as:



An implementation of the macro capacitor could have a class hierarchy as shown in Figure 4.6.

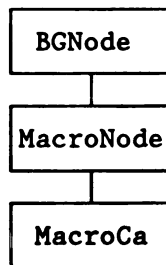


Figure 4.6 Macro Capacitor Class Hierarchy

The MacroNode class would contain all of the methods required to store a macro composition and the methods to distribute messages to the decomposed elements. As discussed above, this capability will be developed during phase IV. This means that this class would not play a direct role in determining the effectiveness of a macro capacitor.

The MacroCa class represents the capacitor in its gyrator-inertia form. This class would require the following:

- A new class added to the class hierarchy.
- Methods to create the GY-In composition.
- Methods to handle the user interfaces when talking about the capacitor. For example, the engineer will refer to a capacitor in terms of two types of equations:
 1. effort = $f(\text{displacement})$ and
 2. $d/dt(\text{displacement})$ = flow.

The GY-In equivalent will have to convert this to its stored representation. Specifically, for the inertia:

1. flow = $f(\text{momentum})$ and
2. $d/dt(\text{momentum})$ = effort.

These methods would handle the differences between the external and internal viewpoints of the capacitor.

- Just as in the atomic version, the method that handles the menu for node choices will have to be modified.

Note: The macro version of the capacitor still has a minimal effect on the existing system.

A comparison between the atomic capacitor and the macro capacitor can now be made in four different categories.

1. Storage

The storage (or run time) memory requirements would be similar. However, it appears the atomic version would take less space due to the complexity of trying to coordinate both the external and internal views of the macro version.

2. Execution Time

Atomic version would run much faster on all but trivial messages received.

3. Development-Implementation Time

Atomic capacitor took less than 1 hour to code and test. Work on the macro capacitor lasted for about three hours without any code written.

4. Code Readability-Understandability

The atomic version would be much easier to comprehend. The atomic capacitor methods show how the capacitor actually behaves while the macro capacitor basically does transformations between external and internal perspectives.

If the storage requirements are similar, and the execution time, the development-implementation time and the code readability-understandability are in favor of the atomic nodes, then it is appropriate to conclude that,

for this object oriented application, atomic versions of the standard bond graph elements are superior to the macro versions.

4.3.2.3 Additional Function Capabilities

The functions available for the mathematical definitions of the node objects after phase I was straight forward. Each function type (gain, sin, etc.) had its own class. Subclassing was used to inherit instance variables, but methods to define the equations for each function could not be inherited. The extension to this is to just add more classes. However, this turns out to be inefficient and not desired from a users perspective. The two reasons for this are:

1. Each new function will require a new class and anywhere from one to five new methods. If the function library grows to an expected 50+ functions, one can see that the run time storage requirements are not ideal. (This makes a big difference when working on a PC.)
2. For the engineer to add a new class, he/she would have to learn some of the object oriented language. This could be held to a minimum by automatic class creation and some standard instance variable handling methods (such as setting and retrieving). However, the actual function definition methods would still have to be written by the engineer. The number of methods would be small (one to

three), but they would require knowledge of the object oriented methodology and portions of the current implementation. This applies particularly to implementations of the bond graph, node and bond objects.

To solve these problems, a new implementation of the functions was done during phase II. The old class hierarchy dealing with functions was removed and a totally different one was added to the system (see Figure 4.7).

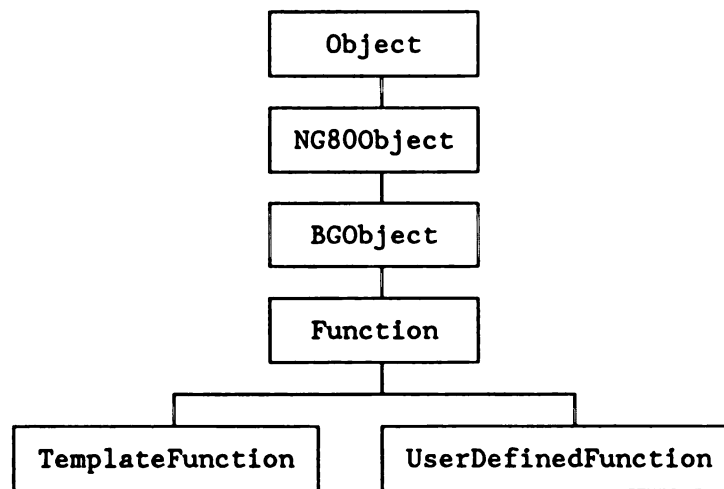


Figure 4.7 Phase II Function Class Hierarchy

The two types of functions identified were pre-defined functions and specialized functions (those functions one might use for testing but not needed in a function library for other use). Each of these types are defined by separate classes.

The pre-defined functions (or template functions, class TemplateFunction) consist of inputs, outputs, parameters and a math specification of how

these define the specific function. This behavior is handled generically by methods of the TemplateFunction class. The specifics for each function type are stored in a standard ASCII text file. This immediately does three things:

1. Hundreds of specific function types can be defined without effecting the run time memory requirements.
2. Engineers can create new function types without knowing anything about the object oriented language. All they need to learn is the format of the ASCII text files. From there they can:
 - a. Use any text editor to define new functions.
 - b. Use the create/modify function option developed in phase II that simplifies the task.
3. Allows for multiple-input/multiple-output functions (previously only single-input/single-output functions were allowed).

The specialized functions (user defined functions, class UserDefinedFunction) are used for the engineer to specify functions exactly as desired. The engineer is responsible for correctness of all input and output variables and associated parameters. When defining one of these objects a text editing window handles a user definition

interface. This also has two benefits:

1. No object oriented programming knowledge is required.
2. New functions can be tested easily.

To increase the usefulness of these function classes, methods were developed to convert between the two function types. For example, if you have a user defined function object and you decide that you would like it as a template function, a conversion process is included to aid in the process. No object oriented programming is required, just knowledge of the text file format requirements.

The only modifications required of the existing system in order to incorporate these new functions were to the node classes. These changes consisted of:

1. Modify node methods `assignFunction`. The variables for input, output and parameters had to be changed to collections in order to be compatible with the new `TemplateFunction` class. This would have been required eventually if multiple-input/multiple-output functions were ever desired.
2. Modify node methods `modifyFunction`. This method was changed to handle the difference when the user switches a node's function from a template function to a user defined function (or the reverse).

4.3.3 Phase III: Block Diagrams and View-Controller Pairs

4.3.3.1 System Capabilities

The first part of phase III consisted of adding block diagram capabilities to OOBProc. Block diagrams consist of blocks and signals (for example, see Figure 2.1). Blocks represent functional operations based on signal inputs. Signals represent a single variable information transfer process from one node to another. Adding block diagrams allows, for example, feedback control mechanisms to be added to bond graph models.

The second part of phase III consisted of adding two new view-controller pairs in order to further investigate the MVC concept for interactive applications. This helped to further define what constitutes the actual model versus how to interact with and display that model.

4.3.3.2 Signals

Signals are a specific type of arc, just as bonds are. Thus adding signals involves subclassing the BGArc class, as shown in Figure 4.8. The Signal class required just five new methods, three of which are used for displaying the signal.

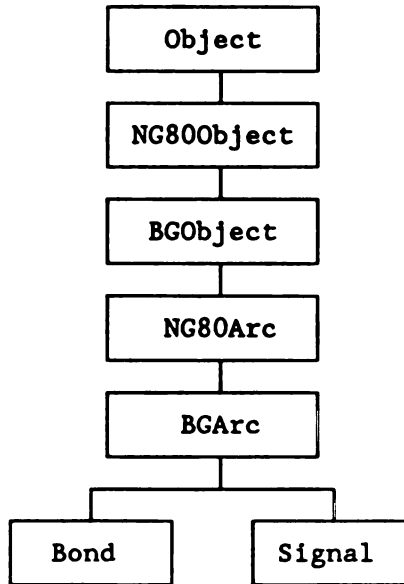


Figure 4.8 Arc Class Hierarchy

Adding the Signal class had a far greater effect on the system than adding new multiports. The greatest effect was on the nodes. The main differences was in how the nodes were going to handle different types of arcs connected to them.

For example, there are restrictions on an arc as to what node types it may be attached to. A signal's `from_node` must be either a block, a 1-junction or a 0-junction. Additionally, a signal's `to_node` must either be a block or a multiport of type Se, Sf, R, C, I, TF or GY. Bonds can only be attached to multiports. To satisfy these restrictions one could either let the arcs decide if they are valid for the particular node or let the node decide if the particular arc is valid. Both ways were tested and it was determined that a few simple methods for the nodes could efficiently check the proper restrictions. An example for this will be given when the blocks are discussed.

Another node change consisted of defining two instance variables for the nodes, called `bondPorts` and `signalPorts`. These store the node's attached bonds and signals, respectively. At first, the attached bonds and signals were stored in the same instance variable, but it became rather cumbersome to keep searching this variable to separate the signals and bonds. Due to encapsulation, changing the storage of the nodes ports was an internal manner to the nodes and thus did not require changes throughout the system. This did however, require two new methods and seven modified methods within the `BGNode` class.

Finally, adding signals helped to refine the `BGArc` class to be more generic. For example, reversing an arc might seem to be something that all arcs would want to do in a general manner. However, for this system reversing a signal is not allowed because signals, as used in block diagrams, are very much one directional and changing that direction would rarely make sense. Also, reversing a bond has an influence on its attached node's equations. Thus, reversing an arc is handled only by the specific subclasses and not the abstract classes (e.g., `BGArc`) as originally designed.

4.3.3.3 Blocks

Adding blocks to `OOBProc` resulted in a new abstract layer of classes for nodes (see Figure 4.9). The classes `Block` and `Multiport` were added to handle abstract behavior of blocks and multiports, respectively, that could not be captured by the single class `BGNode`.

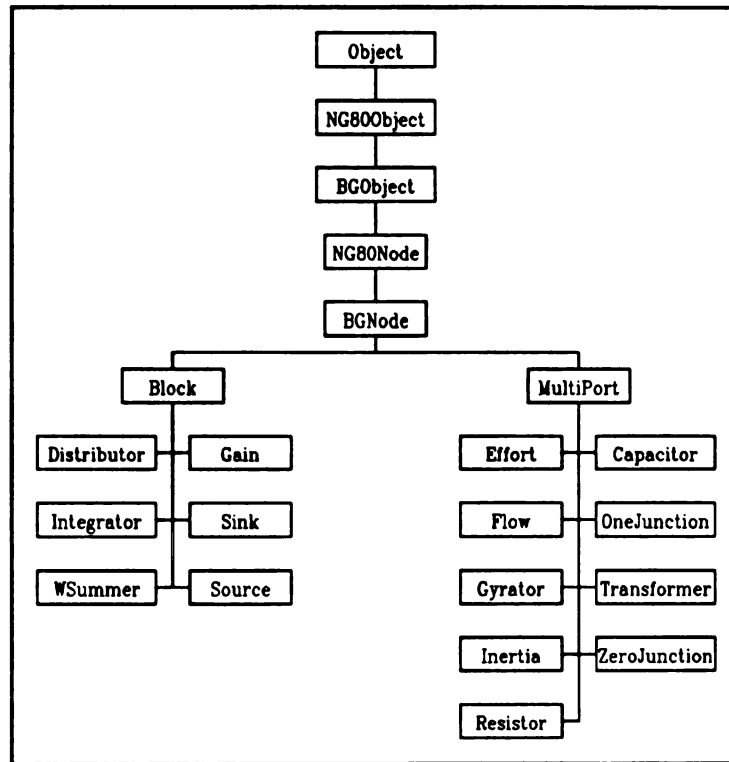


Figure 4.9 Node Class Hierarchy

For example, refer to the restrictions placed on an arc's `from_node` and `to_node` discussed previously. Both classes, `Block` and `Multiport`, implement the method '`checkIfBondIsValidPort`'. If a block object receives this message then it will return false. Conversely, if a multiport object receives this message it will return true. This is because a bond can only be attached to a multiport. An important point here concerning object oriented programming is that nowhere in this code is an `if..then..else..` statement to check if the response should be true or false. Each object knows its own behavior and does not need to perform control statements.

Another example is the icon type associated with a node. A blocks icon is

a box around its label, while a multiport icon is just its label defined within an invisible rectangle.

Due to the simplicity of a block, the input and output variables from any specific block can be handled by the Block class. Recall that each specific multiport had to determine its own input and output variables.

The specific block types, subclasses of the Block class, shown in Figure 4.9, required a total of ten new methods for all classes combined. For example, the 'assignFunction' method required re-defining by most of the Block subclasses. The default function for a block is the gain. Classes Distributor, Integrator and WSummer all re-defined this default to match their appropriate function.

The node and arc class hierarchy changes had no effect on the functions that the nodes use and a minor affect on the BGGraph which use the arcs and nodes as part of its definition. The BGGraph class changes were actually further refinements of the object oriented methodology.

For example, removing a node requires the removal of all of its attached arcs. Previously, the BGGraph method 'removeNode:' sent the message 'removeArc:' to each arc listed in the nodes ports list. By separating port collection of a node into signalPorts and bondPorts, it was discovered that the previous method was actually accessing the nodes data structure and thus violating encapsulation. The correction was to have the BGGraph method ask the node for its connected arcs, leaving the implementation details to the node.

Finally, the changes to the view-controller consisted of: (1) menu changes to include the blocks and arcs (1 method); (2) enforcing the restrictions associated with an arcs from_node and to_node (5 methods), and (3) generalizing the methods associated with adding, removing and modifying nodes so as not to be dependent upon the node type (3 methods).

4.3.3.4 View-Controller Pairs

Two new view-controller pairs were added to OOBProc in order to further refine the distinctions between the model, view and controller that constitute the MVC concept. These consisted of a node and an arc view-controller. See Figure 4.10 for example. The node view lists the nodes by type followed by their label. This is done similarly for the arc view.

The graphical view-controller works under the concept of first picking a menu option and then picking the appropriate elements to apply the method to. The node and arc view-controllers work under the concept of first picking the appropriate element and then picking what to do with that element from an associated menu.

This experiment helped to refine two important concepts of the MVC. Those were the responsibility of the controller and broadcasting model changes to the views for updating.

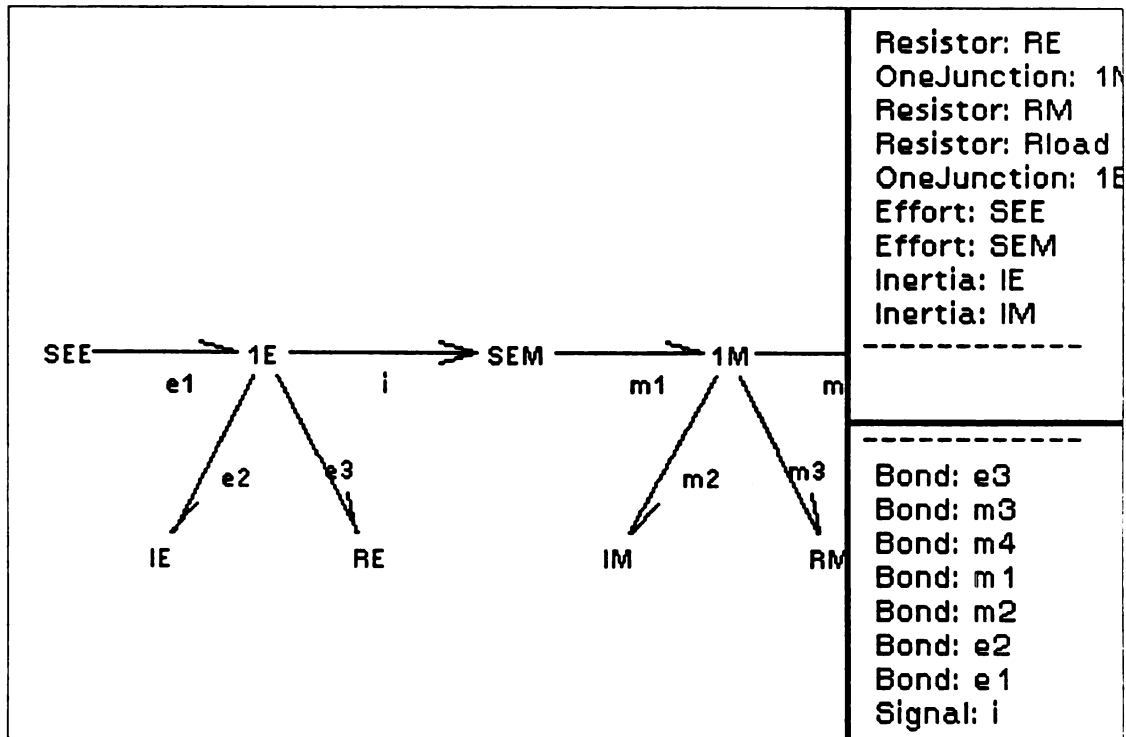


Figure 4.10 Multiple Views of a Bond Graph

For example, removing a node is possible from both the graph view and the node view. Their respective controllers should have minimally duplicated code to handle this task. Previously, without the node view-controller, the graph controller was processing part of the removal process. The node view-controller addition clarified for the BGGraph class (the model) what was its responsibility in removing a node.

When there was only one view, the graphical view, model changes obviously would affect only that view. This sometimes led to having the controller update the view rather than the model broadcasting a change to that single view. Although this might be considered 'sloppy' programming, it is not at all obvious when this occurs if you don't have multiple views to ensure proper updating.

4.3.4 Phase IV: Macros

Implementing topological macros was the final development stage of OOBProc. As discussed in chapter III, topological macros are used to build up complex systems from simpler systems.

A macro is a component in a system graph that consists of multiports, blocks, bonds, signals and other macro components. A macro can not contain itself.

A macro is similar to a node in the following ways:

1. It requires a unique label.
2. It requires a graphical representation.
3. Causality assignment is meaningful to it, through its multiports and bonds.
4. Functional assignments are meaningful to it, through its nodes.
5. It contains a set of ports (i.e., connection points to the outside environment).
6. A set of system equations is associated with it.

Another argument could be made about how the macro behaves very similarly to a graph, that being:

1. It contains a collection of nodes and arcs.
2. Assigning causality requires individual assignment of its multiports.
3. Getting the system equations requires collecting them from its individual nodes.

From this, it appears that multiple inheritance may apply for the macro. That is, the macro component could be a subclass of both BGGraph and BGNode. Unfortunately, Smalltalk-80 does not support multiple inheritance, so this could not be tested.

It was decided to implement the macro as a node, class MacroNode, a subclass of BGNode. There were two main reasons for this were. First, conceptually a graph consists of nodes and arcs. If a graph contains macros then it is reasonable to think of the macro as a node. Secondly, a majority of the user interfaces required for nodes can be used as is for macros if they are treated as nodes. Since model-view-controller implementations are rather complex, re-use of the current interface is desired. However, macros do require additional user interface tools.

The required modifications to the existing system in order to incorporate macros fall into two groups. Those are further refinement of the

encapsulation of each object and additional user interface features.

In the previous section it was discussed how multiple view-controller pairs helped to refine the responsibilities of the model, views and controllers. Similarly, macro nodes helped to refine the responsibility of the graph model and the nodes and arcs that compose the graph. For example, a few methods in the BGGraph class were found to assume the structure of the nodes that define the graph. With the addition of the MacroNode class these methods had to be modified to request behavior from its nodes rather than using their assumed data structure. Once again, this is a matter of responsibility driven design versus data-driven design discussed in section 4.1.1.

The user interface (view-controllers) needed new methods to display and access the internal structure of the macro node. For example, to modify a nodes functional specification an atomic node needs to be selected. If the user selects a macro node, the internal structure of that macro is presented so as to allow the user to select one of its nodes. This process continues until the user selects an atomic node.

To handle the different selection of nodes, the method to select a node needed to be reconstructed as three separate functions. Those were (1) to select an atomic node, as in `modify_function`; (2) to select a macro node, as in `view_macro_structure`; and (3) to select either a macro or atomic node, as in `delete_node`. Each node menu option had to be examined to see which of these three methods to use for selecting a node. The appropriate modifications were then made.

In summary, this chapter has presented some of the details of the object oriented implementation of OOBProc. The implementation was done in four separate phases. It was shown that incorporating small modifications, like adding new functions or node types, and large enhancements, like block diagrams and macros, are possible with only minor disruptions to the existing system. These and other observations made during this research will be discussed in the next two chapters.

CHAPTER V
CHARACTERISTICS OF OBJECT ORIENTED PROGRAMMING IN
DYNAMIC PHYSICAL SYSTEM SIMULATION

Chapters III and IV demonstrated that object oriented programming can be used to implement a fairly extensive bond graph processor. As was pointed out in chapter I, the goal of the research is to determine the advantages and disadvantages of doing so. This chapter details some of the benefits of using object oriented programming relative to a more conventional language like FORTRAN or C.

The concentrated effort was on developing a bond graph processor using Smalltalk-80 versus FORTRAN. Generalizations are made, where possible, to extend comparisons to include general programming of dynamic physical system simulations using object oriented techniques.

The first section (5.1) discusses the software development process and the tools supplied by the languages that support this process. Section 5.2 explains how object oriented programming supports user customized software in an efficient manner. The reusability of software components is then discussed in section 5.3.

The first phase of the bond graph processor was programmed in the object oriented language Actor. Section 5.4 discusses how the transition from

Actor to Smalltalk-80 was made easy by using object oriented design techniques. Finally, a few qualitative reasons for using object oriented programming are given in section 5.5.

The theme developed in this chapter is that object oriented languages provide good tools and constructs to help structure an actual software implementation, while FORTRAN-like languages do not. The value of this structure is pointed out through the benefits detailed in this chapter. They include: (1) a means for constructing an efficient and consistent user interface; (2) storing and using data; (3) software organization; (4) user-customizable software; (5) controlled modifications; (6) a software developing environment unmatched in FORTRAN; (7) a reduction of possible errors in the system, and (8) reusable software components from project to project. The problems associated with these tools and constructs are presented in chapter VI.

5.1 Development

Developing simulation software is a complex and time consuming process. The language chosen to write the software must provide the means for effective development. This includes: (1) language constructs for things such as user interfaces and data storage; (2) a flexible environment with tools to write, test and modify code; (3) robustness, meaning not only ways to find and eliminate errors but to avoid them in the first place, and (4) productivity, meaning programmers should be able to produce quality software in short periods of time (i.e., efficiently).

It is shown in this section that object oriented languages provide these capabilities on a greater scale than FORTRAN does. Specifically, Smalltalk-80 is shown to provide tools for software development that are far superior to FORTRAN's.

5.1.1 Language Constructs

A software language provides means for developing user interfaces, storing and using data, and organizing the software. This section explains how object oriented languages can provide very useful aids in handling these tasks while FORTRAN like languages give the developer practically no help in these areas.

5.1.1.1 Processing Input Requests

Interactive software written in any language requires some type of user interface. This is the portion of the code that intercepts user inputs and calls the appropriate code to perform the requested action. In most applications, menus are organized in levels so that a choice in one menu brings up another menu. A language that provides capabilities to effectively handle this process, which is often complex, would have a clear advantage over one that does not. Smalltalk-80 provides this capability; FORTRAN does not.

In FORTRAN one has basically two choices: (1) Write your own menu

5.1.1.2 Interactive Windowing

Multiple displays on the screen at the same time are valuable in simulation software. With multiple windows you could, for example, show things like: a graphical view of your model, a display of one of the node's functions (using multiple views for it), and a display for prompting the user for input to modify the function, all at the same time. Designing interactive software without these features today is not very common.

Unfortunately, FORTRAN does not provide any direct means for providing interactive windowing. To write these features using FORTRAN is a formidable task. In fact, it is not possible without actually calling machine code (which you would have to write). Calling a windowing software package is a possibility, but then you would have to learn another language. Integrating distinct software languages is not recommended practice.

As described in the previous chapter, Smalltalk-80 provides what is called the Model-View-Controller (MVC). It is an implementation of an interactive windowing environment. It is written totally in Smalltalk and the developer is provided with the several dozen classes that define it. A new language does not have to be learned by the programmer.

Coupling interactive windowing and processing input requests for each separate window becomes quite a complex task. However, this is required

processor with multiple if...then...elseif statements spread around dozens of subroutines along with the use of multiple Read/Write/Format statements or (2) call specially designed packages (languages in themselves) that handle user interfaces (for example, Microsoft Windows or X-Window^{Xx}. Neither option is particularly attractive to someone faced with the task of developing a new user interface.

Smalltalk-80 provides multiple classes that are designed specifically to handle user interrupts. Writing a complex hierarchical menu requires one method that basically lays out the menu with regard to its appearance. When the user drags the mouse (interactive pointing device) over the menu, if the mouse points to a high level menu option (those with sub-menus) the sub-menu is automatically brought up onto the screen. Thus, no processing is required from the developer to change between levels of the menu hierarchy.

Associated with each low level menu option (one that requires specific action) is a key word. When the user selects that option with the mouse the key word is sent as a parameter to the method responsible for the whole menu. This method is often extremely short due to a system method called 'perform:'. This perform: method will cause the execution of any method whose name is passed to it as a parameter. Thus, if the key word associated with the low level menu option is the same name as the method you want performed because of its choice, then basically all you need to do to 'process' the menu is to send the message 'perform: keyword'. Essentially no checks (if...then..elseif) are required to process the users requests.

5.1.1.2 Interactive Windowing

Multiple displays on the screen at the same time are valuable in simulation software. With multiple windows you could, for example, show things like: a graphical view of your model, a display of one of the node's functions (using multiple views for it), and a display for prompting the user for input to modify the function, all at the same time. Designing interactive software without these features today is not very common.

Unfortunately, FORTRAN does not provide any direct means for providing interactive windowing. To write these features using FORTRAN is a formidable task. In fact, it is not possible without actually calling machine code (which you would have to write). Calling a windowing software package is a possibility, but then you would have to learn another language. Integrating distinct software languages is not recommended practice.

As described in the previous chapter, Smalltalk-80 provides what is called the Model-View-Controller (MVC). It is an implementation of an interactive windowing environment. It is written totally in Smalltalk and the developer is provided with the several dozen classes that define it. A new language does not have to be learned by the programmer.

Coupling interactive windowing and processing input requests for each separate window becomes quite a complex task. However, this is required

because that is what constitutes the user interface, a critical part of any software package. Since Smalltalk-80 provides both capabilities individually the integration is not only feasible, it is one of the major strengths of the language. That is, it provides an interactive environment for both developers and users. FORTRAN is sorely missing these features. (The interactive development environment will be discussed in detail later.)

5.1.1.3 Data Storage and Usage

Storing and using data is an important part of any software package. In a bond graph processor the graph description, as described in the previous chapter, must be completely stored and any portions of the graph must be retrievable by the code that is to operate on it. For example, to change a node's function the software must easily retrieve the data representing the specific node to be changed. The way in which a language stores the data and how the data is retrieved can make a big difference in the complexity of the code that performs operations on the data.

The tools to do this are provided in FORTRAN. They include common blocks and passing parameters. Common blocks store groups of data as defined by the programmer. There is no formalized organization for the groups and data names are limited to seven characters. This makes organizing and retrieving the data in an orderly fashion very cumbersome. For example, 'What data is stored where?', is continuously asked by programmers not extremely familiar with the software. The usage of common block data also

creates another problem in that every subroutine can have access to the data enabled. Keeping control over how the data is used and modified is essentially impossible. Modifications to the system data structure can have effects in literally hundreds of subroutines in a large package.

Here we have hit upon object oriented programming's number one strength: encapsulation. First, data is stored in groups known as objects. While there is no guarantee that the objects make sense as groups of data, if you program using object oriented design concepts as described in the previous chapter then the objects will hopefully make sense and be easily understood by other programmers. For example, storing a bond graph as a collection of nodes and arcs is quite natural. As another example, storing a simple node in terms of its name, its type, its attached arcs and its mathematical definition is logical.

Second, the methods that operate on the data are also stored with the data in the object. This means that common blocks are not needed in object oriented programming. However, parameters are still passed between code segments but are greatly reduced because you no longer need to send parameters that deal with outputs from a subroutine or switches that tell the subroutine what to do (very common in FORTRAN call statements). For example, when requesting the system equations from a bond graph you could simply send the message 'getSystemEquations' to the bond graph object. You would not have to locate all the common blocks that store the required data (e.g., the nodes, arcs, connections, functions associated with the nodes, etc) because the bond graph object stores its definition itself. But this is not overly complex. The bond graph is composed of nodes, so

all it has to do is send the same message 'getSystemEquations' to its nodes and collect the results. The nodes would then be responsible for their portion of the process. Each object is responsible for itself and need not concern itself with the data storage and usage of all the other objects in the system.

This last point means that modifications to the system can be localized and not have far reaching effects on the system as was pointed out in FORTRAN systems above. The effects of modifications will be discussed again when looking at the programmer's development environment.

By examining a portion of the node structure it can be reasoned that storing data as objects is easier to define, work with and understand. Figures 5.1 and 5.2 show the data associated with nodes in general and some specific nodes used in the radar pedestal example, respectively.

Name	String name for ease of communication with the user.
Type	Specific type (e.g., capacitor or inertia). An object doesn't really store this data because this is what it is.
Regime	Examples: M - mechanical, E - electrical, H - hydraulic
Icon	A graphical picture of the node to be used for display. FORTRAN doesn't have icon-like descriptions. It would probably have to draw each node each time it is to be displayed.
Ports	List of the attached bonds and signals.
Function	The function(s) associated with the node. The data associated with the function follows. FORTRAN would store these separately. Smalltalk-80 stores the function as one object.
Type	Function type (e.g., gain, polynomial, sin, summer).
Outputs	Output variable(s) from the function(s).
Inputs	Input variable(s) to the function(s).
Parameters	Parameters used in the function(s).

Figure 5.1 Generic Data Defining a Node

	Name	Type	Regime	Icon	Ports	Functions			
						Type	Outputs	Inputs	Parameters
Node1	SEE	Effort	Electrical	aForm1	ctrl, e1	Gain	E.e1	ctrl	20
Node2	Controller	WSummer	General	aForm2	ref, theta, w2, ctrl	Summer	ctrl	ref, w2 theta	1, -1, -1
Node3	CS	Capacitor	Rotational	aForm3	s2	Gain	E.s2	Q.s2	5000

Figure 5.2 Data Defining Specific Nodes

The horizontal rows in Figure 5.2 represent each node. The vertical columns represent specific data attributes. Due to FORTRAN's limited data types and encapsulation (e.g., reals, integers, logicals, strings, and lists or arrays, as long as the members all have the same data type) the only practical way to store the data is by the columns shown. Often, this is not very useful. For example, what good is the list containing the regimes for each node? By itself, it's meaningless. To get the data associated with a particular node, you must first find its index and then search through each vertical list for the appropriate data.

A node object would store the data by the horizontal rows. In this way the data for a node is kept in a logical unit. There is no restriction on using different data types in the object's instance variables (technical term for an object's data).

This way of storing data is often the reason why it is claimed that object oriented programming follows the way we think about the real world. Objects in the real world contain themselves and this property is captured in object oriented languages.

5.1.1.4 Software Organization

Simulation software requires thousands of lines of executable code. Dividing this code into manageable pieces is an undisputed requirement. Different languages provide different means for organizing the software. Some ways are definitely more desirable than others.

FORTRAN provides a bare bones means for organizing software. You can write as many subroutines as you want as long as you link them all together when making the executable module. FORTRAN does not provide you with any means for organizing these subroutines. That function is left to the software design process and the operating system that the system is developed on.

This is usually done on a small scale. For example, each person or group decides on personal organization standards. One might decide to limit the number of subroutines in a file to five and the number of files per directory to twenty. To help find specific routines it would be decided that the first two characters of the subroutine and file names are to be key coded. MT would stand for math related code, ST would stand for code computing statistics (or should those be MT also?). Anyone who has programmed a great deal knows how this becomes unwieldy once you get into large numbers of subroutines (100+). Such size is required for simulation code. Trying to standardize this for a large group of programmers becomes a costly and time consuming task. The point is: FORTRAN doesn't really care and provides essentially no tools for organization.

Software design processes offer tools like flow charts and data flow diagrams [67], [68]. These help in determining what subroutines to write and how, hopefully, to eliminate duplication of code. This also provides a means for documenting the system. The problem is that keeping up with this external documentation is extremely difficult over the life cycle of a software package, and is usually not done. What is needed are languages that are self documenting. That is, the structure of the software should represent itself clearly enough so that extensive external documentation is not needed.

Smalltalk-80 provides many tools for organizing (and re-organizing) the software. First, objects are created from classes. Classes are defined by inheriting from previously defined classes and then adding anything not captured by the superclass. The resulting class hierarchy describes the entire system. Smalltalk provides the means for viewing, creating and modifying the entire hierarchy or just portions of it.

Second, in Smalltalk-80 you can organize your classes into class categories independent of the class hierarchy. This allows one to partition classes into easily accessed groups. For example, the developed bond graph processor required 41 new classes. These classes were broken down into 6 class categories, those being:

1. BG-Core: High level core of the bond graph processor.
2. BG-Nodes: The various multiport nodes.
3. BD-Nodes: The various block diagram nodes.
4. BG-Arcs: Specific types of arcs.

5. BG-Functions: Classes dealing with functions.

6. BG-Utilities: Supporting classes.

Third, within a specific class there are method categories that allow you to organize the methods of a particular class into easily accessed groups. For example, the BGGraph class (the Model) has 35 methods. To work more efficiently with these methods 8 method categories were created, those being:

1. accessing

Assigning and retrieving instance variables.

2. causality

Methods dealing with adding and removing causality from the model.

3. checking

Various methods to check if the model is correct or not (e.g., check to see if all arcs have two nodes attached).

4. initialize

Methods called when creating or resetting defaults of the model.

5. inquiry

Methods that return information about the model (e.g., return system equations or return all nodes in my system).

6. modifications

Methods that modify the model's instance variables (e.g., add an arc or remove a node).

7. releasing

Methods that clean up the model when it is about to be removed.

8. storing

Methods used to store a description of the model in a text file.

These software organizational tools were extremely valuable in developing OOBProc. In fact, the ease in finding specific portions of code encouraged incremental development (discussed in the previous chapter). Since code was well partitioned it generally was not a formidable task to rewrite portions of the system. In other words, the Smalltalk-80 system is self documenting.

Object oriented programming promotes these kinds of tools in general. For example, class categories and method categories are nothing more than objects that contain a group of classes and a group of methods, respectively. The behavior of these objects allow you to access and edit the appropriate classes or methods. Once again this leads to a good development environment, which will be discussed in the next section.

5.1.2 Environment

The development environment is very important to the programmer. It conditions how the developer is going to interact with the computer and the software language during the software development process. Effective tools in this area contribute enormously to programmers' productivity. This area is important to all software development and not just to physical system simulation. Thus specific reference to the bond graph processor are not made in this section.

Languages like FORTRAN or C do not, in general, provide tools for the developmental environment. The development consists of editing, compiling, linking and executing the code. You can purchase development environments for these languages that provide some conveniences in these steps, but are still relatively primitive (e.g., Codeview). For example, since FORTRAN does not provide extensive means for software organization (as discussed in 5.1.1.4), how could anyone provide extensive means for editing it?

On the other hand, Smalltalk-80 provides a seamless environment (i.e., one that has a uniform operating mode). This is done by providing classes (thus, the objects) that handle editing, compiling and execution. Recall that everything is an object. Editing, compiling and execution are done by sending the appropriate objects the necessary messages. This consistency, once mastered, makes working with and figuring out the language and its capabilities very straight forward.

Note that to a large extent linking is not required in Smalltalk-80 because messages are bound to the appropriate method during execution. This is called late binding.

Smalltalk-80 provides the classes for an extensive development environment. This is done through four main areas referred to as: (1) browser; (2) inspector; (3) debugger, and (4) utilities. All of these have numerous capabilities. . (I seem to find a new one at least every week.) Here just a brief overview is given to make the point that object oriented languages lend themselves to useful developmental environments more readily than do FORTRAN or similar languages.

All four tools are based on the user interface discussed in section 5.1.1 and thus are considered user friendly (i.e., multiple windows, multiple views within a window, pull down menus, etc.).

First, the browser is provided to look at, modify and add new software components. These may be class categories, classes, method categories, and methods. The tools provided in the browser allow you to organize your software as discussed in section 5.1.1.4 in an efficient manner. In fact, the browser's view (see model-view-controller in section 4.1.3) of the software is one reason many, including myself, claim that modifications to the system are so encouraged. (See incremental development in section 4.1.2.)

There are multiple types of browsers provided. Each type is dependent upon what it is that you wish to 'browse'. For example, one type of

browser can handle class categories, classes, method categories, and methods all together. Another type of browser is provided just to examine and edit all the methods in the system with the same name.

Second, the inspector allows one to examine and send messages to any object. This provides an internal look and the means to modify the system's data base at any time, which is not possible in FORTRAN.

Looking at the internal data base is very useful when creating new objects to see if they are being formed correctly. It is also very useful in determining what existing objects (ones that you can make from the supplied classes) are really composed of. The added capability of being able to send these objects messages within the inspector allows you to check (or determine) the behavior of any object directly under your control and not through an application.

Normally, in FORTRAN, you write out the data structure within the appropriate routines that use the data if you want to examine the data. There is no means for then 'sending' this data to another subroutine to help determine the behavior, as there is with an inspector.

Third, the debugger allows one to inspect objects and browse methods during execution of some message. This is useful, for example, when an error occurs. This tool combines some of the browser's capabilities and all of the inspector capabilities into one feature that lets the developer examine the current execution and change it if so desired.

The debugger is not just an 'error' handler. It can be invoked at any time, either through an error, a halt statement in your code, or simply by typing control-c at the keyboard. When a halt occurs (i.e., a debugger is invoked) you can: (1) modify any method or object that led to the halt and then resume execution; (2) look around the system at that point and then either resume execution or terminate the current process, or (3) if an error has occurred, fix the error before proceeding, as in (1).

Finally, Smalltalk-80 provides various utilities that are very useful during development. The three most useful utilities for making a more consistent system are:

1. senders

Collects and makes a browser out of all the methods that send a particular message.

2. implementors

Collects and makes a browser out of all the methods with the same name.

3. messages

Collects and makes a specialized list out of all the messages sent by a particular method. If a choice from this list is made then it uses this choice for the implementor's utility described above.

These utilities are nothing more than easily accessed messages that are

sent to specific methods. Once again, everything is an object and using anything in the system is just knowing what message to send to what object. Providing a full set of objects and easily accessed messages makes the programming environment extremely productive.

In summary, the browser, inspector, debugger and utilities compose a useful and productive development environment in Smalltalk-80 unmatched by anything in FORTRAN.

5.1.3 Robustness

Using object oriented programming is claimed to reduce the errors that occur in software development. In this section an argument is made to support that claim. Three different issues have been identified: (1) user interface; (2) data storage, and (3) the environment. Each one of these will be discussed separately.

First, recall the discussion in section 5.1.1.1 about Smalltalk-80's built in menu handler. The developer codes a complex hierarchical menu in one relatively simple method. The handling of the menu is left to the system. This provides a consistent user interface.

In FORTRAN, you have menu handling spread throughout the entire system, in many different subroutines. Writing and maintaining these subroutines becomes quite a task for a complicated system (e.g., a bond graph processor). To provide a consistent user interface is difficult because

the subroutines are often written by different people and over a period of years. This provides an opportunity for errors and inconsistencies to enter the system that does not exist in Smalltalk-80.

Second, recall the discussion in section 5.1.1.3 about data storage and usage. Storing and passing data in common blocks leaves the data unprotected from the possibility that even the most remote subroutine might modify the data in a way unacceptable by another subroutine, perhaps after multiple processes. Finding and eliminating these types of errors are difficult activities in FORTRAN-like systems.

Encapsulation reduces these types of errors and helps in finding ones that do occur. Since an object is responsible for its own behavior it can control undesirable changes to its data, thus reducing errors caused by unprotected data. Also, while finding bugs in software is often very difficult, encapsulation helps in this task. If an object is not behaving appropriately the best place to look for errors is with that object itself. The Smalltalk-80 environment aids in this process, as discussed next.

Finally, the environment (section 5.1.2) not only helps in finding and eliminating errors, but it helps prevent them from ever becoming part of the system in the first place. As discussed in section 5.1.2, the inspector, debugger and utilities can be used together to determine exactly how an object is behaving (or mis-behaving). Thus, finding the errors and then using these tools to eliminate the errors are much easier than in a language like FORTRAN.

Additionally, encapsulation allows for pre-testing objects before they are put into the system. For example, the functions assigned to the nodes were developed and tested relatively easy outside of the bond graph processor before they were ever assigned to any node. Also, nodes existed and were used long before their functions were developed (incremental development, section 4.1.2). It is true that subroutines can be tested in FORTRAN before they are used, but the language does not provide for any reasonable means to do so. One must develop one's own test package.

The Smalltalk-80 environment encourages the creation of new objects as you go in order to test new methods immediately upon their writing. Recall that Smalltalk-80 is an environment. Once you make something (class, method, etc.) it immediately becomes part of the environment and can be used extensively. There is no compile-link time-consuming process required to test new concepts. This pre-testing can considerably reduce the errors that are often found when finally piecing a large system together.

In summary, although statistics for reducing errors were not accumulated it has been reasonably argued that using Smalltalk-80's user interface tools, encapsulation, and the environment can not only help find and eliminate errors effectively, they can also help to prevent them from occurring in the first place. This makes for a more robust system.

5.1.4 Time

Proving, with statistics, that developing a software package in an object oriented language is quicker than in a conventional language like FORTRAN is beyond the scope of this research. However, this section describes just how productive Smalltalk-80 was in OOBProc. The author's experience is a strong indication that time efficiency was achieved.

OOBProc was developed in Smalltalk-80 over a period of five months working on it part time. This five month period includes learning Smalltalk-80 without any training and without the benefit of having others near by for help. It was totally self learned. The total estimated time for development (including learning) is 300 hours. This, I believe, is an indication that using object oriented programming techniques is very productive.

A little background will put this development time in perspective. First, a thorough understand of bond graphs had been obtained prior to the beginning of the research. Second, 21 months prior to the Smalltalk-80 implementation was spent on learning object oriented programming. Finally, a basic bond graph processor (essentially phase I) without any graphics was developed using the language Actor. None of the Actor code was transportable to Smalltalk-80 but the design experience definitely helped. This will be discussed in section 5.4.

What this means is that with a sound foundation in the application domain and the techniques of object oriented programming one can produce powerful

software in a relatively short period of time.

5.2 Enhancements

One of the statements made at the beginning of this dissertation was that the object oriented environment allows for user customizable software and for different levels of development. This section will attempt to show the validity of this statement by discussing the enhancements made to OOBProc from a developer's point-of-view. Recall the discussion in section 3.1.4 where enhancements were described from a user's point-of-view on how to make changes and additions to the software.

5.2.1 Types

The specific types of enhancements singled out for user customization are: (1) functions; (2) nodes, and (3) methods. Each one of these will be discussed in detail.

5.2.1.1 Functions

To allow user customizable functions in FORTRAN is a complex job. The two basic choices are: (a) Develop an in line FORTRAN compiler that will accept user written FORTRAN subroutines and incorporate them into the system, or (b) let the user be responsible for compiling and linking the

whole system together so new subroutines can be added. Neither option has been proven feasible for the every day engineer. Generally, a systems programmer would be the only one with enough knowledge to do such a task regularly.

In OOBProc, capabilities for user customizable functions were programmed into the system without a great deal of trouble, relative to the FORTRAN choices.

Associated with most node types is a mathematical function. The choices of functions desired for a bond graph processor are quite large (40-60). In order to satisfy the user's demand this system was built to allow for easy creation of new function types. This is handled by a class called TemplateFunction. A template function defines generically a function that is created specifically upon demand by the node. How this works is:

1. Function definitions are stored in text files on the hard disk.
2. A Node has an instance variable that points to a TemplateFunction object. This object stores the type of function it represents (e.g., sin) and the appropriate input variables, output variables and parameters.
3. When asked for its mathematical definition the TemplateFunction object reads in the function definition from file and substitutes its specific variables for the

generic ones.

In this way a user can define as many functions as can fit on the hard disk (literally thousands). Special utilities are provided in OOBProc to help the user define new functions in a way that the TemplateFunction can understand them. However, the function files are independent of the software, meaning that you could use any text editor outside of the simulation package to create new functions. If these new functions follow the prescribed format (see Figure 3.15) and are in the proper directory, then OOBProc will automatically incorporate them into possible choices for node functions. This last point is subtle; the user need not tell the bond graph processor explicitly that a new function was created.

A special function type (UserDefinedFunction) allows the user to define specific one-time functions to associate with a given node. This allows the user to experiment with a function before actually making a template definition for it.

5.2.1.2 Nodes

There are three types of nodes allowed in the current bond graph processor, namely, blocks, macro nodes and multiport nodes. A macro node is a specific type of node containing multiple nodes and arcs. Blocks and multiports are abstract nodes, meaning they represent general behavior of more refined node types. Blocks represent functions that produce signal outputs based on specified inputs. For example, a gain function, an

integrator, and a weighted summer are all specific types of blocks. A multiport is a node type that represents the mathematical aspects of a physical component based on power. For example, an inertia represents a component that stores and releases energy, and a resistor represents a component that dissipates energy. The node class hierarchy was shown in Figure 4.9 for OOBProc.

New node types can be added to the system in multiple ways:

1. Add a new block type.
2. Add a new multiport type.
3. Add a specialized type of macro node.
4. Add a new type of node independent of the existing ones.

The first two choices would be rather straightforward. For example, let's say we wanted to add a capacitor to the initial bond graph system from phase I, as was done during phase II (see section 4.3.2). A capacitor is a multiport and thus would be a subclass of MultiPort (class name Capacitor). Just by creating this subclass the following occurs:

1. The Capacitor inherits basic node behavior from class BGNode.
2. The Capacitor inherits basic multiport behavior from class MultiPort.
3. The Controller of the bond graph processor will

automatically handle the new node type because it is independent of the specific node types. However, you would have to add the choice Capacitor to its menu (one line of code).

4. The View would require no modifications because BGNode and Multiport node already handle any view related issues for this type of node.
5. The Model would require no modifications because it handles nodes generically and a new type of this kind would be no different then existing ones.

In order for this new node type to behave like a capacitor (e.g., a spring component) you would have to specialize five methods. Those are

1. assign desired causality

Bond graphs have something called causality that determines the proper form of the node equations. A capacitor has a preferred causality. Overwriting its inherited causality would allow the capacitor to behave as desired.

2. inputs and outputs

The input and output variables of a multiport are based on causality and the type of node it is. A capacitor would be required to supply its specific input and output variables (e.g., force and displacement).

3. `getSysEqn`

This would return the specific equations that represent a capacitor.

4. `stateEquation`

One of the capacitors system equations is its state equation.

This method would supply that particular equation.

Adding a new type of block to the system is actually much simpler than adding a multiport since a block does not have to deal with causality. Its input and output variables are determined by topology and thus can be handled by superclass methods.

Adding a specialized macro node would probably start by inheriting from the existing `MacroNode` class. Methods would have to be written to override the existing behavior in order to determine its specific behavior. This could possibly be a complex addition depending upon how different the new macro type would be from the existing type. Adding the original `MacroNode` class was a complex addition.

Adding a new node type that may not be invented yet would definitely require some thinking. It is probably safe to say that the Model-View-Controller would handle this new node type like it does the other nodes. Additional methods would have to be created in order to handle any specialized user interaction associated with the new node type not present in the existing nodes. For example, when the macro node was added the methods to create, view and expand the macro were required.

These concepts did not exist when just blocks and multiports were around. It is predicted that the class BGNode will still handle the basic behavior of the new node type. (At least it did when blocks were added and then when macro nodes were added.)

In summary, adding new multiport or block nodes would be relatively simple. Changes to the system would be minimal and localized so that the existing system would require no modifications (except adding a menu choice, which is simple). Adding more complex node types could become complex, but the basic node behavior foundation would not have to be redone.

Object oriented languages allow for these capabilities because of their two main features as discussed in section 2.2.2, those being encapsulation and inheritance. Inheritance allows for reusability of the basic node behavior and encapsulation provides for minimal effects on the existing system due to changes.

By following good object oriented design methodologies (see section 4.1) the developer automatically provides the framework for adding new specialized classes to a system. Here, by forming a node class hierarchy, as was shown in Figure 4.9, the ease of adding new node types was designed into the system from the start.

5.2.1.3 Methods

Adding new procedures to the bond graph processor is another important type of enhancement. For example, if someone wanted to determine the subgraph consisting of all of the one junctions, zero junctions, transformers and gyrators (called the general junction structure) this could be done by someone with the proper programming training. You would not have to learn the entire system to implement this new capability. You would, however, need to know how to do the following:

1. Install a user interface to the new option.
 - a. Add the requested action to the appropriate controllers' menus.
 - b. Write the controller method(s) to direct the execution of the requested action. For this case, the method would contain two parts: (i) Request from the model its junction structure, and (ii) display the results in a meaningful way. For example, store them in a file, make a simple displayable list with them, etc.
2. Write the method in the Model that actually collects the required nodes and arcs.

A knowledge of how the graphs data is stored would be required. You would have to loop through all of the nodes

and collect the appropriate ones and their attached internal bonds. A simple request to the nodes themselves inquiring if they are part of the junction structure or not would ensure that the method would be general enough to handle all node types (existing or future ones).

These types of enhancements would not affect any existing behavior. They would add new behavior, such as nodes responding if they are part of a junction structure or not. This could be handled at the Node class level for the majority of the node types. Thus only nodes that are actually junction structure nodes would have to be specifically changed. Knowing the current system would definitely help in re-using existing methods and techniques.

5.2.2 Required Expertise

Section 1.1 identified three types of users involved with the software, namely, the engineer, the analyst, and the system programmer. One of the goals of using object oriented programming is to allow for customizable software. This would mean that each of these users could reasonably make useful modifications to the system without considerable effort and/or training.

The previous section discussed types of enhancements. Here we want to discuss the expertise required to make such enhancements.

Adding new functions to the system was shown to be quite straight forward and independent of the software language used to write the simulation package. The format for new functions is not particularly complex, and anyone who works with a computer could easily learn how to edit text. This implies that the every day engineer would be able to modify/create functions required for applications on an as needed basis.

It should be noted that adding new functions was specifically programmed for in the bond graph processor. There doesn't seem to be any specific restriction that would prevent other enhancements to be programmed for and thus made accessible to the engineer. However, investigating this possibility is left for future research.

Adding new node types was shown to involve basic programming knowledge. But due to encapsulation and inheritance it is kept to a minimum. The analyst of today is usually familiar with specialized programming because generic simulation packages rarely contain all of the specifics required at individual work places. The limited amount of work required to add multiport or block nodes to the system would mean the analyst could undertake such a task with some basic training.

Adding new methods to a system was shown to involve a considerable amount of understanding of the entire system. Unless a new method was fairly simple, modifying the system in this way would probably require a systems programmer.

The boundaries between the engineer, analyst and programmer are not as

clearly defined as may be implied above. Very simple tasks could be performed by the engineer who is involved with current programming techniques. Figure 5.3 shows a matrix of the type of enhancements versus who would have enough expertise to make those enhancements. The ratings in the boxes indicate the complexity of the type of enhancements that one could make.

	Engineer	Analyst	Systems Programmer
Capabilities programmed for (e.g., functions)	medium	high	high
New subclasses	low	medium	high
New methods	low	low	medium

Figure 5.3 Complexity of User Enhancements

Adding new methods of high complexity would probably require the original software developers. This accounts for the rating of 'medium' for the complexity of new methods for the systems programmer.

5.2.3 Controlled Access

So far we have discussed enhancement types and the expertise required to make those enhancements. The final discussion point is based on controlling the actual enhancements made to a system.

In FORTRAN this is easy. Either provide them with the source code or don't. You either have complete access or no access to the system in order to make enhancements.

In object oriented languages you theoretically can control the access to the system. For example, maybe an engineer is capable of adding new functions, but do you really want them to? Changes to the system can cause problems. Discussion of this topic is deferred until chapter VI.

Six different levels of access to the software code have been identified for the bond graph processor. These are depicted in Figure 5.4. The capabilities are accumulative as the access becomes greater.

Access	Capabilities Relative to Bond Graph Processor
None	Add or modify functions.
Browse	Investigate how the system is put together.
Add Subclasses	Add new node types, particularly multiport and block node types.
Limited Write	Add new methods. Add new view/controller pairs to interact with the model differently.
Expanded Write	Change existing methods.
Full Access	No restrictions.

Figure 5.4 Controlled Access to Enhancement Capabilities

A good result from this is that as the complexity of the desired enhancement increases, so does the need to access more of the system. This is significant because it correlates well with the previous section on the expertise required to make enhancements. In other words, the changes a user can make theoretically can match with the changes they are

allowed to make. Users can be given just the tools they need without undo complexity.

5.3 Reusability

There are three basic types of reusability, namely, inheritance, portability between hardware configurations, and reuse of software components between projects. This section focuses on object oriented programming's support of these issues.

First, inheritance as a form of reusability was discussed in detail in section 5.2. This dealt with the ease of enhancements due to inheriting behavior from superclasses. For example, adding a multiport node was straightforward due to inheriting basic node behavior from class BGNode and inheriting more refined behavior from class MultiPort.

Second, portability between hardware platforms is an important issue due to the lack of standards in the hardware industry. Writing and re-writing software to meet different hardware configurations is a complex task. One of the strengths of the C language is its portability. It would be a step backwards to lose this by using an object oriented language.

Some object oriented languages are portable. For example, C++ is just as portable as C. Smalltalk-80 is currently available on the Macintosh, MS-DOS PC's, Sun workstations, Hewlett-Packard workstations and Apollo workstations. Source code written using the base Smalltalk-80 system is

generally portable between the different platforms. However, some object oriented languages are restricted to specific platforms. For example, Actor is available only on IBM compatible PC's.

In general, object oriented programming can ease portability by encapsulating objects that deal with the specific hardware configurations. This would help minimize the amount of work required to support multiple hardware platforms from the same language.

Finally, reusability of software components between projects would be very beneficial to software development in general. For example, if an electrical circuit simulator was to be programmed could portions of the existing bond graph processor system be used? If the underlying model has been developed well enough then the answer is yes. An electrical circuit can be represented by a graph in a similar manner to a bond graph.

OOBProc was built upon a generic graph system called NodeGraph-80 (see section 4.2). NodeGraph-80 was used as a template for creating the working bond graph simulator. As specifics to the bond graph were implemented it turned out that a majority of the methods from NodeGraph-80 were overridden by subclasses. Although the reusability wasn't very high (as promised by the object oriented methodology), the code was extremely useful as a template and for designing through iteration. During the development phases there was almost always a working system. This aided greatly in writing and testing new code. This incremental development is advocated by most of the object oriented literature (see section 4.1.2).

It is for these reasons that I believe that portions of OOBProc are reusable for a generic graph simulator. However, to be really effective NodeGraph-80 should be modified to include things learned from OOBProc's implementation. The major changes would be:

1. Eliminate specific calls to class methods. Subclassing existing classes requires re-writing all methods that reference the superclass specifically.
2. Incorporate more encapsulation. Too many methods were found to be accessing another class's data structure. For example, locations of nodes were assumed always to be handled in the same way. When a change was made to this format some 33 methods had to be modified because they accessed data structures directly.
3. Provide multi-level graphs.
4. Allow multiple types of arc displays similarly to the multiple types of node displays. Also, do not rely upon the nodes to 'clean up' the arc display.
5. Provide generic message handling (those messages that are given to the user). For example, the incorrectly picking an arc message appeared in 8 different methods. When this notice was modified, all 8 methods had to be modified.

These and other changes were made to the bond graph system, but not to NodeGraph-80 itself. I believe these changes are directly associated with generic graphs and not specific to bond graphs. Thus, by incremental development, NodeGraph-80 can become more and more reusable between projects that use graph structures for their underlying models.

In summary, object oriented programming supports reusability of software components to aid in making enhancements, for portability between hardware platforms, and for re-use in new projects.

5.4 Portability Between Object Oriented Languages

The first part of this research was done using the object oriented language Actor. The system developed in Actor contained a basic bond graph processor, without graphics, blocks or macros. However, enough work had been done that losing it and starting over did not seem very attractive. This section discusses how the object oriented methodology is transferable between different languages.

Actor is a pure object oriented language similar to Smalltalk-80. This means that everything (essentially) is an object. The major difference between the two languages is in syntax. Actor code looks like a combination of Pascal and C. In fact, it was developed to look like that in order to help the transition from conventional methods to object oriented methods. Smalltalk-80's syntax is entirely different than conventional languages, and designed that way on purpose. What this means

is that none of the Actor code was transportable to Smalltalk-80.

However, the structure of the Actor system was very useful in writing the initial portions of the Smalltalk-80 implementation. This refers to the self documenting property of object oriented systems discussed in section 5.1.1. The class hierarchy and the class definitions captured in Actor were essentially the same as were used in Smalltalk-80. The difference mostly involved re-programming the specific methods to be in the correct syntax. The environment and incremental development properties of Smalltalk-80 made this transformation quite efficient.

Re-writing a FORTRAN system, say in C, would be a major undertaking. Since the structure of a FORTRAN system is not self documenting there would be essentially no template to follow. Of course, if proper data flow diagrams and flow charts were kept up to date then they would definitely come in handy. However, this is external documentation which has proven to be quite a burden on programmers in general, and so often is neglected. Also, data flow diagrams and flow charts are not part of the FORTRAN system as classes and class hierarchies are in object oriented systems.

In summary, although changing languages in the middle of a project is not recommended, it is nice to know that the structure of an object oriented implementation is relatively easy to transfer between two different object oriented languages. Properly designed class hierarchies and class protocols are independent of the language used to implement them. This kind of structure does not exist in a conventional language.

5.5 Qualitative Reasons

The prior discussions in this chapter on development, enhancements and reusability as they relate to Smalltalk-80 lead to the qualitative observation that programming using an object oriented language is less frustrating and more fun than working with conventional programming languages. Since OOBProc is such a large system, justifying this claim is probably not possible. However, this section examines a smaller application to let the reader get a better feeling for the claim.

The example chosen is to invert a square matrix using both Smalltalk-80 and using FORTRAN. This problem was chosen because it is straight forward, very useful to engineers, and the author needed this capability for another project.

First, let us discuss the Smalltalk-80 implementation. Found on ParcBench (Smalltalk bulletin board) were a collection of classes dealing with matrices. Downloading these from the bulletin board and then loading them into Smalltalk-80 took about 10 minutes. This illustrates reusability (section 5.3) at its best.

By browsing through these matrix classes it was easily determined how to make a new matrix; how to send math requests to it (e.g., *, +, invert), and how to print out the matrix in a nice format. This took less than 5 minutes. The Smalltalk-80 developer's environment was extremely useful (section 5.1).

Using a work window the code in Figure 5.5 was produced in about 5 minutes. The code is relatively straight forward once you realize that 'Transcript show:' is a standard message used to print output to a particular window in the environment.

```
| error m mPrime mm |
error _ [ :message | Transcript cr; show: message; cr ].

"Create the matrix m and print it on the Transcript."
m _ Matrix rows: #(
(1 3 2)
(2 6 9)
(3 8 8)).
Transcript cr; show: 'Matrix m'.
Transcript show: m prettyPrintedString.

"Invert the matrix m to get matrix mPrime, print mPrime."
mPrime _ m invertSquareMatrixIfSingular: [ error value:
'Trouble' ].
Transcript cr; show: 'm inverse'.
Transcript show: mPrime prettyPrintedString.

"Multiply m by mPrime to check if inversion worked."
mm _ m * mPrime.
Transcript cr; show: 'm * m inverse'.
Transcript show: mm prettyPrintedString.
```

Figure 5.5 Smalltalk-80 Code for Inverting a Matrix

Executing the code requires highlighting the text and picking 'do it' from a standard pop-up menu. This took all of 10 seconds. The results are shown in Figure 5.6.

Modifications to the matrix are easy. Just point the cursor to the matrix entries and text edit them. Re-execute as above and you can invert as many matrices as desired. Each one took less than a minute (including editing and execution time).

Total 'project' time was less than 25 minutes.

Matrix m		
1	3	2
2	6	9
3	8	8
m inverse		
(-24/5)	(-8/5)	3
(11/5)	(2/5)	-1
(-2/5)	(1/5)	0
m * m inverse		
1	0	0
0	1	0
0	0	1

Figure 5.6 Results From Executing Matrix Inversion Code

The FORTRAN implementation turned out not to be as convenient. Being familiar with a standard set of routines for matrices written in FORTRAN called LINPACK it seemed, at first, reusability would make this as simple as Smalltalk-80's implementation. What follows is an outline of what happened.

1. Find the routine in a directory of about 50 files that handles inversion. After selective browsing with a standard VAX editor a file called SGEDI was found.
2. Determine all the subroutines that are called within SGEDI; otherwise the system won't compile and link together.
3. Write the main routine.
 - a. Initially hardwire the matrix to be predefined. Add 'easy' modifications later.
 - b. Call SGEDI(A,LDA,N,IPUT,DET,WORK,JOB)

This is the calling sequence to SGEDI. Determine what each parameter means and make sure you declare them

correctly. Doing this it is found that one of the parameters must be an output variable from a subroutine that must be called before calling SGEDI. Which one? It turns out you can either call SGEFO or SGEFA. They both will produce the required input for SGEDI.

OK, call SGEFA first. Get its calling tree for compiling and linking, determine parameters, etc.

- c. Using Format statements display the results. How do you get a matrix displayed nicely with Format statements? Not easily. (There must be a subroutine to do this, but where is it?)
4. Compile and link the required subroutines. Compiling files from different directories is not trivial. Write a command file to handle compiling and linking. Chances are you're going to need it multiple times.

Fix compile and linking errors. Incorrect Format statements, missing some files in the link stage. Must be more careful.
5. Fix format statements for nicer display. Re-compile, link and execute. Success, a victory.
6. Enhancements, in order to compete with the Smalltalk-80 implementation: (1) Output initial matrix, and (2) multiply the resulting matrix by the input matrix and display outcome for verification that inverse worked.
7. How do you change the matrix? Edit the main program, compile, link and execute. No good? Program a better method, either: (1) Read in the matrix from a file, so the user 'only' has to edit that file and then execute the program, (2) prompt for the entries, or (3) do both. Let the user pick which option to follow. Provide multiple runs by looping through the main program until user says to stop.

Total 'project' time was less than 2 hours, not including item 7 (I just didn't have the patience).

Section 5.1 talked about robustness and productivity without statistics.

Errors like 4. above did not occur in the Smalltalk-80 implementation.

Almost 2 hours for the FORTRAN implementation doesn't even compare to the Smalltalk-80 25 minute implementation.

The bond-graph/block-diagram processor implementation had a lot of nice development stages that went similarly to the matrix inversion project. This explains why it is claimed that object oriented programming is less frustrating and more fun than conventional programming.

CHAPTER VI

CONCLUSIONS

6.1 Benefits of Object Oriented Programming

This section discusses the main benefits of using object oriented programming for physical system simulation as discovered during the development of OOBProc.

First, the development platform of an object oriented implementation is readily extendible. This was shown by conducting a four phase implementation of OOBProc. Each phase introduced new capabilities to the system without major revisions of the existing system. The concentrated efforts were on developing the new classes that implemented the new behavior. Modifications to the existing system consisted mainly of refinement of previously defined behavior. Polymorphism was crucial in allowing new classes to be used correctly without modify existing code.

Second, object oriented programming lends itself to user customizable software. This was detailed in section 5.2. The encapsulation of objects and the ability to inherit from existing classes allows the user to comprehend enough of the system to make additions without becoming a professional programmer.

Third, implementations follow closely the world they are modeling. This is extremely helpful because the design process can fully concentrate on the problem being solved and not on transforming a theoretical solution of the problem into a software implementation. Both sections 4.3 and 5.1 referred to this. For example, Figure 5.2 showed the data that defines a node. As an object, a node contains all of this data and the behavior of a node. Object oriented design (section 4.1) is an important step in the process.

Fourth, implementations contain self documenting features. This was discussed in section 5.1.1.4. With tools like class categories, class hierarchies and method categories, along with encapsulation, the structure of an object oriented implementation is represented by that implementation and not by extensive external documentation.

Fifth, classes readily lend themselves to reusability, as discussed in section 5.3. The two types of reusability most applicable to this work were inheritance and reuse of software components. Reuse of code reduces code bulk, code complexity and development time.

Sixth, classes can be tested easily throughout their development and are thus more robust. The Smalltalk-80 environment (section 5.1.2) and encapsulation of objects can allow for new classes (objects) to be thoroughly tested before they become part of a system. This pre-testing of objects contributes to a more robust system (section 5.1.3).

Seventh, control statements are drastically reduced, thus reducing code complexity and preventing bugs from being introduced into the system. Being able to send the same message to different objects (i.e., polymorphism) eliminates a large number of control statements, particularly in the user interface (section 5.1.1.1).

Finally, object oriented programming is less frustrating and more fun than conventional programming. Although this statement is highly subjective, an argument was made in section 5.5 to support it.

6.2 Difficulties with Object Oriented Programming

This section discusses the main difficulties encountered with object oriented programming during the development of OOBProc.

First, violating encapsulation is easy to do and very dangerous. For example, a view object in OOBProc contains an instance variable named `nodeLocations`. This variable is an instance of the class `Dictionary`. Its key is a node object and the corresponding value is the node's location in the view. To access a node's location all you need to do is to send the message `'nodeLocations at: aNode'` to the view. This message violates encapsulation because it assumes what the variable `nodeLocations` is. The message should be something more like `'locationOfNode: aNode'`. Leaving the view with the responsibility of getting the actual location.

After phase II it was desired to test a new way of storing node locations. It was determined that 54 methods directly accessed the views instance variable `nodeLocations` in some way (36 of those methods were from `NodeGraph-80`). A change in the storage could not be made until the violation of encapsulation described above was fixed. A few methods like `'locationOfNode:'` were written for the view and then all 54 methods were updated to use the appropriate message. Even with the utilities of the `Smalltalk-80` environment (section 5.1.2) determining and correcting these methods was time consuming.

Wirfs-Brock and Wilkerson [69] discuss this issue in some detail. Their solution is one of setting recommended practices to be used by programmers. However, as was pointed out in section 4.3.3, it is not always obvious when you are violating encapsulation.

Second, determining how something actually gets done is a complex task. Objects are responsible only for themselves, which is good, but they tend to send a large volume of messages to other objects (mostly their instance variables). The message passing between objects is often a long road from the original request until the desired behavior is achieved. To determine this process a great deal of effort is often required.

There are methods available to determine the message passing sequence. One can use the debugger to stop the process at any location and examine the calling sequence dynamically. One can also use the available searching utilities to locate the methods that respond to a given message. Using this recursively allows one to create a calling tree structure of

the sequence. Cunningham and Beck [70] describe a notation for diagramming the message sending dialog that takes place between objects. Although all of these tools are useful none of them seem to illustrate the message passing sequences effectively and efficiently.

Third, some aspects of physical systems are hard to clearly define in terms of objects. For example, causality is a procedure applied to a bond graph that indicates the direction of the effort and flow variables on the bonds to be used by the multiports to determine proper equation formats. Additionally, there is a graphical display of causality required. A change in the bond graph model could have far reaching effects on the proper causality (and thus the equation structure) for that bond graph. Causality is clearly not an object, but there is data and behavior associated with assigning and using causality for a bond graph. Determining which objects are responsible for that data and behavior is complex.

Finally, due to the large number of classes that can be supplied with an object oriented language the learning curve is rather steep. Specifically, Smalltalk-80 has approximately 250 classes and thousands of methods. Learning the system enough to make effective use of inheritance and reusability from a programmer's view can not be done over night.

6.3 Unanswered Questions and Future Research

First, numerical solution techniques using object oriented programming were not investigated in this research but is none the less critical for simulation software. Other work has been done in this area [58], [59] but much more research is needed.

Second, there is a question as to the run time speed of object oriented software. This is a critical issue associated with the cpu intensive processes required for numerical solutions. Both Thomas [71] and Peskin et al [72] begin to address this issue. Peskin has shown for a few specific cases that calling a C program within Smalltalk for computations lead to a time savings of a factor of about 100. Thomas states that "Using cache technology allows Smalltalk systems to execute a message send in half the time of an equivalent C procedure call." However, Ungar [73] disputes this claim but does agree that advanced computer architectures will lead to faster object oriented implementations.

Third, bond graphs and block diagrams model physical systems. The future in simulation software, as discussed in section 2.1.1.5, will focus on integrating computer aided design (CAD) systems with simulation software. What will be needed are methods to capture descriptions of actual physical systems and components through CAD and to formulate a corresponding mathematical model suitable for the desired simulation. This could be a bond-graph/block-diagram, a finite element model or some other type of mathematically based model. Object oriented programming should be effective in this endeavor.

Fourth, predicting behavior is one of the purposes of simulation. This requires numerous response display capabilities. Ideally this should be highly user customizable. With the many different graphical capabilities of Smalltalk-80 and the user customizable features demonstrated in OOBProc, the desired capabilities for response display should be attainable. Again, Peskin et al [72] have begun work in this area.

Fifth, the current methodology used for simulation includes formulating a set of system equations corresponding to a model, solving those equations and then relating the results to the original model. In the real world, no "master simulator" (to our knowledge) is formulating and solving sets of equations to determine behavior. Each object in the physical world controls itself based on its environment. Object oriented programming and parallel processing may have the potential to actually simulate the physical world as we know it. This might be done by, for example, having objects determine their own motions based on their surrounding environment and to have these objects relay their responses to that environment. This is currently being done with discrete time simulation.

Sixth, it is not clear at this time how one would effectively implement a software version control process with an object oriented system. For example, if a group of, say ten, engineers were using the same object oriented software system and each wanted to customize the software, a controlled process of doing this must be enforced or else conflicts in the software will arise. Section 5.2.3 discussed controlled access on an individual bases, but how this would work in a group is unclear.

Seventh, the proliferation of object oriented languages is immense (section 2.2.2). Which will be the important languages of tomorrow is unclear. The question remains as to which language a software development organization should use for engineering applications. The safe choice at this time, outside of the AI world, appears to be C++ with its strong roots in the currently popular C language.

Finally, this research has shown that new physical system simulation software should be written using object oriented techniques. What remains to be answered is: At what point should old software be re-written to take advantages of the newer technology?

APPENDICES

APPENDIX A

INTRODUCTION TO BOND GRAPHS

Bond graphs are used for structured modeling of energy and power based physical system [23], [24], [25]. The recognized benefits of using the bond graph technique are as follows:

1. Bond graphs can be used to model a wide range of systems, including mechanical, electrical, magnetic, hydraulic, and thermal systems.
2. Bond graphs provide coupling mechanisms between domains. For example, the electrical-mechanical rotation conversion of a motor.
3. Bond graphs are a graphical technique derived directly from the physical system model, rather than through the equations of the system. This provides the following:
 - a. This gives the trained engineer direct insight into the physical system by examining the bond graph without interpreting any equations.
 - b. Large problems are more understandable depicted graphically rather than by a large set of equations.
 - c. Modifications made to a system are more easily incorporated into a graphical structure than a derived set of equations.
 - d. Graphical structures lend themselves to useful abstract (macro) models of the system.
4. Bond graphs lead to state space differential equations (i.e., first order ordinary differential equations). This provides the following:
 - a. Numerical solution techniques are well developed for these types of equations.
 - b. State space equations readily lend themselves to feedback control techniques.
5. Once a model of the dynamic system is obtained in bond-graph/block-diagram form it can be handled by computer software, such as ENPORT, CAMP, and TUTSIM [27], [28], [29].

To minimize the complexity of this Appendix, bond graphs will be described in terms of mechanical translation and electrical systems only.

Before beginning the discussion about bond graphs it is important to understand the variables used in the methodology. There are four general variables used, these are: effort, flow, momentum and displacement. Figure A.1 shows the terminology used in this Appendix for the different variables. Note the re-use of some variables, for example q can represent general displacements or the charge in an electrical system. The context in which a variable is used should make the distinction possible.

General	e Effort	f Flow	p Momentum	q Displacement
Mechanical Translation	F Force	V Velocity	P Lin. Momentum	X Lin. Displace.
Electrical	e Voltage	i Current	λ Flux Linkage	q Charge


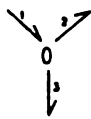
Figure A.1 Variables Used For Bond Graphs

A bond graph consists of nodes and bonds. The bonds indicate energy transfer between the nodes. The bonds 'carry' the four variables effort, flow, momentum and displacement at all times. The bond is depicted by a straight line with a half arrow on one end. This arrow is called the power stroke and represents the direction of positive power. The power associated with a bond is equal to the effort times the flow of that bond.

The nodes of the bond graph describe both the physical laws of the components and the constraints the components exert on each other. These nodes are often called multiports.

The basic multiports consist of the capacitor (C), inertia (I), resistor (R), source of effort (Se), source of flow (Sf), 1-Junction (1), 0-Junction (0), transformer (TF) and gyrator (GY). Figure A.2 lists the physical meaning, bond graph representation and mathematical definitions for these multiports in terms of mechanical translation and electrical components.

Learning how to construct bond graph models is beyond the scope of this Appendix. However, two examples are looked at to demonstrate the meaning of the resulting bond graphs.

Capacitor	$\longrightarrow C$	Mechanical: Spring Electrical: Capacitor	$F = k \cdot X, \dot{X} = V$ $e = 1/C \cdot q, \dot{q} = i$
Inertia	$\longrightarrow I$	Mechanical: Mass Electrical: Inductor	$V = 1/m \cdot P, \dot{P} = F$ $i = 1/L \cdot \lambda, \dot{\lambda} = e$
Resistor	$\longrightarrow R$	Mechanical: Damper Electrical: Resistor	$F = b \cdot V$ $e = R \cdot i$
Source of Effort	$Se \longrightarrow$	Mechanical: Force Electrical: Voltage	$F = F(\text{time})$ $e = e(\text{time})$
Source of Flow	$Sf \longrightarrow$	Mechanical: Velocity Electrical: Current	$V = V(\text{time})$ $i = i(\text{time})$
1-Junction		Mechanical: see (a) Electrical: see (b)	$F1 + F2 - F3 = 0$ $V1 = V2 = V3$ $e1 + e2 - e3 = 0$ $i1 = i2 = i3$
0-Junction		Mechanical: see (c) Electrical: see (d)	$F1 = F2 = F3$ $V1 - V2 - V3 = 0$ $e1 = e2 = e3$ $i1 - i2 - i3 = 0$
Transformer	$\xrightarrow{1} TF \xrightarrow{2}$ mod:1	Mechanical: Fulcrum Electrical: Transformer	$F1 = \text{mod} \cdot F2$ $V2 = \text{mod} \cdot V1$ $e1 = \text{mod} \cdot e2$ $i2 = \text{mod} \cdot i1$
Gyrator	$\xrightarrow{1} GY \xrightarrow{2}$ K	Mechanical: N/A Electrical: Gyrator	$e1 = K \cdot i2$ $e2 = K \cdot i1$

Linear relationships are used for the equations; k , C , m , L , b , R , mod and K are the parameters associated with the linear equations.

- (a) The 1-Junction for mechanical translation systems handles two properties; (1) the sum of all the forces acting on a particular object is equal to zero (including inertial forces), and (2) the velocity at connection points between components are the same.
- (b) The 1-Junction for electrical systems handles two properties; (1) the sum of the voltage drop around any loop is zero, and (2) the current in any line is the same throughout.
- (c) The 0-Junction for mechanical translation systems handles two properties; (1) velocity constraints, where a simple algebraic velocity equation must be satisfied (i.e., the sum of the velocities must be equal to zero), and (2) the force in a component is the same throughout that component (e.g., the force in a spring).
- (d) The 0-Junction for electrical systems handles two properties; (1) the current going into a connection must equal the current coming out of that connection, and (2) the voltage drop across parallel lines are equal.

Figure A.2 Bond Graph Multiports

MECHANICAL TRANSLATION EXAMPLE

Figure A.3 shows a simple two degree of freedom spring- mass-damper system with an applied force. Figure A.4 is a corresponding bond graph of the physical system.

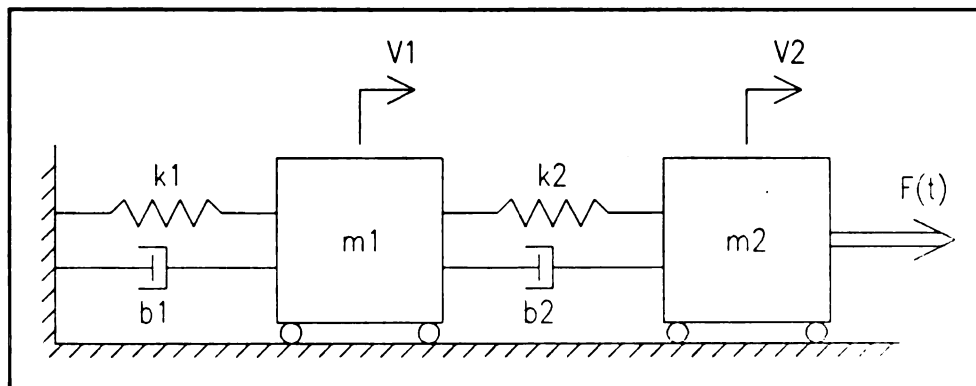


Figure A.3 Two Degree of Freedom Spring-Mass-Damper System

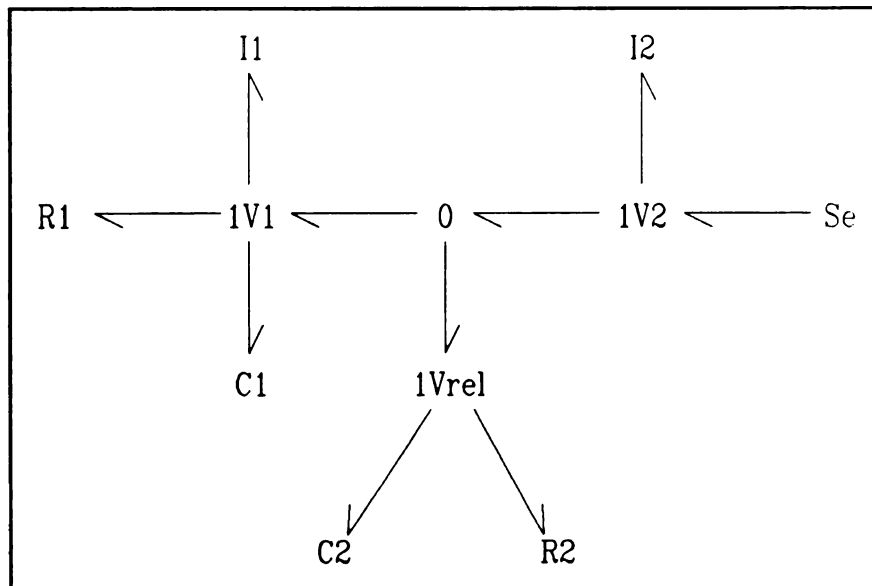


Figure A.4 Bond Graph of System Figure A.3

The bond graph components of Figure A.4 have the following meanings:

C1, C2

The springs 1 and 2, respectively.

I1, I2

The masses 1 and 2, respectively.

R1, R2

The dampers 1 and 2, respectively.

Se

The applied force $F(t)$.

1V1

The velocity of mass 1, end point of spring 1, end point of damper 1, and end points of spring 2 and damper 2 are all equal to V_1 .

The sum of the forces (including inertial) on mass 1 is equal to zero.

1V2

The velocity of mass 2, applied force location, and end points of spring 2 and damper 2 are all equal to V_2 .

The sum of the forces (including inertial) on mass 2 is equal to zero.

1Vrel

The relative velocity between mass 1 and mass 2 is the same as the relative velocity of spring 2 and damper 2.

The total force of spring 2 and damper 2 is, magically, equal to the force in spring 2 plus the force in damper 2.

0

The total force applied to mass 1 due to spring 2 and damper 2 is equal to the total force applied to mass 2 also due to spring 2 and damper 2.

The relative velocity of 1Vrel is equal to velocity of mass 2 minus the velocity of mass 1.

ELECTRICAL CIRCUIT EXAMPLE

Figure A.5 shows a simple electrical circuit with a current source in series with a capacitor, both of which are in parallel with an inductor and a resistor. Figure A.6 shows a bond graph representation of this electrical circuit.

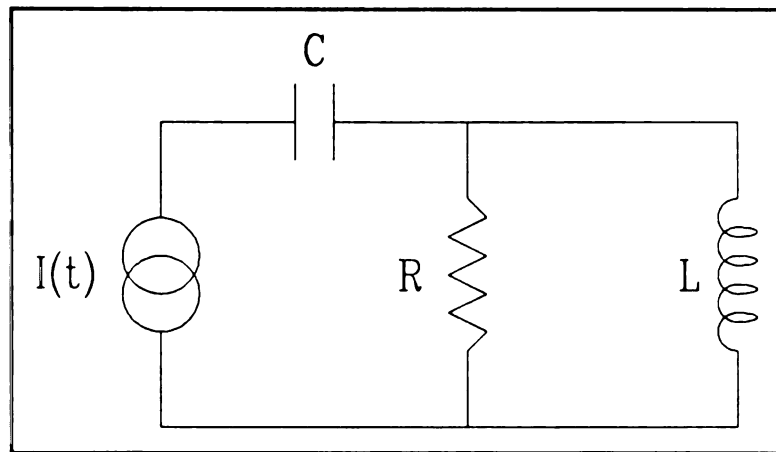


Figure A.5 Schematic of an Electrical Circuit

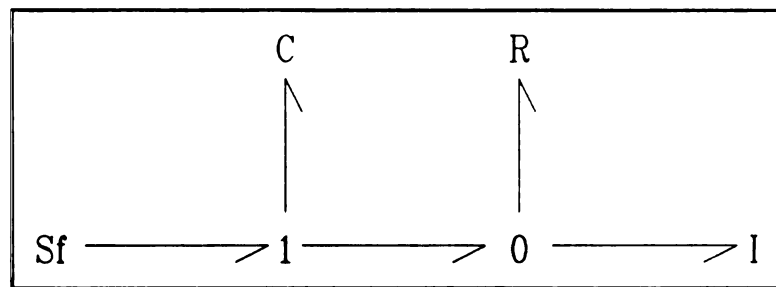


Figure A.6 Bond Graph Representation of the Circuit

The bond graph components of Figure A.6 have the following meanings:

Sf

The current source applied to the system, $I(t)$.

C

The capacitor.

R

The resistor.

I

The inductor.

1

The current source and capacitor are in series (meaning they have common currents) and their voltage drops added together is equal to the voltage drop of the rest of the system.

0

The resistor is in parallel with the inductor and also in parallel with the remainder of the system (i.e., the current source and capacitor combined). This means these all have the same voltage drop. Also, the current from the source/capacitor line is equal to the sum of the currents in the resistor and inductor lines.

APPENDIX B

INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

Object oriented programming is a relatively new idea in software programming that concentrates on the objects of a system rather than on the procedures that manipulate data, as conventional engineering software does (e.g., FORTRAN or PL/I) [1], [3], [8].

BASIC CONCEPTS

Defining object oriented programming precisely is difficult. There are many different interpretations of this idea. At an abstract level all agree that it is a set of principles, practices and procedures used to compose an effective computer implementation to solve a given problem. Some details of object oriented programming used in this dissertation are given below. The author believes that these definitions represent the majority view as derived from the literature written on the subject [4].

Object oriented programming is an abstract idea that proposes that large complex problems can be broken down into simple, easy to manage problems with the relatively few concepts of objects, classes, encapsulation, inheritance and messages.

Objects combine data and the code that operates on that data into a single useable structure. An object captures the state and the behavior of something. The code that operates on the data, or that represents the behavior, are called methods.

The way to manipulate an object's data is to send that object a message. If the message is understood by the object it will perform the requested action using its appropriate method(s). If not, the object will respond that the message was not understood. The localization of data manipulation by the object itself is known as encapsulation. One cannot manipulate an object's data structure without sending it a message.

Objects sharing common properties are grouped together in a class. A class provides a template for objects so that common objects are always stored and manipulated in a consistent manner. The class feature provides a consistency in the software that typically is lacking in more conventional languages. An object is an instance of a class.

In order to take further advantage of common data structures and methods, a class hierarchy is introduced. A particular class can have subclasses (dependents) and superclasses (parents and/or ancestors). This family-tree-like structure is referred to as the class hierarchy.

The advantage of a class hierarchy is that an object (instance of a particular class) inherits all of the properties of its superclasses. This means that the object not only has available to it its own data and methods, it also has access to the data structures and methods associated

with its superclasses. On the other hand, an object knows nothing about any of its subclasses.

An example of a class could be the class of rigid bodies. This class could be denoted by the symbol `RigidBody`. A specific rigid body, denoted by `aRigidBody`, is an instance of the `RigidBody` class. The methods of the `RigidBody` class would handle any messages that would be sent to `aRigidBody`. For example, if you wanted to draw `aRigidBody` on a graphics terminal you might send the message 'draw' to `aRigidBody`. If the `RigidBody` class had a method that understands this message then `aRigidBody` would perform the requested action using its specific data. In this example `aRigidBody` would draw itself using its geometric description data (i.e., its width, height, orientation angle and center location).

Taking the `RigidBody` class one step further, a simple class hierarchy can be identified. For example, one might identify two types of bodies: rigid bodies and deformable bodies. Here a class hierarchy might consist of three classes: `Body`, `RigidBody` and `DeformableBody`. This is depicted in Figure B.1. `Body` is the superclass of the other two classes, while `RigidBody` and `DeformableBody` are subclasses of `Body`. The `Body` class contains data structures and methods that are associated with both of its subclasses. These would be inherited by all instances of both `RigidBody` and `DeformableBody`. In other words, the attributes held in common by the subclasses are stored in their superclasses.

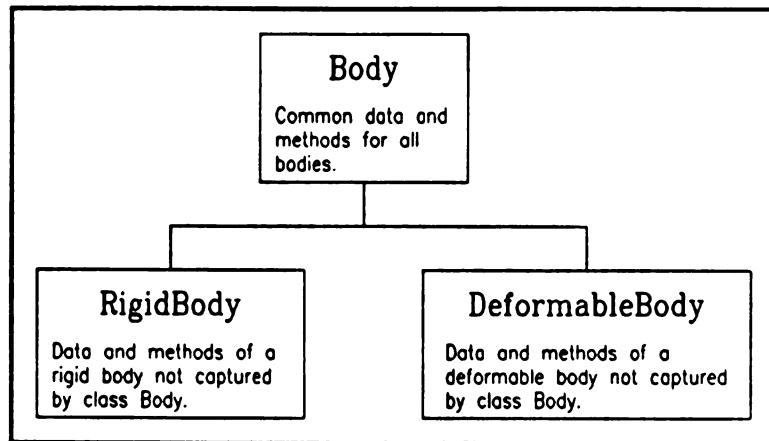


Figure B.1 A Simple Class Hierarchy

THE CONCEPTUAL DIFFERENCE

Conventional programming languages (e.g., FORTRAN, Pascal) organize their programs around procedures with data being passed from procedure to procedure. Object oriented programming focuses attention on the data as (fixed) objects. Messages are passed from object to object. The objects operate on themselves using their own methods based on the messages they receive.

From a certain point-of-view object oriented programming is the inverse of conventional programming. It recognizes that the data is an essential part of a system whereas conventionally the procedures are treated in this role. Figure B.2, taken from [74], gives us a better visualization of this concept. Notice also the encapsulation of the data in (b) by its methods. This data protection is sorely missing from (a).

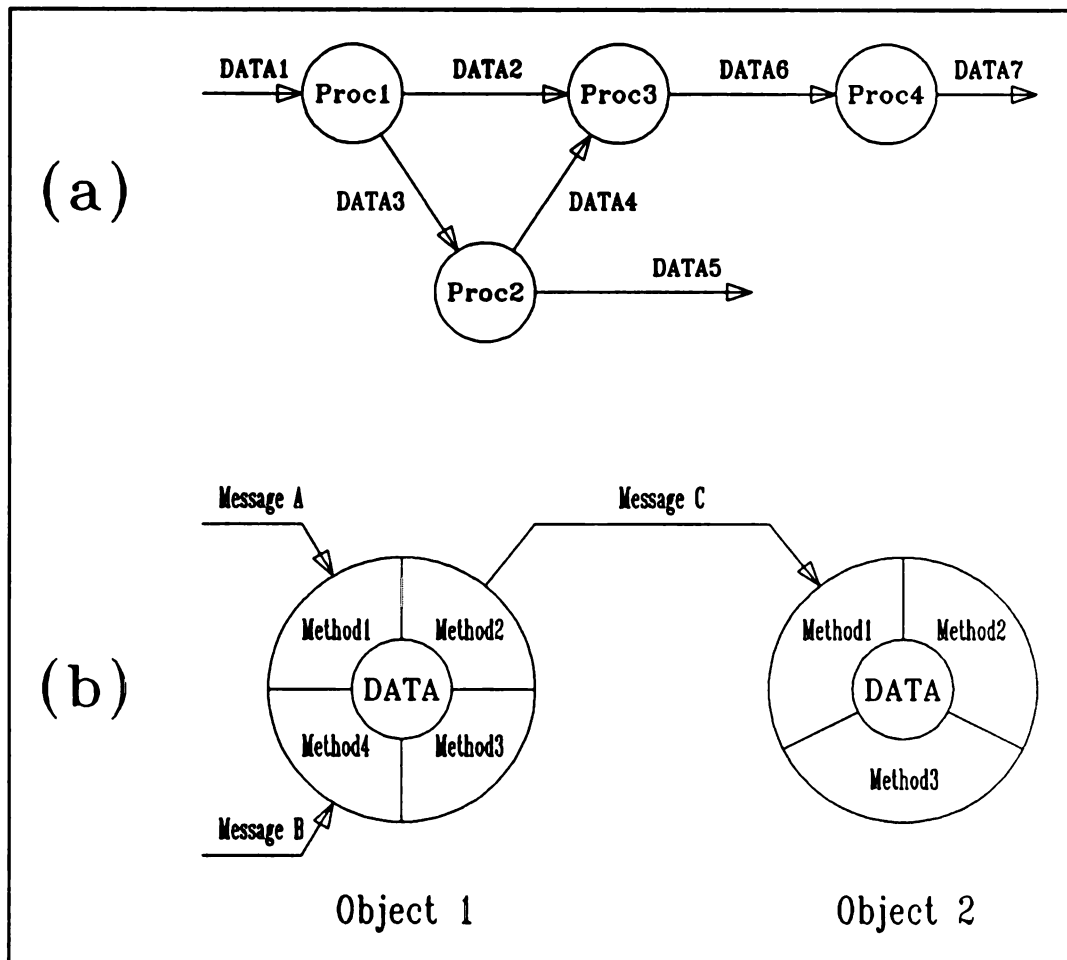


Figure B.2 (a) Procedural vs. (b) Object Oriented Organization

APPENDIX C

OOBProc IMPLEMENTATION DETAILS

This appendix gives some of the details of the classes developed during this research. Figure C.1 shows the main class hierarchy for OOBProc. Classes beginning with NG80 (NG80Object, NG80Arc, NG80Node, NG80View and NG80Controller) are part of the NodeGraph-80 software package used to help develop OOBProc. Classes Object, View, Controller and MouseMenuController are part of the base Smalltalk-80 system. The remaining classes shown were developed by the author.

In total, 41 new classes were created for OOBProc, 30 of which are shown in Figure C.1. The remaining 11 classes are supporting classes for the 30 main ones.

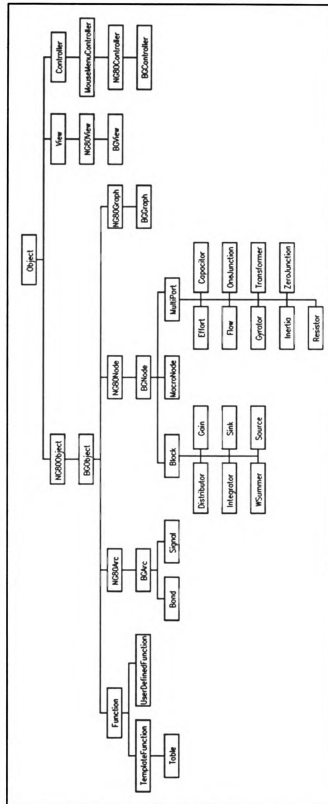


Figure C.1 OOBProc Class Hierarchy

A method was written to produce a description of each class without showing the detailed code of the class. This includes the class comment, the class category, the superclass, the subclasses, the instance variables, the class variables, the method categories and the method protocols for each class. What follows in this Appendix are the class descriptions for each of the 41 classes developed for OOBProc. These are organized in the following format:

Model-View-Controller Classes

Node Classes

Arc Classes

Function Classes

Miscellaneous Classes

The Model-View-Controller classes are composed of the following classes:

BGGraph

BGView

BGController

ArcView

ArcController

NodeView

NodeController

class name: BGGraph

class comment:

This class is used as the model for the bond graph processor.
J. Reid 4/89

class category: BG-Core

superclass: NG80Graph

subclasses:

instance variables:

bgComplete
causality
selectedArcIndex
selectedNodeIndex

class variables:

method categories followed by the instance methods:

accessing
bgComplete
bgComplete:
causality
causality:
locations
locations:
nodes
nodes:
selectedArcIndex
selectedArcIndex:
selectedNodeIndex
selectedNodeIndex:
causality
assignCausality
clearCausality
checking
checkBgComplete
checkIfLabelIsUnique:
checkVariable:
initialize
initialize
inquiry
getAllArcs
getAllNodes
getSysEqns
getUniqueLabel:defaultIsOK:
listOfArcs
listOfNodes
modifications
addArc:
addNode:

- editLabel:
- relabelArc:
- removeArc:
- removeNode:
- renameNode:
- replaceOldNode:withNewNode:
- releasing
 - release
- storing
 - saveOn:
 - stringDefinition

class method categories followed by the class methods:

- checking
 - checkVariable:
- instance creation
 - fromFile:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

```

class name: BGView

class comment:
    This class is used as the graphical view for the bond
    graph processor. J.Reid 4/89

class category: BG-Core

superclass: NG80View

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:
    accessing
        canvas
        nodeLocations
        nodeLocations:
    controller access
        defaultControllerClass
    displaying
        displayView
    displaying arcs
        calculateDisplayPts:to:middlePts:
        displayArc:
        displayArcOnCanvasXOR:
        displayArcXOR:
        displayBoldArcs:
        displayFatArc:
        endPtsOfAnArc:
        fastDisplayArcsXOR:
        reshapeDisplayArcXOR:
        unDisplayArc:
    displaying nodes
        unDisplayNodes:
    inquiry
        boundingBox
        centerOfCanvas
        centerOfView
        displayLocationOfNode:
        locationOfNode:
        locationsIncludesNode:
        nodeResponsibleForDisplay:
    modifications
        addNode:atLocation:
        addNodes:andArcs:at:
        insertNodes:andArcs:at:
        insertNodesScaled:andArcs:at:
        moveNode:to:
        pasteNodes:andArcs:at:

```

```

        pasteNodesScaled:andArcs:at:
        removeNode:
        updateCanvasSize
updating
        update:
        updateNodes

```

class method categories followed by the class methods:

```

example
    example
    example2
instance creation
    openMultipleViewsOn:
    openOn:
    openOnMacroNodeToPickAtomicNode:
    openSubGraphOn:
    openViewOnMacroNode:
system - fileOut
    classDescriptionBGBD
    classDescriptionNG80
    fileOutBGBD
    fileOutCategories
    fileOutClassDescription
    fileOutClassDescription:file:
    fileOutNG80
    fileOutST80

```

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990


```

class name: BGController

class comment:
    This class is used as the Controller for a bond graph
    processor. J. Reid 4/89

class category: BG-Core

superclass: NG80Controller

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:
    controlling
        viewHasCursor
        yellowButtonActivity
    menu messages
        convertUserFunction
        editTemplateFunction
        inspectFunctions
        notImplemented
        openGraphView
        openMultipleViews
    menu messages (arcs)
        addBond
        addSignal
        changeArcNodes
        convertPolyToSpline
        convertSplineToPoly
        editArc
        inspectArc
        polylineArc
        rectilinearPolylineArc
        relabelArc
        removeArc
        reverseArcDirection
        splineArc
        straightArc
    menu messages (nodes)
        addNode:
        changeNodeType
        copyNodes
        createMacroNode
        cutNodes
        editMacroNode
        expandMacroNode
        inspectNode
        modifyNodeFunction
        pasteMacroFromFile

```

```

    pasteNode
    pasteNodeOld
    removeNode
    removeNodes
    renameNode
    saveMacroToFile
    viewMacroNode
menu messages (graph)
    assignCausality
    clearCausality
    clearGraph
    getSysEqns
    inspectGraph
    inspectMVC
    pasteFromFile
    saveToFile
    scaleToRectangle
    scaleToView
    zoomIn
    zoomOut
prompt user
    displayTempMessage:
    getUniqueLabel:
    pickPoint:
    restoreScreen:
shaping arcs
    pointFromUserStartingAt:
    polylineFromUserFor:
    rectilinearPolylineFromUserFor:
utilities
    autoRelabel:
    changeBondNodes:
    changeLabel:
    changeSignalNodes:
    pasteFromFile:
    scale:zoomPoint:
utilities (nodes)
    copyNodes:
    moveNode:
    moveNodesAfterPaste:
    pickArc
    pickAtomicNode
    pickMacroNode
    pickNode
    placeNode:
    selectAtomicNodeStartingAt:
    selectNodeStartingAt:

class method categories followed by the class methods:
    class initialization
        initializePPS
time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
            on 13 February 1990

```

class name: ArcView

class comment:

This class implements the view for the arc MVC. The model is, at this point, a BGOBJECT. This view/controller of the model lists the arcs in the model and handles specific behavior that can be performed on those arcs.
J. Reid 6/89

class category: BG-ViewCtrls

superclass: ListView

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

updating
update:

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

```

class name: ArcController

class comment:
    This class implements the controller for the arc MVC.
    J. Reid    6/89

class category: BG-ViewCtrls

superclass: ListController

subclasses:

instance variables:

class variables:
    ArcYellowButtonMenu
    ArcYellowButtonMessages
    NoArcYellowButtonMenu
    NoArcYellowButtonMessages

method categories followed by the instance methods:
    controlling
        redButtonActivity
        yellowButtonActivity
    menu messages
        inspectArc
        noArcSelected
        relabelArc
        removeArc
    private
        changeModelSelection:

class method categories followed by the class methods:
    initialize
        initialize

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
            on 13 February 1990

```

class name: NodeView

class comment:

This class implements the view for the node MVC. The model is, at this point, a BGObject. This view/controller of the model lists the nodes in the model and handles specific behavior that can be performed on those nodes.

J. Reid 6/89

class category: BG-ViewCtrls

superclass: ListView

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

updating

update:

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: NodeController

class comment:

 This class implements the controller for the node MVC.
 J. Reid 6/89

class category: BG-ViewCtrls

superclass: ListController

subclasses:

instance variables:

class variables:

 NodeYellowButtonMessages
 NodeYellowButtonMenu
 NoNodeYellowButtonMenu
 NoNodeYellowButtonMessages

method categories followed by the instance methods:

 controlling
 redButtonActivity
 yellowButtonActivity

 menu messages
 inspectNode
 modifyNodeFunction
 noNodeSelected
 removeNode
 renameNode

 node utilities
 getAtomicNodeFrom:

 private
 changeModelSelection:

class method categories followed by the class methods:

 initialize
 initialize

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
 on 13 February 1990

The Node classes are composed of the following classes (indentations indicate inheritance structure):

BGNode

Block

Distributor

Gain

Integrator

Sink

Source

WSummer

MacroNode

MultiPort

Capacitor

Effort

Flow

Gyrator

Inertia

OneJunction

Resistor

Transformer

ZeroJunction

class name: BGNode

class comment:

The BGNode class is an abstract superclass used to implement the data and behavior of the nodes found in a bond graph and/or block diagram. J. Reid 5/89

class category: BG-Core

superclass: NG80Node

subclasses:

Block
MacroNode
MultiPort

instance variables:

bondPorts
signalPorts
function
partOfAMacro

class variables:

method categories followed by the instance methods:

accessing
bondPorts
bondPorts:
function
function:
icon
partOfAMacro
partOfAMacro:
signalPorts
signalPorts:
causality
assignArbitraryCausality
assignDesiredCausality
assignRequiredCausality
extendCausality
checking
checkFunction
copying
deepCopy
deepCopyIntoMacro:
initialize
assignFunction
initialize
initializeWith:
inquiry
arcsConnectedTo
boundingBoxOfIcon
findNodeWithLabel:

- findNodeWithUniqueID:
- getAllArcs
- getAllNodes
- getSysEqns
- includesInDefinition:
- modifications
 - addPort:
 - addPortKeepFunction:
 - changeLabel:
 - modifyFunction
 - removePort:
- releasing
 - release
- storing
 - saveOn:
 - stringDefinition

class method categories followed by the class methods:
instance creation
fromFile:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

```

class name: Block

class comment:
    This class is an abstract superclass of block diagram
    blocks. J. Reid 6/89

class category: BD-Nodes

superclass: BNode

subclasses:
    WSummer
    Distributor
    Source
    Gain
    Sink
    Integrator

instance variables:

class variables:

method categories followed by the instance methods:
    accessing
        iconType
    causality
        assignArbitraryCausality
    checking
        checkBgComplete
        checkIfBondIsValidPort
        checkIfSignalIsValidFromNode
        checkIfSignalIsValidToNode
    inquiry
        inputs
        outputs

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
            on 13 February 1990

```

class name: Distributor

class comment:

This class represents the block diagram distributor element.
It has a single signal input and multiple signal outputs,
all equal to the input. J. Reid 12/89

class category: BD-Nodes

superclass: Block

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

causality
 assignArbitraryCausality
checking
 checkBgComplete
inquiry
 getSysEqns

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: Gain

class comment:

This class is used as a block diagram gain element. The output signal is equal to the input signal multiplied by a constant (default). Behaves exactly like the standard Block node (currently). J. Reid 7/89

class category: BD-Nodes

superclass: Block

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: Integrator

class comment:

This class is used as a block diagram integrator element.
The integrator integrates an input signal and produces
the result as its output signal. J. Reid 7/89

class category: BD-Nodes

superclass: Block

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

initialize
assignFunction

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: Sink

class comment:

 This class implements the block diagram sink element.
 It has exactly one signal input and no outputs. J.
 Reid 2/90

class category: BD-Nodes

superclass: Block

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

 checking
 checkBgComplete
 causality
 assignArbitraryCausality
 inquiry
 getSysEqns

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
 on 13 February 1990

class name: Source

class comment:

This class implements the block diagram source element.
It has exactly one signal output based on a user defined
function of time (default). J. Reid 7/89

class category: BD-Nodes

superclass: Block

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

checking
 checkBgComplete
inquiry
 inputs

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: WSummer

class comment:

This class implements the block diagram weighted summer block. It produces a signal output based on the sum of multiple signal inputs, each multiplied by a constant.
J. Reid 7/89

class category: BD-Nodes

superclass: Block

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

checking
 checkBgComplete
initialize
 assignFunction

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: MacroNode

class comment:

This class implements macro modeling capabilities. A macro node is composed of other node objects, including other macro nodes. A macro node can not contain itself. J.

Reid 7/89

class category: BG-Core

superclass: BGNode

subclasses:

instance variables:

arcs
nodes
locations

class variables:

method categories followed by the instance methods:

accessing
arcs
arcs:
iconType
locations
locations:
nodes
nodes:
causality
assignArbitraryCausality
assignDesiredCausality
assignRequiredCausality
checking
checkBgComplete
copying
deepCopyIntoMacro:
initialize
initialize
inquiry
arcsConnectedTo
findNodeWithLabel:
findNodeWithUniqueID:
getAllArcs
getAllNodes
getSysEqns
includes:
includesInDefinition:
releasing
release
storing
stringDefinition

class method categories followed by the class methods:
instance creation
fromFile:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: MultiPort

class comment:

This class is used as an abstract superclass for bond graph multiport elements (e.g., I, Se, R, GY, 1, 0 , etc.). J. Reid 5/89

class category: BG-Nodes

superclass: BGNode

subclasses:

- Effort
- Inertia
- Gyrator
- Flow
- ZeroJunction
- OneJunction
- Transformer
- Capacitor
- Resistor

instance variables:

class variables:

method categories followed by the instance methods:

- accessing
 - iconType
 - label:
- checking
 - checkBgComplete
 - checkIfBondIsValidPort
 - checkIfSignalIsValidFromNode
 - checkIfSignalIsValidToNode

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: Capacitor

class comment:

This class represents the bond graph capacitor element.
J. Reid 10/89

class category: BG-Nodes

superclass: MultiPort

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

- causality
 - assignDesiredCausality
- inquiry
 - getSysEqns
 - inputs
 - outputs
 - stateEquation

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: Effort

class comment:

This class represents the bond graph Source of Effort
element. J. Reid 5/89

class category: BG-Nodes

superclass: MultiPort

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

causality
 assignRequiredCausality
inquiry
 inputs
 outputs

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

```
class name: Flow

class comment:
    This class represents the bond graph Source of Flow element.
    J. Reid 11/89

class category: BG-Nodes

superclass: MultiPort

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:
    causality
        assignRequiredCausality
    inquiry
        inputs
        outputs

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
            on 13 February 1990
```

class name: Gyrator

class comment:

This class represents the bond graph gyrator element.
J. Reid 5/89

class category: BG-Nodes

superclass: MultiPort

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

- causality
 - assignArbitraryCausality
 - extendCausality
- checking
 - checkBgComplete
- initialize
 - assignFunction
- inquiry
 - getSysEqns
 - inputs
 - outputs
- modifications
 - changeLabel:

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

```

class name: Inertia

class comment:
    This class represents the bond graph inertia element.
    J. Reid    5/89

class category: BG-Nodes

superclass: MultiPort

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:
    causality
        assignDesiredCausality
    inquiry
        getSysEqns
        inputs
        outputs
        stateEquation

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
            on 13 February 1990

```


class name: OneJunction

class comment:

This class represents the bond graph 1-Junction element.
J. Reid 5/89

class category: BG-Nodes

superclass: MultiPort

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

- causality
 - causalBond
 - extendCausality
- checking
 - checkBgComplete
 - checkIfSignalIsValidFromNode
 - checkIfSignalIsValidToNode
- inquiry
 - effortEqn
 - getSysEqns

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: Resistor

class comment:

This class represents the bond graph resistor element.
J. Reid 5/89

class category: BG-Nodes

superclass: MultiPort

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

causality
 assignArbitraryCausality
inquiry
 inputs
 outputs

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

```

class name: Transformer

class comment:
    This class represents the bond graph transformer element.
    J. Reid 10/89

class category: BG-Nodes

superclass: MultiPort

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:
    causality
        assignArbitraryCausality
        extendCausality
    checking
        checkBgComplete
    initialize
        assignFunction
    inquiry
        getSysEqns
        inputs
        outputs
    modifications
        changeLabel:

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
            on 13 February 1990

```

class name: ZeroJunction

class comment:

 This class represents the bond graph 0-Junction element.
 J. Reid 10/89

class category: BG-Nodes

superclass: MultiPort

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

- causality
 - causalBond
 - extendCausality
- checking
 - checkBgComplete
 - checkIfSignalIsValidFromNode
 - checkIfSignalIsValidToNode
- inquiry
 - effortEqn
 - flowEqn
 - getSysEqns

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
 on 13 February 1990

The Arc classes are composed of the following classes (indentations indicate inheritance structure):

 BGArc

 Bond

 Signal

class name: BGArc

class comment:

The BGArc class is an abstract superclass used to implement common data and behavior of the bonds and signals found in a bond graph and/or block diagram. J. Reid 5/89

class category: BG-Core

superclass: NG80Arc

subclasses:

Signal

Bond

instance variables:

class variables:

method categories followed by the instance methods:

accessing

label:

causality

clearCausality

copying

copy

initialize

initializeFrom:to:label:

initKeepFunctionFrom:to:

modifications

replaceOldNode:withNewNode:

releasing

release

storing

saveOn:

stringDefinition

class method categories followed by the class methods:

instance creation

fromFile:for:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

```

class name: Bond

class comment:
    This class represents the bond graph bond element.  J.
    Reid 5/89

class category: BG-Arcs

superclass: BGArc

subclasses:

instance variables:
    causalNode
    oldCausalNodeLabel

class variables:

method categories followed by the instance methods:
    accessing
        causalNode
        causalNode:
        label:
        oldCausalNodeLabel
        oldCausalNodeLabel:
    causality
        assignCausality:node:
        clearCausality
    checking
        checkVariable:
    copying
        copy
    displaying
        polylineDisplayOn:from:to:inside:using:
        splineDisplayOn:from:to:inside:using:
        straightDisplayOn:from:to:inside:using:
    modifications
        reverseDirection
    releasing
        release
    storing
        stringDefinition

class method categories followed by the class methods:
    instance creation
        fromFile:for:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
            on 13 February 1990

```

class name: Signal

class comment:

This class represents the block diagram signal element.
J. Reid 5/89

class category: BG-Arcs

superclass: BGArc

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

checking

checkVariable:

displaying

polylineDisplayOn:from:to:inside:using:

splineDisplayOn:from:to:inside:using:

straightDisplayOn:from:to:inside:using:

modifications

reverseDirection

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

The Function classes are composed of the following classes (indentations indicate inheritance structure):

Function

 TemplateFunction

 Table

 UserDefinedFunction

FunctionController

FunctionView

class name: Function

class comment:

This class is used as an abstract superclass for mathematical definitions. Basically used for a convenient superclass hierarchy. J. Reid 5/89

class category: BG-Functions

superclass: BGOBJECT

subclasses:

UserDefinedFunction
TemplateFunction

instance variables:

class variables:

method categories followed by the instance methods:

modifications
changeVariable:to:
modFunctionOld:
modifyFunction:

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: TemplateFunction

class comment:

This class implements generic function definitions that are stored in text files. Specific input variables, output variables and parameters are stored, along with the function type. When needed, the specific variables are substituted into the generic definitions to produce complete mathematical functions. J. Reid 5/89

class category: BG-Functions

superclass: Function

subclasses:

Table

instance variables:

inputs
outputs
parameters

class variables:

method categories followed by the instance methods:

accessing
inputs
inputs:
outputs
outputs:
parameters
parameters:
copying
deepCopy
function handling
getFileName
readContentsOfEntireFunction
readFunction
initialize
name:inputs:outputs:
setDefaultParameters
inquiry
getSysEqns
getVarSysEqns:numberVariables:
modifications
changeVariable:to:
convertToTemplate
modifyFunction
modifyInputs
modifyParameters
updateFunction
releasing
release

storing
stringDefinition

class method categories followed by the class methods:
instance creation
fromFile:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: Table

class comment:

This class implements a table function. That is, a set of (x,y) points. This could have been implemented by a template function, but it is being used as a test case for more advanced functional capabilities. J. Reid 11/89

class category: BG-Functions

superclass: TemplateFunction

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

- function handling
 - readContentsOfEntireFunction
- initialize
 - setDefaultParameters
- inquiry
 - getSysEqns
- modifications
 - modifyParameters
 - updateFunction

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: UserDefinedFunction

class comment:

This class is used to store a user defined function.
 The user is responsible for the existence of each variable
 and the correctness of the function. The function definition
 is stored as text in i.v. systemEqns. J. Reid 6/89

class category: BG-Functions

superclass: Function

subclasses:

instance variables:

systemEqns

class variables:

method categories followed by the instance methods:

accessing

systemEqns

systemEqns:

inquiry

getSysEqns

inputs

modifications

convertToTemplate

modifyFunction

modifyFunction:

stringChanged:

releasing

release

storing

stringDefinition

class method categories followed by the class methods:

instance creation

fromFile:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
 on 13 February 1990

class name: FunctionController

class comment:

 This class is used to control the modifications of a Function.
 J. Reid 6/89

class category: BG-Functions

superclass: Controller

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

 controlling
 startUp

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
 on 13 February 1990

class name: FunctionView

class comment:

This class is used to view the function while it is being
worked on by the function. J. Reid 6/89

class category: BG-Functions

superclass: View

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

controller access

defaultControllerClass

class method categories followed by the class methods:

instance creation

openOn:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

The Miscellaneous class are composed of the following classes:

BGIcon

BGObject

NodeAndPointDictionary

NodeCollection

StringEditorController

StringEditorView

class name: BGIcon

class comment:

This class is used for the icons that are associated with
the nodes in the bond graph processor. (Display form
- picture.) J. Reid 4/89

class category: BG-Utilities

superclass: NG80Icon

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

copying

deepCopy

icon creation

createIconOfType:withLabel:

createInvisibleRectangleIconWithLabel:

createRectangleIconWithLabel:

createRoundedRectangleIconWithLabel:

inquiry

boundingBox

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: BGObject

class comment:

This class is used as an abstract superclass for the main components that make up a bond graph. (eg, bonds, signals, multiports, inertias, icons, functions, etc.) J. Reid
5/89

class category: BG-Core

superclass: NG80Object

subclasses:

NG80Arc
NG80Icon
NG80Node
NG80Graph
Function

instance variables:

label

class variables:

method categories followed by the instance methods:

accessing
label
label:
modifications
changeLabel:
notifying
inform:
messageHandling:from:
printing
printOn:

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

class name: NodeAndPointDictionary

class comment:

This class is a dictionary with: keys: Instances of
 BGNode (or subclasses of) values: Points It is used
 for storing a point directly with a particular node.
 For example, the location of the node on a Form to be
 displayed - e.g., BGView. It provides the methods that
 one would want to apply to such a Dictionary - e.g.,
 Calculating the bounding box. J. Reid 8/89

class category: BG-Utilities

superclass: Dictionary

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

copying
 deepCopy
 inquiry
 boundingBox
 boundingBoxIncludingArcs:
 internalArcs
 locationOfNode:
 locationsIncludesNode:
 nodes
 modifications
 addNode:atLocation:
 moveNode:to:
 removeNode:
 release
 release

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
 on 13 February 1990

class name: NodeCollection

class comment:

This class is used as a an ordered collection of nodes.
 It's members are assumed to be instances of BGNode
 (or subclasses of). It provides the methods that one
 would want to apply to a collection of such nodes. J.
 Reid 8/89

class category: BG-Utilities

superclass: OrderedCollection

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

inquiry
 findNodeWithLabel:
 findNodeWithUniqueID:
 includesInDefinition:
 internalArcs
 release
 release

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
 on 13 February 1990

class name: StringEditorController

class comment:

This class is used as the controller for StringEditorView.
It differs from TextCollectorController in only one
aspect. It tells the views stringOwner that a change
has been made and accepted to its string. J. Reid 7/89

class category: BG-ViewCtrls

superclass: TextCollectorController

subclasses:

instance variables:

class variables:

method categories followed by the instance methods:

menu messages
accept

class method categories followed by the class methods:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

LIST OF REFERENCES

class name: StringEditorView

class comment:

This class is used to edit a string using a window. The difference between this class and TextCollectorView is that StringEditorView can be used in stream by any object that wants to edit a string. StringEditorController reports the edited string back to the string owner object with the message ''stringChanged: newString''. J. Reid
7/89

class category: BG-ViewCtrls

superclass: TextCollectorView

subclasses:

instance variables:

stringOwner

class variables:

method categories followed by the instance methods:

accessing
stringOwner
stringOwner:
controller access
defaultControllerClass

class method categories followed by the class methods:

instance creation
openWithString:label:owner:

time stamp: Smalltalk-80, Version 2.3 of 13 June 1988
on 13 February 1990

LIST OF REFERENCES

- [1] B.J. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.
- [2] L.S. Levy, *Taming the Tiger: Software Engineering and Software Economics*, Springer-Verlag, 1987.
- [3] BYTE, Special Issue on Object-Oriented Programming, August 1986.
- [4] G.E. Peterson, *Object-Oriented Computing*, Computer Society of the IEEE, Volume 1 and 2, 1987.
- [5] M. Stefik and D.G. Bobrow, "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, Winter 1986, 40-62.
- [6] B.J. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology," *IEEE Software*, January 1984, 50-61.
- [7] M. Floyd, "A Class Act," *Dr. Dobbs's Journal*, April 1989, 58-64.
- [8] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.
- [9] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1984.
- [10] B. Stroustrup, "A Better C?," *BYTE*, August 1988, 215- 216D.
- [11] B.P. Zeigler, *Theory of Modelling and Simulation*, Krieger Publishing Co., Malabar, Florida, 1976.
- [12] W.E. Biles, "Introduction To Simulation," *Proceedings of the 1987 Winter Simulation Conference*, December 1987, 7-15.
- [13] P.F. Roth, "Discrete, Continuous and Combined Simulation," *Proceedings of the 1987 Winter Simulation Conference*, December 1987, 25-29.
- [14] J. Banks and J.S. Carson, *Discrete-Event System Simulation*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.
- [15] R.J. Ord-Smith and J. Stephenson, *Computer Simulation of Continuous Systems*, Cambridge University Press, 1975.

- [16] I. Bausch-Gall, "Continuous System Simulation Languages," 3rd Seminar on Advanced Vehicle System Dynamics, Amalfi, Italy, May 1986, 347-366.
- [17] D. Conner, "Mixed Analog-Digital Simulators," *EDN*, July 20, 1989, 160-166.
- [18] "MCAE Enters The Picture at Polaroid," *Mechanical Engineering*, October 1989, Vol 111, No. 10, 50-53.
- [19] J. McLeod, "Computer Modelling and Simulation: The Changing Challenge," *Simulation*, March 1986, 114-118.
- [20] K. Ogata, *Modern Control Engineering*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1970.
- [21] Y. Takahashi, M.J. Rabins and D.M. Auslander, *Control and Dynamic Systems*, Addison-Wesley Publishing Co., 1972.
- [22] J. Van de Vegte, *Feedback Control Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.
- [23] D. Karnopp and R. Rosenberg, *System Dynamics: A Unified Approach*, John Wiley and Sons, NY, 1975.
- [24] R. Rosenberg and D. Karnopp, *Introduction to Physical System Dynamics*, McGraw-Hill, NY, 1983.
- [25] A.M. Bos and P.C. Breedveld, "1985 Update of the Bond Graph Bibliography," *J. Franklin Institute*, Vol. 319, No. 1/2, 1985, 269-286.
- [26] Z. Zalewski and R.C. Rosenberg, "Simulation of Engineering Models Containing Bond Graphs and Block Diagrams," *Proc. 1986 ASME Joint PVP & Computers in Engineering Conf.*, 1986.
- [27] R. Rosenberg, *The ENPORT QuickGuide*, ROSENCODE Associates, 1987.
- [28] J.J. Granda, "Computer Generation of Physical System Differential Equations Using Bond Graphs," *J. Franklin Institute*, Vol. 319, No. 1/2, 1985, 243-255.
- [29] J.J.A.J. Beukeboom, J.J. van Dixhoorn and J.W. Meerman, "Simulation of Mixed Bond Graphs and Block Diagrams on Personal Computers Using TUTSIM," *J. Franklin Institute*, Vol. 319, No. 1/2, 1985, 257-267.
- [30] R.C. Rosenberg and Z. Zalewski, "Macro Modeling of Engineering Systems," *ASME Paper 86-WA/DSC-12*, Presented at the Winter Annual Meeting, Anaheim, California, 1986.
- [31] W.S. Scott, et al, editors, *1986 VLSI Tools*, Computer Science Division, EECS Department, University of California at Berkeley, 1986.

- [32] T. Quarles, et al, *SPICE 3B1 User's Guide*, EECS Department, University of California, Berkeley, CA, 1987.
- [33] Intergraph Corporation, *Mechanical Engineering Design System I and II (MEDS) Course Guides*, Huntsville, Alabama, 1988.
- [34] Mechanical Dynamics, Inc., *ADAMS User's Manual*, MDI, October 1987, Ann Arbor, MI.
- [35] E.J. Haug, *Computer Aided Kinematics and Dynamics of Mechanical Systems*, Allyn and Bacon, Needham Heights, Mass., 1989.
- [36] T. Nelson, "The Zoo Story," *Creative Computing*, October 1980, 62-70.
- [37] US Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983.
- [38] N. Wirth, *Programming in Modula-2*, Springer-Verlag, 1983.
- [39] E. Seidewitz, "Object-Oriented Programming in Smalltalk and Ada," *OOPSLA '87 Proceedings*, October 1987, 202-213.
- [40] S.R. Ladd, "Comparing Modula-2 and C++," *Dr. Dobb's Journal*, January 1989, 62-68.
- [41] M. Rettig, et al, "Object-Oriented Programming in AI: New Choices," *AI Expert*, January 1989, 53-70.
- [42] C. Schaffert, et al, "An Introduction to Trellis/Owl," *OOPSLA '86 Proceedings*, September 1986, 9-16.
- [43] C. Duff, *Introducing Actor*, 43 page booklet about Actor, The Whitewater Group, 1987.
- [44] S. Dewhurst and K. Stark, *Programming in C++*, Prentice Hall, Englewood Cliffs, NJ, 1989.
- [45] E.R. Tello, "Objective-C," *Dr. Dobb's Journal*, August 1988, 56-69.
- [46] D.G. Bobrow, et al, "CommonLoops: Merging Lisp and Object-Oriented Programming," *OOPSLA '86 Proceedings*, September 1986, 17-29.
- [47] D.A. Moon, "Object-Oriented Programming with Flavors," *OOPSLA '86 Proceedings*, September 1986, 1- 8.
- [48] J.P. Jacky and I.J. Kalet, "An Object-Oriented Programming Discipline for Standard Pascal," *Communications of the ACM*, September 1987, Vol. 30, Number 9, 772-776.
- [49] M. Floyd, "Turbo Pascal with Objects," *Dr. Dobb's Journal*, July 1989, 56-63.

- [50] J. Rothenberg, "Object-Oriented Simulation: Where Do We Go From Here?" *Proceedings of the 1986 Winter Simulation Conference*, December 1986, 464-469.
- [51] V. Knapp, "The Smalltalk Simulation Environment," *Proceedings of the 1986 Winter Simulation Conference*, December 1986, 125-128.
- [52] O.M. Ulgen and T. Thomasma, "Simulation Modeling in an Object-Oriented Environment Using Smalltalk-80," *Proceedings of the 1986 Winter Simulation Conference*, December 1986, 474-484.
- [53] R.P. Rich, "Message Oriented Simulation Language," *Proceedings of the 1988 Summer Computer Simulation Conference*, July 1988, 303-308.
- [54] B.P. Zeigler, "Hierarchical, Modular Discrete-Event Modelling in an Object-Oriented Environment," *Simulation*, November 1987, 219-230.
- [55] S. Ruiz-Mier and J. Talavage, "A Hybrid Paradigm for Modeling of Complex Systems," *Simulation*, April 1987, 135-141.
- [56] T.S. Larkin, et al, "Simulation and Object-Oriented Programming: The Development of SERB," *Simulation*, September 1988, 93-100.
- [57] B. Gates, et al, "A Demon Facility for Object- Oriented Simulation Languages," *Proceedings of the 1988 Summer Computer Simulation Conference*, July 1988, 667-673.
- [58] A. Guasch and R.C. Huntsinger, "Object Oriented Continuous System Simulation," *Proceedings of the 1989 Summer Computer Simulation Conference*, July 1989, 562-565.
- [59] J. Sung, *The Development of a Kinematic Solver Based on Object-Oriented Programming Principles*, Master's Thesis, Mechanical Engineering, Michigan State University, Fall 1989.
- [60] P.G. Kaumbutho, *A Bond Graph Model for Simulating the Performance of a Farm Tractor*, PhD Dissertation, Agricultural Engineering, Michigan State University, 1987.
- [61] E.J. Haug, "Elements and Methods of Computational Dynamics," *Computer Aided Analysis and Optimization of Mechanical System Dynamics*, NATO ASI Serie, Vol. F9, Springer-Verlag, 1984, 3-38.
- [62] D. Karnopp, "Lagrange's Equations for Complex Bond Graph Systems," *Journal of Dynamic Systems, Measurement, and Control*, December 1977, 300-306.
- [63] I. Jacobson, "Object Oriented Development in an Industrial Environment," *OOPSLA '87 Proceedings*, October 1987, 183-191.
- [64] R. Wirfs-Brock and B. Wilkerson, "Object-Oriented Design: A Responsibility-Driven Approach," *OOPSLA '89 Proceedings*, 1989.

- [65] S.T. Pope, et al, "Object-Oriented Approaches to the Software Lifecycle Using the Smalltalk-80 System as a CASE Toolkit," *1987 Fall Joint Computer Conference, ACM-IEEE*, 1987, 13-20.
- [66] S.S. Adams, *NodeGraph-80 Version 1.0 User Manual*, Knowledge Systems Corporation, 1987.
- [67] M. Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press, New York, 1980.
- [68] S.L. Pfleeger, *Software Engineering*, Macmillan Publishing Company, New York, 1987.
- [69] A. Wirfs-Brock and B. Wilkerson, "Variables Limit Reusability," *Journal of Object-Oriented Programming*, May/June 1989, Vol. 2, No. 1, 34-40.
- [70] W. Cunningham and K. Beck, "A Diagram for Object-Oriented Programs," *OOPSLA '86 Proceedings*, September 1986, 361-367.
- [71] D. Thomas, "The Time/Space Requirements of Object-Oriented Programs," *Journal of Object-Oriented Programming*, March/April 1989.
- [72] R.L. Peskin, et al, "Smalltalk - The Next Generation Scientific Computing Interface?," *Mathematics and Computers in Simulation*, Vol. 31, Numbers 4 & 5, October 1989, 371-381.
- [73] D. Ungar, Letter to the Editor, *Journal of Object-Oriented Programming*, September/October 1989, Vol. 2, No. 3, 76-77.
- [74] R. Wilson, "Object-Oriented Languages Reorient Programming Techniques," *Computer Design*, November 1, 1987, 52-62.

MICHIGAN STATE UNIV. LIBRARIES



31293007849197